

Science, Math, and Code for Realistic Effects

2nd Edition



Physics for Game Developers



O'REILLY®

*David M. Bourg
& Bryan Bywalec*

www.allitebooks.com

Physics for Game Developers

If you want to enrich your game's experience with physics-based realism, the expanded edition of this classic book details physics principles applicable to game development. You'll learn about collisions, explosions, sound, projectiles, and other effects used in games on Wii, PlayStation, Xbox, smartphones, and tablets. You'll also get a handle on how to take advantage of various sensors such as accelerometers and optical tracking devices.

Authors David Bourg and Bryan Bywalec show you how to develop your own solutions to a variety of problems by providing technical background, formulas, and a few code examples. This updated book is indispensable whether you work alone or as part of a team.

- Refresh your knowledge of classical mechanics, including kinematics, force, kinetics, and collision response
- Explore rigid body dynamics, using real-time 2D and 3D simulations to handle rotation and inertia
- Apply concepts to real-world problems: model the behavior of boats, airplanes, cars, and sports balls
- Enhance your games with digital physics, using accelerometers, touch screens, GPS, optical tracking devices, and 3D displays
- Capture 3D sound effects with the OpenAL audio API

David Bourg, owner of MiNO Marine—a Naval architecture and marine services firm—also formed a company in the 1990s that developed children's games, casino games, and various PC to Mac ports. He's the co-author of *AI for Game Programmers* (O'Reilly).

Bryan Bywalec is an architect at MiNO Marine, where accurate simulation of the physical world is necessary on a daily basis. In his passion for physics, he enjoys modding games (like Kerble Space Program) that places physics on center stage.

US \$44.99

CAN \$47.99

ISBN: 978-1-449-39251-2



5 4 4 9 9
9 781449 392512



"Physics for Game Developers has a wealth of information based on real world physics problems that is immediately usable in game code."

—Paul Zirkle
*Lead Games Engineer
at Disney Interactive*

Twitter: @oreillymedia
facebook.com/oreilly

O'REILLY®
oreilly.com

SECOND EDITION

Physics for Game Developers

David M. Bourg and Bryan Bywalec

O'REILLY®

Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo

www.allitebooks.com

Physics for Game Developers, Second Edition

by David M. Bourg and Bryan Bywalec

Copyright © 2013 David M. Bourg and Bryan Bywalec. All rights reserved.

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (<http://my.safaribooksonline.com>). For more information, contact our corporate/institutional sales department: 800-998-9938 or corporate@oreilly.com.

Editors: Andy Oram and Rachel Roumeliotis

Indexer: Lucie Haskins

Production Editor: Christopher Hearse

Cover Designer: Randy Comer

Copyeditor: Rachel Monaghan

Interior Designer: David Futato

Proofreader: Amanda Kersey

Illustrator: Rebecca Demarest

April 2013: Second Edition

Revision History for the Second Edition:

2013-04-09: First release

See <http://oreilly.com/catalog/errata.csp?isbn=9781449392512> for release details.

Nutshell Handbook, the Nutshell Handbook logo, and the O'Reilly logo are registered trademarks of O'Reilly Media, Inc. *Physics for Game Developers*, 2nd Edition, the image of a cat and mouse, and related trade dress are trademarks of O'Reilly Media, Inc.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and O'Reilly Media, Inc., was aware of a trademark claim, the designations have been printed in caps or initial caps.

While every precaution has been taken in the preparation of this book, the publisher and authors assume no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein.

ISBN: 978-1-449-39251-2

[LSI]

Table of Contents

Preface.....	xı
--------------	----

Part I. Fundamentals

1. Basic Concepts.....	3
Newton's Laws of Motion	3
Units and Measures	4
Coordinate System	6
Vectors	7
Derivatives and Integrals	8
Mass, Center of Mass, and Moment of Inertia	9
Newton's Second Law of Motion	20
Inertia Tensor	24
Relativistic Time	29
2. Kinematics.....	35
Velocity and Acceleration	36
Constant Acceleration	39
Nonconstant Acceleration	41
2D Particle Kinematics	42
3D Particle Kinematics	45
X Components	46
Y Components	47
Z Components	48
The Vectors	48
Hitting the Target	49
Kinematic Particle Explosion	54
Rigid-Body Kinematics	61
Local Coordinate Axes	62

Angular Velocity and Acceleration	62
3. Force.....	71
Forces	71
Force Fields	72
Friction	73
Fluid Dynamic Drag	75
Pressure	76
Buoyancy	77
Springs and Dampers	79
Force and Torque	80
Summary	83
4. Kinetics.....	85
Particle Kinetics in 2D	87
Particle Kinetics in 3D	91
X Components	94
Y Components	95
Z Components	95
Cannon Revised	95
Rigid-Body Kinetics	99
5. Collisions.....	103
Impulse-Momentum Principle	104
Impact	105
Linear and Angular Impulse	112
Friction	115
6. Projectiles.....	119
Simple Trajectories	120
Drag	124
Magnus Effect	132
Variable Mass	138

Part II. Rigid-Body Dynamics

7. Real-Time Simulations.....	143
Integrating the Equations of Motion	144
Euler's Method	146
Better Methods	153

Summary	159
8. Particles.....	161
Simple Particle Model	166
Integrator	169
Rendering	170
The Basic Simulator	170
Implementing External Forces	172
Implementing Collisions	175
Particle-to-Ground Collisions	175
Particle-to-Obstacle Collisions	181
Tuning	186
9. 2D Rigid-Body Simulator.....	189
Model	190
Transforming Coordinates	197
Integrator	198
Rendering	200
The Basic Simulator	201
Tuning	204
10. Implementing Collision Response.....	205
Linear Collision Response	206
Angular Effects	213
11. Rotation in 3D Rigid-Body Simulators.....	227
Rotation Matrices	228
Quaternions	232
Quaternion Operations	234
Quaternions in 3D Simulators	239
12. 3D Rigid-Body Simulator.....	243
Model	243
Integration	247
Flight Controls	250
13. Connecting Objects.....	255
Springs and Dampers	257
Connecting Particles	258
Rope	258
Connecting Rigid Bodies	265
Links	265

Rotational Restraint	275
14. Physics Engines.....	281
Building Your Own Physics Engine	281
Physics Models	283
Simulated Objects Manager	284
Collision Detection	285
Collision Response	286
Force Effectors	287
Numerical Integrator	288
<hr/>	
Part III. Physical Modeling	
15. Aircraft.....	293
Geometry	294
Lift and Drag	297
Other Forces	302
Control	303
Modeling	305
16. Ships and Boats.....	321
Stability and Sinking	323
Stability	323
Sinking	325
Ship Motions	326
Heave	327
Roll	327
Pitch	328
Coupled Motions	328
Resistance and Propulsion	328
General Resistance	328
Propulsion	334
Maneuverability	335
Rudders and Thrust Vectoring	336
17. Cars and Hovercraft.....	339
Cars	339
Resistance	339
Power	340
Stopping Distance	341
Steering	342

Hovercraft	345
How Hovercraft Work	345
Resistance	347
Steering	350
18. Guns and Explosions.....	353
Projectile Motion	353
Taking Aim	355
Zeroing the Sights	357
Breathing and Body Position	360
Recoil and Impact	361
Explosions	362
Particle Explosions	363
Polygon Explosions	366
19. Sports.....	369
Modeling a Golf Swing	370
Solving the Golf Swing Equations	373
Billiards	378
Implementation	380
Initialization	383
Stepping the Simulation	386
Calculating Forces	388
Handling Collisions	393

Part IV. Digital Physics

20. Touch Screens.....	403
Types of Touch Screens	403
Resistive	403
Capacitive	404
Infrared and Optical Imaging	404
Exotic: Dispersive Signal and Surface Acoustic Wave	404
Step-by-Step Physics	404
Resistive Touch Screens	404
Capacitive Touch Screens	408
Example Program	410
Multitouch	410
Other Considerations	411
Haptic Feedback	411
Modeling Touch Screens in Games	411

Difference from Mouse-Based Input	412
Custom Gestures	412
21. Accelerometers.....	413
Accelerometer Theory	414
MEMS Accelerometers	416
Common Accelerometer Specifications	417
Data Clipping	417
Sensing Orientation	418
Sensing Tilt	420
Using Tilt to Control a Sprite	420
Two Degrees of Freedom	421
22. Gaming from One Place to Another.....	427
Location-Based Gaming	427
Geocaching and Reverse Geocaching	428
Mixed Reality	428
Street Games	428
What Time Is It?	429
Two-Dimensional Mathematical Treatment	429
Location, Location, Location	433
Distance	433
Great-Circle Heading	435
Rhumb Line	436
23. Pressure Sensors and Load Cells.....	439
Under Pressure	440
Example Effects of High Pressure	440
Button Mashing	442
Load Cells	444
Barometers	448
24. 3D Display.....	451
Binocular Vision	451
Stereoscopic Basics	454
The Left and Right Frustums	454
Types of Display	458
Complementary-Color Anaglyphs	458
Linear and Circular Polarization	459
Liquid-Crystal Plasma	462
Autostereoscopy	463
Advanced Technologies	465

Programming Considerations	467
Active Stereoization	467
Passive Stereoization	469
25. Optical Tracking.....	471
Sensors and SDKs	472
Kinect	472
OpenCV	473
Numerical Differentiation	474
26. Sound.....	477
What Is Sound?	477
Characteristics of and Behavior of Sound Waves	481
Harmonic Wave	481
Superposition	483
Speed of Sound	484
Attenuation	485
Reflection	486
Doppler Effect	488
3D Sound	489
How We Hear in 3D	489
A Simple Example	491
A. Vector Operations.....	495
B. Matrix Operations.....	507
C. Quaternion Operations.....	517
Bibliography.....	529
Index.....	535

Preface

Who Is This Book For?

Simply put, this book is targeted at computer game developers who do not have a strong mechanics or physics background, charged with the task of incorporating *real physics* in their games.

As a game developer, and very likely as a gamer yourself, you've seen products being advertised as "ultra-realistic," or as using "real-world physics." At the same time you, or perhaps your company's marketing department, are wondering how you can spice up your own games with such realism. Or perhaps you want to try something completely new that requires you to explore real physics. The only problem is that you threw your college physics text in the lake after final exams and haven't touched the subject since. Maybe you licensed a really cool physics engine, but you have no idea how the underlying principles work and how they will affect what you're trying to model. Or, perhaps you are charged with the task of tuning someone else's physics code but you really don't understand how it works. Well then, this book is for you.

Sure you could scour the Internet, trade journals, and magazines for information and how-to's on adding physics-based realism to your games. You could even fish out that old physics text and start from scratch. However, you're likely to find that either the material is too general to be applied directly, or too advanced requiring you to search for other sources to get up to speed on the basics. This book will pull together the information you need and will serve as the starting point for you, the game developer, in your effort to enrich your game's content with physics-based realism.

This book is not a recipe book that simply gives sample code for a miscellaneous set of problems. The Internet is full of such example programs (some very good ones we might add). Rather than give you a collection of specific solutions to specific problems, our aim is to arm you with a thorough and fundamental understanding of the relevant topics such that you can formulate your own solutions to a variety of problems. We'll do this by explaining, in detail, the principles of physics applicable to game development, and

by providing complimentary hand calculation examples in addition to sample programs.

What We Assume You Know

Although we don't assume that you are a physics expert, we do assume that you have at least a basic college level understanding of classical physics typical of non-physics and non-engineering majors. It is not essential that your physics background is fresh in your mind as the first several chapters of this book review the subjects relevant to game physics.

We also assume that you are proficient in trigonometry, vector, and matrix math, although we do include reference material in the appendices. Further, we assume that you have at least a basic college level understanding of calculus, including integration and differentiation of explicit functions. Numerical integration and differentiation is a different story, and we cover these techniques in detail in the later chapters of this book.

Mechanics

Most people that we've talked to when we was developing the concept for this book immediately thought of flight simulators when the phrases "real physics" and "real-time simulation" came up. Certainly cutting edge flight simulations are relevant in this context; however, many different types of games, and specific game elements, stand to benefit from physics-based realism.

Consider this example: You're working on the next blockbuster hunting game complete with first-person 3D, beautiful textures, and an awesome sound track to set the mood, but something is missing. That something is realism. Specifically, you want the game to "feel" more real by challenging the gamer's marksmanship, and you want to do this by adding considerations such as distance to target, wind speed and direction, and muzzle velocity, among others. Moreover, you don't want to fake these elements, but rather, you'd like to realistically model them based on the principles of physics. Gary Powell, with MathEngine Plc, put it like this "The illusion and immersive experience of the virtual world, so carefully built up with high polygon models, detailed textures and advanced lighting, is so often shattered as soon as objects start to move and interact."¹ "It's all about interactivity and immersiveness," says Dr. Steven Collins, CEO of Havok.com.² We think both these guys or right on target. Why invest so much time and

1. At the time of this book's first edition, Gary Powell worked for MathEngine Plc. Their products included Dynamics Toolkit 2 and Collision Toolkit 1, which handled single and multiple body dynamics. Currently the company operates under the name CM Labs.
2. At the time of this book's first edition, Dr. Collins was the CEO of Havok.com. Their technology handled rigid body, soft body, cloth, and fluid and particle dynamics. Intel purchased Havok in 2005.

effort making your game world look as realistic as possible, but not take the extra step to make it behave just as realistically?

Here are a few examples of specific game elements that stand to benefit, in terms of realism, from the use of real physics:

- The trajectory of rockets and missiles including the effects of fuel burn off
- The collision of objects such as billiard balls
- The effects of gravitation between large objects such as planets and battle stations
- The stability of cars racing around tight curves
- The dynamics of boats and other waterborne vehicles
- The flight path of a baseball after being struck by a bat
- The flight of a playing card being tossed into a hat

This is by no means an exhaustive list, but just a few examples to get you in the right frame of mind, so to speak. Pretty much anything in your games that bounces around, flies, rolls, slides, or isn't sitting dead still can be realistically modeled to create compelling, believable content for your games.

So how can this realism be achieved? By using physics, of course, which brings us back to the title of this section, the subject of *mechanics*. Physics is a vast field of science that covers many different, but related subjects. The subject most applicable to realistic game content is the subject of mechanics, which is really what's meant by "real physics."

By definition, mechanics is the study of bodies at rest and in motion, and of the effect of forces on them. The subject of mechanics is subdivided into *statics*, which specifically focuses on bodies at rest, and *dynamics*, which focuses on bodies in motion. One of the oldest and most studied subjects of physics, the formal origins of mechanics can be traced back more than 2000 years to Aristotle. An even earlier treatment of the subject was formalized in *Problems of Mechanics*, but the origins of this work are unknown. Although some of these early works attributed some physical phenomena to magical elements, the contributions of such great minds as Galileo, Kepler, Euler, Lagrange, d'Alembert, Newton, and Einstein, to name a few, have helped develop our understanding of this subject to such a degree that we have been able to achieve the remarkable state of technological advancement that we see today.

Because you want your game content to be alive and active, we'll primarily look at bodies in motion and will thus delve into the details of the subject of dynamics. Within the subject of dynamics there are even more specific subjects to investigate, namely, *kine-matics*, which focuses on the motion of bodies without regard to the forces that act on the body, and *kinetics*, which considers both the motion of bodies and the forces that act on or otherwise affect bodies in motion. We'll take a very close look at these two subjects throughout this book.

Digital Physics

This book's first edition focused exclusively on mechanics. More than a decade after its release we've broadened our definition of game physics to include *digital physics* not in the cosmological sense but in the context of the physics associated with such devices as smart phones and their unique user interaction experience. As more platforms such as the Wii, PlayStation, X Box and smart phones come out and are expanded developers will have to keep up with and understand the new input and sensors technologies that accompany these platforms in order to keep producing fresh gaming experiences. But you shouldn't look at this as a burden, and instead look at it as an opportunity to enhance the user's interactive experience with your games.

Arrangement of This Book

Physics-based realism is not new to gaming, and in fact many games on the shelves these days advertise their physics engines. Also, many 3D modeling and animation tools have physics engines built in to help realistically animate specific types of motion. Naturally, there are magazine articles that appear every now and then that discuss various aspects of physics-based game content. In parallel, but at a different level, research in the area of real-time rigid body³ simulation has been active for many years, and the technical journals are full of papers that deal with various aspects of this subject. You'll find papers on subjects ranging from the simulation of multiple, connected rigid bodies to the simulation of cloth. However, while these are fascinating subjects and valuable resources, as we hinted earlier, many of them are of limited immediate use to the game developer as they first require a solid understanding of the subject of mechanics requiring you to learn the basics from other sources. Further, many of them focus primarily on the mathematics involved in solving the equations of motion and don't address the practical treatment of the forces acting on the body or system being simulated.

We asked John Nagle, with Animats, what is, in his opinion, the most difficult part of developing a physics-based simulation for games and his response was developing numerically stable, robust code.⁴ Gary Powell echoed this when he told me that minimizing the amount of parameter tuning to produce stable, realistic behavior was one of the most difficult challenges. We agree; speed and robustness in dealing with the mathematics of bodies in motion are crucial elements of a simulator. And on top of that, so are completeness and accuracy in representing the interacting forces that initiate and

3. A rigid body is formally defined as a body, composed of a system of particles, whose particles remain at fixed distances from each other with no relative translation or rotation among particles. Although the subject of mechanics deals with flexible bodies and even fluids such as water, we'll focus our attention on bodies that are rigid.
4. At the time of this book's first edition, John Nagle was the developer of Falling Bodies, a dynamics plug-in for Softimage|3D.

perpetuate the simulation in the first place. As you'll see later in this book, forces govern the behavior of objects in your simulation and you need to model them accurately if your objects are to behave realistically.

This prerequisite understanding of mechanics and the real world nature of forces that may act on a particular body or system have governed the organization of this book. Generally, this book is organized in four parts with each building on the material covered in previous parts:

Part I, Fundamentals

A mechanics refresher, comprising Chapters 1 through 6.

Chapter 1, Basic Concepts

This warm up chapter covers the most basic of principles that are used and referred to throughout this book. The specific topics addressed include mass and center of mass, Newton's Laws, inertia, units and measures, and vectors.

Chapter 2, Kinematics

This chapter covers such topics as linear and angular velocity, acceleration, momentum, and the general motion of particles and rigid bodies in two and three dimensions.

Chapter 3, Force

The principles of force and torque are covered in this chapter, which serves as a bridge from the subject of kinematics to that of kinetics. General categories of forces are discussed including drag forces, force fields, and pressure.

Chapter 4, Kinetics

This chapter combines elements of Chapters 2 and 3 to address the subject of kinetics and explains the difference between kinematics and kinetics. Further discussion treats the kinetics of particles and rigid bodies in two and three dimensions.

Chapter 5, Collisions

In this chapter we'll cover particle and rigid body collision response, that is, what happens after two objects run in to each other.

Chapter 6, Projectiles

This chapter will focus on the physics of simple projectiles laying the ground work for further specific modeling treatment in later chapters.

Part II, Rigid-Body Dynamics

An introduction to real time simulations, comprising Chapters 7 through 14.

Chapter 7, Real-Time Simulations

This chapter will introduce real-time simulations and detail the core of such simulations—the numerical integrator. Various methods will be presented and coverage will include stability and tuning.

Chapter 8, Particles

Before diving into rigid body simulations, this chapter will show how to implement a particle simulation, which will be extended in the next chapter to include rigid bodies.

Chapter 9, 2D Rigid-Body Simulator

This chapter will extend the particle simulator from the previous chapter showing how to implement rigid bodies, which primarily consists of adding rotation and dealing with the inertia tensor.

Chapter 10, Implementing Collision Response

Collision detection and response will be combined to implement real-time collision capabilities in the 2D simulator.

Chapter 11, Rotation in 3D Rigid-Body Simulators

This chapter will address how to handle rigid body rotation in 3D including how to deal with the inertia tensor. Then we'll show the reader how to extend the 2D simulator to 3D.

Chapter 12, 3D Rigid-Body Simulator

Multiple unconnected bodies will be incorporated in the simulator in this chapter. Introduction of multiple bodies requires resolution of multiple rigid body collisions, which can be very tricky. Issues of stability and realism will be covered.

Chapter 13, Connecting Objects

Taking things a step further, this chapter will show how to join rigid bodies forming connected bodies, which may be used to simulate human bodies, complex vehicles that may blow apart, among many other game objects. Various connector types will be considered.

Chapter 14, Physics Engines

In this chapter, specific aspects of automobile performance are addressed, including aerodynamic drag, rolling resistance, skidding distance, and roadway banking.

Part III, Physical Modeling

A look at some real world problems, comprising Chapters 15 through 19.

Chapter 15, Aircraft

This chapter focuses on the elements of flight including propulsor forces, drag, geometry, mass, and most importantly lift.

Chapter 16, Ships and Boats

The fundamental elements of floating vehicles are discussed in this chapter, including floatation, stability, volume, drag, and speed.

Chapter 17, Cars and Hovercraft

In this chapter, specific aspects of automobile performance are addressed, including aerodynamic drag, rolling resistance, skidding distance, and roadway banking. Additionally hovercraft shares some of the same characteristics of both cars and boats.

This chapter will consider those characteristics that distinguish the hovercraft as a unique vehicle. Topics covered include hovering flight, aerostatic lift, and directional control

Chapter 18, Guns and Explosions

This chapter will focus on the physics of guns including power, recoil, and projectile flight. Since we generally want things to explode when hit with a large projectile, this chapter will also address the physics of and modeling explosions.

Chapter 19, Sports

This chapter will focus on the physics of ball sports such as baseball, golf, and tennis. Coverage will go beyond projectile physics and include such topics as including pitching, bat swing, bat-ball impact, golf club swing and club ball impact, plus tennis racket swinging and racket/ball impacts.

Part IV, Digital Physics

Chapters in this part of the book will explain the physics behind accelerometers, touch screens, GPS and other gizmos showing the reader how to leverage these elements in their games, comprising Chapters 20 through 26.

Chapter 20, Touch Screens

Touch screens facilitate virtual tactile interfaces with mobile device games, such as those made for the iPhone. This chapter will explain the physics of touch screen and how the reader can leverage this interface in their games particularly with respect to virtual physical interaction with game elements through gesturing.

Chapter 21, Accelerometers

Accelerometers are now widely used in mobile devices and game controllers allowing virtual physical interaction between players and game objects. This chapter will explain how accelerometers work, what data they provide and how that data can be manipulated with respect to virtual physical interaction with game elements. Topics covered will include, but not be limited to integration of acceleration data to derive velocities and displacements and rotations.

Chapter 22, Gaming from One Place to Another

Mobile devices commonly have GPS capabilities and this chapter will explain the physics of the GPS system including relativistic effects. Further, GPS data will be explained and this chapter showing the reader how to manipulate that data for virtual interaction with game elements. For example, we'll show the reader how to differentiate GPS data to derive speed and acceleration among other manipulations.

Chapter 23, Pressure Sensors and Load Cells

Pressure sensing devices are used in games as a means of allowing players to interact with game elements, for example, the Wii balance board uses pressure sensors allowing players to interact with the Wii Fit game. This chapter will explain the physics behind such pressure sensors, what data they generate, and how to manipulate that data for game interaction.

Chapter 24, 3D Display

The new PlayStation Move and Microsoft's Kinect use optical tracking systems to detect the position of players' game controllers or gestures. This chapter will explain the physics behind optical tracking and how to leverage this technology in games.

Chapter 25, Optical Tracking

As televisions and handheld game consoles race to implement 3D displays, several different technologies are being developed. By understanding the physics of the glasses dependent stereoscopic displays, the new "glasses free" autostereoscopic displays, and looking forward to holography and volumetric displays, developers will be better positioned to leverage these effects in their games.

Chapter 26, Sound

Sound is a particularly important part of a game's immersive experience; however, to date no book on game physics addresses the physics of sound. This chapter will focus on sound physics including such topics of sound speed and the Doppler Effect. Discussions will also include why sound physics is often ignored in games, for example, when simulating explosions in outer space.

Appendix A, Vector Operations

This appendix shows you how to implement a C++ class that captures all of the vector operations that you'll need to when writing 2D or 3D simulations.

Appendix B, Matrix Operations

This appendix implements a class that captures all of the operations you need to handle 3x3 matrices.

Appendix C, Quaternion Operations

This appendix implements a class that captures all of the operations you need to handle quaternions when writing 3D rigid body simulations.

Part I, Fundamentals focuses on fundamental topics in Newtonian mechanics such as kinematics and kinetics. Kinematics deals with the motion of objects. We'll cover both linear and angular velocity and acceleration. Kinetics deals with forces and resulting motion. Part I serves as a primer for **Part II, Rigid-Body Dynamics** that covers rigid body dynamics. Readers already versed in classical mechanics can skip **Part I, Fundamentals** without loss of continuity.

Part II, Rigid-Body Dynamics focuses on rigid body dynamics and development of both single and multi-body simulations. This part covers numerical integration, real-time simulation of particles and rigid bodies, and connected rigid bodies. Generally, this part covers what most game programmers consider elements of a physics engine.

Part III, Physical Modeling focuses on physical modeling. The aim of this part is to provide valuable physical insight for the reader so they can make better judgments on what to include in their models and what they can safely leave out without sacrificing physical realism. We cannot and do not attempt to cover all the possible things you

might want to simulate. Instead we cover several typical things you may try to simulate in a game such as aircraft, boats, sports balls, among others with the purpose of giving you some insight into the physical nature of those things and some of the choices you must make when developing suitable models.

Part IV, Digital Physics covers digital physics in a broad sense. This is an exciting topic as it relates to the technologies associated with mobile platforms, such as smart phones like the iPhone, and ground breaking game systems such as the Nentendo Wii. Chapters in this part of the book will explain the physics behind accelerometers, touch screens, GPS and other gizmos showing the reader how to leverage these elements in their games. We recognize that these topics are not what most game programmers typically think about when they think of game physics; however, the technologies covered play an increasingly important role in modern mobile games and we feel it important to explain the underlying physics behind them with the hope that you'll be better able to leverage these technologies in your games.

In addition to resources pertaining to real-time simulations, the **Bibliography** at the end of this book will provide sources of information on mechanics, mathematics, and other specific technical subjects, such as books on aerodynamics.

Conventions Used in This Book

The following typographical conventions are used in this book:

Constant width

Used to indicate command-line computer output, code examples, Registry keys, and keyboard accelerators (see “Keyboard Accelerators” later in this book).

Constant width italic

Used to indicate variables in code examples.

Italic

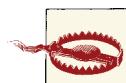
Introduces new terms and to indicate URLs, variables, filenames and directories, commands, and file extensions.

Bold

Indicates vector variables.



This icon signifies a tip, suggestion, or general note.



This icon indicates a warning or caution.

We use boldface type to indicate a vector quantity, such as force, \mathbf{F} . When referring to the magnitude only of a vector quantity, we use standard type. For example, the magnitude of the vector force, \mathbf{F} , is F with components along the coordinate axes, F_x , F_y , and F_z . In the code samples throughout the book, we use the * (asterisk) to indicate vector dot product, or scalar product, operations depending on the context, and we use the ^ (caret) to indicate vector cross product.

Using Code Examples

This book is here to help you get your job done. In general, if this book includes code examples, you may use the code in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing a CD-ROM of examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*Physics for Game Developers, 2nd Edition* by David M. Bourg and Bryan Bywalec (O'Reilly). Copyright 2013 David M. Bourg and Bryan Bywalec, 978-1-449-39251-2."

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at permissions@oreilly.com.

Safari® Books Online



Safari Books Online (www.safaribooksonline.com) is an on-demand digital library that delivers expert **content** in both book and video form from the world's leading authors in technology and business.

Technology professionals, software developers, web designers, and business and creative professionals use Safari Books Online as their primary resource for research, problem solving, learning, and certification training.

Safari Books Online offers a range of **product mixes** and pricing programs for **organizations**, **government agencies**, and **individuals**. Subscribers have access to thousands of books, training videos, and prepublication manuscripts in one fully searchable database from publishers like O'Reilly Media, Prentice Hall Professional, Addison-Wesley Professional, Microsoft Press, Sams, Que, Peachpit Press, Focal Press, Cisco Press, John Wiley & Sons, Syngress, Morgan Kaufmann, IBM Redbooks, Packt, Adobe Press, FT Press, Apress, Manning, New Riders, McGraw-Hill, Jones & Bartlett, Course Technol-

ogy, and dozens **more**. For more information about Safari Books Online, please visit us [online](#).

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)
707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at <http://oreil.ly/Physics-GameDev2>.

To comment or ask technical questions about this book, send email to bookquestions@oreilly.com.

For more information about our books, courses, conferences, and news, see our website at <http://www.oreilly.com>.

Find us on Facebook: <http://facebook.com/oreilly>

Follow us on Twitter: <http://twitter.com/oreillymedia>

Watch us on YouTube: <http://www.youtube.com/oreillymedia>

Acknowledgments

We want to thank Andy Oram, the editor of this edition of the book, for his skillful review of our writing and his insightful comments and suggestions, not to mention his patience. We also want to express my appreciation to O'Reilly for agreeing to take on this project giving us the opportunity to expand on the original edition. Furthermore, special thanks go to all of the production and technical staff at O'Reilly.

We'd also like to thank the technical reviewers, Christian Stober and Paul Zirkle, whose valuable insight added much to this edition.

Individually, David would like to thank his loving wife and best friend, Helena, for her endless support and encouragement, and his wonderful daughter, Natalia, for making every day special.

Bryan would like to thank his co-author David for the opportunity to help with the second edition and would also like to thank his parents, Barry and Sharon, for raising him to be curious about the world. Lastly, he would like to thank his fiancée, Anne Hasuly, for her support without which many chapters would still be half-finished.

PART I

Fundamentals

Part I focuses on fundamental topics in Newtonian mechanics such as *kinematics* and *kinetics*. Kinematics deals with the motion of objects; we'll cover both linear and angular velocity and acceleration. Kinetics deals with forces and resulting motion. **Part I** serves as a primer for **Part II**, which covers rigid-body dynamics. Readers already versed in classical mechanics can skip **Part I** without loss of continuity.

CHAPTER 1

Basic Concepts

As a warm-up, this chapter will cover the most basic of the principles that will be used and referenced throughout the remainder of this book. First, we'll introduce Newton's laws of motion, which are very important in the study of mechanics. Then we'll discuss units and measures, where we'll explain the importance of keeping track of units in your calculations. You'll also have a look at the units associated with various physical quantities that you'll be studying. After discussing units, we'll define our general coordinate system, which will serve as our standard frame of reference. Then we'll explain the concepts of mass, center of mass, and moment of inertia, and show you how to calculate these quantities for a collection, or combination, of masses. Finally, we'll discuss Newton's second law of motion in greater detail, take a quick look at vectors, and briefly discuss relativistic time.

Newton's Laws of Motion

In the late 1600s (around 1687), Sir Isaac Newton put forth his philosophies on mechanics in his *Philosophiae Naturalis Principia Mathematica*. In this work Newton stated the now-famous laws of motion, which are summarized here:

Law I

A body tends to remain at rest or continue to move in a straight line at constant velocity unless acted upon by an external force. This is the so-called concept of inertia.

Law II

The acceleration of a body is proportional to the resultant force acting on the body, and this acceleration is in the same direction as the resultant force.

Law III

For every force acting on a body (action) there is an equal and opposite reacting force (reaction), where the reaction is collinear to the acting force.

These laws form the basis for much of the analysis in the field of mechanics. Of particular interest to us in the study of dynamics is the second law, which is written:

which is the famous expression of Newton's second law of motion. We will take a closer look at this equation later.

By no means did we just derive this famous formula. What we did was check its dimensional consistency (albeit in reverse), and all that means is that any formulas you develop to represent a force acting on a body had better come out to a consistent set of units in the form $(M)(L/T^2)$. This may seem trivial at the moment; however, when you start looking at more complicated formulas for the forces acting on a body, you'll want to be able to break down these formulas into their component dimensions so you can check their dimensional consistency. Later we will use actual units, from the SI (*le Système international d'unités*, or International System of Units) for our physical quantities. Of course, there are other unit systems, but unless you want to show these values to your gamers, it really does not matter which system you use in your games. Again, what is important is consistency.

To help clarify this point, consider the formula for the friction drag on a body moving through a fluid, such as water:

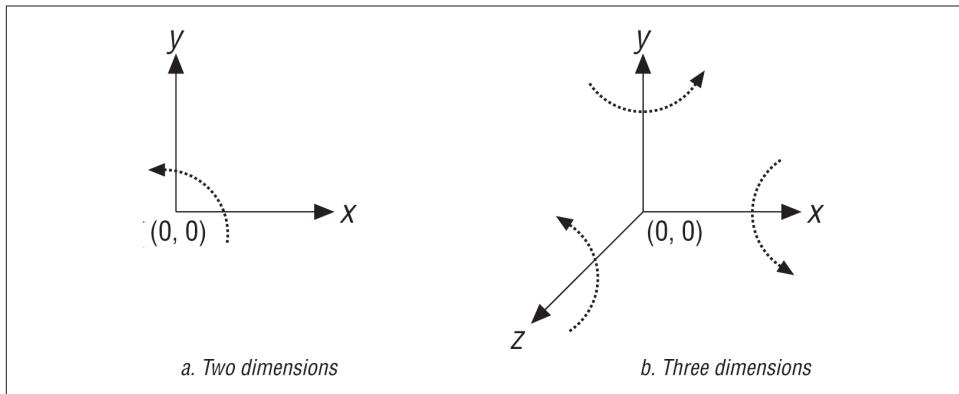


Figure 1-1. Right-handed coordinate system

In three dimensions we will use the coordinate system shown in Figure 1-1(b), where rotations about the x-axis are positive from positive y to positive z , rotations about the y -axis are positive from positive z to positive x , and rotations about the z -axis are positive from positive x to positive y .

Vectors

Let us take you back for a moment to your high school math class and review the concept of *vectors*. Essentially, a vector is a quantity that has both magnitude as well as direction. Recall that a *scalar*, unlike a vector, has only magnitude and no direction. In mechanics, quantities such as force, velocity, acceleration, and momentum are vectors, and you must consider both their magnitude and direction. Quantities such as distance, density, viscosity, and the like are scalars.

With regard to notation, we'll use boldface type to indicate a vector quantity, such as force, \mathbf{F} . When referring to the magnitude only of a vector quantity, we'll use standard type. For example, the magnitude of the vector force, \mathbf{F} , is F with components along the coordinate axes, F_x , F_y , and F_z . In the code samples throughout the book, we'll use the * (asterisk) to indicate vector dot product, or scalar product, operations depending on the context, and we'll use the ^ (caret) to indicate vector cross product.

Because we will be using vectors throughout this book, it is important that you refresh your memory on the basic vector operations, such as vector addition, dot product, and cross product, among others. For your convenience (so you don't have to drag out that old math book), we've included a summary of the basic vector operations in [Appendix A](#). This appendix provides code for a `Vector` class that contains all the important vector math functionality. Further, we explain how to use specific vector operations—such as the dot-product and cross-product operations—to perform some common and

useful, calculations. For example, in dynamics you'll often have to find a vector perpendicular, or *normal*, to a plane or contacting surface; you use the cross-product operation for this task. Another common calculation involves finding the shortest distance from a point to a plane in space; you use the dot-product operation here. Both of these tasks are described in [Appendix A](#), which we encourage you to review before delving too deeply into the example code presented throughout the remainder of this book.

Derivatives and Integrals

If you're not familiar with calculus, or The Calculus, don't let the use of derivatives and integrals in this text worry you. While we'll write equations using derivatives and integrals, we'll show you explicitly how to deal with them computationally throughout this book. Without going into a dissertation on all the properties and applications of derivatives and integrals, let's touch on their physical significance as they relate to the material we'll cover.

You can think of a derivative as the rate of change in one variable with respect to another variable, or in other words, derivatives tells you how fast one variable changes as some other variable changes. Take speed, for example. A car travels at a certain speed covering some distance in a certain period of time. Its speed, on average, is the distance traveled over a specific time interval. If it travels a distance of 60 kilometers in one hour, then its average speed is 60 kilometers an hour. When we're doing simulations, the ones you'll see later in this book, we're interested in what the car is doing over very short time intervals. As the time interval gets really small and we consider the distance traveled over that very short period of time, we're looking at *instantaneous* speed. We usually write such relations using symbols like the following:

moving on to the second slice, estimating its volume and adding that to the volume of the first slice; and then moving on to the third, and fourth, and so on, aggregating the volume of the loaf as you move toward the other end. Integration applies this technique to infinitely thin slices of volume to compute the volume of any arbitrary shape. The same techniques apply to other computations—for example, computing areas, *inertias*, masses, and so on, and even aggregating distance traveled over successive small slices of time, as you'll see later. In fact, this latter application is the inverse of the derivative of distance with respect to time, which gives speed. Using integration and differentiation in this way allows you to work back and forth when computing speed, acceleration, and distance traveled, as you'll see shortly. In fact, we'll use these concepts heavily throughout the rest of this book.

Mass, Center of Mass, and Moment of Inertia

The properties of a body—*mass*, *center of mass*, and *moment of inertia*, collectively called *mass properties*—are absolutely crucial to the study of mechanics, as the linear and angular¹ motion of a body and a body's response to a given force are functions of these mass properties. Thus, in order to accurately model a body in motion, you need to know or be capable of calculating these mass properties. Let's look at a few definitions first.

In general, people think of mass as a measure of the amount of matter in a body. For our purposes in the study of mechanics, we can also think of mass as a measure of a body's resistance to motion or a change in its motion. Thus, the greater a body's mass, the harder it will be to set it in motion or change its motion.

In laymen's terms, the center of mass (also known as *center of gravity*) is the point in a body around which the mass of the body is evenly distributed. In mechanics, the center of mass is the point through which any force can act on the body without resulting in a rotation of the body.

Although most people are familiar with the terms *mass* and *center of gravity*, the term *moment of inertia* is not so familiar; however, in mechanics it is equally important. The mass moment of inertia of a body is a quantitative measure of the radial distribution of the mass of a body about a given axis of rotation. Analogous to mass being a measure of a body's resistance to linear motion, mass moment of inertia (also known as *rotational inertia*) is a measure of a body's resistance to rotational motion.

Now that you know what these properties mean, let's look at how to calculate each.

For a given body made up of a number of particles, the total mass of the body is simply the sum of the masses of all elemental particles making up the body, where the mass of

1. *Linear motion* refers to motion in space without regard to rotation; *angular motion* refers specifically to the rotation of a body about any axis (the body may or may not be undergoing linear motion at the same time).

each elemental particle is its mass density times its volume. Assuming that the body is of uniform density, then the total mass of the body is simply the density of the body times the total volume of the body. This is expressed in the following equation:

nators, thus dropping out of the equations. Recall that the weight of an object is its mass times the acceleration due to gravity, g , which is 9.8 m/s^2 at sea level.

The formulas for calculating the total mass and center of gravity for a system of discrete point masses can conveniently be written in vector notation as follows:

```

}
CombinedCG = FirstMoment / TotalMass;

```

Now that the combined center of gravity location has been found, you can calculate the relative position of each point mass as follows:

```

for(i=0; i<_NUMELEMENTS; i++)
{
    Element[i].correctedPosition = Element[i].designPosition -
        CombinedCG;
}

```

To calculate mass moment of inertia, you need to take the second moment of each elemental mass making up the body about each coordinate axis. The second moment is then the product of the mass times distance squared. That distance is not the distance to the elemental mass centroid along the coordinate axis as in the calculation for center of mass, but rather the perpendicular distance from the coordinate axis, about which we want to calculate the moment of inertia, to the elemental mass centroid.

Referring to [Figure 1-2](#) for an arbitrary body in three dimensions, when calculating moment of inertia about the x-axis, I_{xx} , this distance, r , will be in the yz-plane such that $r_x^2 = y^2 + z^2$. Similarly, for the moment of inertia about the y-axis, I_{yy} , $r_y^2 = z^2 + x^2$, and for the moment of inertia about the z-axis, I_{zz} , $r_z^2 = x^2 + y^2$.

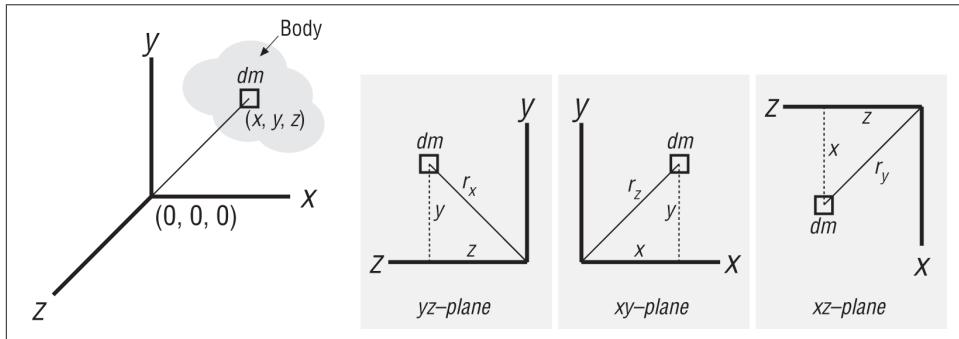


Figure 1-2. Arbitrary body in 3D

The equations for mass moment of inertia about the coordinate axes in 3D are:

the center of mass of the body, but you want to know the moment of inertia, I , about an axis some distance from but parallel to this neutral axis. In this case, you can use the transfer of axes, or *parallel axis theorem*, to determine the moment of inertia about this new axis. The formula to use is:

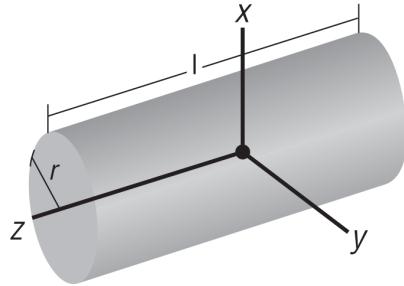


Figure 1-3. Circular cylinder: $I_{xx} = I_{yy} = (1/4) mr^2 + (1/12) ml^2$; $I_{zz} = (1/2) mr^2$

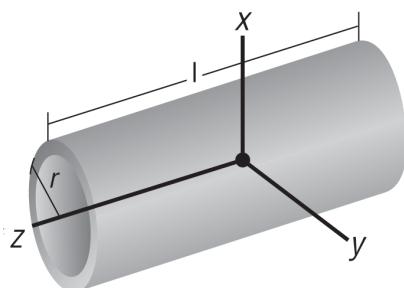


Figure 1-4. Circular cylindrical shell: $I_{xx} = I_{yy} = (1/2) mr^2 + (1/12) ml^2$; $I_{zz} = mr^2$

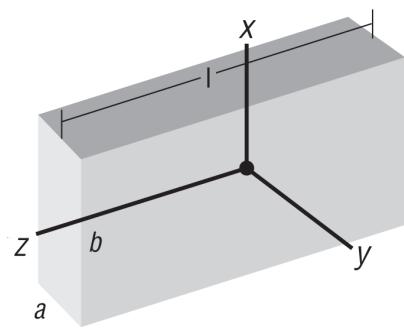


Figure 1-5. Rectangular cylinder: $I_{xx} = (1/12) m(a^2 + l^2)$; $I_{yy} = (1/12) m(b^2 + l^2)$; $I_{zz} = (1/12) m(a^2 + b^2)$

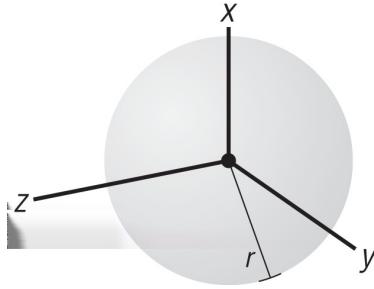


Figure 1-6. Sphere: $I_{xx} = I_{yy} = I_{zz} = (2/5) mr^2$

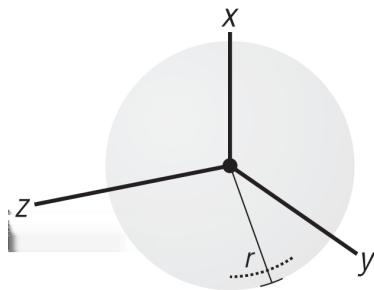


Figure 1-7. Spherical shell: $I_{xx} = I_{yy} = I_{zz} = (2/3) mr^2$

As you can see, these formulas are relatively simple to implement. The trick here is to break up a complex body into a number of smaller, simpler representative geometries whose combination will approximate the complex body's inertia properties. This exercise is largely a matter of judgment considering the desired level of accuracy.

Let's look at a simple 2D example demonstrating how to apply the formulas discussed in this section. Suppose you're working on a top-down-view auto racing game where you want to simulate the automobile sprite based on 2D rigid-body dynamics. At the start of the game, the player's car is at the starting line, full of fuel and ready to go. Before starting the simulation, you need to calculate the mass properties of the car, driver, and fuel load at this initial state. In this case, the *body* is made up of three components: the car, driver, and full load of fuel. Later during the game, however, the mass of this body will change as fuel burns off and the driver gets thrown after a crash! For now, let's focus on the initial condition, as illustrated in Figure 1-8.

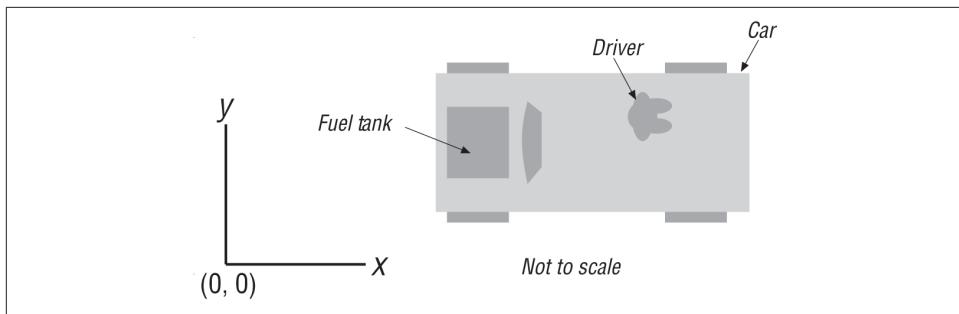


Figure 1-8. Example body consisting of car, driver, and fuel

The properties of each component in this example are given in [Table 1-2](#). Note that length is measured along the x-axis, width along the y-axis, and height would be coming out of the screen. Also note that the coordinates—in the form (x, y) —to the centroid of each component are referenced to the global origin.

Table 1-2. Example properties

Car	Driver (seated)	Fuel
Length = 4.70 m	Length = 0.90 m	Length = 0.50 m
Width = 1.80 m	Width = 0.50 m	Width = 0.90 m
Height = 1.25 m	Height = 1.10 m	Height = 0.30 m
Weight = 17,500 N	Weight = 850 N	Density of fuel = 750 kg/m ³
Centroid = (30.5, 30.5) m	Centroid = (31.50, 31.00) m	Centroid = (28.00, 30.50) m

The first mass property we want to calculate is the mass of the body. This is a simple calculation since we are already given the weight of the car and the driver. The only other component of weight we need is that of the fuel. Since we are given the mass density of the fuel and the geometry of the tank, we can calculate the volume of the tank and multiply by the density and the acceleration due to gravity to get the weight of the fuel in the tank. This yields 920.6 N of fuel, as shown here:

Now, the total weight of the body is:

Notice how the calculations for the I_{cg} of the driver and the fuel are dominated by their md^2 terms. In this example, the local inertia of the driver and fuel is only 2.7% and 2.1%, respectively, of their corresponding md^2 terms.

Finally, we can obtain the total moment of inertia of the body about its own neutral axis by summing the I_{cg} contributions of each component as follows:



Newton's Second Law of Motion

As we stated in the first section of this chapter, Newton's second law of motion is of particular interest in the study of mechanics. Recall that the equation form of Newton's second law is:

where i represents the i th particle making up the body, ω is the angular velocity of the body about the axis under consideration, and $(\mathbf{r}_i \times m_i(\boldsymbol{\omega} \times \mathbf{r}_i))$ is the angular momentum of the i th particle, which has a magnitude of $m_i\omega r_i^2$. For rotation about a given axis, this equation can be rewritten in the form:

-
2. In this case, I will be a second-rank tensor, which is essentially a 3×3 matrix. A vector is actually a tensor of rank one, and a scalar is actually a tensor of rank zero.
-

and a piece of woven or knitted cloth. Take the sheet of paper and, holding it flat, pull on it softly from opposing ends. Try this length-wise, width-wise, and along a diagonal. You should observe that the paper seems just as strong, or stretches about the same, in all directions. It is isotropic; therefore, only a single scalar constant is required to represent its strength for all directions.

Now, try to find a piece of cloth with a simple, relatively loose weave where the threads in one direction are perpendicular to the threads in the other direction. Most neckties will do. Try the same pull test that you conducted with the sheet of paper, pulling the cloth along each thread direction and then at a diagonal to the threads. You should observe that the cloth stretches more when you pull it along a diagonal to the threads as opposed to pulling it along the direction of the run of the threads. The cloth is anisotropic in that it exhibits different elastic (or strength) properties depending on the direction of pull; thus, a collection of vector quantities (a tensor) is required to represent its strength for all directions.

In the context of this book, the property under consideration is a body's moment of inertia, which in 3D requires nine components to fully describe it for any arbitrary rotation. Moment of inertia is not a strength property as in the paper and cloth example, but it is a property of the body that varies with the axis of rotation. Since nine components are required, moment of inertia will be generalized in the form of a 3×3 matrix (i.e., a *second-rank tensor*) later in this book.

We need to mention a few things at this point regarding coordinates, which will become important when you're writing your real-time simulator. Both of the equations of motion have, so far, been written in terms of global coordinates and not body-fixed coordinates. That's OK for the linear equation of motion, where you can track the body's location and velocity in the global coordinate system. However, from a computational point of view, you don't want to do that for the angular equation of motion for bodies that rotate in three dimensions.³ The reason is because the moment of inertia term, when calculated with respect to global coordinates, actually changes depending on the body's position and orientation. This means that during your simulation you'll have to recalculate the inertia matrix (and its inverse) a lot, which is computationally inefficient. It's better to rewrite the equations of motion in terms of local (attached to the body) coordinates so you have to calculate the inertia matrix (and its inverse) only once at the start of your simulation.

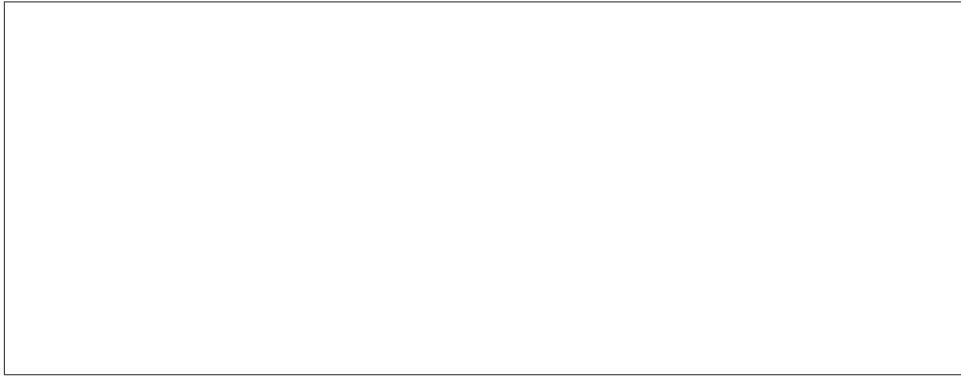
In general, the time derivative of a vector, \mathbf{V} , in a fixed (nonrotating) coordinate system is related to its time derivative in a rotating coordinate system by the following:

3. In two dimensions, it's OK to leave the angular equation of motion as it's shown here since the moment of inertia term is simply a constant scalar quantity.

The $(\omega \times V)$ term represents the difference between V 's time derivative as measured in the fixed coordinate system and V 's time derivative as measured in the rotating coordinate system. We can use this relation to rewrite the angular equation of motion in terms of local, or body-fixed, coordinates. Further, the vector to consider is the angular momentum vector H_{cg} . Recall that $H_{cg} = I\omega$ and its time derivative are equal to the sum of moments about the body's center of gravity. These are the pieces you need for the angular equation of motion, and you can get to that equation by substituting H_{cg} in place of V in the derivative transform relation as follows:

Expanding the triple vector product term yields:

You already know that **I** represents the moment of inertia, and the terms that should look familiar to you already are the moment of inertia terms about the three coordinate axes, I_{xx} , I_{yy} , and I_{zz} . The other terms are called *products of inertia* (see [Figure 1-9](#)):



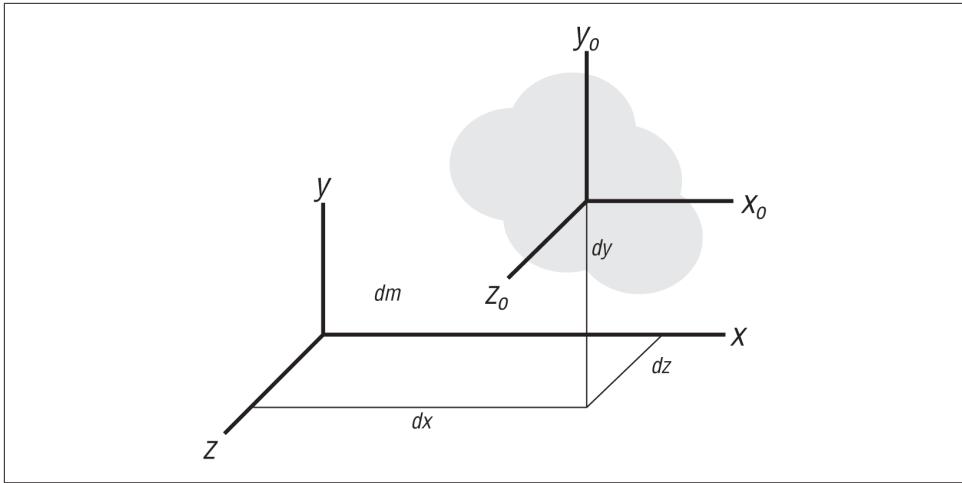


Figure 1-10. Transfer of axes

For the simple geometries shown earlier, each coordinate axis represented a plane of symmetry, and products of inertia go to zero about axes that represent planes of symmetry. You can see this by examining the product of inertia formulas, where, for example, all of the (xy) terms in the integral will be cancelled out by each corresponding $-(xy)$ term if the body is symmetric about the y -axis, as illustrated in [Figure 1-11](#).

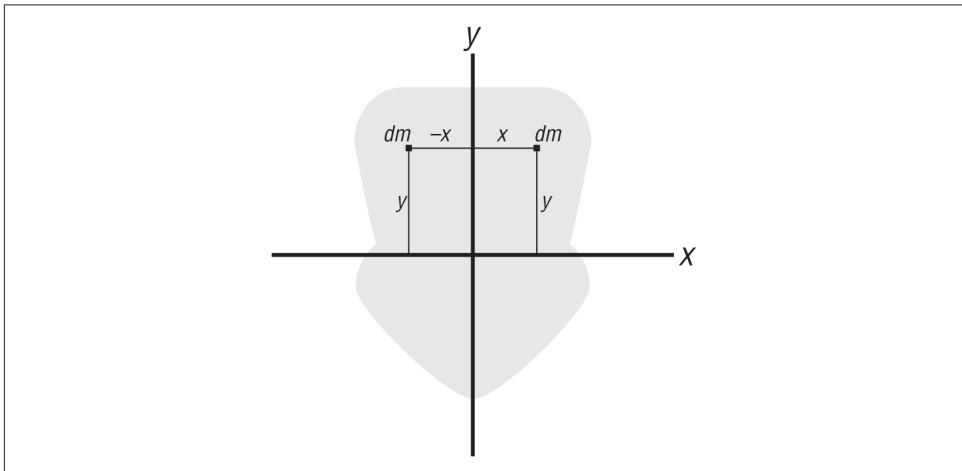


Figure 1-11. Symmetry

For composite bodies, however, there may not be any planes of symmetry, and the orientation of the principal axes will not be obvious. Further, you may not even want to use the principal axes as your local coordinate axes for a given rigid body since it may

be awkward to do so. For example, consider the airplane from the FlightSim discussion in [Chapter 7](#), where you'll have the local coordinate design axes running, relative to the pilot, fore and aft, up and down, and left and right. This orientation is convenient for locating the parts of the wings, tail, elevators, etc. with respect to one another, but these axes don't necessarily represent the principal axes of the airplane. The end result is that you'll use axes that are convenient and deal with the nonzero products of inertia (which, by the way, can be either positive or negative).

We already showed you how to calculate the combined moments of inertia for a composite body made up of a few smaller elements. Accounting for the product of inertia terms follows the same procedure except that, typically, your elements are such that their local product of inertia terms are zero. This is the case only if you represent your elements by simple geometries such as point masses, spheres, rectangles, etc. That being the case, the main contribution to the rigid body's products of inertia will be due to the transfer of axes terms for each element.

Before looking at some sample code, let's first revise the element structure to include a new term to hold the element's local moment of inertia as follows:

```
typedef struct _PointMass
{
    float mass;
    Vector designPosition;
    Vector correctedPosition;
    Vector localInertia;
} PointMass;
```

Here we're using a vector to represent the three local moment of inertia terms and we're also assuming that the local products of inertia are zero for each element.

The following code sample shows how to calculate the inertia tensor given the component elements:

```
float Ixx, Iyy, Izz, Ixy, Ixz, Iyz;
Matrix3x3 InertiaTensor;

Ixx = 0; Iyy = 0; Izz = 0;
Ixy = 0; Ixz = 0; Iyz = 0;

for (i = 0; i<_NUMELEMENTS; i++)
{
    Ixx += Element[i].LocalInertia.x +
        Element[i].mass * (Element[i].correctedPosition.y *
        Element[i].correctedPosition.y +
        Element[i].correctedPosition.z *
        Element[i].correctedPosition.z);

    Iyy += Element[i].LocalInertia.y +
        Element[i].mass * (Element[i].correctedPosition.z *
        Element[i].correctedPosition.z +
```

```

Element[i].correctedPosition.x *
Element[i].correctedPosition.x);

Izz += Element[i].LocalInertia.z +
Element[i].mass * (Element[i].correctedPosition.x *
Element[i].correctedPosition.x +
Element[i].correctedPosition.y *
Element[i].correctedPosition.y);

Ixxy += Element[i].mass * (Element[i].correctedPosition.x *
Element[i].correctedPosition.y);

Ixzx += Element[i].mass * (Element[i].correctedPosition.x *
Element[i].correctedPosition.z);

Ixyz += Element[i].mass * (Element[i].correctedPosition.y *
Element[i].correctedPosition.z);
}

// e11 stands for element on row 1 column 1, e12 for row 1 column 2, etc.
InertiaTensor.e11 = Ixx;
InertiaTensor.e12 = -Ixxy;
InertiaTensor.e13 = -Ixzx;

InertiaTensor.e21 = -Ixxy;
InertiaTensor.e22 = Iyy;
InertiaTensor.e23 = -Ixyz;

InertiaTensor.e31 = -Ixzx;
InertiaTensor.e32 = -Ixyz;
InertiaTensor.e33 = Izz;

```

Note that the inertia tensor is calculated about axes that pass through the combined center of gravity for the rigid body, so be sure to use the corrected coordinates for each element relative to the combined center of gravity when applying the transfer of axes formulas.

We should also mention that this calculation is for the inertia tensor in body-fixed coordinates, or local coordinates. As we discussed earlier in this chapter, it is better to rewrite the angular equation of motion in terms of local coordinates and use the local inertia tensor to save some number crunching in your real-time simulation.

Relativistic Time

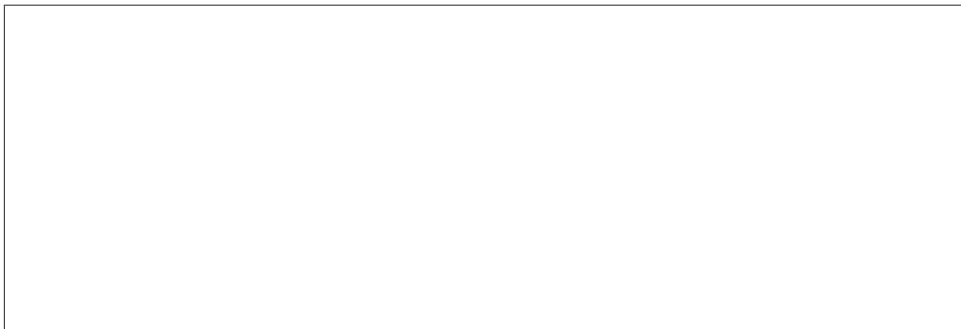
To allow for a thorough understanding of how advanced space vehicles work as well as give you a mechanism by which to alter time in your games, we would like to offer a brief introduction to the theory of relativity, and particularly its effect on time. In our everyday experience, it is safe to assume that the clock on your wall is ticking at the same rate as the clock on our wall as we write this. However, the reason we all know the name

Albert Einstein is that he had the foresight to abandon time as a constant. Instead he postulated that light travels at the same speed regardless of the motion of the source.

That is to say, if you shine a flashlight in a vacuum, the electromagnetic radiation it emits in the form of visible light travels at a set velocity of c (299,792,458 m/s). Now, if you take that same flashlight and put it on the nose of a rocket traveling at half that speed directly at you, you might expect that light is traveling at you with a velocity of $1.5c$. Yet, the rocket-powered flashlight would still be observed as emitting light at a velocity of c . As Einstein's theory of special relativity matured, the postulate has been reformulated to state that there is a maximum speed at which information can be transferred in the space-time continuum, a principal called *locality*. As electromagnetic radiation has no mass,⁴ it travels at this maximum speed in a vacuum.

The most startling consequence of the theory is that time is no longer absolute. The postulate that the speed of light is constant for all frames of reference requires that time slow down, or *dilate*, as velocity increases. It is actually fairly easy to demonstrate this result.

The following example depicts a conceptual clock. A beam of light is bouncing between two mirrors. The time it takes for the beam of light to start from one mirror, bounce off the second, and return to the first constitutes one “tick” of this clock. That tick can be calculated as:



4. Photons, the particle form of electromagnetic radiation, can have relativistic mass but are hypothesized to have no “rest mass.” To avoid getting into quantum electrodynamics, here we’ll just consider them without mass.

Now suppose that you are above the mirrors as they speed past you to the right. Then the clock would look something like [Figure 1-13](#).

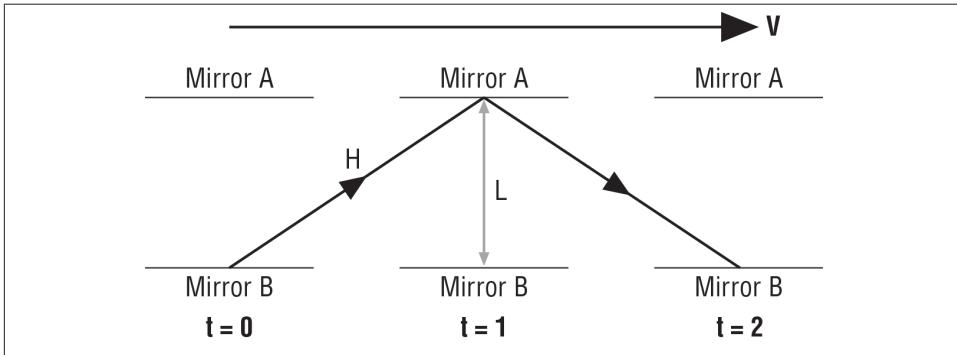


Figure 1-13. Stationary with respect to the clock

One tick of the clock is now defined as twice the distance of the hypotenuse over the speed of light. Clearly H must be larger than L , so we see that the clock with the relative velocity will take longer to tick than if you were moving with the clock.

If this isn't clear, we can also come to the same conclusion a different way. If we define the speed of light as the amount of time it takes for the light beam to travel the distance between the mirrors divided by the time it took to travel that distance, we see that:

Now, besides those implications to games involving space flight or high-velocity travel, time dilation is also important to some surprising digital electronic applications. For instance, the Global Positioning System (GPS), described in detail in [Chapter 22](#), must take relativistic time dilation into account when calculating position. The satellite's high speed slows the clock compared to your watch on Earth; however, being farther up the Earth's gravity causes it to tick faster than a terrestrial clock. The specifics of this combined effect are discussed in [Chapter 22](#).

Another point you might find interesting is that it is now easy to see how the “you can't travel faster than light” rule is a result of the theory of relativity. Should you accelerate such that your velocity, v , is equal to c , the Lorentz transformation attempts to divide by zero. For games where faster-than-light travel is a practical necessity, you will have to imagine a mechanism to prevent this but be able to break the rules with style.

CHAPTER 2

Kinematics

In this chapter we'll explain the fundamental aspects of the subject of kinematics. Specifically, we'll explain the concepts of linear and angular displacement, velocity, and acceleration. We've prepared an example program for this chapter that shows you how to implement the kinematic equations for particle motion. After discussing particle motion, we go on to explain the specific aspects of rigid-body motion. This chapter, along with the next chapter on force, is prerequisite to understanding the subject of kinetics, which you'll study in [Chapter 4](#).

In the preface, we told you that kinematics is the study of the motion of bodies without regard to the forces acting on the body. Therefore, in kinematics, attention is focused on position, velocity, and acceleration of a body, how these properties are related, and how they change over time.

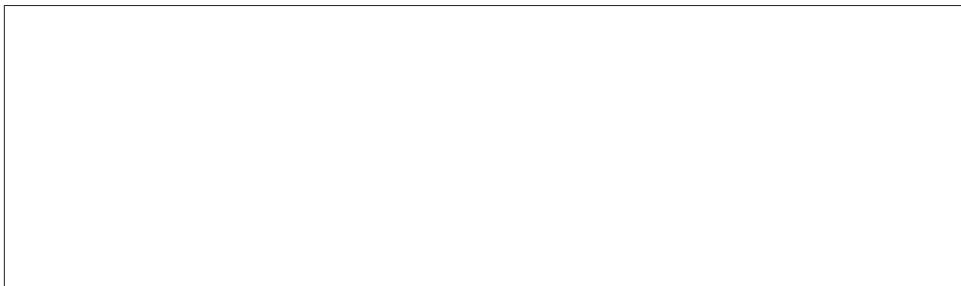
Here you'll look at two types of bodies, particles and rigid bodies. A rigid body is a system of particles that remain at fixed distances from one another with no relative translation or rotation among them. In other words, a rigid body does not change its shape as it moves—or any changes in its shape are so small or unimportant that they can safely be neglected. When you are considering a rigid body, its dimensions and orientation are important, and you must account for both the body's linear motion and its angular motion.

A particle, on the other hand, is a body that has mass but whose dimensions are negligible or unimportant in the problem being investigated. For example, when considering the path of a projectile or a rocket over a great distance, you can safely ignore the body's dimensions when analyzing its trajectory. When you are considering a particle, its linear motion is important, but the angular motion of the particle itself is not. Think of it this way: when looking at a particle, you are zooming way out to view the big picture, so to speak, as opposed to zooming in as you do when looking at the rotation of rigid bodies.

Whether you are looking at problems involving particles or rigid bodies, there are some important kinematic properties common to both. These are, of course, the object's position, velocity, and acceleration. The next section discusses these properties in detail.

Velocity and Acceleration

In general, velocity is a vector quantity that has magnitude and direction. The magnitude of velocity is speed. Speed is a familiar term—it's how fast your speedometer says you're going when driving your car down the highway. Formally, speed is the rate of travel, or the ratio of distance traveled to the time it took to travel that distance. In math terms, you can write:



or whether or not it is traveling at a constant 60 mi/hr. It could very well be that the car was accelerating (or decelerating) over that 30 m distance.

To more precisely analyze the motion of the car in this example, you need to understand the concept of *instantaneous* velocity. Instantaneous velocity is the specific velocity at a given instant in time, not over a large time interval as in the car example. This means that you need to look at very small Δt 's. In math terms, you must consider the limit as Δt approaches 0—that is, as Δt gets infinitesimally small. This is written as follows:

Taking the limit as Δt goes to 0 gives the instantaneous acceleration:

velocity and the second derivative of distance traveled with respect to time is acceleration, which is the same as the first derivative of velocity with respect to time.

Constant Acceleration

One of the simplest classes of problems in kinematics involves constant acceleration. A good example of this sort of problem involves the acceleration due to gravity, g , on objects moving relatively near the earth's surface, where the gravitational acceleration is a constant 9.81 m/s^2 . Having constant acceleration makes integration over time relatively easy since you can pull the acceleration constant out of the integrand, leaving just dt .

Integrating the relationship between velocity and acceleration described earlier when acceleration is constant yields the following equation for instantaneous velocity:

You can derive a similar formula for displacement as a function of velocity, acceleration, and time by integrating the differential equation:



To find: Given these: Use this:

$$v_1 \quad a, v_2, \Delta s \quad v_1 = \sqrt{v_2^2 - 2a\Delta s}$$

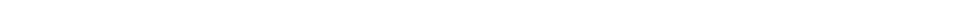
In cases where acceleration is not constant, but is some function of time, velocity, or position, you can substitute the function for acceleration into the differential equations shown earlier to derive new equations for instantaneous velocity and displacement. The next section considers such a problem.

Nonconstant Acceleration

A common situation that arises in real-world problems is when drag forces act on a body in motion. Typically, drag forces are proportional to velocity squared. Recalling the equation of Newton's second law of motion, $F = ma$, you can deduce that the acceleration induced by these drag forces is also proportional to velocity squared

Later we'll show you some techniques to calculate this sort of drag force, but for now let the functional form of drag-induced acceleration be:

It follows that:



If the distance to the target, n , equals 500 m and the muzzle velocity, v_m , equals 800 m/sec, then the equations for t_{hit} and d give:

We'll show you how to set up the kinematic equations for this problem by treating each vector component separately at first and then combining these components.

X Components

The x components here are similar to those in the previous section's rifle example in that there is no drag force acting on the shell; thus, the x component of acceleration is 0, which means that the x component of velocity is constant and equal to the x component of the muzzle velocity as the shell leaves the cannon. Note that since the cannon barrel may not be horizontal, you'll have to compute the x component of the muzzle velocity, which is a function of the direction in which the cannon is aimed.

The muzzle velocity vector is:

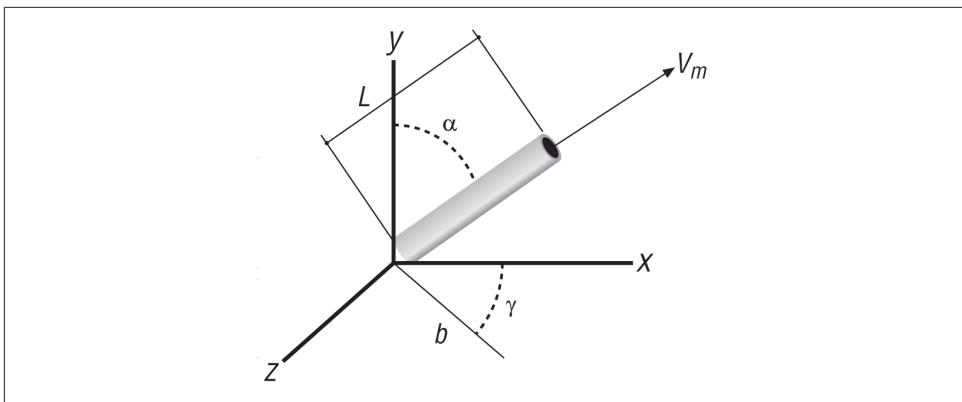


Figure 2-4. Cannon orientation

Using these angles, it follows that the projection, *b*, of the cannon length, *L*, onto the *x*-*z* plane is:

Observe here that the displacement vector essentially gives the position of the shell's center of mass at any given instant in time; thus, you can use this vector to plot the shell's trajectory from the cannon to the target.

Hitting the Target

Now that you have the equations fully describing the shell's trajectory, you need to consider the location of the target in order to determine when a direct hit occurs. To show you how to do this, we've prepared a sample program that implements these kinematic equations along with a simple bounding box collision detection method for checking whether or not the shell has struck the target. Basically, at each time step where we calculate the position of the shell after it has left the cannon, we check to see if this position falls within the bounding dimensions of the target object represented by a cube.

The sample program is set up such that you can change all of the variables in the simulation and view the effects of your changes. [Figure 2-5](#) shows the main screen for the cannon example program, with the governing variables shown on the left. The upper illustration is a bird's-eye view looking down on the cannon and the target, while the lower illustration is a profile (side) view.

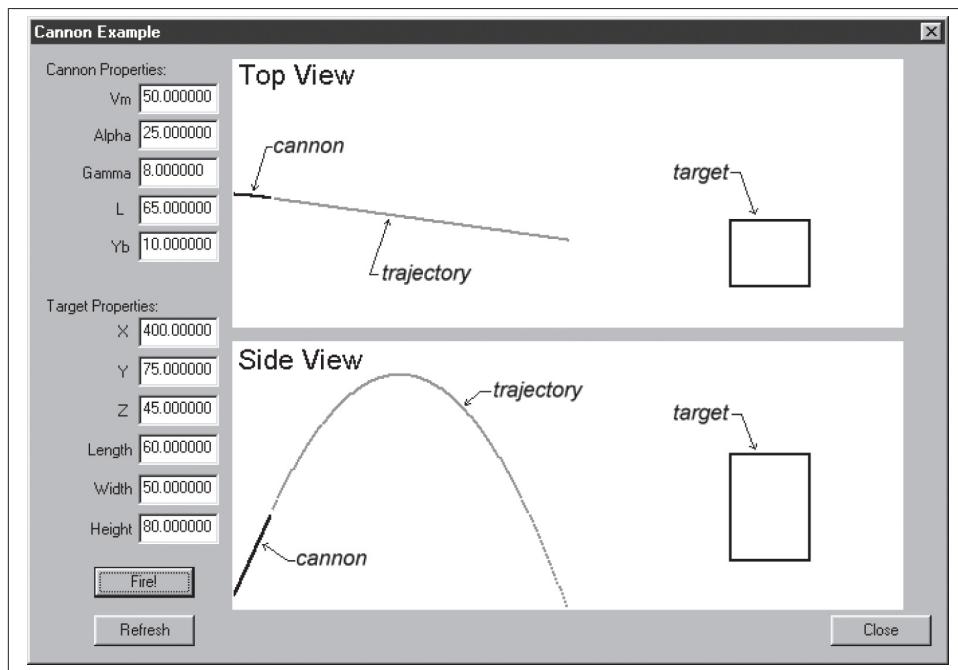


Figure 2-5. Cannon sample program main window

You can change any of the variables shown on the main window and press the Fire button to see the resulting flight path of the shell. A message box will appear when you hit the target or when the shell hits the ground. The program is set up so you can repeatedly change the variables and press Fire to see the result without erasing the previous trial. This allows you to gauge how much you need to adjust each variable in order to hit the target. Press the Refresh button to redraw the views when they get too cluttered.

Figure 2-6 shows a few trial shots that we made before finally hitting the target.

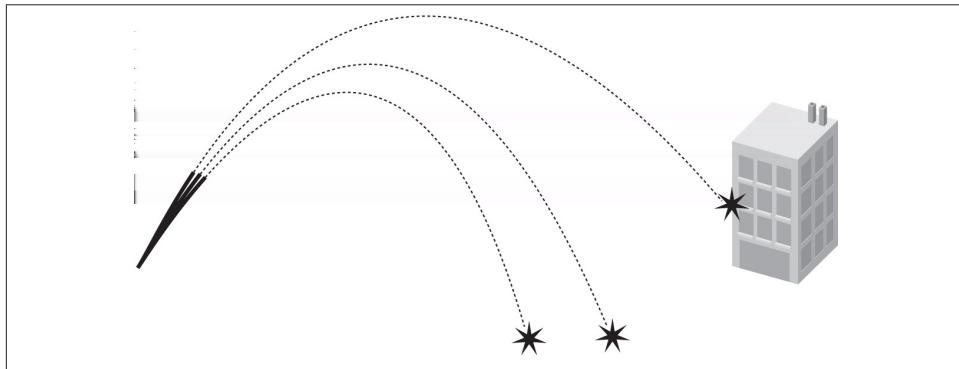


Figure 2-6. Trial shots (profile view)

The code for this example is really quite simple. Aside from the overhead of the window, controls, and illustrations setup, all of the action takes place when the Fire button is pressed. In pseudocode, the Fire button's pressed event handler looks something like this:

```
FIRE BUTTON PRESSED EVENT:  
  
Fetch and store user input values for global variables,  
Vm, Alpha, Gamma, L, Yb, X, Y, Z, Length, Width, Height...  
  
Initialize the time and status variables...  
status = 0;  
time = 0;  
  
Start stepping through time for the simulation  
until the target is hit, the shell hits  
the ground, or the simulation times out...  
  
while(status == 0)  
{  
    // do the next time step  
    status = DoSimulation();
```

```

        Update the display...

    }

    // Report results
    if (status == 1)
        Display DIRECT HIT message to the user...

    if (status == 2)
        Display MISSED TARGET message to the user...

    if (status == 3)
        Display SIMULATION TIMED OUT message to the user...

```

The first task is to simply get the new values for the variables shown on the main window. After that, the program enters a `while` loop, stepping through increments of time and recalculating the position of the shell projectile using the formula for the displacement vector, s , shown earlier. The shell position at the current time is calculated in the function `DoSimulation`. Immediately after calling `DoSimulation`, the program updates the illustrations on the main window, showing the shell's trajectory. `DoSimulation` returns 0, keeping the `while` loop going, if there has not yet been a collision or if the time has not yet reached the preset time-out value.

Once the `while` loop terminates by `DoSimulation` returning nonzero, the program checks the return value from this function call to see if a hit has occurred between the shell and the ground or the shell and the target. Just so the program does not get stuck in this `while` loop, `DoSimulation` will return a value of 3, indicating that it is taking too long.

Now let's look at what's happening in the function `DoSimulation` (we've also included here the global variables that are used in `DoSimulation`).

```

//-----
// Define a custom type to represent
// the three components of a 3D vector, where
// i represents the x component, j represents
// the y component, and k represents the z
// component
//-----
typedef struct TVectorTag
{
    double i;
    double j;
    double k;
} TVector;

//-----
// Now define the variables required for this simulation
//-----
double      Vm;      // Magnitude of muzzle velocity, m/s

```

```

double      Alpha; // Angle from y-axis (upward) to the cannon.
                // When this angle is 0, the cannon is pointing
                // straight up, when it is 90 degrees, the cannon
                // is horizontal
double      Gamma; // Angle from x-axis, in the x-z plane to the cannon.
                  // When this angle is 0, the cannon is pointing in
                  // the positive x-direction, positive values of this angle
                  // are toward the positive z-axis
double      L;    // This is the length of the cannon, m
double      Yb;   // This is the base elevation of the cannon, m

double      X;    // The x-position of the center of the target, m
double      Y;    // The y-position of the center of the target, m
double      Z;    // The z-position of the center of the target, m
double      Length; // The length of the target measured along the x-axis, m
double      Width; // The width of the target measured along the z-axis, m
double      Height; // The height of the target measured along the y-axis, m

TVector     s;    // The shell position (displacement) vector

double      time; // The time from the instant the shell leaves
                  // the cannon, seconds
double      tInc; // The time increment to use when stepping through
                  // the simulation, seconds

double      g;    // acceleration due to gravity, m/s^2

//-----//  

// This function steps the simulation ahead in time. This is where the kinematic  

// properties are calculated. The function will return 1 when the target is hit,  

// and 2 when the shell hits the ground (x-z plane) before hitting the target;  

// otherwise, the function returns 0.  

//-----//  

int      DoSimulation(void)
//-----//  

{
    double      cosX;
    double      cosY;
    double      cosZ;
    double      xe, ze;
    double      b, Lx, Ly, Lz;
    double      tx1, tx2, ty1, ty2, tz1, tz2;

    // step to the next time in the simulation
    time+=tInc;

    // First calculate the direction cosines for the cannon orientation.
    // In a real game, you would not want to put this calculation in this
    // function since it is a waste of CPU time to calculate these values
    // at each time step as they never change during the sim. We only put them
    // here in this case so you can see all the calculation steps in a single
    // function.
}

```

```

b = L * cos((90-Alpha) *3.14/180); // projection of barrel onto x-z plane
Lx = b * cos(Gamma * 3.14/180); // x-component of barrel length
Ly = L * cos(Alpha * 3.14/180); // y-component of barrel length
Lz = b * sin(Gamma * 3.14/180); // z-component of barrel length

cosX = Lx/L;
cosY = Ly/L;
cosZ = Lz/L;

// These are the x and z coordinates of the very end of the cannon barrel
// we'll use these as the initial x and z displacements
xe = L * cos((90-Alpha) *3.14/180) * cos(Gamma * 3.14/180);
ze = L * cos((90-Alpha) *3.14/180) * sin(Gamma * 3.14/180);

// Now we can calculate the position vector at this time
s.i = Vm * cosX * time + xe;
s.j = (Yb + L * cos(Alpha*3.14/180)) + (Vm * cosY * time) -
      (0.5 * g * time * time);
s.k = Vm * cosZ * time + ze;

// Check for collision with target
// Get extents (bounding coordinates) of the target
tx1 = X - Length/2;
tx2 = X + Length/2;
ty1 = Y - Height/2;
ty2 = Y + Height/2;
tz1 = Z - Width/2;
tz2 = Z + Width/2;

// Now check to see if the shell has passed through the target
// We're using a rudimentary collision detection scheme here where
// we simply check to see if the shell's coordinates are within the
// bounding box of the target. This works for demo purposes, but
// a practical problem is that you may miss a collision if for a given
// time step the shell's change in position is large enough to allow
// it to "skip" over the target.
// A better approach is to look at the previous time step's position data
// and to check the line from the previous position to the current position
// to see if that line intersects the target bounding box.
if( (s.i >= tx1 && s.i <= tx2) &&
   (s.j >= ty1 && s.j <= ty2) &&
   (s.k >= tz1 && s.k <= tz2) )
   return 1;

// Check for collision with ground (x-z plane)
if(s.j <= 0)
   return 2;

// Cut off the simulation if it's taking too long
// This is so the program does not get stuck in the while loop
if(time>3600)
   return 3;

```

```
    return 0;  
}
```

We've commented the code so that you can readily see what's going on. This function essentially does four things: 1) increments the time variable by the specified time increment, 2) calculates the initial muzzle velocity components in the x-, y-, and z-directions, 3) calculates the shell's new position, and 4) checks for a collision with the target using a bounding box scheme or the ground.

Here's the code that computes the shell's position:

```
// Now we can calculate the position vector at this time  
s.i = Vm * cosX * time + xe;  
s.j = (Yb + L * cos(Alpha*3.14/180)) + (Vm * cosY * time) -  
    (0.5 * g * time * time);  
s.k = Vm * cosZ * time + ze;
```

This code calculates the three components of the displacement vector, *s*, using the formulas that we gave you earlier. If you wanted to compute the velocity and acceleration vectors as well, just to see their values, you should do so in this section of the program. You can set up a couple of new global variables to represent the velocity and acceleration vectors, just as we did with the displacement vector, and apply the velocity and acceleration formulas that we gave you.

That's all there is to it. It's obvious by playing with this sample program that the shell's trajectory is parabolic in shape, which is typical *projectile motion*. We'll take a more detailed look at this sort of motion in [Chapter 6](#).

Even though we put a comment in the source code, we must reiterate a warning here regarding the collision detection scheme that we used in this example. Because we're checking only the current position coordinate to see if it falls within the bounding dimensions of the target cube, we run the risk of skipping over the target if the change in position is too large for a given time step. A better approach would be to keep track of the shell's previous position and check to see if the line connecting the previous position to the new one intersects the target cube.

Kinematic Particle Explosion

At this point you might be wondering how particle kinematics can help you create realistic game content unless you're writing a game that involves shooting a gun or a cannon. If so, let us offer you a few ideas and then show you an example. Say you're writing a football simulation game. You can use particle kinematics to model the trajectory of the football after it's thrown or kicked. You can also treat the wide receivers as particles when calculating whether or not they'll be able to catch the thrown ball. In this scenario you'll have two particles—the receiver and the ball—traveling independently, and you'll have to calculate when a collision occurs between these two particles,

indicating a catch (unless, of course, your player is all thumbs and fumbles the ball after it hits his hands). You can find similar applications for other sports-based games as well.

What about a 3D “shoot ‘em up” game? How could you use particle kinematics in this genre aside from bullets, cannons, grenades, and the like? Well, you could use particle kinematics to model your player when she jumps into the air, either from a run or from a standing position. For example, your player reaches the middle of a catwalk only to find a section missing, so you immediately back up a few paces to get a running head start before leaping into the air, hoping to clear the gap. This long-jump scenario is perfect for using particle kinematics. All you really need to do is define your player’s initial velocity, both speed and take-off angle, and then apply the vector formula for displacement to calculate whether or not the player makes the jump. You can also use the displacement formula to calculate the player’s trajectory so that you can move the player’s viewpoint accordingly, giving the illusion of leaping into the air. You may in fact already be using these principles to model this action in your games, or at least you’ve seen it done if you play games of this genre. If your player happens to fall short on the jump, you can use the formulas for velocity to calculate the player’s impact velocity when she hits the ground below. Based on this impact velocity, you can determine an appropriate amount of damage to deduct from the player’s health score, or if the velocity is over a certain threshold, you can say goodbye to your would-be adventurer!

Another use for simple particle kinematics is for certain special effects like particle explosions. This sort of effect is quite simple to implement and really adds a sense of realism to explosion effects. The particles don’t just fly off in random, straight-line trajectories. Instead, they rise and fall under the influence of their initial velocity, angle, and the acceleration due to gravity, which gives the impression that the particles have mass.

So, let’s explore an example of a kinematic particle explosion. The code for this example is taken from the cannon example discussed previously, so a lot of it should look familiar to you. [Figure 2-7](#) shows this example program’s main window.

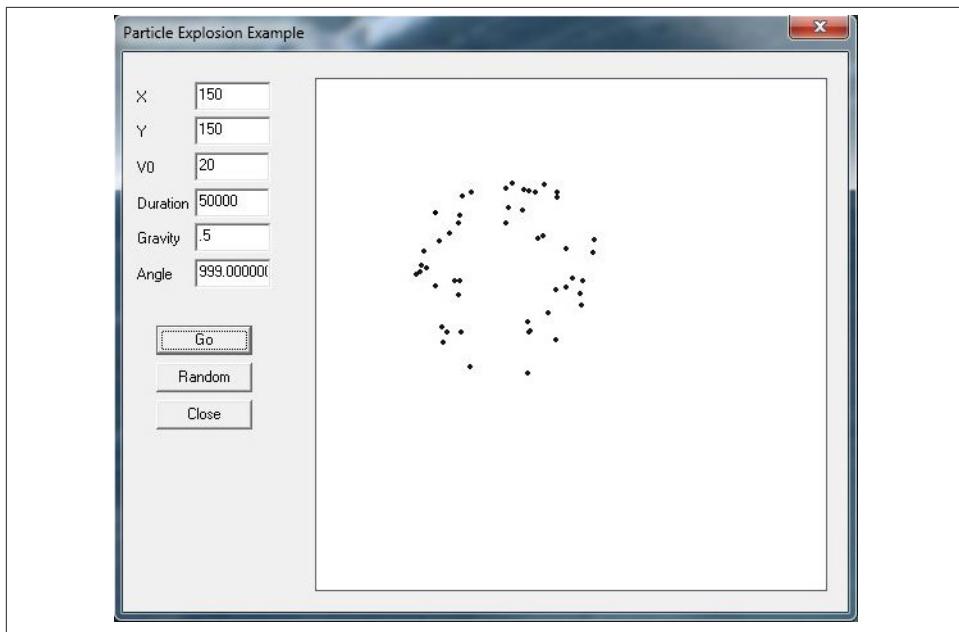


Figure 2-7. Particle explosion example program

The explosion effect takes place in the large rectangular area on the right. While the black dots representing exploding particles are certainly static in the figure, we assure you they move in the most spectacular way during the simulation.

In the edit controls on the left, you specify an x- and y-position for the effect, along with the initial velocity of the particles (which is a measure of the explosion's strength), a duration in milliseconds, a gravity factor, and finally an angle. The angle parameter can be any number between 0 and 360 degrees or 999. When you specify an angle in the range of 0 to 360 degrees, all the particles in the explosion will be launched generally in that direction. If you specify a value of 999, then all the particles will shoot off in random directions. The duration parameter is essentially the life of the effect. The particles will fade out as they approach that life.

The first thing you need to do for this example is set up some structures and global variables to represent the particle effect and the individual particles making up the effect along with the initial parameters describing the behavior of the effect as discussed in the previous paragraph. Here's the code:

```
//-----  
// Define a custom type to represent each particle in the effect.  
//-----  
typedef struct _TParticle  
{  
    float      x;           // x coordinate of the particle
```

```

        float      y;           // y coordinate of the particle
        float      vi;          // initial velocity
        float      angle;       // initial trajectory (direction)
        int       life;         // duration in milliseconds
        int       r;            // red component of particle's color
        int       g;            // green component of particle's color
        int       b;            // blue component of particle's color
        int       time;         // keeps track of the effect's time
        float     gravity;      // gravity factor
        BOOL     Active;        // indicates whether this particle
                                // is active or dead
    } TParticle;

#define      _MAXPARTICLES 50

typedef struct _TParticleExplosion
{
    TParticle   p[_MAXPARTICLES]; // list of particles
                                // making up this effect
    int        V0; // initial velocity, or strength, of the effect
    int        x;  // initial x location
    int        y;  // initial y location
    BOOL      Active; // indicates whether this effect is
                      // active or dead
} TParticleExplosion;

//-----//
// Now define the variables required for this simulation
//-----//
TParticleExplosion    Explosion;

int             xc;          // x coordinate of the effect
int             yc;          // y coordinate of the effect
int             V0;          // initial velocity
int             Duration;    // life in milliseconds
float           Gravity;     // gravity factor (acceleration)
float           Angle;       // indicates particles' direction

```

You can see from this code that the particle explosion effect is made up of a collection of particles. The behavior of each particle is determined by kinematics and the initial parameters set for each particle.

Whenever you press the Go button, the initial parameters that you specified are used to initialize the particle explosion (if you press the Random button, the program randomly selects these initial values for you). This takes place in the function called `CreateParticleExplosion`:

```

///////////////////////////////
/* This function creates a new particle explosion effect.

x,y:      starting point of the effect
Vinit:    a measure of how fast the particles will be sent flying

```

```

        (it's actually the initial velocity of the particles)
life:    life of the particles in milliseconds; particles will
        fade and die out as they approach
        their specified life
gravity: the acceleration due to gravity, which controls the
        rate at which particles will fall
        as they fly
angle:   initial trajectory angle of the particles,
        specify 999 to create a particle explosion
        that emits particles in all directions; otherwise,
        0 right, 90 up, 180 left, etc...
*/
void CreateParticleExplosion(int x, int y, int Vinit, int life,
                             float gravity, float angle)
{
    int i;
    int m;
    float f;

    Explosion.Active = TRUE;
    Explosion.x = x;
    Explosion.y = y;
    Explosion.V0 = Vinit;

    for(i=0; i<_MAXPARTICLES; i++)
    {
        Explosion.p[i].x = 0;
        Explosion.p[i].y = 0;
        Explosion.p[i].vi = tb_Rnd(Vinit/2, Vinit);

        if(angle < 999)
        {
            if(tb_Rnd(0,1) == 0)
                m = -1;
            else
                m = 1;
            Explosion.p[i].angle = -angle + m * tb_Rnd(0,10);
        } else
            Explosion.p[i].angle = tb_Rnd(0,360);

        f = (float) tb_Rnd(80, 100) / 100.0f;
        Explosion.p[i].life = tb_Round(life * f);
        Explosion.p[i].r = 255;//tb_Rnd(225, 255);
        Explosion.p[i].g = 255;//tb_Rnd(85, 115);
        Explosion.p[i].b = 255;//tb_Rnd(15, 45);
        Explosion.p[i].time = 0;
        Explosion.p[i].Active = TRUE;
        Explosion.p[i].gravity = gravity;
    }
}

```

Here you can see that all the particles are set to start off in the same position, as specified by the *x* and *y* coordinates that you provide; however, you'll notice that the initial velocity of each particle is actually randomly selected from a range of *Vinit*/2 to *Vinit*. We do this to give the particle behavior some variety. We do the same thing for the life parameter of each particle so they don't all fade out and die at the exact same time.

After the particle explosion is created, the program enters a loop to propagate and draw the effect. The loop is a `while` loop, as shown here in pseudocode:

```
while(status)
{
    Clear the off screen buffer...

    status = DrawParticleExplosion( );

    Copy the off screen buffer to the screen...
}
```

The `while` loop continues as long as `status` remains `true`, which indicates that the particle effect is still alive. After all the particles in the effect reach their set life, then the effect is dead and `status` will be set to `false`. All the calculations for the particle behavior actually take place in the function called `DrawParticleExplosion`; the rest of the code in this `while` loop is for clearing the off-screen buffer and then copying it to the main window.

`DrawParticleExplosion` updates the state of each particle in the effect by calling another function, `UpdateParticleState`, and then draws the effect to the off-screen buffer passed in as a parameter. Here's what these two functions look like:

```
//-----
// Draws the particle system and updates the state of each particle.
// Returns false when all of the particles have died out.
//-----

BOOL      DrawParticleExplosion(void)
{
    int      i;
    BOOL     finished = TRUE;
    float    r;

    if(Explosion.Active)
        for(i=0; i<_MAXPARTICLES; i++)
    {
        if(Explosion.p[i].Active)
        {
            finished = FALSE;

            // Calculate a color scale factor to fade the particle's color
            // as its life expires
            r = ((float)(Explosion.p[i].life-
                Explosion.p[i].time)/(float)(Explosion.p[i].life));
    }
}
```

```

    ...
    Draw the particle as a small circle...
    ...

    Explosion.p[i].Active = UpdateParticleState(&(Explosion.p[i]),
                                                10);
}
}

if(finished)
    Explosion.Active = FALSE;

return !finished;
}

//-----*/
/* This is generic function to update the state of a given particle.
   p:          pointer to a particle structure
   dtime:      time increment in milliseconds to
               advance the state of the particle

If the total elapsed time for this particle has exceeded the particle's
set life, then this function returns FALSE, indicating that the particle
should expire.
*/
BOOL    UpdateParticleState(TParticle* p, int dtime)
{
    BOOL retval;
    float t;

    p->time+=dtime;
    t = (float)p->time/1000.0f;
    p->x = p->vi * cos(p->angle*PI/180.0f) * t;
    p->y = p->vi * sin(p->angle*PI/180.0f) * t + (p->gravity*t*t)/2.0f;

    if (p->time >= p->life)
        retval = FALSE;
    else
        retval = TRUE;

    return retval;
}

```

`UpdateParticleState` uses the kinematic formulas that we've already shown you to update the particle's position as a function of its initial velocity, time, and the acceleration due to gravity. After `UpdateParticleState` is called, `DrawParticleExplosion` scales down each particle's color, fading it to black, based on the life of each particle and elapsed time. The fade effect is simply to show the particles dying slowly over time instead of disappearing from the screen. The effect resembles the behavior of fireworks as they explode in the night sky.

Rigid-Body Kinematics

The formulas for displacement, velocity, and acceleration discussed in the previous sections apply equally well for rigid bodies as they do for particles. The difference is that with rigid bodies, the point on the rigid body that you track, in terms of linear motion, is the body's center of mass (gravity).

When a rigid body translates with no rotation, all of the particles making up the rigid body move on parallel paths since the body does not change its shape. Further, when a rigid body does rotate, it generally rotates about axes that pass through its center of mass, unless the body is hinged at some other point about which it's forced to rotate. These facts make the center of mass a convenient point to use to track its linear motion. This is good news for you since you can use all of the material you learned on particle kinematics here in your study of rigid-body kinematics.

The procedure for dealing with rigid bodies involves two distinct aspects: 1) tracking the translation of the body's center of mass, and 2) tracking the body's rotation. The first aspect is old hat by now—just treat the body as a particle. The second aspect, however, requires you to consider a few more concepts—namely, local coordinates, angular displacement, angular velocity, and angular acceleration.

For most of the remainder of this chapter, we'll discuss *plane* kinematics of rigid bodies. Plane motion simply means that the body's motion is restricted to a flat plane in space where the only axis of rotation about which the body can rotate is perpendicular to the plane. Plane motion is essentially two-dimensional. This allows us to focus on the new kinematic concepts of angular displacement, velocity, and acceleration without getting lost in the math required to describe arbitrary rotation in three dimensions.

You might be surprised by how many problems lend themselves to plane kinematic solutions. For example, in some popular 3D “shoot 'em up” games, your character is able to push objects, such as boxes and barrels, around on the floor. While the game world here is three dimensions, these particular objects may be restricted to sliding on the floor—a plane—and thus can be treated like a 2D problem. Even if the player pushes on these objects at some angle instead of straight on, you'll be able to simulate the sliding and rotation of these objects using 2D kinematics (and kinetics) techniques.

Local Coordinate Axes

Earlier, we defined the Cartesian coordinate system to use for your fixed global reference, or world coordinates. This world coordinate system is all that's required when treating particles; however, for rigid bodies you'll also use a set of local coordinates fixed to the body. Specifically, this local coordinate system will be fixed at the body's center-of-mass location. You'll use this coordinate system to track the orientation of the body as it rotates.

For plane motion, we require only one scalar quantity to describe the body's orientation. This is illustrated in [Figure 2-8](#).

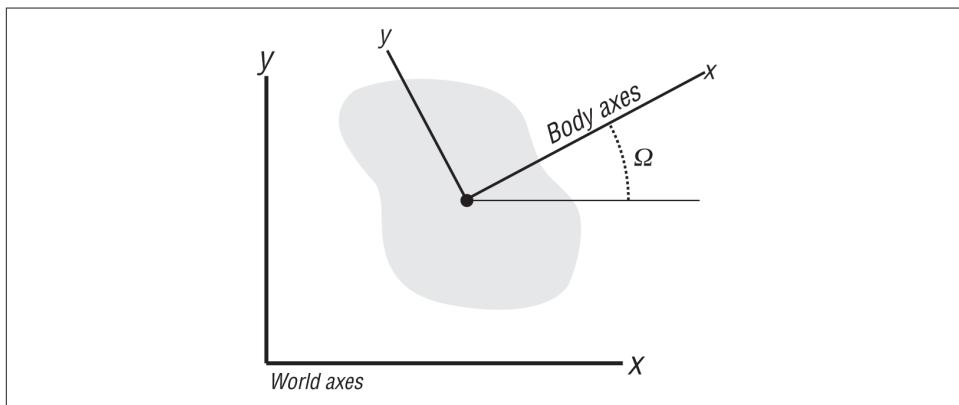


Figure 2-8. Local coordinate axes

Here the orientation, Ω , is defined as the angular difference between the two sets of coordinate axes: the fixed world axes and the local body axes. This is the so-called Euler angle. In general 3D motion there is a total of three Euler angles, which are usually called *yaw*, *pitch*, and *roll* in aerodynamic and hydrodynamic jargon. While these angular representations are easy to visualize in terms of their physical meaning, they aren't so nice from a numerical point of view, so you'll have to look for alternative representations when writing your 3D real-time simulator. These issues are addressed in [Chapter 9](#).

Angular Velocity and Acceleration

In two-dimensional plane motion, as the body rotates, Ω will change, and the rate at which it changes is the angular velocity, ω . Likewise, the rate at which ω changes is the angular acceleration, α . These angular properties are analogous to the linear properties of displacement, velocity, and acceleration. The units for angular displacement, velocity,

and acceleration are radians (rad), radians per sec (rad/s), and radians per second-squared (rad/s²), respectively.

Mathematically, you can write these relations between angular displacement, angular velocity, and angular acceleration as:

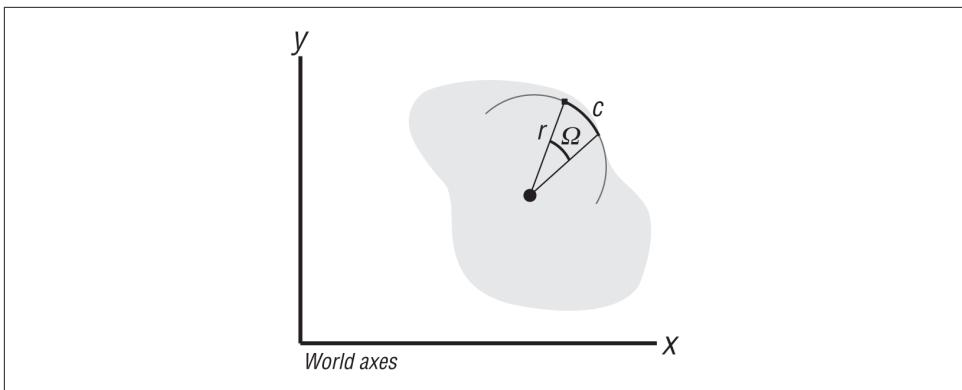


Figure 2-9. Circular path of particles making up a rigid body

The formula relating arc length to angular displacement is:

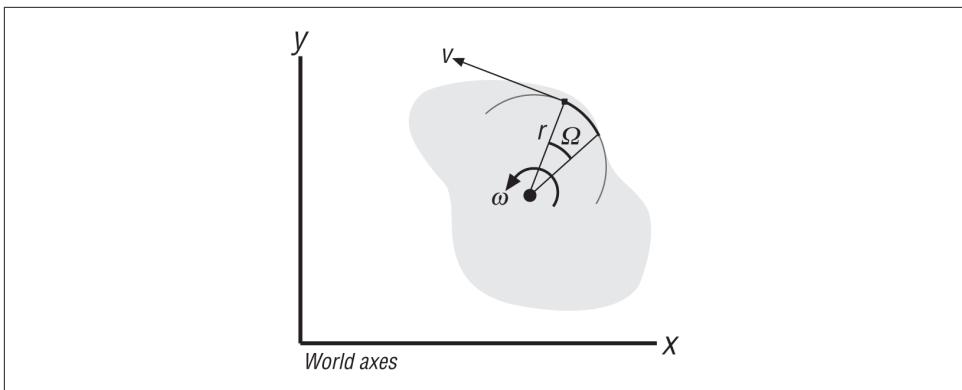


Figure 2-10. Linear velocity due to angular velocity

Differentiating the equation, $v = r \omega$:

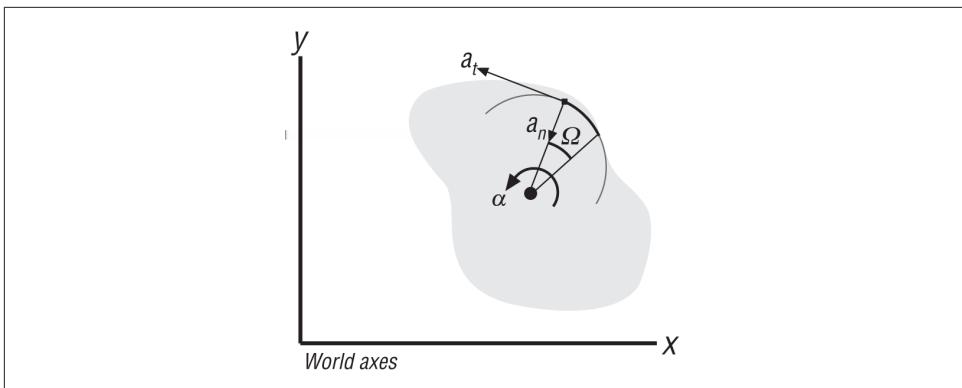


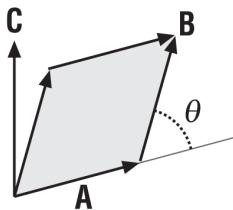
Figure 2-11. Tangential and centripetal acceleration

The formula for the magnitude of centripetal acceleration, a_n , is:

Note that this gives both the magnitude and direction of the linear, tangential velocity. Also, be sure to preserve the order of the vectors when taking the cross product—that is, ω cross \mathbf{r} , and not the other way around, which would give the wrong direction for \mathbf{v} .

Vector Cross Product

Given any two vectors \mathbf{A} and \mathbf{B} , the cross product $\mathbf{A} \times \mathbf{B}$ is defined by a third vector \mathbf{C} with a magnitude equal to $AB \sin \theta$, where θ is the angle between the two vectors \mathbf{A} and \mathbf{B} , as illustrated in the following figure.



what each particle making up the rigid body is doing all the time. Thus, you treat the rigid body's linear motion and its angular motion separately. When you do need to take a close look at specific particles of—or points on—the rigid body, you can do so by taking the motion of the rigid body as a particle and then adding to it the relative motion of the point under consideration.

Figure 2-12 shows a rigid body that is traveling at a speed v_{cg} , where v_{cg} is the speed of the rigid body's center of mass (or center of gravity). Remember, the center of mass is the point to track when treating a rigid body as a particle. This rigid body is also rotating with an angular velocity ω about an axis that passes through the body's center of mass. The vector r is the vector from the rigid body's center of mass to the particular point of interest, P , located on the rigid body.

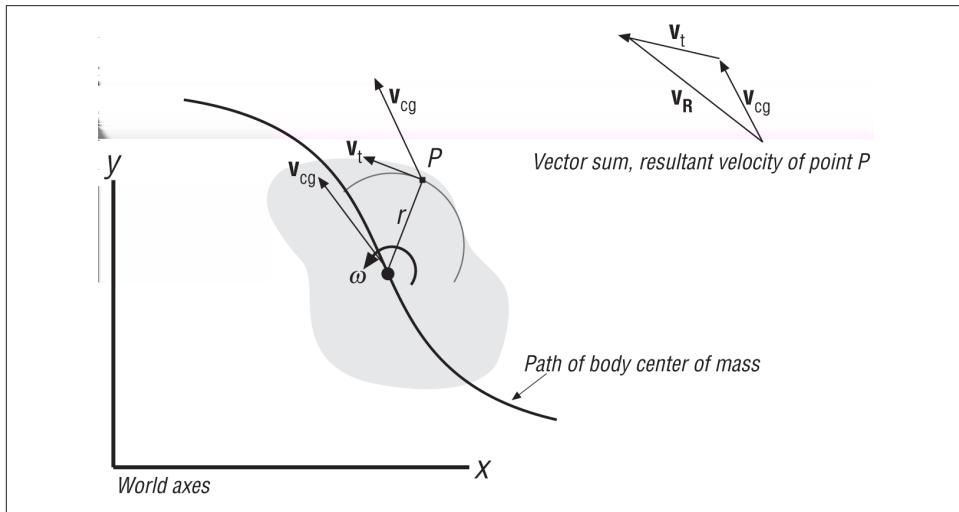


Figure 2-12. Relative velocity

In this case, we can find the resultant velocity of the point, P , by taking the vector sum of the velocity of the body's center of mass and the tangential velocity of point P due to the body's angular velocity ω . Here's what the vector equation looks like:

mass, the tangential acceleration due to the body's angular acceleration, and the centripetal acceleration due to the change in direction of the tangential velocity. In equation form, this looks like:



CHAPTER 3

Force

This chapter is a prerequisite to [Chapter 4](#), which addresses the subject of kinetics. The aim here is to provide you with enough of a background on forces so you can readily appreciate the subject of kinetics. This chapter is not meant to be the final word on the subject of force. In fact, we feel that the subject of force is so important to realistic simulations that we'll revisit it several times in various contexts throughout the remainder of this book. In this chapter, we'll discuss the two fundamental categories of force and briefly explain some important specific types of force. We'll also explain the relationship between force and torque.

Forces

As we mentioned in [Chapter 2](#), you need to understand the concept of force before you can fully understand the subject of kinetics. Kinematics is only half the battle. You are already familiar with the concept of force from your daily experiences. You exert a force on this book as you hold it in your hands, counteracting gravity. You exert force on your mouse as you move it from one point to another. When you play soccer, you exert force on the ball as you kick it. In general, force is what makes an object move, or more precisely, what produces an acceleration that changes the velocity. Even as you hold this book, although it may not be moving, you've effectively produced an acceleration that cancels the acceleration from gravity. When you kick that soccer ball, you change its velocity from, say, 0 when the ball is at rest to some positive value as the ball leaves your foot. These are some examples of externally applied *contact* forces.

There's another broad category of forces, in addition to contact forces, called *field* forces or sometimes *forces at a distance*. These forces can act on a body without actually having to make contact with it. A good example is the gravitational attraction between objects. Another example is the electromagnetic attraction between charged particles. The concept of a force field was developed long ago to help us visualize the interaction between objects subject to forces at a distance. You can say that an object is subjected to the

gravitational field of another object. Thinking in terms of force fields can help you grasp the fact that an object can exert a force on another object without having to physically touch it.

Within these two broad categories of forces, there are specific types of forces related to various physical phenomena—forces due to friction, buoyancy, and pressure, among others. We'll discuss idealizations of several of these types of forces in this chapter. Later in this book, we'll revisit these forces from a more practical point of view.

Before going further, we need to explain the implications of Newton's third law as introduced in [Chapter 1](#). Remember, Newton's third law states that for every force acting on a body, there is an equal and opposite reacting force. This means that forces must exist in pairs—a single force can't exist by itself.

Consider the gravitational attraction between the earth and yourself. The earth is exerting a force—your weight—on you, accelerating you toward its center. Likewise, you are exerting a force on the earth, accelerating it toward you. The huge difference between your mass and the earth's makes the acceleration of the earth in this case so small that it's negligible. Earlier we said you are exerting a force on this book to hold it up; likewise, this book is exerting a force on your hands equal in magnitude but opposite in direction to the force you are exerting on it. You feel this reaction force as the book's weight.

This phenomenon of action-reaction is the basis for rocket propulsion. A rocket engine exerts force on the fuel molecules that are accelerated out of the engine exhaust nozzle. The force required to accelerate these molecules is exerted back against the rocket as a reaction force called *thrust*. Throughout the remainder of this book, you'll see many other examples of action-reaction, which is an important phenomenon in rigid-body dynamics. It is especially important when we are dealing with collisions and objects in contact, as you'll see later.

Force Fields

The best example of a force field or force at a distance is the gravitational attraction between objects. *Newton's law of gravitation* states that the force of attraction between two masses is directly proportional to the product of the masses and inversely proportional to the square of the distances separating the centers of each mass. Further, this law states that the line of action of the force of attraction is along the line that connects the centers of the two masses. This is written as follows:

So far in this book, I've been using the acceleration due to gravity, g , as a constant 9.8 m/s^2 (32.174 ft/s^2). This is true when you are near the earth's surface—for example, at sea level. In reality, g varies with altitude—maybe not by much for our purposes, but it does. Consider Newton's second law along with the law of gravitation for a body near the earth. Equating these two laws, in equation form, yields:



In **Figure 3-1**, the block is resting on the horizontal surface with a small force, F_a , applied to the block on a line of action through the block's center of mass. As this applied force increases, a frictional force will develop between the block and the horizontal surface, tending to resist the motion of the block. The maximum value of this frictional force is:



-
1. *Static* here implies that there is no motion; the block is sitting still with all forces balancing.
 2. The term *dynamic* is sometimes used here instead of *kinetic*.
-

in the technical literature (see the [Bibliography](#) for sources). Note that experimentally determined friction coefficient data will vary, even for the same surface conditions, depending on the specific condition of the material used in the experiments and the execution of the experiment itself.

Fluid Dynamic Drag

Fluid dynamic drag forces oppose motion like friction. In fact, a major component of fluid dynamic drag is friction that results from the relative flow of the fluid over (and in contact with) the body's surface. Friction is not the only component of fluid dynamic drag, though. Depending on the shape of the body, its speed, and the nature of the fluid, fluid dynamic drag will have additional components due to pressure variations in the fluid as it flows around the body. If the body is located at the interface between two fluids (like a ship on the ocean where the two fluids are air and water), an additional component of drag will exist due to the wave generation.

In general, fluid dynamic drag is a complicated phenomenon that is a function of several factors. We won't go into detail in this section on all these factors, since we'll revisit this subject later. However, we do want to discuss how the *viscous* (frictional) component of these drag forces is typically idealized.

Ideal viscous drag is a function of velocity and some experimentally determined *drag coefficient* that's supposed to take into account the surface conditions of the body, the fluid properties (density and viscosity), and the flow conditions. You'll typically see a formula for viscous drag force in the form:

Both of these equations are very simplified and inadequate for practical analysis of fluid flow problems. However, they do offer certain advantages in computer game simulations. Most obviously, these formulas are easy to implement—you need only know the velocity of the body under consideration, which you get from your kinematic equations, and an assumed value for the drag coefficient. This is convenient, as your game world will typically have many different types of objects of all sizes and shapes that would make rigorous analysis of each of their drag properties impractical. If the illusion of realism is all you need, not real-life accuracy, then these formulas may be sufficient.

Another advantage of using these idealized formulas is that you can tweak the drag coefficients as you see fit to help reduce numerical instabilities when solving the equations of motion, while maintaining the illusion of realistic behavior. If real-life accuracy is what you’re going for, then you’ll have no choice but to consider a more involved (read: complicated) approach for determining fluid dynamic drag. We’ll talk more about drag in [Chapter 6](#) through [Chapter 10](#).

Pressure

Many people confuse pressure with force. You have probably heard people say, when explaining a phenomenon, something like, “It pushed with a force of 100 pounds per square inch.” While you understand what they mean, they are technically referring to pressure, not force. Pressure is force per unit area, thus the units *pounds per square inch* (psi) or *pounds per square foot* (psf) and so on. Given the pressure, you’ll need to know the total area acted on by this pressure in order to determine the resultant force. Force equals pressure times area:

Buoyancy

You've no doubt felt the effects of buoyancy when immersing yourself in the bathtub. Buoyancy is why you feel lighter in water than you do in air and why some people can float on their backs in a swimming pool.

Buoyancy is a force that develops when an object is immersed in a fluid. It's a function of the volume of the object and the density of the fluid and results from the pressure differential between the fluid just above the object and the fluid just below the object. Pressure increases the deeper you go in a fluid, thus the pressure is greater at the bottom of an object of a given height than it is at the top of the object. Consider the cube shown in [Figure 3-2](#).

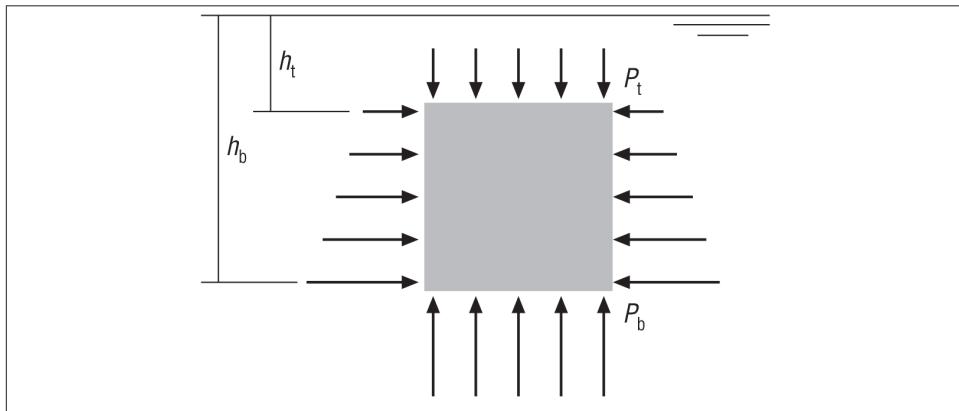


Figure 3-2. Immersed cube

Let s denote the cube's length, width, and height, which are all equal. Further, let h_t denote the depth to the top of the cube and h_b the depth to the bottom of the cube. The pressure at the top of the cube is $P_t = \rho g h_t$, which acts over the entire surface area of the top of the cube, normal to the surface in the downward direction. The pressure at the bottom of the cube is $P_b = \rho g h_b$, which acts over the entire surface area of the bottom of the cube, normal to the surface in the upward direction. Note that the pressure acting on the sides of the cube increases linearly with submergence, from P_t to P_b . Also, note that since the side pressure is symmetric, equal and opposite, the net side pressure is 0, which means that the net side force (due to pressure) is also 0. The same is not true of the top and bottom pressures, which are obviously not equal, although they are opposite.

The force acting down on the top of the cube is equal to the pressure at the top of the cube times the surface area of the top. This can be written as follows:

3. Specific weight is density times the acceleration due to gravity. Typical units are lbs/ft^3 and N/m^3 .
-

that for very light objects with relatively large volumes, the buoyant forces in air may be significant. For example, consider simulating a large balloon.

Springs and Dampers

Springs are structural elements that, when connected between two objects, apply equal and opposite forces to each object. This spring force follows Hooke's law and is a function of the stretched or compressed length of the spring relative to the rest length of the spring and the spring constant of the spring. Hooke's law states that the amount of stretch or compression is directly proportional to the force being applied. The spring constant is a quantity that relates the force exerted by the spring to its deflection:

L is the length of the spring-damper (L , not in bold print, is the magnitude of the vector \mathbf{L}), which is equal to the vector difference in position between the connected points on bodies 1 and 2. If the connected objects are particles, then \mathbf{L} is equal to the position of body 1 minus the position of body 2. Similarly, \mathbf{v}_1 and \mathbf{v}_2 are the velocities of the connected points on bodies 1 and 2. The quantity $(\mathbf{v}_1 - \mathbf{v}_2)$ represents the relative velocity between the connected bodies.

Springs and dampers are useful when you want to simulate collections of connected particles or rigid bodies. The spring force provides the structure, or glue, that holds the bodies together (or keeps them separated by a certain distance), while the damper helps smooth out the motion between the connected bodies so it's not too jerky or springy. These dampers are also very important from a numerical stability point of view in that they help keep your simulations from blowing up. We're getting a little ahead of ourselves here, but we'll show you how to use these spring-dampers in real-time simulations in [Chapter 13](#).

Force and Torque

We need to make the distinction here between force and torque.⁴ Force is what causes linear acceleration, while torque is what causes rotational acceleration. Torque is force times distance. Specifically, to calculate the torque applied by a force acting on an object, you need to calculate the perpendicular distance from the axis of rotation to the line of action of the force and then multiply this distance by the magnitude of the force.

This calculation gives the magnitude of the torque. Typical units for force are pounds, newtons, and tons. Since torque is force times a distance, its units take the form of a length unit times a force unit (e.g., foot-pounds, newton-meters, or foot-tons).

Since both force and torque are vector quantities, you must also determine the direction of the torque vector. The force vector is easy to visualize: its line of action passes through the point of application of the force, with its direction determined by the direction in which the force is applied. As a vector, the torque's line of action is along the axis of rotation, with the direction determined by the direction of rotation and the *right hand rule* (see [Figure 3-3](#)). As noted in [Chapter 2](#), the right hand rule is a simple trick to help you keep track of vector directions—in this case, the torque vector. Pretend to curl the fingers of your right hand around the axis of rotation with your fingertips pointing in the direction of rotation. Now extend your thumb, as though you are giving a thumbs up, while keeping your fingers curled around the axis. The direction that your thumb is pointing indicates the direction of the torque vector. Note that this makes the torque vector perpendicular to the applied force vector, as shown in [Figure 3-3](#).

4. Another common term for torque is *moment*.

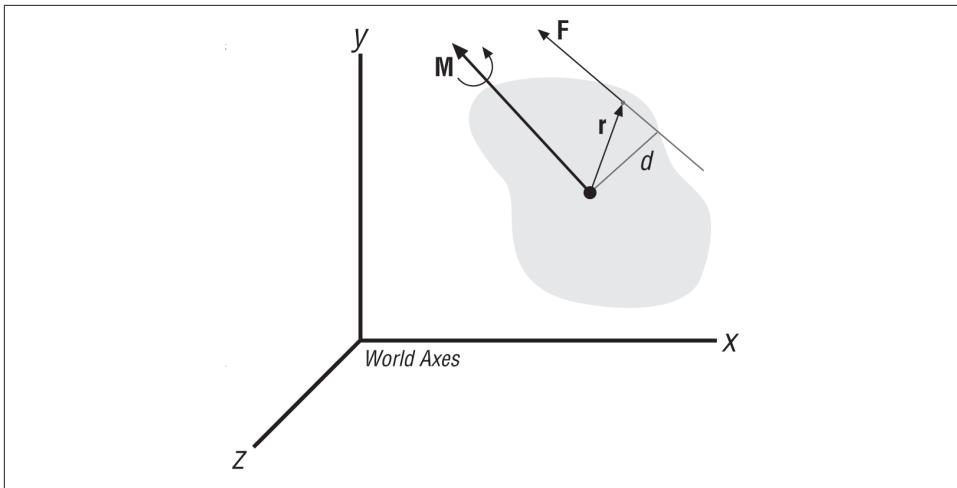


Figure 3-3. Force and torque

We said earlier that you find the magnitude of torque by multiplying the magnitude of the applied force times the perpendicular distance between the axis of rotation and the line of action of the force. This calculation is easy to perform in two dimensions where the perpendicular distance (d in Figure 3-3) is readily calculable.

However, in three dimensions you'll want to be able to calculate torque by knowing only the force vector and the coordinates of its point of application on the body relative to the axis of rotation. You can accomplish this by using the following formula:

Note that the x and y components of the torque vector are 0; thus, the torque moment is pointing directly along the z -axis. The torque vector would be pointing out of the page of this book in this case.

In dynamics you need to consider the sum, or total, of all forces acting on an object separately from the sum of all torques acting on a body. When summing forces, you simply add, vectorally, all of the forces without regard to their point of application. However, when summing torques you must take into account the point of application of the forces to calculate the torques, as shown in the previous example. Then you can take the vector sum of all torques acting on the body.

When you are considering rigid bodies that are not physically constrained to rotate about a fixed axis, any force acting through the body's center of mass will not produce a torque on the body about its center of gravity. In this case, the axis of rotation passes through the center of mass of the body and the vector \mathbf{r} would be 0 (all components 0). When a force acts through a point on the body some distance away from its center of mass, a torque on the body will develop, and the angular motion of the body will be affected. Generally, field forces, which are forces at a distance, are assumed to act through a body's center of mass; thus, only the body's linear motion will be affected unless the body is constrained to rotate about a fixed point. Other contact forces, however, generally do not act through a body's center of mass (they could but aren't necessarily assumed to) and tend to affect the body's angular motion as well as its linear motion.

Summary

As we said earlier, this chapter on forces is your bridge from kinematics to kinetics. Here you've looked at the major force categories—contact forces and force fields—and some important specific types of forces. This chapter was meant to give you enough theoretical background on forces so you can fully appreciate the subject of kinetics that's covered in the next chapter. In [Chapter 15](#) through [Chapter 19](#), you'll revisit the subject of forces from a much more practical point of view when we investigate specific real-life problems. We'll also introduce some new specific types of force in those chapters that we didn't cover here.

CHAPTER 4

Kinetics

Recall that kinetics is the study of the motion of bodies, including the forces that act on them. It's now time that we combine the material presented in the earlier chapters—namely, kinematics and forces—to study the subject of kinetics. As in [Chapter 2](#) on kinematics, we'll first discuss particle kinetics and then go on to discuss rigid-body kinetics.

In kinetics, the most important equation that you must consider is Newton's second law:

This chapter will primarily discuss the first type of problem, where you know the force(s) acting on the body, which is more common to in-game physics. The second type of problem has become important with the advent of motion-based controllers such as the Sony SixAxis and Nintendo Wii Remote. These controllers rely on *digital accelerometers* to directly measure the acceleration of a controller. While this is most often used to find the controller's orientation, it is also possible to integrate the time history of these sensor values to determine velocity and position. Additionally, if you know the mass of the controller or device, you can find the force. Accelerometers are found in most smartphones as well, which also allows for the use of kinematic-based input. So as to not confuse the two types of problems, we'll discuss the second type, with the acceleration as input, in detail in [Chapter 21](#).

Let us stress that you must consider the sum of *all* of the forces acting on the body when solving kinetics problems. These include all applied forces and all reaction forces. Aside from the computational difficulties of solving the equations of motion, one of the more challenging aspects of kinetics is identifying and properly accounting for all of these forces. In later chapters, you'll look at specific problems where we'll investigate the particular forces involved. For now, and for the purpose of generality, let's stick with the idealized forces introduced in the previous chapter.

Here is the general procedure for solving kinetics problems of interest to us:

1. Calculate the body's mass properties (mass, center of mass, and moment of inertia).
2. Identify and quantify all forces and moments acting on the body.
3. Take the vector sum of all forces and moments.
4. Solve the equations of motion for linear and angular accelerations.
5. Integrate with respect to time to find linear and angular velocity.
6. Integrate again with respect to time to find linear and angular displacement.

This outline makes the solution to kinetics problems seem easier than it actually is because there are a number of complicating factors that you'll have to overcome. For example, in many cases the forces acting on a body are functions of displacement, velocity, or acceleration. This means that you'll have to use iterative techniques in order to solve the equations of motion. Further, since you most likely will not be able to derive closed-form solutions for acceleration, you'll have to numerically integrate in order to estimate velocity and displacement at each instant of time under consideration. These computational aspects will be addressed further in [Chapter 7](#) through [Chapter 13](#).

Particle Kinetics in 2D

As in particle kinematics, in particle kinetics you need to consider only the linear motion of the particle. Thus, the equations of motion will consist of equations of the form $\mathbf{F} = m\mathbf{a}$, where motion in each coordinate direction will have its own equation. The equations for 2D particle motion are:

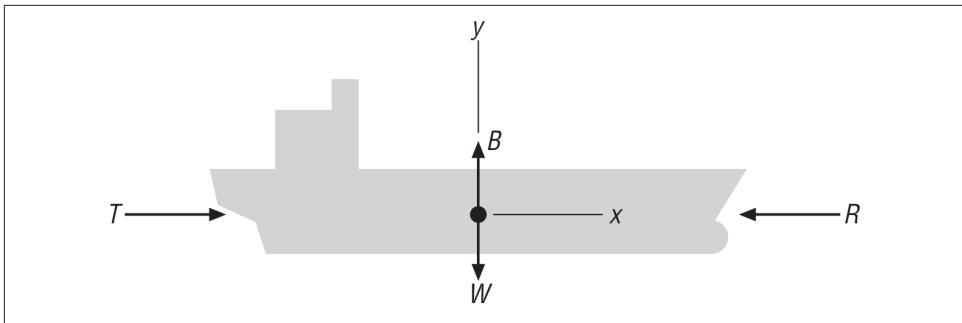


Figure 4-1. Free-body diagram of ship

Notice here that the buoyancy force is exactly equal in magnitude to the ship's weight and opposite in direction; thus, these forces cancel each other out and there will be no motion in the y-direction. This must be the case if the ship is to stay afloat. This observation effectively reduces the problem to a one-dimensional problem with motion in the x-direction, only where the forces acting in the x-direction are the propeller thrust and resistance.

Now you can write the equation (for motion in the x-direction) using Newton's second law, as follows:

- The initial ship speed and displacement are 0 at time 0.
- The propeller thrust is 20,000 thrust units.
- The ship's mass is 10,000 mass units.
- The drag coefficient is 1,000.

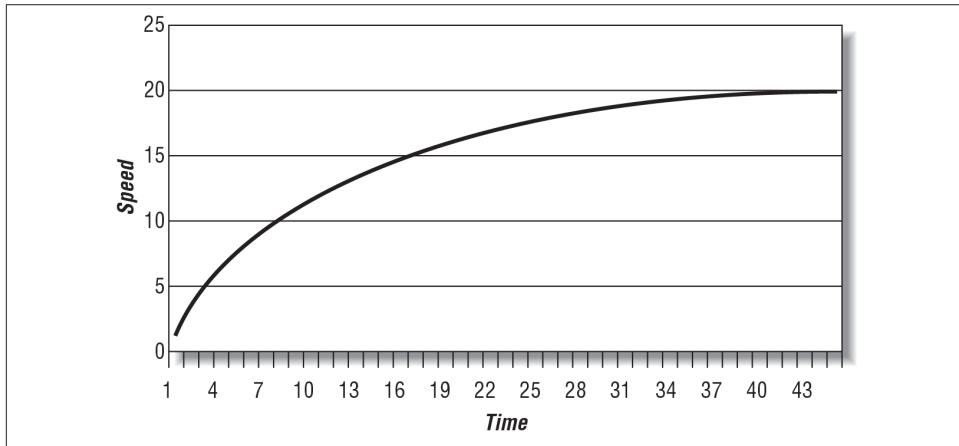


Figure 4-2. Speed versus time

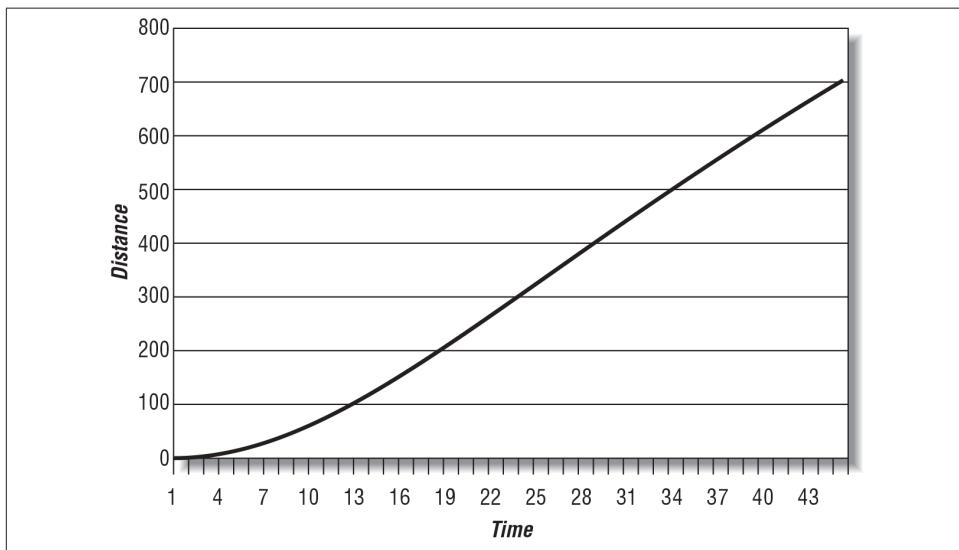


Figure 4-3. Distance versus time

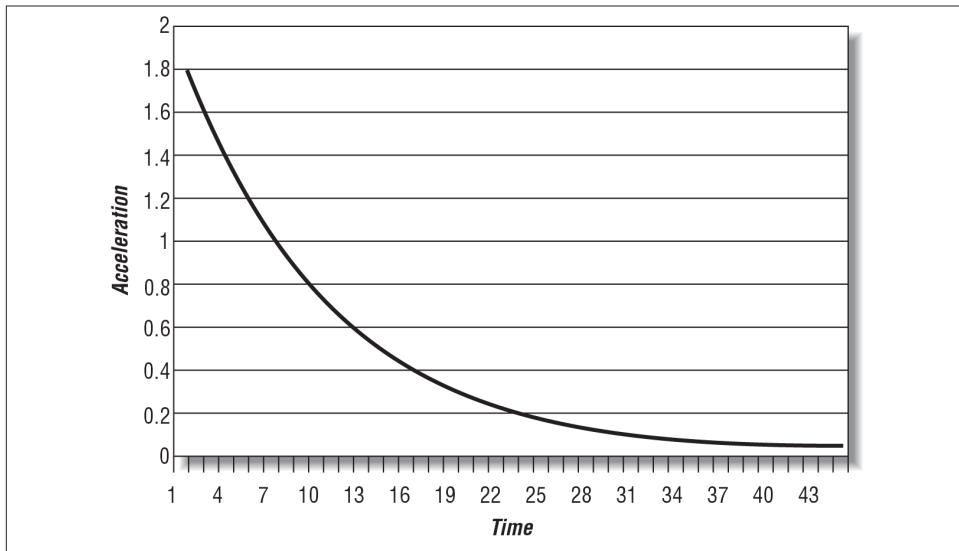


Figure 4-4. Acceleration versus time

You'll notice that the ship's speed approaches the steady state speed of 20 speed units, assuming that the propeller thrust remains constant. This corresponds to a reduction in acceleration from a maximum acceleration at time 0 to no acceleration once the steady speed is achieved.

This example illustrates how to set up the differential equations of motion and integrate them to find velocity, displacement, and acceleration. In this case, you were able to find a closed-form solution—that is, you were able to integrate the equations symbolically to derive new ones. You could do this because we imposed enough constraints on the problem to make it manageable. But you can readily see that if there were more forces acting on the ship, or if the thrust were not held constant but was some function of speed, or if the resistance were a function of speed squared, and so on, the problem gets increasingly complicated—making a closed-form solution much more difficult, if not impossible.

Particle Kinetics in 3D

As in kinematics, extending the equations of motion for a particle to three dimensions is easy to do. You simply need to add one more component and will end up with three equations as follows:

where C_w is the drag coefficient, v_w is the wind speed, and the minus sign means that this force opposes the projectile's motion when the wind is blowing in a direction opposite of the projectile's direction of motion. When the wind is blowing with the projectile—say, from behind it—then the wind will actually help the projectile along instead of impede its motion. In general, C_w is not necessarily equal to the C_d shown in the drag formula. Referring to [Figure 2-3](#), we'll define the wind direction as measured by the angle γ . The x and z components of the wind force can now be written in terms of the wind direction, γ , as follows:

directions will become initial velocities in each direction, and they will be included in the equations of motion once they've been integrated. The initial velocities will show up in the velocity and displacement equations just like they did in the example in [Chapter 2](#). You'll see this in the following sections.

X Components

The first step is to make the appropriate substitutions for the force terms in the equation of motion, and then integrate to find an equation for velocity.

Y Components

For the y components, you need to follow the same procedure shown earlier for the x components, but with the appropriate y -direction forces. Here's what it looks like:

```

//-----//  

int      DoSimulation(void)  

//-----//  

{
    .  

    .  

    .  

    // new local variables:  

    double      sx1, vx1;  

    double      sy1, vy1;  

    double      sz1, vz1;  

    .  

    .  

    .  

    // Now we can calculate the position vector at this time  

    // Old position vector commented out:  

//s.i = Vm * cosX * time + xe;  

//s.j = (Yb + L * cos(Alpha*3.14/180)) + (Vm * cosY * time) -  

    (0.5 * g * time * time);  

//s.k = Vm * cosZ * time + ze;  

    // New position vector calculations:  

sx1 = xe;  

vx1 = Vm * cosX;  

sy1 = Yb + L * cos(Alpha * 3.14/180);  

vy1 = Vm * cosY;  

sz1 = ze;  

vz1 = Vm * cosZ;  

s.i =((m/Cd) * exp(-(Cd * time)/m) * ((-Cw * Vw * cos(GammaW * 3.14/180))/Cd -  

    vx1) - (Cw * Vw * cos(GammaW * 3.14/180) * time) / Cd ) -  

    ( (m/Cd) * ((-Cw * Vw * cos(GammaW * 3.14/180))/Cd - vx1) ) + sx1;  

s.j = sy1 + ( -(vy1 + (m * g)/Cd) * (m/Cd) * exp(-(Cd*time)/m) -  

    (m * g * time) / Cd ) + ( (m/Cd) * (vy1 + (m * g)/Cd) );  

s.k =((m/Cd) * exp(-(Cd * time)/m) * ((-Cw * Vw * sin(GammaW * 3.14/180))/Cd -  

    vz1) - (Cw * Vw * sin(GammaW * 3.14/180) * time) / Cd ) -  

    ( (m/Cd) * ((-Cw * Vw * sin(GammaW * 3.14/180))/Cd - vz1) ) + sz1;  

.  

.  

.  

}

```

To take into account the cross wind and drag, you'll need to add some new global variables to store the wind speed and direction, the mass of the projectile, and the drag

coefficients. You'll also have to add some controls in the dialog window so that you can change these variables when you run the program. [Figure 4-5](#) shows how we added these interface controls in the upper-right corner of the main window.

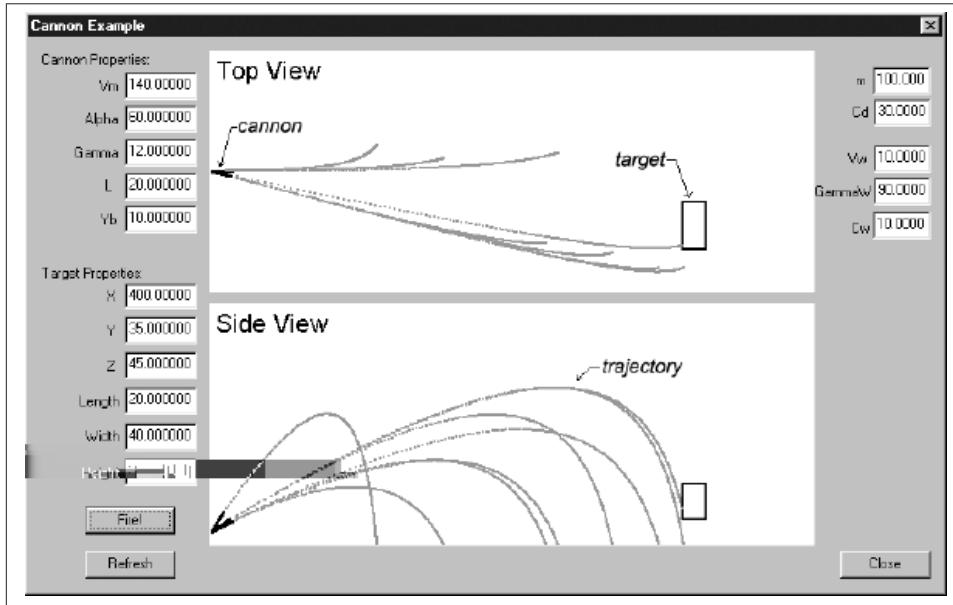


Figure 4-5. Revised cannon example screenshot

We also added these lines to the `DemoDlgProc` function to handle the new wind speed and direction values:

```
//-----//  
LRESULT CALLBACK DemoDlgProc(HWND hDlg, UINT message, WPARAM wParam, LPARAM lParam)  
//-----//  
{  
    .  
    .  
    .  
  
    case WM_INITDIALOG:  
        .  
        .  
        .  
        // New variables:  
        sprintf( str, "%f", m );  
        SetDlgItemText(hDlg, IDC_M, str);  
  
        sprintf( str, "%f", Cd );  
        SetDlgItemText(hDlg, IDC_CD, str);
```

```

        sprintf( str, "%f", Vw );
        SetDlgItemText(hDlg, IDC_VW, str);

        sprintf( str, "%f", GammaW );
        SetDlgItemText(hDlg, IDC_GAMMAW, str);

        sprintf( str, "%f", Cw );
        SetDlgItemText(hDlg, IDC_CW, str);
        .

        .

        case IDC_REFRESH:
        .

        .

        // New variables:
        GetDlgItemText(hDlg, IDC_M, str, 15);
        m = atof(str);

        GetDlgItemText(hDlg, IDC_CD, str, 15);
        Cd = atof(str);

        GetDlgItemText(hDlg, IDC_VW, str, 15);
        Vw = atof(str);

        GetDlgItemText(hDlg, IDC_GAMMAW, str, 15);
        GammaW = atof(str);

        GetDlgItemText(hDlg, IDC_CW, str, 15);
        Cw = atof(str);
        .

        .

        case IDC_FIRE:
        .

        .

        // New variables:
        GetDlgItemText(hDlg, IDC_M, str, 15);
        m = atof(str);

        GetDlgItemText(hDlg, IDC_CD, str, 15);
        Cd = atof(str);

        GetDlgItemText(hDlg, IDC_VW, str, 15);
        Vw = atof(str);

        GetDlgItemText(hDlg, IDC_GAMMAW, str, 15);
        GammaW = atof(str);

```

```

GetDlgItemText(hDlg, IDC_CW, str, 15);
Cw = atof(str);

.

.

}

```

After playing with this example program, you should readily see that the trajectory of the projectile is noticeably different from that typically obtained in the original example. By adjusting the values of the wind speed, direction, and drag coefficients, you can dramatically affect the projectile's trajectory. If you set the wind speed to 0 and the drag coefficients to 1, the trajectory will look like that obtained in the original example, where wind and drag were not taken into account. Be careful, though: don't set the drag coefficient to 0 because this will result in a "divide by zero" error. We didn't put the exception handler in the program, but you can see that it will happen by looking at the displacement vector formulas where the drag coefficient appears in the denominator of several terms.

From a user's perspective, if this were a video game, the problem of hitting the target becomes much more challenging when wind and drag are taken into account. The wind element is particularly interesting because you can change the wind speed and direction during game play, forcing the user to pay careful attention to the wind in order to accurately hit the target.

Rigid-Body Kinetics

You already know from your study of kinematics in [Chapter 2](#) that dealing with rigid bodies adds rotation, or angular motion, into the mix of things to consider. As we stated earlier, the equations of motion now consist of a set of equations that relate forces to linear accelerations and another set of equations that relate moments to angular accelerations. Alternatively, you can think of the equations of motion as relating forces to the rate of change in linear momentum, and moments to the rate of change in angular momentum, as discussed in [Chapter 1](#).

As in kinematics, the procedure for dealing with rigid-body kinetics problems involves two distinct aspects: 1) tracking the translation of the body's center of mass, where the body is treated as a particle, and 2) tracking the body's rotation, where you'll utilize the principles of local coordinates and relative angular velocity and acceleration, as discussed in [Chapter 2](#). Really, the only difference between rigid-body kinematics and kinetics problems is that in kinetics problems we have forces to consider (including their resulting moments).

The vector equations are repeated here for convenience:

where in two dimensions:

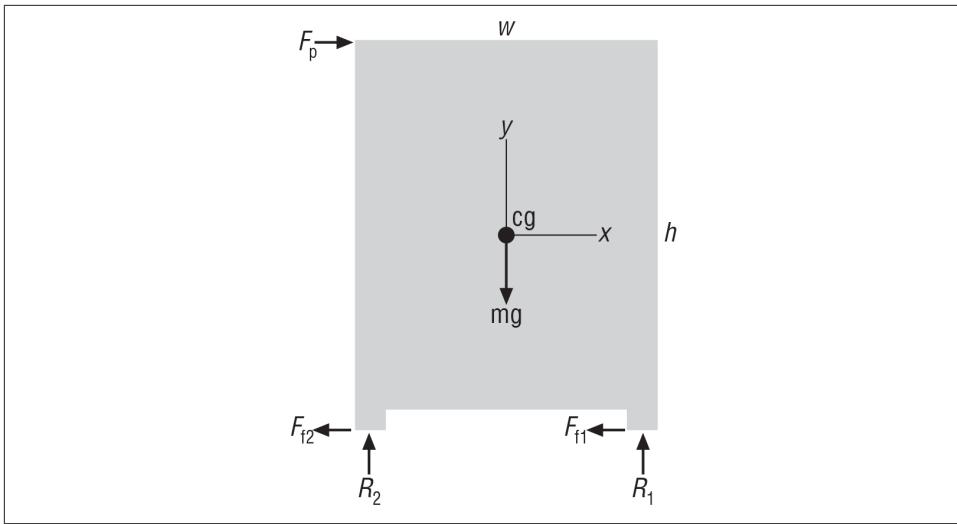


Figure 4-6. Box-free body diagram

In [Figure 4-6](#), F_p is the applied force, R_1 and R_2 are the reaction forces at supports one and two, F_{f1} and F_{f2} are the forces due to friction at points one and two, and mg is the weight of the box.

This is an example of the type of problem where you know something about the motion of the object and have to find the value of one or more forces acting on it. To find the value of the force that will be just enough to start tipping the box, you need to look at the instant when the reaction force at support two is 0. This implies that all of the weight of the box is now supported at point one and the box is starting to rotate over. At this instant, just before it starts to rotate, the angular acceleration of the box is 0. Note that the box's linear acceleration isn't necessarily 0—that is, you can push on the box and it may slide without actually tipping over.

The equations of motion for this problem are:

CHAPTER 5

Collisions

Now that you understand the motion of particles and rigid bodies, you need to consider what happens when they run into each other. That's what we'll address in this chapter; specifically, we'll show you how to handle particle and, more interestingly, rigid-body collision response.

Before moving forward, we need to make a distinction between collision *detection* and collision *response*. Collision detection is a computational geometry problem involving the determination of whether and where two or more objects have collided. Collision response is a kinetics problem involving the motion of two or more objects after they have collided. While the two problems are intimately related, we'll focus solely on the problem of collision response in this chapter. Later, in [Chapter 7](#) through [Chapter 13](#), we'll show you how to implement collision detection and response in various real-time simulations, which draw upon concepts presented in this chapter.

Our treatment of rigid-body collision response in this chapter is based on classical (Newtonian) impact principles. Here, colliding bodies are treated as rigid irrespective of their construction and material. As in earlier chapters, the rigid bodies discussed here do not change shape even upon impact. This, of course, is an idealization. You know from your everyday experience that when objects collide they dent, bend, compress, or crumple. For example, when a baseball strikes a bat, it may compress as much as three-quarters of an inch during the millisecond of impact. Notwithstanding this reality, we'll rely on well-established analytical and empirical methods to approximate rigid-body collisions.

This classical approach is widely used in engineering machine design, analysis, and simulations; however, for rigid-body simulations there is another class of methods,

known as *penalty methods*, at your disposal.¹ In penalty methods, the force at impact is represented by a temporary spring that gets compressed between the objects at the point of impact. This spring compresses over a very short time and applies equal and opposite forces to the colliding bodies to simulate collision response. Proponents of this method say it has the advantage of ease of implementation. However, one of the difficulties encountered in its implementation is numerical instability. There are other arguments for and against the use of penalty methods, but we won't get into the debate here. Instead, we've included several references in the [Bibliography](#) for you to review if you are so inclined. Other methods of modeling collisions exist as well. For example, nonlinear finite element simulations are commonly used to model collisions during product design, such as the impact of a cellphone with the ground. These methods can be quite accurate; however, they are too slow for real-time applications. Further, they are overkill for games.

Impulse-Momentum Principle

Impulse is defined as a force that acts over a very short period of time. For example, the force exerted on a bullet when fired from a gun is an impulse force. The collision forces between two colliding objects are impulse forces, as when you kick a football or hit a baseball with a bat.

More specifically, impulse is a vector quantity equal to the change in momentum. The so-called *impulse-momentum principle* says that the change in moment is equal to the applied impulse. For problems involving constant mass and moment of inertia, you can write:

1. We use the classical approach in this book and are mentioning penalty methods only to let you know that the method we're going to show is not the only one. Roughly speaking, the *penalty* in *penalty methods* refers to the numerical spring constants, which are usually large, that are used to represent the stiffness of the springs and thus the hardness (or softness) of the colliding bodies. These constants are used in the system of equations of motion describing the motion of all the bodies under consideration before and after the collision.

Consider this simple example: a 150 gram (0.15 kg) bullet is fired from a gun at a muzzle velocity of 756 m/s. The bullet takes 0.0008 seconds to travel through the 610 mm (0.610 m) rifle barrel. Calculate the impulse and the average impulsive force exerted on the bullet. In this example, the bullet's mass is a constant 150 grams and its initial velocity is 0, thus its initial momentum is 0. Immediately after the gun is fired, the bullet's momentum is its mass times the muzzle velocity, which yields a momentum of 113.4 kg-m/s. The impulse is equal to the change in momentum, and is simply 113.4 kg-m/s. The average impulse force is equal to the impulse divided by the duration of application of the force, or in this case:

the objects to deform. (See the sidebar “[Kinetic Energy](#)” on page 106 for further details on this topic.) When the deformation in the objects is permanent, energy is lost and thus kinetic energy is not conserved.

Kinetic Energy

Kinetic energy is a form of energy associated with moving bodies. It is equal to the energy required to accelerate the body from rest, which is also equal to the energy required to bring the moving body to a stop. As you might expect, kinetic energy is a function of the body's speed, or velocity, in addition to its mass. The formula for linear kinetic energy is:



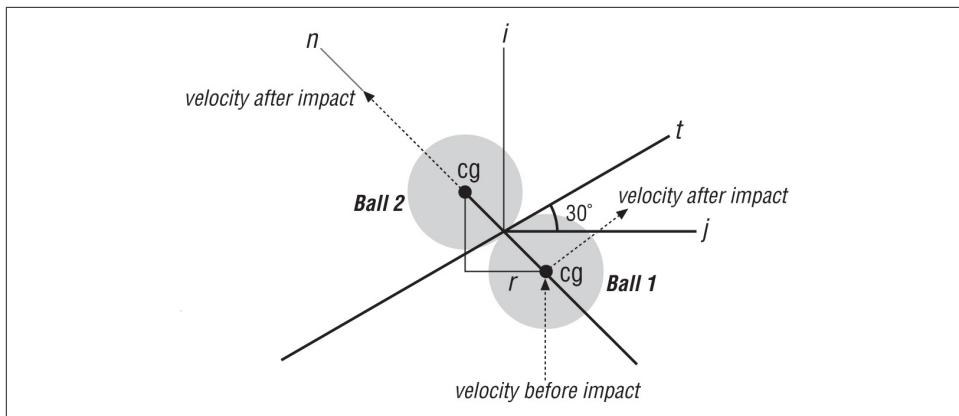


Figure 5-2. Example billiard ball collision

Both balls are a standard 57 mm in diameter, and each weighs 156 grams. Assume that the collision is nearly perfectly elastic and the coefficient of restitution is 0.9. If the velocity of ball 1 when it strikes ball 2 is 6 m/s in the x-direction, as shown in [Figure 5-2](#), calculate the velocities of both balls after the collision assuming that this is a frictionless collision.

The first thing you need to do is recognize that the line of action of impact is along the line connecting the centers of gravity of both balls, which, since these are spheres, is also normal to both surfaces. You can then write the unit normal vector as follows:

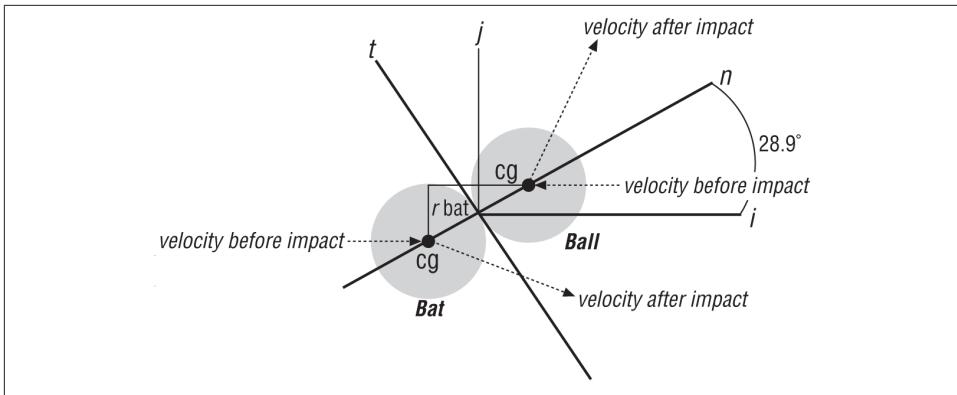


Figure 5-3. Example baseball and bat collision

To a reasonable degree of accuracy, the motion of a baseball bat at the instant of collision can generally be described as independent of the batter—in other words, you can assume that the bat is moving freely and pivoting about a point located near the handle end of the bat. Assume that the ball strikes the bat on the sweet spot—that is, a point near the center of percussion.² Further assume that the bat is swung in the horizontal plane and that the baseball is traveling in the horizontal plane when it strikes the bat. The bat is of standard dimensions with a maximum diameter of 70 mm and a weight of 1.02 kg. The ball is also of standard dimensions with a radius of 37 mm and a weight of 0.15 kg. The ball reaches a speed of 40 m/s (90 mph) at the instant it strikes the bat, and the speed of the bat at the point of impact is 31 m/s (70 mph). For this collision, the coefficient of restitution is 0.46. In the millisecond of impact that occurs, the baseball compresses quite a bit; however, in this analysis assume that both the bat and the ball are rigid. Finally, assume that this impact is frictionless.

As in the previous example, the line of action of impact is along the line connecting the centers of gravity of the bat and ball; thus, the unit normal vector is:

2. The center of percussion is a point located near one of the nodes of natural vibration, and is the point at which, when the bat strikes the ball, no force is transmitted to the handle of the bat. If you've ever hit a baseball incorrectly such that you get a painful vibrating sensation in your hands, then you know what it feels like to miss the center of percussion.

Linear and Angular Impulse

In the previous section, you were able to work through the specific examples by hand using the principle of conservation of momentum and the coefficient of restitution. This approach will suffice if you're writing games where the collision events are well defined and anticipated. However, if you're writing a real-time simulation where objects, especially arbitrarily shaped rigid bodies, may or may not collide, then you'll want to use a more general approach. This approach involves the use of formulas to calculate the actual impulse between colliding objects so that you can apply this impulse to each object, instantly changing its velocity. In this section, we'll derive the equations for impulse, both linear and angular, and we'll show you how to implement these equations in code in [Chapter 10](#).

When you're dealing with particles or spheres, the only impulse formula that you'll need is that for linear impulse, which will allow you to calculate the new linear velocities of the objects after impact. So, the first formula that we'll derive for you is that for linear impulse between two colliding objects, as shown in [Figure 5-4](#).

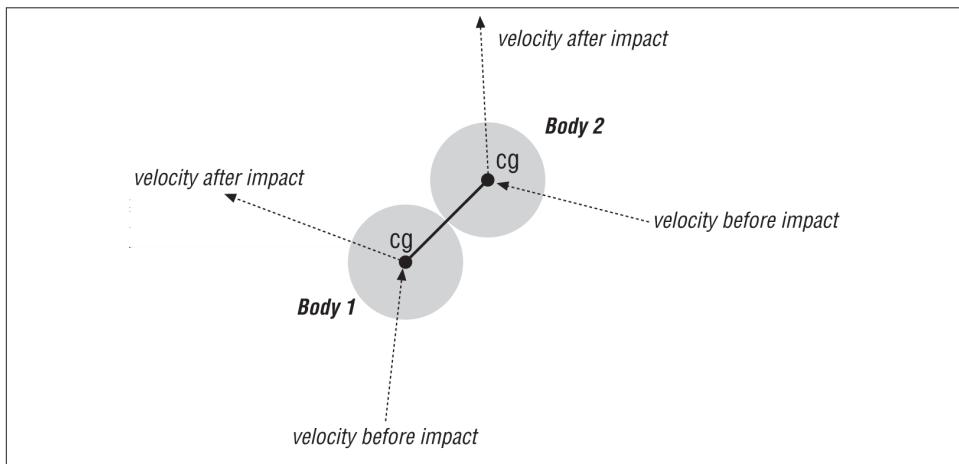


Figure 5-4. Two colliding particles (or spheres)

For now, assume the collision is frictionless and the line of action of the impulse is along the line connecting the centers of mass of the two objects. This line is normal to the surfaces of both objects.

To derive the formula for linear impulse, you have to consider the formula from the definition of impulse along with the formula for coefficient of restitution. Here let J represent the impulse:

Notice that for the second object, the negative of the impulse is applied since it acts on both objects equally but in opposite directions.

When dealing with rigid bodies that rotate, you'll have to derive a new equation for impulse that includes angular effects. You'll use this impulse to calculate new linear and angular velocities of the objects just after impact. Consider the two objects colliding at point P , as shown in [Figure 5-5](#).

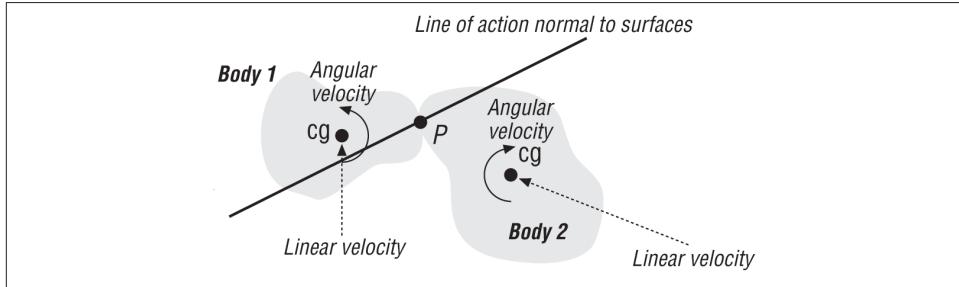


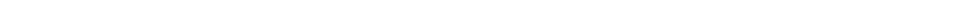
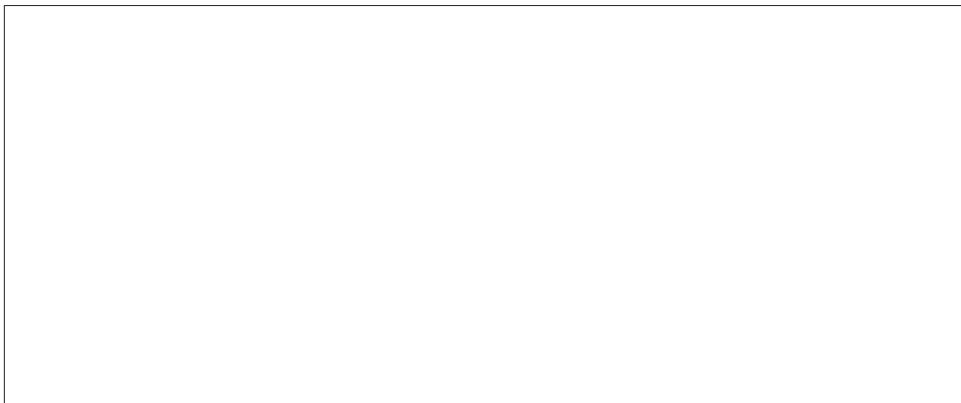
Figure 5-5. Two colliding rigid bodies

There's a crucial distinction between this collision and that discussed earlier. In this case, the velocity at the point of contact on each body is a function of not only the objects' linear velocity but also their angular velocities, and you'll have to recall from [Chapter 2](#) the following formula in order to calculate the velocities at the impact point on each body:

Here we calculate the moment due to the impulse by taking the vector cross product of the impulse with the distance from the body's center of gravity to the point of application of the impulse.

By combining all of these equations with the equation for e and following the same procedure used when deriving the linear impulse formula, you'll end up with a formula for $|J|$ that takes into account both linear and angular effects, which you can then use to find the linear and angular velocities of each body immediately after impact. Here's the result:

friction force is equal to the coefficient of friction. If you assume that the collisions are such that the kinetic coefficient of friction is applicable, then this ratio is constant.



acting on the ball's surface some distance from its center of gravity, it creates a moment (torque) about the ball's center of gravity, which causes the ball to spin. You can develop an equation for the new angular velocity of the ball in terms of the normal impact force or impulse:

CHAPTER 6

Projectiles

This chapter is the first in a series of chapters that discuss specific real-world phenomena and systems, such as projectile motion and airplanes, with the goal of giving you a solid understanding of their real-life behavior. This understanding will help you to model these or similar systems accurately in your games. Instead of relying on purely idealized formulas, we'll present a wide variety of practical formulas and data that you can use. We've chosen the examples in this and later chapters to illustrate common forces and phenomena that exists in many systems, not just the ones we'll be discussing here. For example, while [Chapter 16](#), "Ships and Boats," discusses buoyancy in detail, buoyancy is not limited to ships; any object immersed in a fluid experiences buoyant forces. The same applies for the topics discussed in this chapter and [Chapter 15](#), [Chapter 17](#), [Chapter 18](#), and [Chapter 19](#).

Once you understand what's supposed to happen with these and similar systems, you'll be in a better position to interpret your simulation results to determine if they make sense—that is, if they are realistic enough. You'll also be better educated on what factors are most important for a given system such that you can make appropriate simplifying assumptions to help ease your effort. Basically, when designing and optimizing your code, you'll know where to cut things out without sacrificing realism. This gets into the subject of *parameter tuning*.

Over the next few chapters, we want to give you enough of an understanding of certain physical phenomenon such that you can tune your models for the desired behavior. If you are modeling several similar objects in your simulation but want each one to behave slightly differently, then you have to tune the forces that get applied to each object in order to achieve the varying behavior. Since forces govern the behavior of objects in your simulations, we'll be focusing on force calculations with the intent of showing you how and why certain forces are what they are instead of simply using the idealized formulas discussed in [Chapter 3](#). Parameter tuning isn't just limited to tuning your model's behavior—it also involves dealing with numerical issues, such as numerical

stability in your integration algorithms. We'll discuss these issues more when we show you several simulation examples in [Chapter 7](#) through [Chapter 14](#).

We've devoted this entire chapter to projectile motion because so many physical problems that may find their way into your games fall in this category. Further, the forces governing projectile motion affect many other systems that aren't necessarily projectiles—for example, the drag force experienced by projectiles is similar to that experienced by airplanes, cars, or any other object moving through a fluid such as air or water.

A projectile is an object that is placed in motion by a force acting over a very short period of time, which, as you know from [Chapter 5](#), is also called an impulse. After the projectile is set in motion by the initial impulse during the launching phase, the projectile enters into the projectile motion phase, where there is no longer a thrust or propulsive force acting on it. As you know already from the examples presented in [Chapter 2](#) and [Chapter 4](#), there are other forces that act on projectiles. (For the moment, we're not talking about self-propelled "projectiles" such as rockets since, due to their propulsive force, they don't follow "classical" projectile motion until after they've expended their fuel.)

In the simplest case, neglecting aerodynamic effects, the only force acting on a projectile other than the initial impulsive force is gravitation. For situations where the projectile is near the earth's surface, the problem reduces to a constant acceleration problem. Assuming that the earth's surface is flat—that is, that its curvature is large compared to the range of the projectile—the following statements describe projectile motion:

- The trajectory is parabolic.
- The maximum range, for a given launch velocity, occurs when the launch angle is 45° .
- The velocity at impact is equal to the launch velocity when the launch point and impact point are at the same level.
- The vertical component of velocity is 0 at the apex of the trajectory.
- The time required to reach the apex is equal to the time required to descend from the apex to the point of impact assuming that the launch point and impact point are at the same level.
- The time required to descend from the apex to the point of impact equals the time required for an object to fall the same vertical distance when dropped straight down from a height equal to the height of the apex.

Simple Trajectories

There are four simple classes of projectile motion problems that we'll summarize:

- When the target and launch point are at the same level

- When the target is at a level higher than the launch point
- When the target is at a level lower than the launch point
- When the projectile is dropped from a moving system (like an airplane) above the target

In the first type of problem, the launch point and the target point are located on the same horizontal plane. In [Figure 6-1](#), v_0 is the initial velocity of the projectile at the time of launch, φ is the launch angle, R is the range of the projectile, and h is the height of the apex of the trajectory.

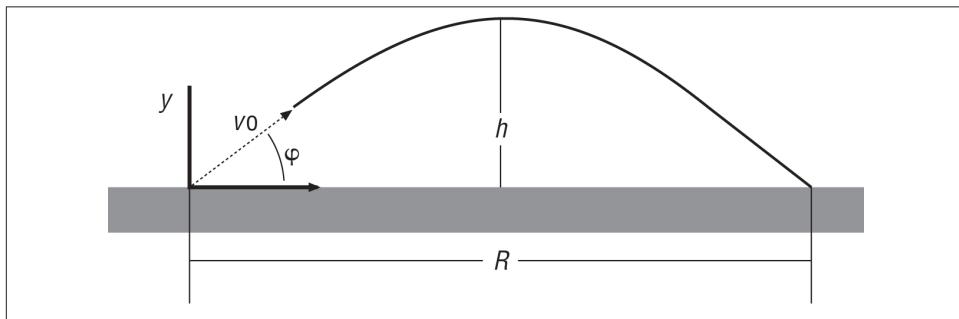


Figure 6-1. Target and launch point at same level

To solve this type of problem, use the formulas shown in [Table 6-1](#). Note, in these formulas t represents any time instant after launch, and T represents the total time from launch to impact.

Table 6-1. Formulas—target and launch point at same level

To calculate:	Use this formula:
$x(t)$	$(v_0 \cos \varphi) t$
$y(t)$	$(v_0 \sin \varphi) t - (g t^2) / 2$
$v_x(t)$	$v_0 \cos \varphi$
$v_y(t)$	$v_0 \sin \varphi - g t$
$v(t)$	$\sqrt{v_0^2 - 2gtv_0 \sin \varphi + g^2t^2}$
h	$(v_0^2 \sin^2 \varphi) / (2 g)$
R	$v_0 T \cos \varphi$
T	$(2 v_0 \sin \varphi) / g$

Remember to keep your units consistent when applying these formulas. If you are working in the SI (metric) system, length and distance values should be in meters (m);

time should be in seconds (s); speed should be in meters per second (m/s); and acceleration should be in meters per second squared (m/s²). In the SI system, g is 9.8 m/s².

In the second type of problem, the launch point is located on a lower horizontal plane than the target. In [Figure 6-2](#), the launch point's y coordinate is lower than the target's y coordinate.

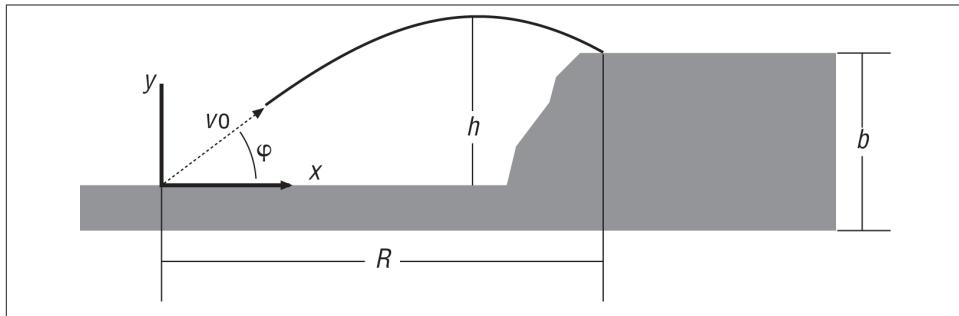


Figure 6-2. Target higher than launch point

For this type of problem, use the formulas shown in [Table 6-2](#). Notice that most of these formulas are the same as those shown in [Table 6-1](#).

Table 6-2. Formulas—target higher than launch point

To calculate: Use this formula:

(t)	$(v_0 \cos \varphi) t$
$y(t)$	$(v_0 \sin \varphi) t - (g t^2) / 2$
$v_x(t)$	$v_0 \cos \varphi$
$v_y(t)$	$v_0 \sin \varphi - g t$
$v(t)$	$\sqrt{v_0^2 - 2gtv_0 \sin \varphi + g^2t^2}$
h	$(v_0^2 \sin^2 \varphi) / (2 g)$
R	$v_0 T \cos \varphi$
T	$(v_0 \sin \varphi) / g + \sqrt{\frac{(2(h - b))}{g}}$

Actually, the only formula that has changed is that for T , where it has been revised to account for the difference in elevation between the target and the launch point.

In the third type of problem, the target is located on a plane lower than the launch point; in [Figure 6-3](#), the target's y coordinate is lower than the launch point's y coordinate.

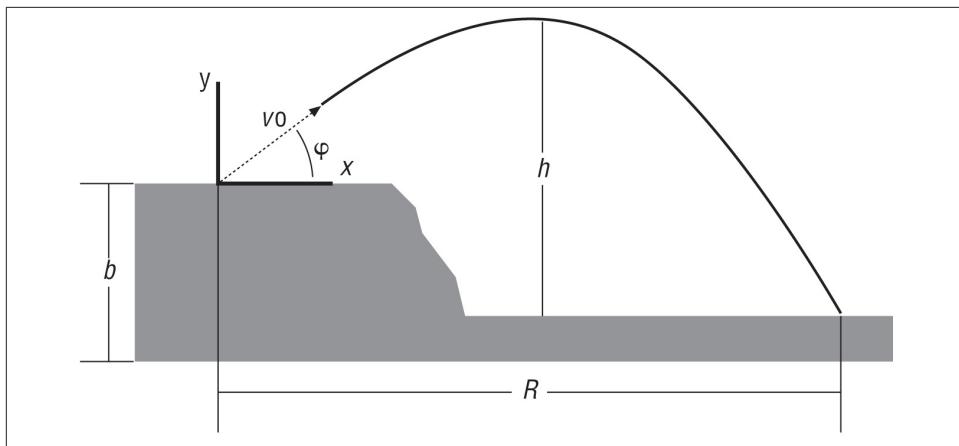


Figure 6-3. Target lower than launch point

Table 6-3 shows the formulas to use for this type of problem. Here again, almost all of the formulas are the same as those shown in [Table 6-1](#).

Table 6-3. Formulas—target lower than launch point

To calculate:	Use this formula:
$x(t)$	$(v_0 \cos \varphi) t$
$y(t)$	$(v_0 \sin \varphi) t - (g t^2) / 2$
$v_x(t)$	$v_0 \cos \varphi$
$v_y(t)$	$v_0 \sin \varphi - g t$
$v(t)$	$\sqrt{v_0^2 - 2gtv_0 \sin \varphi + g^2t^2}$
h	$b + (v_0^2 \sin^2 \varphi) / (2 g)$
R	$v_0 T \cos \varphi$
T	$(v_0 \sin \varphi) / g + \sqrt{\frac{2b}{g}}$

The only formulas that have changed are the formulas for h and T , which have been revised to account for the difference in elevation between the target and the launch point (except this time the target is lower than the launch point).

Finally, the fourth type of problem involves dropping the projectile from a moving system, such as an airplane. In this case, the initial velocity of the projectile is horizontal and equal to the speed of the vehicle dropping it. [Figure 6-4](#) illustrates this type of problem.

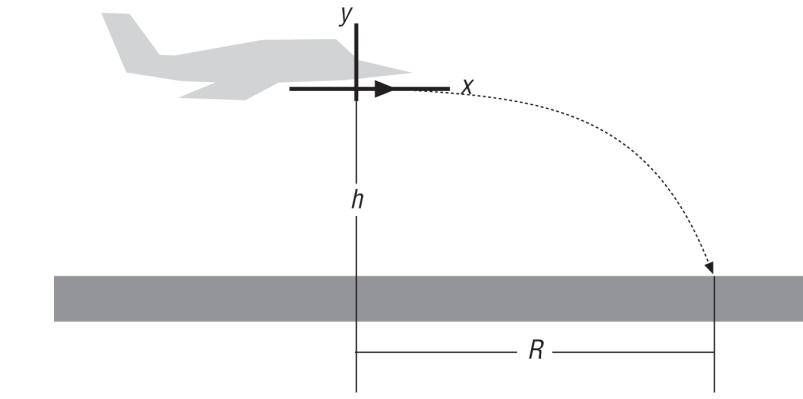


Figure 6-4. Projectile dropped from a moving system

Table 6-4 shows the formulas to use to solve this type of problem. Note here that when v_o is 0, the problem reduces to a simple free-fall problem in which the projectile drops straight down.

Table 6-4. Formulas—projectile dropped from a moving system

To calculate:	Use this formula:
$x(t)$	$v_o t$
$y(t)$	$h - (g t^2) / 2$
$v_x(t)$	v_o
$v_y(t)$	$-g t$
$v(t)$	$\sqrt{v_o^2 + g^2 t^2}$
h	$(g t^2) / 2$
R	$v_o T$
T	$\sqrt{\frac{2h}{g}}$

These formulas are useful if you’re writing a game that does not require a more accurate treatment of projectile motion—that is, if you don’t need or want to consider the other forces that can act on a projectile when in motion. If you are going for more accuracy, you’ll have to consider these other forces and treat the problem as we did in [Chapter 4](#)’s example.

Drag

In [Chapter 3](#) and [Chapter 4](#), we showed you the idealized formulas for viscous fluid dynamic drag as well as how to implement drag in the equations of motion for a pro-

jective. This was illustrated in the example program discussed in [Chapter 4](#). Recall that the drag force is a vector just like any other force and that it acts on the line of action of the velocity vector but in a direction opposing velocity. While those formulas work in a game simulation, as we said before, they don't tell the whole story. While we can't treat the subject of fluid dynamics in its entirety in this book, we do want to give you a better understanding of drag than just the simple idealized equation presented earlier.

Analytical methods can show that the drag on an object moving through a fluid is proportional to its speed, size, shape, and the density and viscosity of the fluid through which it is moving. You can also come to these conclusions by drawing on your own real-life experience. For example, when waving your hand through the air, you feel very little resistance; however, if you put your hand out of a car window traveling at 100 km/h, then you feel much greater resistance (drag force) on your hand. This is because drag is speed dependent. When you wave your hand underwater—say, in a swimming pool—you'll feel a greater drag force on your hand than you do when waving it in the air. This is because water is more dense and viscous than air. As you wave your hand underwater, you'll notice a significant difference in drag depending on the orientation of your hand. If your palm is in line with the direction of motion—that is, you are leading with your palm—then you'll feel a greater drag force than you would if your hand were turned 90 degrees as though you were executing a karate chop through the water. This tells you that drag is a function of the shape of the object. You get the idea.

To facilitate our discussion of fluid dynamic drag, let's look at the flow around a sphere moving through a fluid such as air or water. If the sphere is moving slowly through the fluid, the flow pattern around the sphere would look something like [Figure 6-5](#).

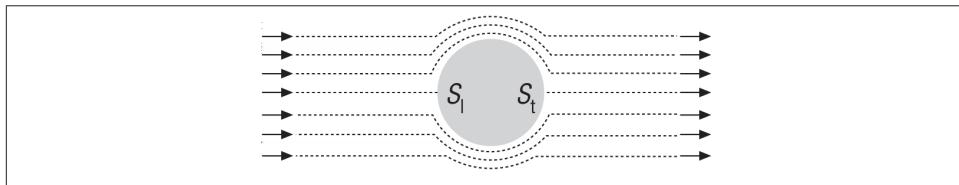


Figure 6-5. Flow pattern around slowly moving sphere

Bernoulli's equation, which relates pressure to velocity in fluid flow, says that as the fluid moves around the sphere and speeds up, the pressure in the fluid (locally) will go down. The equation, presented by Daniel Bernoulli in 1738, applies to frictionless incompressible fluid flow and looks like this:¹

1. In a real fluid with friction, this equation will have extra terms that account for energy losses due to friction.

where P is the pressure at a point in the fluid volume under consideration, γ is the specific weight of the fluid, z is the elevation of the point under consideration, V is the fluid velocity at that point, and g is the acceleration due to gravity. As you can see, if the expression on the left is to remain constant, and assuming that z is constant, then if velocity increases the pressure must decrease. Likewise, if pressure increases, then velocity must decrease.

As you can see in [Figure 6-5](#), the pressure will be greatest at the stagnation point, S_l , and will decrease around the leading side of the sphere and then start to increase again around the back of the sphere. In an ideal fluid with no friction, the pressure is fully recovered behind the sphere and there is a trailing stagnation point, S_t , whose pressure is equal to the pressure at the leading stagnation point. Since the pressure fore and aft of the sphere is equal and opposite, there is no net drag force acting on the sphere.

The pressure on the top and bottom of the sphere will be lower than at the stagnation points since the fluid velocity is greater over the top and bottom. Since this is a case of symmetric flow around the sphere, there will be no net pressure difference between the top and bottom of the sphere.

In a real fluid there is friction, which affects the flow around the sphere such that the pressure is never fully recovered on the aft side of the sphere. As the fluid flows around the sphere, a thin layer sticks to the surface of the sphere due to friction. In this *boundary layer*, the speed of the fluid varies from 0 at the sphere surface to the ideal free stream velocity, as illustrated in [Figure 6-6](#).

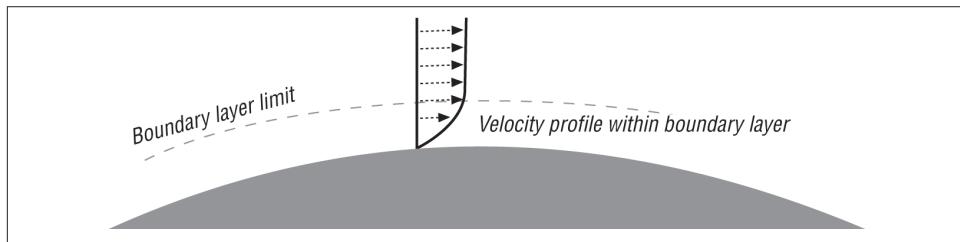


Figure 6-6. Velocity gradient within boundary layer

This velocity gradient represents a momentum transfer from the sphere to the fluid and gives rise to the frictional component of drag. Since a certain amount of fluid is *sticking* to the sphere, you can think of this as the energy required to accelerate the fluid and move it along with the sphere. (If the flow within this boundary layer is laminar, then the viscous shear stress between fluid “layers” gives rise to friction drag. When the flow is turbulent, the velocity gradient and thus the transfer of momentum gives rise to friction drag.)

Moving further aft along the sphere, the boundary layer grows in thickness and will not be able to maintain its adherence to the sphere surface, and it will separate at some point. Beyond this *separation point*, the flow will be turbulent, and this is called the *turbulent wake*. In this region, the fluid pressure is lower than that at the front of the sphere. This pressure differential gives rise to the pressure component of drag. [Figure 6-7](#) shows how the flow might look.

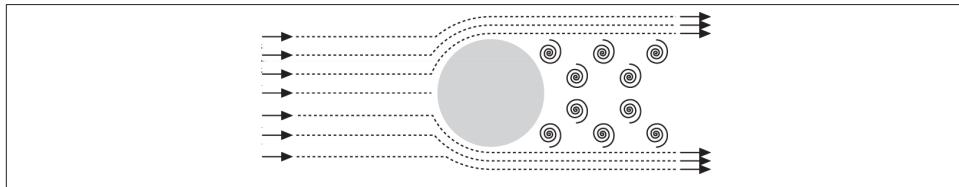


Figure 6-7. Flow pattern around sphere showing separation

For a slowly moving sphere, the separation point will be approximately 80° from the leading edge.

Now, if you were to roughen the surface of the sphere, you'll affect the flow around it. As you would expect, this roughened sphere will have a higher friction drag component. However, more importantly, the flow will adhere to the sphere longer and the separation point will be pushed further back to approximately 115° , as shown in [Figure 6-8](#).

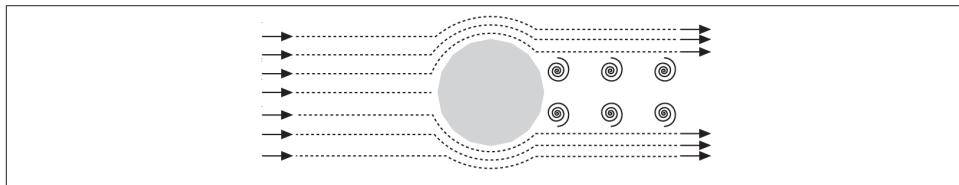


Figure 6-8. Flow around a roughened sphere

This will reduce the size of the turbulent wake and the pressure differential, thus decreasing the pressure drag. It's paradoxical but true that, all other things being equal, a slightly roughened sphere will have less *total* drag than a smooth one. Ever wonder why golf balls have dimples? If so, there's your answer.

The total drag on the sphere depends very much on the nature of the flow around the sphere—that is, whether the flow is laminar or turbulent. This is best illustrated by looking at some experimental data. [Figure 6-9](#) shows a typical curve of the total drag coefficient for a sphere plotted as a function of the *Reynolds number*.

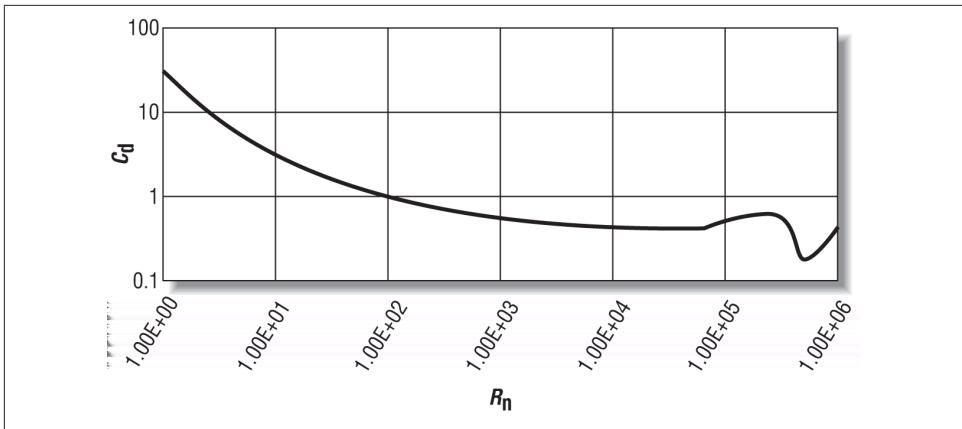


Figure 6-9. Total drag coefficient for a smooth sphere versus Reynolds number²

The Reynolds number (commonly denoted N_r , or R_n) is a dimensionless number that represents the speed of fluid flow around an object. It's a little more than just a speed measure, since it includes a characteristic length for the object and the viscosity and density of the fluid. The formula for the Reynolds number is:

-
2. The curve shown here is intended to demonstrate the trend of C_d versus R_n for a smooth sphere. For more accurate drag coefficient data for spheres and other shapes, refer to any college-level fluid mechanics text, such as Robert L. Daugherty, Joseph B. Franzini, and E. John Finnemore's *Fluid Mechanics with Engineering Applications* (McGraw-Hill).

diameter. A more useful application of this scaling technique is estimating the viscous drag on ship or airplane appendages based on model test data obtained from wind tunnel or tow tank experiments.

The Reynolds number is used as an indicator of the nature of fluid flow. A low Reynolds number generally indicates laminar flow, while a high Reynolds number generally indicates turbulent flow. Somewhere in between, there is a transition range where the flow makes the transition from laminar to turbulent flow. For carefully controlled experiments, this transition (*critical*) Reynolds number can consistently be determined. However, in general the ambient flow field around an object—that is, whether it has low or high turbulence—will affect when this transition occurs. Further, the transition Reynolds number is specific to the type of problem being investigated (for example, whether you're looking at flow within pipes, the flow around a ship, or the flow around an airplane, etc.).

We calculate the total drag coefficient, C_d , by measuring the total resistance, R_b , from tests and using the following formula:

reduction in drag. This is a result of the flow becoming fully turbulent with a corresponding reduction in pressure drag.

In the Cannon2 example in [Chapter 4](#), we implemented the ideal formula for air drag on the projectile. In that case we used a constant value of drag coefficient that was arbitrarily defined. As we said earlier, it would be better to use the formula presented in this chapter for total drag along with the total drag coefficient data shown in [Figure 6-9](#) to estimate the drag on the projectile. While this is more “accurate,” it does complicate matters for you. Specifically, the drag coefficient is now a function of the Reynolds number, which is a function of velocity. You’ll have to set up a table of drag coefficients versus the Reynolds number and interpolate this table given the Reynolds number calculated at each time step. As an alternative, you can fit the drag coefficient data to a curve to derive a formula that you can use instead; however, the drag coefficient data may be such that you’ll have to use a piecewise approach and derive curve fits for each segment of the drag coefficient curve. The sphere data presented herein is one such case. The data does not lend itself nicely to a single polynomial curve fit over the full range of the Reynolds number. In such cases, you’ll end up with a handful of formulas for drag coefficient, with each formula valid over a limited range of Reynolds numbers.

While the Cannon2 example does have its limitations, it is useful to see the effects of drag on the trajectory of the projectile. The obvious effect is that the trajectory is no longer parabolic. You can see in [Figure 6-10](#) that the trajectory appears to drop off much more sharply when the projectile is making its descent after reaching its apex height.

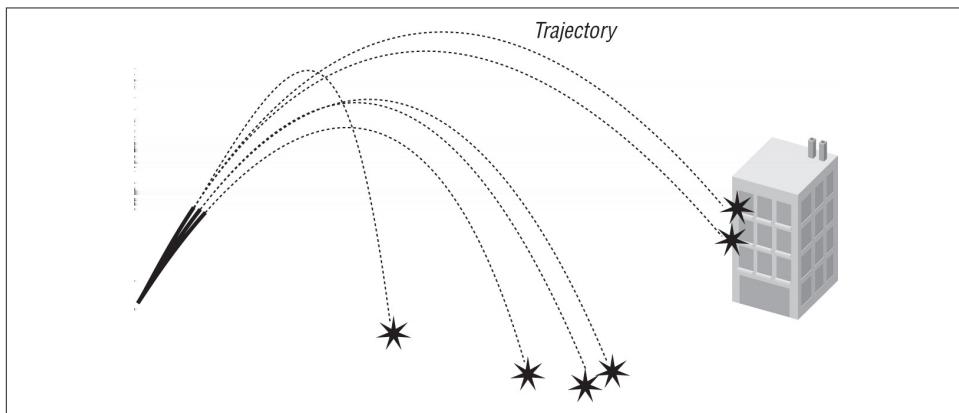
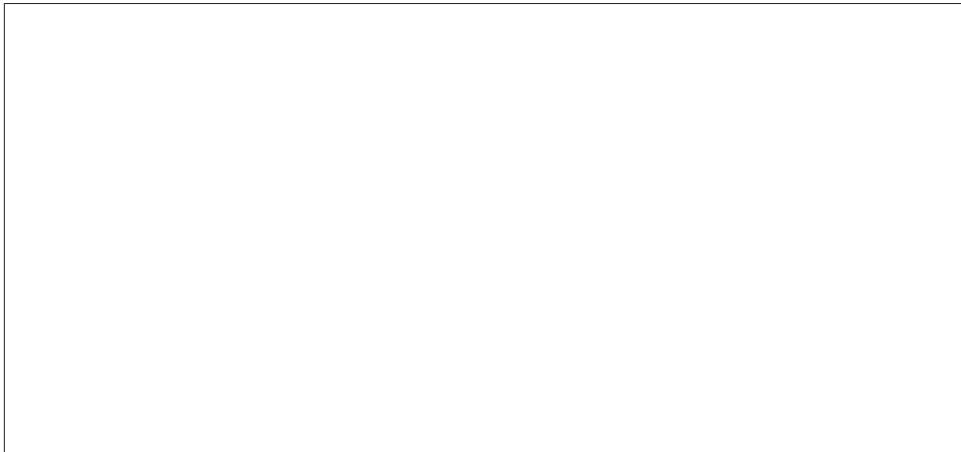


Figure 6-10. Cannon2 example, trajectories

Another important effect of drag on trajectory (this applies to objects in free fall as well) is the fact that drag will limit the maximum vertical velocity attainable. This limit is the so-called *terminal velocity*. Consider an object in free fall for a moment. As the object accelerates toward the earth at the gravitation acceleration, its velocity increases. As

velocity increases, so does drag since drag is a function of velocity. At some speed the drag force retarding the object's motion will increase to a point where it is equal to the gravitational force that's pulling the object toward the earth. In the absence of any other forces that may affect motion, the net acceleration on the object is 0, and it continues its descent at the constant terminal velocity.

Let us illustrate this further. Go back to the formula we derived for the y component (vertical component) of velocity for the projectile modeled in the `Cannon2` example. Here it is again so you don't have to flip back to [Chapter 4](#):



The trick in applying this formula is in determining the right value for the drag coefficient. Just for fun, let's assume a drag coefficient of 0.5 and calculate the terminal velocity for several different objects. This exercise will allow you to see the influence of the object's size on terminal velocity. [Table 6-5](#) gives the terminal velocities for various objects in free fall using an air density of 1.225 kg/m^3 (air at standard atmospheric pressure at 15°C). Using this equation with density in kg/m^3 means that m must be in kg , g in m/s^2 , and A in m^2 in order to get the terminal speed in m/s . We went ahead and converted from m/s to kilometers per hour (km/h) to present the results in [Table 6-5](#). The weight of each object shown in this table is simply its mass, m , times g .

Table 6-5. Terminal velocities for various objects

Object	Weight (N)	Area (m^2)	Terminal velocity (km/h)
Skydiver in free fall	801	0.84	201
Skydiver with open parachute	801	21.02	40
Baseball (2.88 in diameter)	1.42	4.19×10^{-3}	121
Golf ball (1.65 in diameter)	0.5	1.40×10^{-3}	116
Raindrop (0.16 in diameter)	3.34×10^{-4}	1.29×10^{-5}	32

Although we've talked mostly about spheres in this section, the discussions on fluid flow generally apply to any object moving through a fluid. Of course, the more complex the object's geometry, the harder it is to analyze the drag forces on it. Other factors such as surface condition, and whether or not the object is at the interface between two fluids (such as a ship in the ocean) further complicate the analysis. In practice, scale model tests are particularly useful. In the [Bibliography](#), we give several sources where you can find more practical drag data for objects other than spheres.

Magnus Effect

The *Magnus effect* (also known as the *Robbins effect*) is quite an interesting phenomenon. You know from the previous section that an object moving through a fluid encounters drag. What would happen if that object were now spinning as it moved through the fluid? For example, consider the sphere that we talked about earlier and assume that while moving through a fluid such as air or water, it spins about an axis passing through its center of mass. What happens when the sphere spins is the interesting part—it actually generates lift! That's right, *lift*. From everyday experience, most people usually associate lift with a wing-like shape such as an airplane wing or a hydrofoil. It is far less well known that cylinders and spheres can produce lift as well—that is, as long as they are spinning. We'll use the moving sphere to explain what's happening here.

From the previous section on drag, you know that for a fast-moving sphere there will be some point on the sphere where the flow separates, creating a turbulent wake behind the sphere. Recall that the pressure acting on the sphere within this turbulent wake is

lower than the pressure acting on the leading surface of the sphere, and this pressure differential gives rise to the pressure drag component. When the sphere is spinning—say, clockwise—about a horizontal axis passing through its center, as shown in [Figure 6-12](#), the fluid passing over the top of the sphere will be sped up while the fluid passing under the sphere will be retarded.

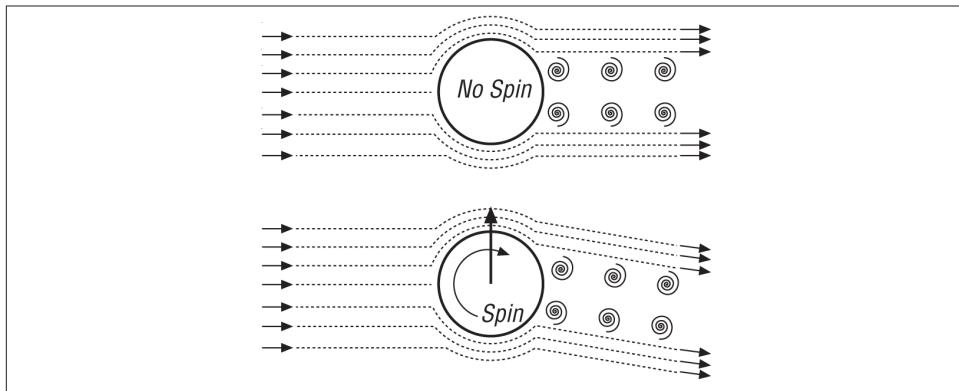


Figure 6-12. Spinning sphere

Remember, because of friction, there is a thin boundary layer of fluid that attaches to the sphere's surface. At the sphere's surface, the velocity of the fluid in the boundary layer is 0 relative to the sphere. The velocity increases within the boundary layer as you move further away from the sphere's surface. In the case of the spinning sphere, there is now a difference in fluid pressure above and below the sphere due to the increase in velocity above the sphere and the decrease in velocity below the sphere. Further, the separation point on the top side of the sphere will be pushed further back along the sphere. The end result is an asymmetric flow pattern around the sphere with a net lift force (due to the pressure differential) perpendicular to the direction of flow. If the surface of the sphere is roughened a little, not only will frictional drag increase, but this lift effect will increase as well.

Don't let the term *lift* confuse you into thinking that this force always acts to lift, or elevate, the sphere. The effect of this lift force on the sphere's trajectory is very much tied to the axis of rotation about which the sphere is spinning as related to the direction in which the sphere is traveling (that is, its angular velocity).

The magnitude of the Magnus force is proportional to the speed of travel, rate of spin, density of fluid, size of the object, and nature of the fluid flow. This force is not easy to calculate analytically, and as with many problems in fluid dynamics, you must rely on experimental data to accurately estimate it for a specific object under specific conditions. There are, however, some analytical techniques that will allow you to approximate the Magnus force. Without going into the theoretical details, you can apply the *Kutta-*

Joukouski therorem to estimate the lift force on rotating objects such as cylinders and spheres. The Kutta-Joukouski theorem is based on a frictionless idealization of fluid flow involving the concept of circulation around the object (like a vortex around the object). You can find the details of this theory in any fluid dynamics text (we give some references in the [Bibliography](#)), so we won't go into the details here. However, we will give you some results.

For a spinning circular cylinder moving through a fluid, you can use this formula to estimate the Magnus lift force:

velocity, and spin rate. Further, experiments show that the drag coefficient is also affected by spin.

For example, consider a golf ball struck perfectly (right!) such that the ball spins about a horizontal axis perpendicular to its direction of travel while in flight. In this case the Magnus force will tend to lift the ball higher in the air, increasing its flight time and range. For a golf ball struck such that its initial velocity is 58 m/s with a take-off angle of 10 degrees, the increase in range due to Magnus lift is on the order of 59 meters; thus, it's clear that this effect is significant. In fact, over the long history of the game of golf, people have attempted to maximize this effect. In the late 1800s, when golf balls were still made with smooth surfaces, players observed that used balls with roughened surfaces flew even better than smooth balls. This observation prompted manufacturers to start making balls with rough surfaces so as to maximize the Magnus lift effect. The dimples that you see on modern golf balls are the result of many decades of experience and research and are thought to be optimum.

Typically a golf ball takes off from the club with an initial velocity on the order of 76 m/s, with a backspin on the order of 60 revolutions per second (rps). For these initial conditions, the corresponding Magnus lift coefficient is within the range of 0.1 to 0.35. Depending on the spin rate, this lift coefficient can be as high as 0.45, and the lift force acting on the ball can be as much as 50% of the ball's weight.

If the golf ball is struck with a less-than-perfect stroke (that's more like it), the Magnus lift force may work against you. For example, if your swing is such that the ball leaves the club head spinning about an axis that is not horizontal, then the ball's trajectory will curve, resulting in a slice or a draw. If you top the ball such that the upper surface of the ball is spinning away from you, then the ball will tend to curve downward much more rapidly, significantly reducing the range of your shot.

As another example, consider a baseball pitched such that it's spinning with topspin about a horizontal axis perpendicular to its direction of travel. Here the Magnus force will tend to cause the ball to curve in a downward direction, making it drop more rapidly than it otherwise would without spin. If the pitcher spins the ball such that the axis of rotation is not horizontal, then the ball will curve out of the vertical plane. Another trick that pitchers use is to give the ball backspin, making it appear (to the batter) to actually rise. This rising fastball does not *actually* rise, but because of the Magnus lift force it falls much less rapidly than it would without spin.

For a typical pitched speed and spin rate of 45 m/s and 30 rps, respectively, the lift force can be up to 33% of the ball's weight. For a typical curveball, the lift coefficient is within the range of 0.1 to 0.2, and for flyballs it can be up to 0.4.

These are only two examples; however, you need not look far to find other examples of the Magnus force in action. Think about the behavior of cricket balls, soccer balls, tennis balls, or ping-pong balls when they spin in flight. Bullets fired from a gun with a rifling

barrel also spin and are affected by this Magnus force. There have even been sailboats built with tall, vertical, rotating cylindrical “sails” that use the Magnus force for propulsion. We’ve also seen technical articles describing a propeller with spinning cylindrical blades instead of airfoil-type blades.

To further illustrate the Magnus effect, we’ve prepared a simple example program that simulates a ball being thrown with varying amounts of backspin (or topspin). This example is based on the cannon example, so here again, the code should look familiar to you. In this example we’ve neglected drag, so the only forces that the ball will see are due to gravity and the Magnus effect. We did this to isolate the lift-generating effect of spin and to keep the equations of motion clearer.

Since most of the code for this example is identical, or very similar, to the previous cannon examples, we won’t repeat it here. We will, however, show you the global variables used in this simulation along with a revised `DoSimulation` function that takes care of the equations of motion:

```
//-----//  
// Global variables required for this simulation  
//-----//  
TVector      V1;      // Initial velocity (given), m/s  
TVector      V2;      // Velocity vector at time t, m/s  
double       m;       // Projectile mass (given), kg  
TVector      s1;      // Initial position (given), m  
TVector      s2;      // The projectile's position (displacement) vector, m  
double       time;    // The time from the instant the projectile  
                      // is launched, s  
double       tInc;    // The time increment to use when stepping  
                      // through the simulation, s  
double       g;       // acceleration due to gravity (given), m/s^2  
double       spin;    // spin in rpm (given)  
double       omega;   // spin in radians per second  
double       radius;  // radius of projectile (given), m  
  
#define      PI      3.14159f  
#define      RHO     1.225f      // kg/m^3  
  
//-----//  
int      DoSimulation(void)  
//-----//  
{  
    double      C = PI * PI * RHO * radius * radius * radius * omega;  
    double      t;  
  
    // step to the next time in the simulation  
    time+=tInc;  
    t = time;  
  
    // Calc. V2:  
    V2.i = 1.0f/(1.0f-(t/m)*(t/m)*C*C) * (V1.i + C * V1.j * (t/m) -
```

```

        C * g * (t*t)/m);
V2.j = V1.j + (t/m)*C*V2.i - g*t;

// Calc. S2:
s2.i = s1.i + V1.i * t + (1.0f/2.0f) * (C/m * V2.j) * (t*t);
s2.j = s1.j + V1.j * t + (1.0f/2.0f) * ( ((C*V2.i) - m*g)/m ) * (t*t);

// Check for collision with ground (x-z plane)
if(s2.j <= 0)
    return 2;

// Cut off the simulation if it's taking too long
// This is so the program does not get stuck in the while loop
if(time>60)
    return 3;

return 0;
}

```

The heart of this simulation is the lines that calculate `V2` and `s2`, the instantaneous velocity and position of the projectile, respectively. The equations of motion here come from the 2D kinetic equations of motion including gravity, as discussed in [Chapter 4](#), combined with the following formula (shown earlier) for estimating the Magnus lift on a spinning sphere:

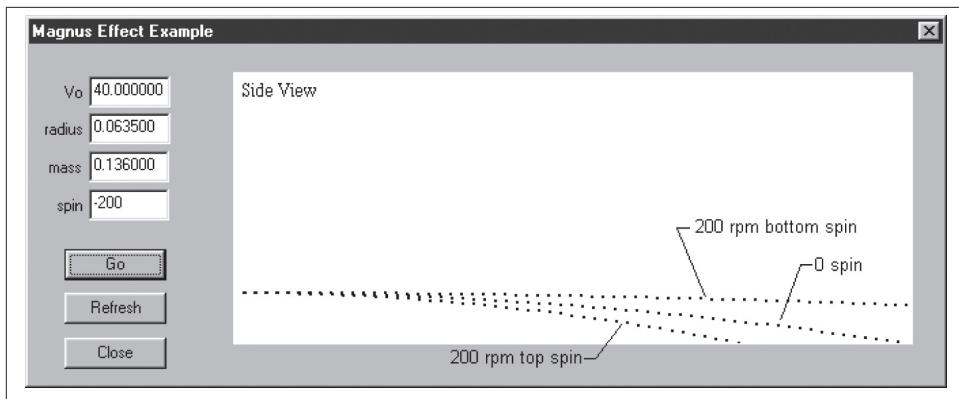


Figure 6-13. Magnus effect sample program

Variable Mass

In Chapter 1 we mentioned that some problems in dynamics involve variable mass. We'll look at variable mass here since it applies to self-propelled projectiles such as rockets. When a rocket is producing thrust to accelerate, it loses mass (fuel) at some rate. When all of the fuel is consumed (burnout), the rocket no longer produces thrust and has reached its maximum speed. After burnout you can treat the trajectory of the rocket just as you would a non-self-propelled projectile, as discussed earlier. However, while the rocket is producing thrust, you need to consider its mass change since this will affect its motion.

In cases where the mass change of the object under consideration is such that the mass being expelled or taken in has 0 absolute velocity—like a ship consuming fuel, for example—you can set up the equations of motion as you normally would, where the sum of the forces equals the rate of change in momentum. However, in this case mass will be a function of time, and your equations of motion will look like this:

where u is the relative velocity between the expelled mass and the object (the rocket in this case).

For a rocket traveling straight up, neglecting air resistance and the pressure at the exhaust nozzle, the only force acting on the rocket is due to gravity. But the rocket is expelling mass (burning fuel). How it expels this mass is not important here, since the forces involved are internal to the rocket; we need to consider only the external forces. Let the fuel burn rate be $-m'$. The equation of motion (in the vertical direction) for the rocket is as follows:

PART II

Rigid-Body Dynamics

Part II focuses on rigid-body dynamics and development of both single- and multibody simulations. This part covers numerical integration, real-time simulation of particles and rigid bodies, and connected rigid bodies. Generally, this part covers what most game programmers consider elements of a physics engine.

CHAPTER 7

Real-Time Simulations

This chapter is the first in a series of chapters designed to give you a thorough introduction to the subject of real-time simulation. We say *introduction* because the subject is too vast and complex to adequately treat in a few chapters; however, we say *thorough* because we'll do more than touch on real-time simulations. In fact, we'll walk you through the development of two simple simulations, one in two dimensions and the other in three dimensions.

What we hope to do is give you enough of an understanding of this subject so that you can pursue it further with confidence. In other words, we want you to have a solid understanding of the fundamentals before jumping in to use someone else's physics engine, or venturing out to write your own.

In the context of this book, a real-time simulation is a process whereby you calculate the state of the object (or objects) you're trying to represent on the fly. You don't rely on prescribed motion sequences to animate your object, but instead you rely on your physics model, the equations of motion, and your differential equation solver to take care of the motion of your object as the simulation progresses. This sort of simulation can be used to model rigid bodies like the airplane in our `FlightSim` example, or flexible bodies such as cloth and human figures. Perhaps one of the most fundamental aspects of implementing a real-time rigid-body simulator is solving the equations of motion using numerical integration techniques. For this reason, we'll spend this entire chapter explaining the numerical integration techniques that you'll use later in the 2D and 3D simulators that we'll develop.

If you refer back for a moment to [Chapter 4](#), where we outlined a generic procedure for solving kinetics problems, you'll see that we've covered a lot of ground so far. The preceding chapters have shown you how to estimate mass properties and develop the governing equations of motion. This chapter will show you how to solve the equations of motion in order to determine acceleration, velocity, and displacement. We'll follow this

chapter up with several showing you how to implement both 2D and 3D rigid-body simulations.

Integrating the Equations of Motion

By now you should have a thorough understanding of the dynamic equations of motion for particles and rigid bodies. If not, you may want to go back and review [Chapter 1](#) through [Chapter 4](#) before reading this one. The next aspect of dealing with the equations of motion is actually solving them in your simulation. The equations of motion that we've been discussing can be classified as ordinary differential equations. In [Chapter 2](#) and [Chapter 4](#), you were able to solve these differential equations explicitly since you were dealing with simple functions for acceleration, velocity, and displacement. This won't be the case for your simulations. As you'll see in later chapters, force and moment calculations for your system can get pretty complicated and may even rely on tabulated empirical data, which will prevent you from writing simple mathematical functions that can be easily integrated. This means that you have to use numerical integration techniques to approximately integrate the equations of motion. We say *approximately* because solutions based on numerical integration won't be exact and will have a certain amount of error depending on the chosen method.

We're going to start with a rather informal explanation of how we'll apply numerical integration because it will be easier to grasp. Later we'll get into some of the formal mathematics. Take a look at the differential equation of linear motion for a particle (or rigid body's center of mass):

It is important to notice here that this does not give a formula for instantaneous velocity; instead, it gives you only an approximation of the change in velocity. Thus, to approximate the actual velocity of your particle (or rigid body), you have to know what its velocity was before the time change Δt . At the start of your simulation, at time 0, you have to know the starting velocity of your particle. This is an initial condition and is required in order to uniquely define your particle's velocity as you step through time using this equation:¹

1. In mathematics, this sort of problem is termed an *initial value problem*.

Even though we used the linear equation of motion for a particle, this integration technique (and the ones we'll show you later) applies equally well to the angular equations of motion.

Euler's Method

The preceding explanation of Euler's method was, as we said, informal. To treat Euler's method in a more mathematically rigorous way, we'll look at the Taylor series expansion of a general function, $y(x)$. Taylor's theorem lets you approximate the value of a function at some point by knowing something about that function and its derivatives at some other point. This approximation is expressed as an infinite polynomial series of the form:

In this case, the first truncated term, $((\Delta t)^2 / 2!) v''(t)$, dominates the truncation error, and this method is said to have an error of order $(\Delta t)^2$.

Geometrically, Euler's method approximates a new value, at the current step, for the function under consideration by extrapolating in the direction of the derivative of the function at the previous step. This is illustrated in [Figure 7-1](#).

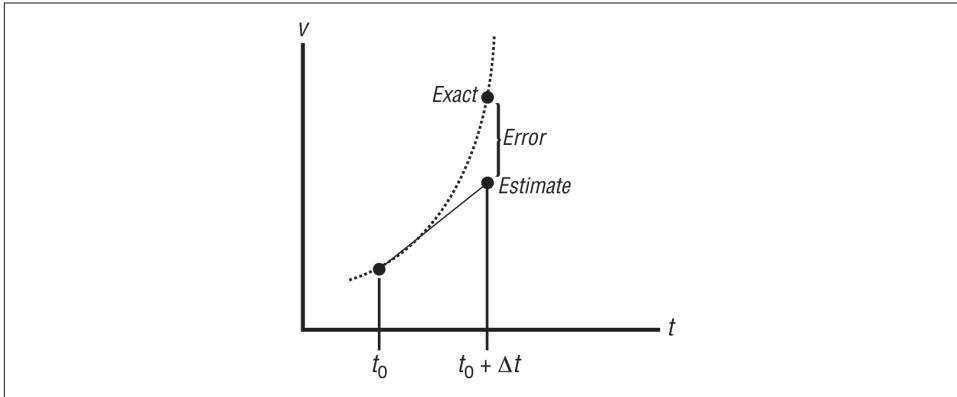


Figure 7-1. Euler integration step

[Figure 7-1](#) illustrates the truncation error and shows that Euler's method will result in a polygonal approximation of the smooth function under consideration. Clearly, if you decrease the step size, you increase the number of polygonal segments and better approximate the function. As we said before, though, this isn't always efficient to do since the number of computations in your simulation will increase and round-off error will accumulate more rapidly.

To illustrate Euler's method in practice, let's examine the linear equation of motion for the ship example of [Chapter 4](#):

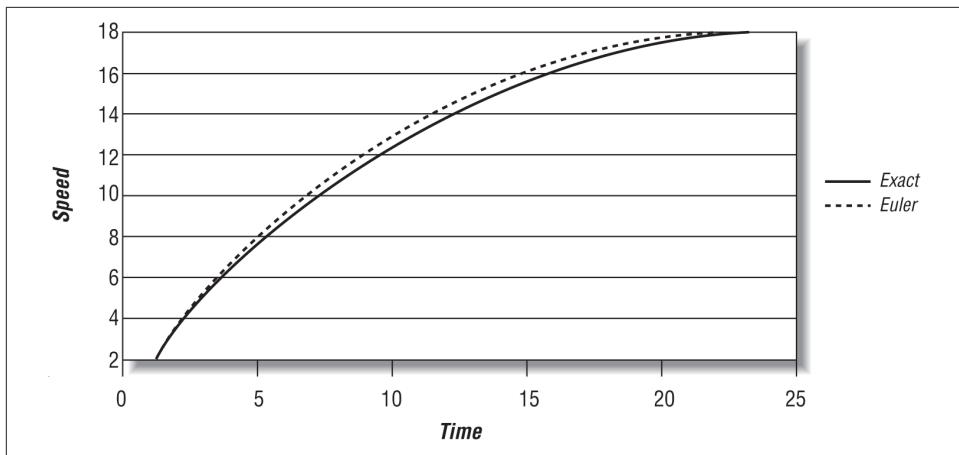


Figure 7-2. Euler integration comparison

Zooming in on this graph allows you to see the error in the Euler approximation. This is shown in [Figure 7-3](#).

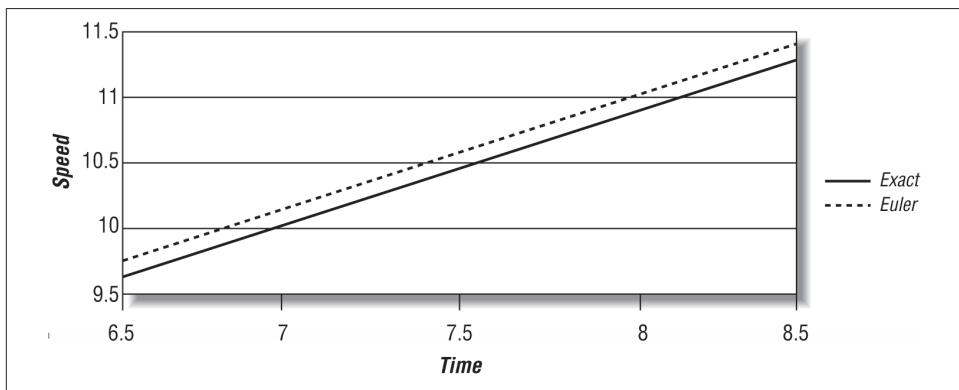


Figure 7-3. Euler error

[Table 7-1](#) shows the numerical values of speed versus time for the range shown in [Figure 7-3](#). Also shown in [Table 7-1](#) is the percent difference, the error, between the exact solution and the Euler solution at each time step.

Table 7-1. Exact solution versus Euler solution

Time (s)	Velocity, exact (m/s)	Velocity, Euler (m/s)	Error
6.5	9.559084	9.733158	1.82%
7	10.06829	10.2465	1.77%
7.5	10.55267	10.73418	1.72%
8	11.01342	11.19747	1.67%
8.5	11.4517	11.63759	1.62%

As you can see, the truncation error in this example isn't too bad. It could be better, though, and we'll show you some more accurate methods in a moment. Before that, however, you should notice that in this example Euler's method is also stable—that is, it converges well with the exact solution as shown in Figure 7-4, where we've carried the time range out further.

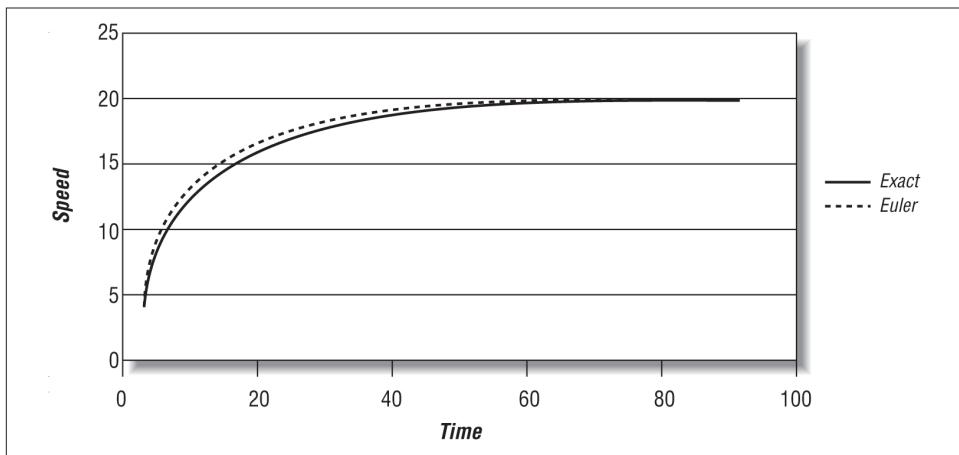


Figure 7-4. Convergence

Here's a code snippet that implements Euler's method for this example:

```
// Global Variables
float T;      // thrust
float C;      // drag coefficient
float V;      // velocity
float M;      // mass
float S;      // displacement

.

.

.

// This function progresses the simulation by dt seconds using
```

```

// Euler's basic method
void StepSimulation(float dt)
{
    float      F;      // total force
    float      A;      // acceleration
    float      Vnew;   // new velocity at time t + dt
    float      Snew;   // new position at time t + dt

    // Calculate the total force
    F = (T - (C * V));

    // Calculate the acceleration
    A = F / M;

    // Calculate the new velocity at time t + dt
    // where V is the velocity at time t
    Vnew = V + A * dt;

    // Calculate the new displacement at time t + dt
    // where S is the displacement at time t
    Snew = S + Vnew * dt;

    // Update old velocity and displacement with the new ones
    V = Vnew;
    S = Snew;
}

```

Although Euler's method is stable in this example, that isn't always so, depending on the problem you're trying to solve. This is something that you must keep in mind when implementing any numerical integration scheme. What we mean by *stable* here is that, in this case, the Euler solution converges with the exact solution. An unstable solution could manifest errors in two ways. First, successive values could oscillate above and below the exact solution, never quite converging on it. Second, successive values could diverge from the exact solution, creating a greater and greater error over time.

Take a look at [Figure 7-5](#). This figure shows how Euler's method can become very unstable. What you see in the graph represents the vibratory motion of a spring-mass system. This is a simple dynamical system that should exhibit regular sinusoidal motion.

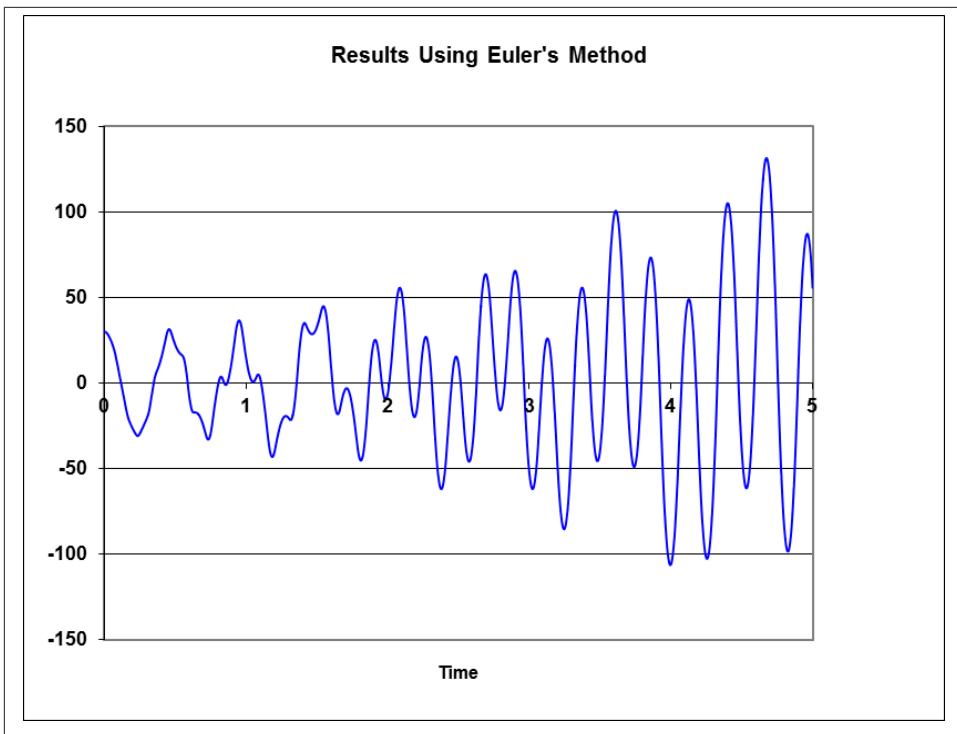


Figure 7-5. Unstable results using Euler's method

It's clear from the figure that using Euler's method yields terribly unstable results. You can see how the motion amplitude continues to grow. If this were a game, say, where you have a few objects connected by springs interacting with one another, then this sort of instability would manifest itself by wildly unrealistic motion of those objects. Worse yet, the simulation could blow up, numerically speaking.

Often, your choice of step size affects stability where smaller step sizes tend to eliminate or minimize instability and larger steps aggravate the problem. If you're working with a particularly unwieldy function, you might find that you have to decrease your step size substantially in order to achieve stability. This, however, increases the number of computations you need to make. One way around this difficulty is to employ what's called an *adaptive step size method*, in which you change your step size on the fly depending on the magnitude of a predicted amount of truncation error from one step to the next. If the truncation error is too large, then you back up a step, decrease your step size, and try again.

One way to implement this for Euler's method is to first take a step of size Δt to obtain an estimate at time $t + \Delta t$, and then take two steps (starting from time t again) of size $\Delta t/2$ to obtain another estimate at time $t + \Delta t$. Since we've been talking velocity in the

examples so far, let's call the first estimate v_1 and the second estimate v_2 .² A measure of the truncation error is then:

-
2. Even though we're talking about velocity and time here, these techniques apply to any function—for example, displacement versus time, etc.

```

A = F / M;
V2 = V1 + A * (dt/2);

// Estimate the truncation error
et = absf(V1 - V2);

// Estimate a new step size
dtnew = dt * SQRT(eto/et);

if (dtnew < dt)
{ // take at step at the new smaller step size
    F = (T - (C * V));
    A = F / M;
    Vnew = V + A * dtnew;
    Snew = S + Vnew * dtnew;
} else
{ // original step size is okay
    Vnew = V1;
    Snew = S + Vnew * dt;
}

// Update old velocity and displacement with the new ones
V = Vnew;
S = Snew;
}

```

Better Methods

At this point, you might be wondering why you can't simply use more terms in the Taylor series to reduce the truncation error of Euler's method. In fact, this is the basis for several integration methods that offer greater accuracy than Euler's basic method for a given step size. Part of the difficulty associated with picking up more terms in the Taylor's series expansion is in being able to determine the second, third, fourth, and higher derivatives of the function you're trying to integrate. The way around this problem is to perform additional Taylor series expansions to approximate the derivatives of the function under consideration and then substitute those values back into your original expansion.

Taking this approach to include one more Taylor term beyond the basic Euler method yields a so-called *improved Euler method* that has a reduced truncation error on the order of $(\Delta t)^3$ instead of $(\Delta t)^2$. The formulas for this method are:

Here y is a function of t , and y' is the derivative as a function of t and possibly of y depending on the equations you're trying to solve, and Δt the step size.

To make this clearer for you, we'll show these formulas in terms of the ship example equation of motion of [Chapter 4](#), the same example that we discussed in the previous section. In this case, velocity is approximated by the following formulas:

We can carry out this procedure of taking on more Taylor terms even further. The popular Runge-Kutta method takes such an approach to reduce the truncation error to the order of $(\Delta t)^5$. The integration formulas for this method are as follows:

```

// Calculate the new velocity at time t + dt
// where V is the velocity at time t
Vnew = V + (k1 + 2*k2 + 2*k3 + k4) / 6;

// Calculate the new displacement at time t + dt
// where S is the displacement at time t
Snew = S + Vnew * dt;

// Update old velocity and displacement with the new ones
V = Vnew;
S = Snew;
}

```

To show you how accuracy is improved over the basic Euler method, we've superimposed integration results for the ship example using these two methods over those shown in [Figure 7-2](#) and [Figure 7-3](#). [Figure 7-6](#) and [Figure 7-7](#) show the results, where [Figure 7-7](#) is a zoomed view of 7-6.

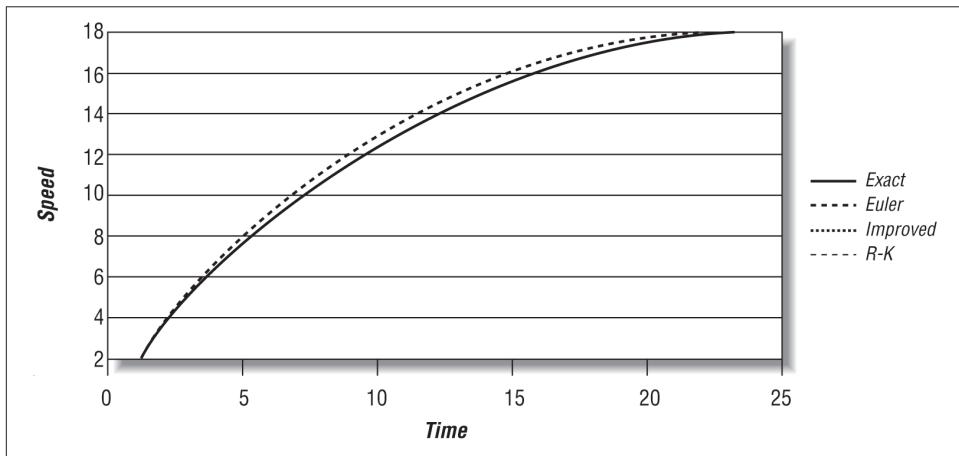


Figure 7-6. Method comparison

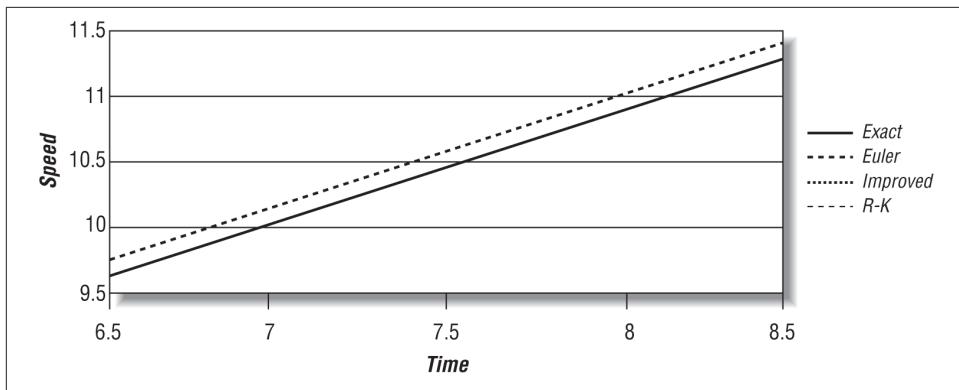


Figure 7-7. A closer look

As you can see from these figures, it's impossible to discern the curves for the improved Euler and Runge-Kutta methods from the exact solution because they fall almost right on top of each other. These results clearly show the improvement in accuracy over the basic Euler method, whose curve is distinct from the other three. Over the interval from 6.5 to 8.5 seconds, the average truncation error is 1.72%, 0.03%, and $3.6 \times 10^{-6}\%$ for Euler's method, the improved Euler method, the Runge-Kutta method, respectively. It is obvious, based on these results, that for this problem, the Runge-Kutta method yields substantially better results for a given step size than the other two methods. Of course, you pay for this accuracy, since you have several more computations per step in the Runge-Kutta method.

Both of these methods are generally more stable than Euler's method, which is a huge benefit in real-time applications. Recall our discussion earlier about the stability of Euler's method. [Figure 7-5](#) showed the results of applying Euler's method to an oscillating dynamical system. There, the motion results that should be sinusoidal were wildly erratic (i.e., unstable). Applying the improved Euler method, or the Runge-Kutta method, to the same problem yields stable results, as shown in [Figure 7-8](#).

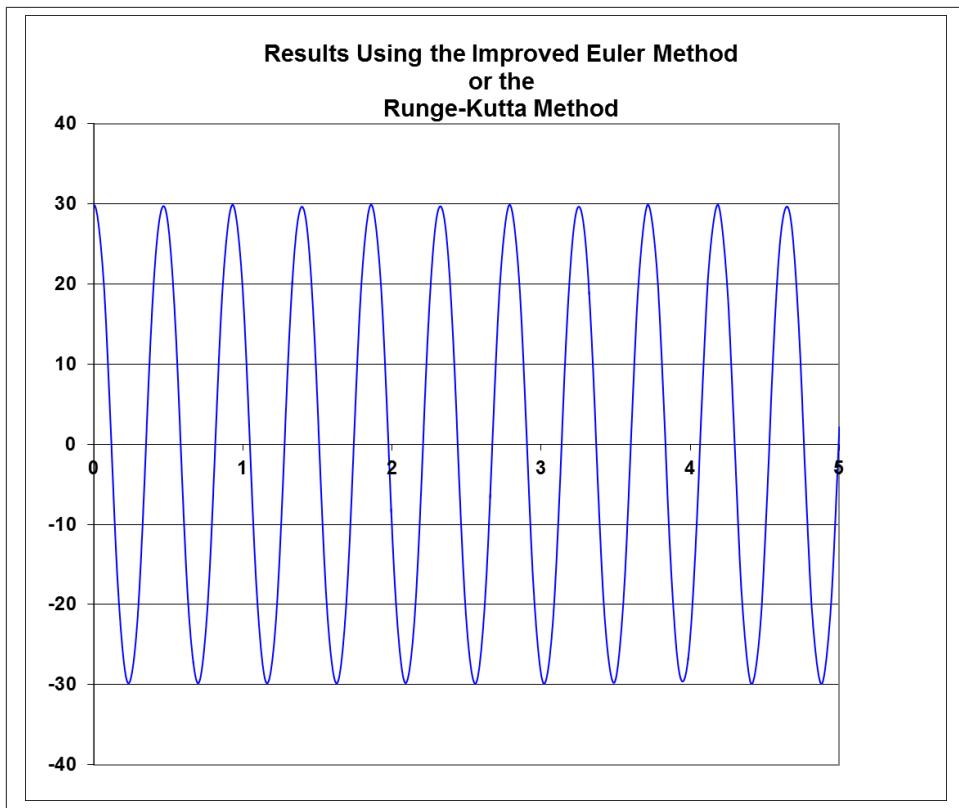


Figure 7-8. Stable results using the improved Euler or the Runge-Kutta methods

Here the oscillatory motion is clearly sinusoidal, as it should be. The results for this particular problem are almost identical whether you use the improved Euler method or the Runge-Kutta method. Since for this problem the results of both methods are virtually the same, you can save computational time and memory using the improved Euler method versus the Runge-Kutta method. This can be a significant advantage for real-time games. Remember the Runge-Kutta method requires four derivative computations per time step.

These methods aren't the only ones at your disposal, but they are the most common. The Runge-Kutta method is particularly popular as a general-purpose numerical integration scheme. Other methods attempt to improve computational efficiency even further—that is, they are designed to minimize truncation error while still allowing you to take relatively large step sizes so as to reduce the number of steps you have to take in your integration. Still other methods are especially tailored for specific problem types. We cite some pretty good references for further reading on this subject in the [Bibliography](#).

Summary

At this point you should be comfortable with the terms that appear in the equations of motion and be able to calculate terms like the sum of forces and moments, mass, and inertia. You should also have a solid understanding of basic numerical integration techniques. Implementing these techniques in code is really straightforward since they are composed of simple polynomial functions. The hard part is developing the derivative function for your problem. In the case of the equations of motion, the derivative function will include all your force and moment calculations for the particle or rigid body that you are modeling. You'll see some more numerical integration code when you get to [Chapter 8](#), [Chapter 9](#), and [Chapter 11](#).

CHAPTER 8

Particles

In this chapter we'll show you how to apply what you've learned in [Chapter 7](#) in a simple particle simulator. Before getting to the specifics of the example we'll present, let's consider particles in general. Particles are simple idealizations that can be used to simulate all sorts of phenomena or special effects within a game. For example, particle simulations are often used to simulate smoke, fire, and explosions. They can also be used to simulate water, dust clouds, and swarms of insects, among many other things. Really, your imagination is the only limit. Particles lend themselves to simulating both discrete objects like bouncing balls and continua like water. Plus, you can easily ascribe an array of attributes to particles depending on what you're modeling.

For example, say, you're modeling fire using particles. Each particle will rise in the air, and as it cools its color will change until it fades away. You can tie the particle's color to its temperature, which is modeled using thermodynamics. The attribute you'd want to track is the particle's temperature. In a previous work, *AI for Game Programmers* (O'Reilly), this book's coauthor David M. Bourg used particles to represent swarms of insects that would swarm, flock, chase, and evade depending on the artificial intelligence (AI). The AI controlled their behavior, which was then implemented as a system of particles using principles very similar to what you'll see in this chapter.

Particles are not limited to collections of independent objects either. Later in this book, you'll learn how to connect particles together using springs to create deformable objects such as cloth. Particles are extremely versatile, and you'll do well to learn how to leverage their simplicity.

You can use particles to model sand in a simple phone application that simulates an hourglass. Couple this sand model with the accelerometer techniques you'll learn about in [Chapter 21](#), and you'll be able to make the sand flow by turning the phone over.

You can easily use particles to simulate bullets flying out of a gun. Imagine a Gatling gun spewing forth a hail of lead, all simulated using simple particles. Speaking of spew-

ing, how about using particles to simulate debris flung from an erupting volcano as a special effect in your adventure game set in prehistoric times? Remember the Wooly Willy toy? To make particles a direct part of game play, consider a diversionary application where you drag piles of virtual magnetic particles around a portrait photograph, giving someone a lovely beard or mustache much like Wooly Willy.

Hopefully, you're now thinking of creative ways to use particles in your games. So, let's address implementation. There are two basic ingredients to implementing a particle simulator: the particle *model* and the *integrator*. (Well, you could argue that a third basic ingredient is the *renderer*, where you actually draw the particles, but that's more graphics than physics, and we're focusing on modeling and integrating in this book.)

The model very simply describes the attributes of the particles in the simulation along with their rules of behavior. We mean this in the physics sense and not the AI sense in this book, although in general the model you implement may very well include suitable AI rules. Now, the integrator is responsible for updating the state of the particles throughout the simulation. In this chapter, the particles' states will be described by their position and velocity at any given time. The integrator will update each particle's state under the influence of several external stimuli—forces such as gravity, aerodynamic drag, and collisions.

The rest of this chapter will walk you through the details of a simple particle simulation in an incremental manner. The first task will be to simulate a set of particles falling under the influence of gravity alone. Even though this sounds elementary, such an example encompasses all of the basic ingredients mentioned earlier. Once gravity is under control, we'll show you how to implement still air drag and wind forces to influence the particles' motion. Then, we'll make things more interesting by showing you how to implement collision response between the particles and a ground plane plus random obstacles. This collision stuff will draw on material presented in [Chapter 5](#), so be sure to read that chapter first if you have not already done so.

[Figure 8-1](#) through [Figure 8-4](#) show a few frames of this example simulation complete with obstacles and collisions. Use your imagination here to visualize the particles falling under the influence of gravity until they impact the circular objects, at which time they bounce around and ultimately settle on the ground.

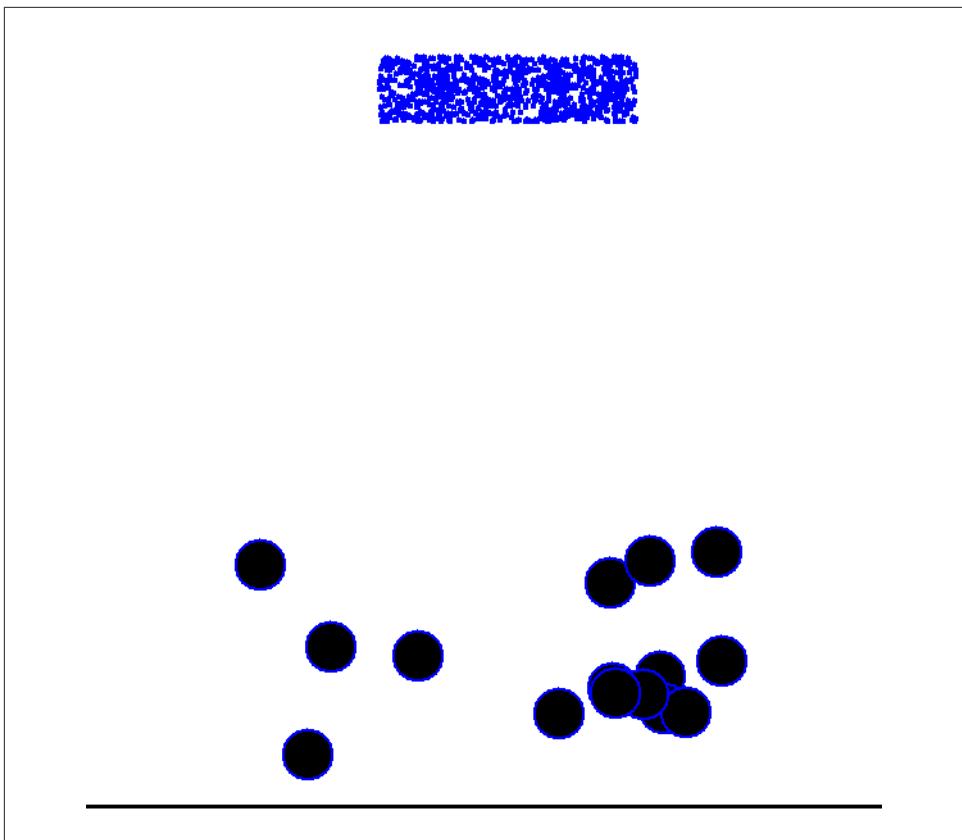


Figure 8-1. Particles falling under the influence of gravity

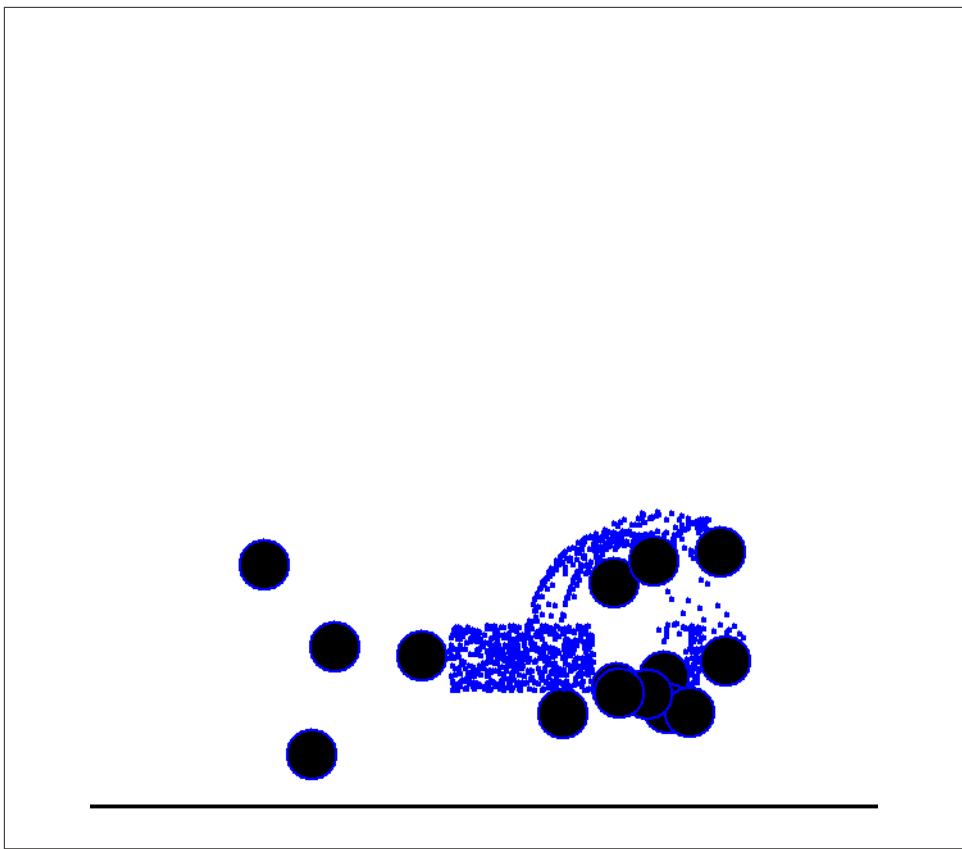


Figure 8-2. Particles impacting circular objects

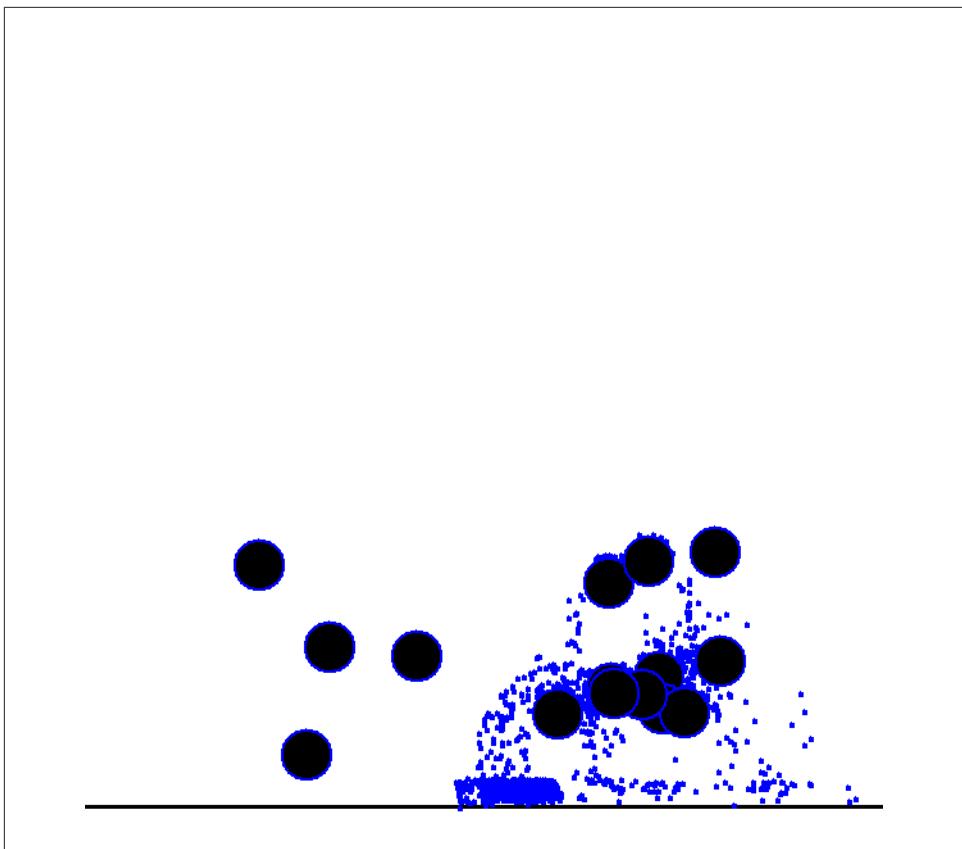


Figure 8-3. More collisions

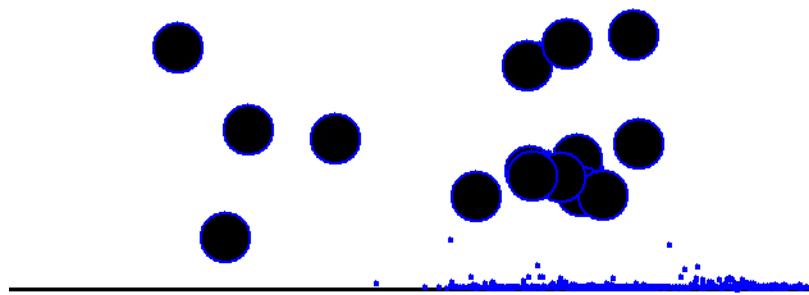


Figure 8-4. Particles coming to rest on the ground plane

While working through this chapter, keep in mind that everything you're learning here will be directly applicable to 2D and 3D simulations. Chapters following this one will build on the material covered here. We'll focus on two dimensions in this chapter and later in the book we'll show you how to extend the simulation to 3D. Actually, for particle simulations it's almost trivial to make the leap from 2D to 3D. Trust us on this for now.

Simple Particle Model

The particle model we'll begin with is very simple. All we want to achieve at first is to have the particles fall under the influence of gravity. The particles will be initialized with an altitude above a ground plane. Upon the start of the simulation, gravity will act on each particle, continuously causing each to accelerate toward the ground plane, gaining speed as it goes. Imagine holding a handful of small rocks up high and then releasing them. Simple, huh?

There are several particle attributes we must consider even for this simple example. Our model assumes that each particle has mass, and a set diameter (we're assuming our particles are circles in 2D or spheres in 3D), occupies some position in space, and is traveling at some velocity. Additionally, each particle is acted upon by some net external force, which is the aggregate of all forces acting on the particle. These forces will be gravity alone to start with, but will eventually include drag and impact forces. We set up a `Particle` class to encapsulate these attributes as follows:

```
class Particle {  
public:  
    float fMass;      // Total mass  
    Vector vPosition; // Position  
    Vector vVelocity; // Velocity  
    float fSpeed;     // Speed (magnitude of the velocity)  
    Vector vForces;   // Total force acting on the particle  
    float fRadius;    // Particle radius used for collision detection  
    Vector vGravity;  // Gravity force vector  
  
    Particle(void);   // Constructor  
    void CalcLoads(void); // Aggregates forces acting on the particle  
    void UpdateBodyEuler(double dt); // Integrates one time step  
    void Draw(void);   // Draws the particle  
};
```

Most of these attributes are self-explanatory given the comments we've included. Notice that several of these attributes are `Vector` types. These vectors are defined in the custom math library we've included in [Appendix A](#). This type makes managing vectors and performing arithmetic operations with them a breeze. Take a look at [Appendix A](#) to see what this custom type does. We'll just remind you of the data structure `Vector` uses: three scalars called *x*, *y*, and *z* representing the three dimensions of a location or of a movement in some direction. The *z* component will always be set to 0 in this chapter's examples.

You should have noticed the `fSpeed` property in the `Particle` class. This property stores the magnitude of the velocity vector, the particle's speed. We'll use this later when computing aerodynamic drag forces. We've also included a `Vector` type property called `vGravity`, which stores the gravity force vector defining the magnitude and the direction in which the gravity force acts. This is not really necessary, as you could hardcode the gravity force vector or use a global variable; however, we've included it here to illustrate some creative flexibility. For example, you could redefine the gravity vector in a game that uses accelerometer input to determine gravity's direction with respect to a particular device's orientation (see [Chapter 21](#)). And you may have a game where some particles react to different gravities depending on their type, which can be of your own concoction.

Aside from properties, you'll notice several methods in the `Particle` class. The constructor is trivial. It sets everything to 0 except the particle's mass, radius, and the gravity force vector. The following code illustrates how everything is initialized:

```
Particle::Particle(void)
{
    fMass = 1.0;
    vPosition.x = 0.0;
    vPosition.y = 0.0;
    vPosition.z = 0.0;
    vVelocity.x = 0.0;
    vVelocity.y = 0.0;
    vVelocity.z = 0.0;
    fSpeed = 0.0;
    vForces.x = 0.0;
    vForces.y = 0.0;
    vForces.z = 0.0;
    fRadius = 0.1;
    vGravity.x = 0;
    vGravity.y = fMass * _GRAVITYACCELERATION;
}
```

Now is probably a good time to explain the coordinate system we've assumed. Our world origin is located at the lower-left corner of the example program's window with positive *x* pointing to the right and positive *y* pointing up. The acceleration due to gravity acts downward (i.e., in the negative *y*-direction). We're using the SI system of units and have defined the acceleration due to gravity as follows:

```
#define _GRAVITYACCELERATION -9.8f
```

That's 9.8 m/s^2 in the negative *y*-direction. We've set the mass of each particle to 1 kg by default, which means the force due to gravity is 1 kg times 9.8 m/s^2 , or 9.8 newtons of force. We've set the radius of each particle to one-tenth of a meter. These masses and radii are really arbitrary; you can set them to anything suitable for what you're modeling.

The `CalcLoads` method is responsible for computing all the loads—forces—acting on the particle, with the exception of impact forces (we'll handle those later). For now, the only force acting on the particles is that due to gravity, or simply, the weight of each particle. `CalcLoads` is very simple at this point:

```
void Particle::CalcLoads(void)
{
    // Reset forces:
    vForces.x = 0.0f;
    vForces.y = 0.0f;

    // Aggregate forces:
    vForces += vGravity;
}
```

The first order of business is to reset the `vForces` vector. `vForces` is the vector containing the net force acting on the particle. All of these forces are aggregated in `CalcLoads`, as shown by the line `vForces += vGravity`. Again, so far, the only force to aggregate is that due to gravity.

Integrator

The `UpdateBodyEuler` method integrates the equations of motion for each particle. Since we're dealing with particles, the only equation of motion we need concern ourselves with is that for translation; rotation does not matter for particles (at least not for us here). The following code sample shows `UpdateBodyEuler`.

```
void Particle::UpdateBodyEuler(double dt)
{
    Vector a;
    Vector dv;
    Vector ds;

    // Integrate equation of motion:
    a = vForces / fMass;

    dv = a * dt;
    vVelocity += dv;

    ds = vVelocity * dt;
    vPosition += ds;

    // Misc. calculations:
    fSpeed = vVelocity.Magnitude();
}
```

As the name of this method implies, we've implemented Euler's method of integration as described in [Chapter 7](#). Using this method, we simply need to divide the aggregate forces acting on a particle by the mass of the particle to get the particle's acceleration. The line of code `a = vForces / fMass` does just this. Notice here that `a` is a `Vector`, as is `vForces`. `fMass` is a scalar, and the `/` operator defined in the `Vector` class takes care of dividing each component of the `vForces` vector by `fMass` and setting the corresponding components in `a`. The change in velocity, `dv`, is equal to acceleration times the change in time, `dt`. The particle's new velocity is then computed by the line `vVelocity += dv`. Here again, `vVelocity` and `dv` are `Vectors` and the `+=` operator takes care of the vector arithmetic. This is the first actual integration.

The second integration takes place in the next few lines, where we determine the particle's displacement and new position by integrating its velocity. The line `ds = vVelocity * dt` determines the displacement, or change in the particle's position, and the line `vPosition += ds` computes the new position by adding the displacement to the particle's old position.

The last line in `UpdateBodyEuler` computes the particle's speed by taking the magnitude of its velocity vector.

For demonstration purposes, using Euler's method is just fine. In an actual game, the more robust method described in [Chapter 7](#) is advised.

Rendering

In this example, rendering the particles is rather trivial. All we do is draw little circles using Windows API calls wrapped in our own functions to hide some of the Windows-specific code. The following code snippet is all we need to render the particles.

```
void Particle::Draw(void)
{
    RECT r;
    float drawRadius = max(2, fRadius);

    SetRect(&r, vPosition.x - drawRadius,
            _WINHEIGHT - (vPosition.y - drawRadius),
            vPosition.x + drawRadius,
            _WINHEIGHT - (vPosition.y + drawRadius));
    DrawEllipse(&r, 2, RGB(0,0,0));
}
```

You can use your own rendering code here, of course, and all you really need to pay close attention to is converting from world coordinates to window coordinates. Remember, we've assumed our world coordinate system origin is in the lower-left corner of the window, whereas the window drawing coordinate system has its origin in the upper-left corner of the window. To transform coordinates in this example, all you need to do is subtract the particle's y-position from the height of the window.

The Basic Simulator

The heart of this simulation is handled by the `Particle` class described earlier. However, we need to show you how that class is used in the context of the main program.

First, we define a few global variables as follows:

```
// Global Variables:
int     FrameCounter = 0;
Particle Units[_MAX_NUM_UNITS];
```

`FrameCounter` counts the number of time steps integrated before the graphics display is updated. How many time steps you allow the simulation to integrate before updating the display is a matter of tuning. You'll see how this is used momentarily when we discuss the `UpdateSimulation` function. `Units` is an array of `Particle` types. These will represent moving particles in the simulation—the ones that fall from above and bounce off the circular objects we'll add later.

For the most part, each unit is initialized in accordance with the `Particle` constructor shown earlier. However, their positions are all at the origin, so we make a call to the following `Initialize` function to randomly distribute the particles in the upper-middle portion of the screen within a rectangle of width `_SPAWN_AREA_R*4` and a height of `_SPAWN_AREA_R`, where `_SPAWN_AREA_R` is just a global `define` we made up.

```
bool Initialize(void)
{
    int i;

    GetRandomNumber(0, _WINWIDTH, true);

    for(i=0; i<_MAX_NUM_UNITS; i++)
    {
        Units[i].vPosition.x = GetRandomNumber(_WINWIDTH/2-_SPAWN_AREA_R*2,
                                                _WINWIDTH/2+_SPAWN_AREA_R*2, false);
        Units[i].vPosition.y = _WINHEIGHT -
            GetRandomNumber(_WINHEIGHT/2-_SPAWN_AREA_R,
                           _WINHEIGHT/2, false);
    }

    return true;
}
```

OK, now let's consider `UpdateSimulation` as shown in the code snippet that follows. This function gets called every cycle through the program's main message loop and is responsible for cycling through all the `Units`, making appropriate function calls to update their positions, and rendering the scene.

```
void UpdateSimulation(void)
{
    double dt = _TICKTIME;
    int     i;

    // initialize the back buffer
    if(FrameCounter >= _RENDER_FRAME_COUNT)
    {
        ClearBackBuffer();
    }

    // update the particles (Units)
    for(i=0; i<_MAX_NUM_UNITS; i++)
    {
        Units[i].CalcLoads();
        Units[i].UpdateBodyEuler(dt);

        if(FrameCounter >= _RENDER_FRAME_COUNT)
        {
            Units[i].Draw();
        }
    }
}
```

```

    if(Units[i].vPosition.x > _WINWIDTH) Units[i].vPosition.x = 0;
    if(Units[i].vPosition.x < 0) Units[i].vPosition.x = _WINWIDTH;
    if(Units[i].vPosition.y > _WINHEIGHT) Units[i].vPosition.y = 0;
    if(Units[i].vPosition.y < 0) Units[i].vPosition.y = _WINHEIGHT;
}

// Render the scene if required
if(FrameCounter >= _RENDER_FRAME_COUNT) {
    CopyBackBufferToWindow();
    FrameCounter = 0;
} else
    FrameCounter++;
}

```

The two local variables in `UpdateSimulation` are `dt` and `i`. `i` is trivial and serves as a counter variable. `dt` represents the small yet finite amount of time, in seconds, over which each integration step is taken. The global `define_TIMESTEP` stores the time step, which we have set to 0.1 seconds. This value is subject to tuning, which we'll discuss toward the end of this chapter in the section "[“Tuning” on page 186](#)".

The next segment of code checks the value of the frame counter, and if the frame counter has reached the defined number of frames, stored in `_RENDER_FRAME_COUNT`, then the back buffer is cleared to prepare it for drawing upon and ultimately copying to the screen.

The next section of code under the comment `update the particles` does just that by calling the `CalcLoads` and `UpdateBodyEuler` methods of each `Unit`. These two lines are responsible for updating all the forces acting on each particle and then integrating the equation of motion for each particle.

The next few lines within the `for` loop draw each particle if required and wrap each particle's position around the window extents should they progress beyond the edges of the window. Note that we're using window coordinates in this example.

Implementing External Forces

We'll add a couple of simple external forces to start with—still air drag, and wind force. We'll use the formulas presented in [Chapter 3](#) to approximate these forces, treating them in a similar manner. Recall that still air drag is the aerodynamic drag force acting against an object moving at some speed through still air. Drag always acts to resist motion. While we'll use the same formulas to compute a wind force, recall that wind force may not necessarily act to impede motion. You could have a tailwind pushing an object along, or the wind could come from any direction with components that push the object sideways. In this example we'll assume a side wind from left to right, acting to push the particles sideways, with the still air drag resisting their falling motion. When we add collisions later, this same drag formulation will act to resist particle motion in any direction in which they travel as they bounce around.

The formula we'll use to model still air drag is:

```
    vForces += vWind;  
}
```

So after the force due to gravity is added to the aggregate, two new local variables are declared. `vDrag` is a vector that will represent the still air drag force. `fDrag` is the magnitude of that drag force. Since we know the drag force vector is exactly opposite to the particle's velocity vector, we can equate `vDrag` to negative `vVelocity` and then normalize `vDrag` to obtain a unit vector pointing in a direction opposite of the particle's velocity. Next we compute the magnitude of the drag force using the formula shown earlier. This line handles that:

```
fDrag = 0.5 * _AIRDENSITY * fSpeed * fSpeed *  
(3.14159 * fRadius * fRadius) * _DRAGCOEFFICIENT;
```

Here, `_AIRDENSITY` is a global `define` representing the density of air, which we have set to 1.23 kg/m^3 (standard air at 15°C). `fSpeed` is the particle's speed: the magnitude of its velocity. The `3.14159 * fRadius * fRadius` line represents the projected area of the particle assuming the particle is a sphere. And finally, `_DRAGCOEFFICIENT` is a drag coefficient that we have set to 0.6. We picked this value from the chart of drag coefficient for a smooth sphere versus the Reynolds number shown in [Chapter 6](#). We simply eyeballed a value in the Reynolds number range from $1\text{e}4$ to $1\text{e}5$. You have a choice here of tuning the drag coefficient value to achieve some desired effect, or you can create a curve fit or lookup table to select a drag coefficient corresponding to the Reynolds number of the moving particle.

Now that we have the magnitude of the drag force, we simply multiply that force by the unit drag vector to obtain the final drag force vector. This vector then gets aggregated in `vForces`.

We handle the wind force in a similar manner with a few differences in the details. First, since we know the unit wind force vector is in the positive x -direction (i.e., it acts from left to right), we can simply set the x component of the wind force vector, `vWind`, to the magnitude of the wind force. We compute that wind force using the same formula we used for still air drag with the exception of using the wind speed instead of the particle's speed. We used `_WINDSPEED`, a global `define`, to represent the wind speed, which we have set to 10 m/s (about 20 knots).

Finally, the wind force is aggregated in `vForces`.

At this stage the particles will fall under the influence of gravity, but not as fast as they would have without the drag force. And now they'll also drift to the right due to the wind force.

Implementing Collisions

Adding external forces made the simulation a little more interesting. However, to really make it pop, we're going to add collisions. Specifically, we'll handle particle-to-ground collisions and particle-to-object collisions. If you have not yet read [Chapter 5](#), which covers collisions, you should because we'll implement principles covered in that chapter here in the example. We'll implement enough collision handling in this example to allow particles to bounce off the ground and circular objects, and we'll come back to collision handling in more detail in [Chapter 10](#). The material in this chapter should whet your appetite. We'll start with the easier case of particle-to-ground collisions.

Particle-to-Ground Collisions

Essentially what we're aiming to achieve with particle-to-ground collision detection is to prevent the particles from passing through a ground plane specified at some y coordinate. Imagine a horizontal impenetrable surface that the particles cannot pass through. There are several things we must do in order to detect whether a particle is indeed colliding with the ground plane. If so, then we need to handle the collision, making the particles respond in a suitable manner.

The left side of [Figure 8-5](#) illustrates a collision scenario. It's easy to determine whether or not a collision has taken place. Over a given simulation time step, a particle may have moved from some previous position (its position at the previous time step) to its current position. If this current position puts the centroid coordinate of the particle within one particle radius of the ground plane, then a collision might be occurring. We say *might* because the other criteria we need to check in order to determine whether or not a collision is happening is whether or not the particle is moving toward the ground plane. If the particle is moving toward the ground plane and it's within one radius of the ground plane, then a collision is occurring. It may also be the case that the particle has passed completely through the ground plane, in which case we assume a collision has occurred.

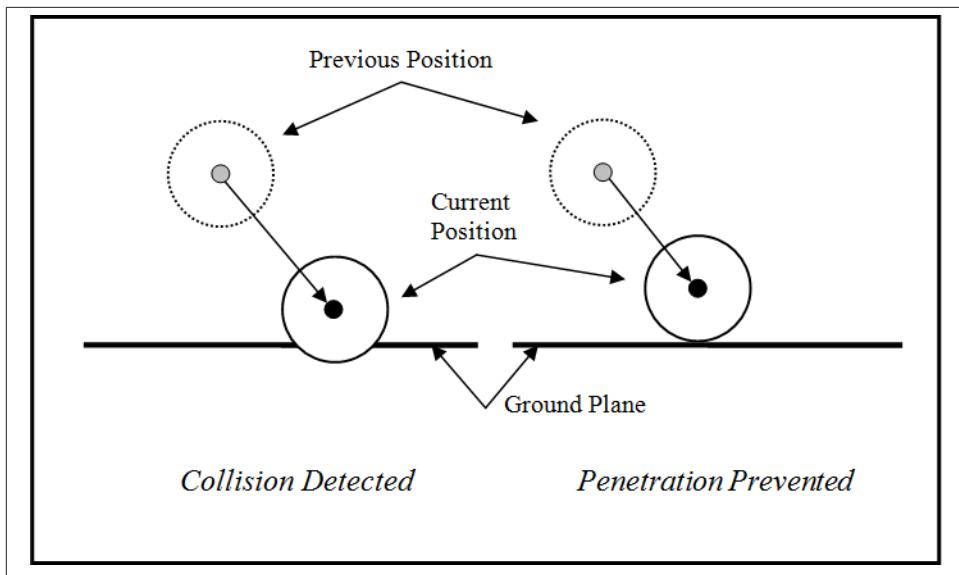


Figure 8-5. Particle-to-ground collision

To prevent such penetration of the ground plane, we need to do two things. First, we must reposition the particle so that it is just touching the ground plane, as shown on the right side of [Figure 8-5](#). Second, we must apply some impact force resulting from the collision in order to force the particle to either stop moving down into the ground plane or to move away from the ground plane. All these steps make up collision detection and response.

There are several changes and additions that we must make to the example code in order to implement particle-to-ground collision detection and response. Let's begin with the `Particle` class.

We've added three new properties to `Particle`, as follows:

```
class Particle {
    .
    .
    .
    Vector vPreviousPosition;
    Vector vImpactForces;
    bool bCollision;
    .
    .
    .
};
```

`vPreviousPosition` is used to store the particle's position at the previous time step—that is, at time $t - dt$. `vImpactForces` is used to aggregate all of the impact forces acting

on a particle. You'll see later that it is possible for a particle to collide with more than one object at the same time. `bCollision` is simply a flag that is used to indicate whether or not a collision has been detected with the particle at the current time step. This is important because when a collision occurs, at that instant in time, we assume that the only forces acting on the particle are the impact forces; all of the other forces—gravity, drag, and wind—are ignored for that time instant. We use `bCollision` in the updated `CalcLoads` method:

```
void Particle::CalcLoads(void)
{
    // Reset forces:
    vForces.x = 0.0f;
    vForces.y = 0.0f;

    // Aggregate forces:
    if(bCollision) {
        // Add Impact forces (if any)
        vForces += vImpactForces;
    } else {
        // Gravity
        vForces += vGravity;

        // Still air drag
        Vector vDrag;
        float fDrag;

        vDrag -= vVelocity;
        vDrag.Normalize();
        fDrag = 0.5 * _AIRDENSIY * fSpeed * fSpeed *
            (3.14159 * fRadius * fRadius) * _DRAGCOEFFICIENT;
        vDrag *= fDrag;
        vForces += vDrag;

        // Wind
        Vector vWind;
        vWind.x = 0.5 * _AIRDENSIY * _WINDSPEED * _WINDSPEED *
            (3.14159 * fRadius * fRadius) * _DRAGCOEFFICIENT;
        vForces += vWind;
    }
}
```

The only difference between this version of `CalcLoads` and the previous one is that we added the conditional `if(bCollision) { ... } else { ... }`. If `bCollision` is true, then we have a collision to deal with and the only forces that get aggregated are the impact forces. If there is no collision, if `bCollision` is false, then the non-impact forces are aggregated in the usual manner.

You may have caught that we are aggregating impact forces in this example. This is an alternate approach to the one shown in [Chapter 5](#). There we showed you how to calculate an impulse and change an object's velocity in response to a collision, using conservation

of momentum. Well, we're still calculating impulses just like in [Chapter 5](#); however, in this example, we're going to compute the impact force based on that impulse and apply that force to the colliding objects. We'll let the numerical integrator integrate that force to derive the colliding particle's new velocities. Either method works, and we'll show you an example of the former method later. We're showing the latter method here just to illustrate some alternatives. The advantage of this latter method is that it is easy to compute impact forces due to multiple impacts and let the integrator take care of them all at once.

Now, with these changes made to `Particle`, we need to add a line of code to `UpdateSimulation`, as shown here:

```
void UpdateSimulation(void)
{
    .
    .
    .

    // update computer controlled units:
    for(i=0; i<_MAX_NUM_UNITS; i++)
    {
        Units[i].bCollision = CheckForCollisions(&(Units[i]));
        Units[i].CalcLoads();
        Units[i].UpdateBodyEuler(dt);
        .
        .
        .
    } // end i-loop

    .
    .
    .
}
```

The new line is `Units[i].bCollision = CheckForCollisions(&(Units[i]));`. `CheckForCollisions` is a new function that takes the given unit, whose pointer is passed as an argument, and checks to see if it's colliding with anything—in this case, the ground. If a collision is detected, `CheckForCollisions` also computes the impact force and returns `true` to let us know a collision has occurred. `CheckForCollisions` is as follows:

```
bool CheckForCollisions(Particle* p)
{
    Vector n;
    Vector vr;
    float vrn;
    float J;
    Vector Fi;
    bool hasCollision = false;
```

```

// Reset aggregate impact force
p->vImpactForces.x = 0;
p->vImpactForces.y = 0;

// check for collisions with ground plane
if(p->vPosition.y <= (_GROUND_PLANE+p->fRadius)) {
    n.x = 0;
    n.y = 1;
    vr = p->vVelocity;
    vrn = vr * n;
    // check to see if the particle is moving toward the ground
    if(vrn < 0.0) {
        J = -(vr*n) * (_RESTITUTION + 1) * p->fMass;
        Fi = n;
        Fi *= J/_Timestep;
        p->vImpactForces += Fi;

        p->vPosition.y = _GROUND_PLANE + p->fRadius;
        p->vPosition.x = ((_GROUND_PLANE + p->fRadius -
                            p->vPreviousPosition.y) /
                           (p->vPosition.y - p->vPreviousPosition.y) *
                           (p->vPosition.x - p->vPreviousPosition.x)) +
                           p->vPreviousPosition.x;

        hasCollision = true;
    }
}

return hasCollision;
}

```

`CheckForCollisions` makes two checks: 1) it checks to see whether or not the particle is making contact or passing through the ground plane; and 2) it checks to make sure the particle is actually moving toward the ground plane. Keep in mind a particle could be in contact with the ground plane right after a collision has been handled with the particle moving away from the ground. In this case, we don't want to register another collision.

Let's consider the details of this function, starting with the local variables. `n` is a vector that represents the unit normal vector pointing from the ground plane to the particle colliding with it. For ground collisions, in this example, the unit normal vector is always straight up since the ground plane is flat. This means the unit normal vector will always have an `x` component of 0 and its `y` component will be 1.

The `Vector vr` is the relative velocity vector between the particle and the ground. Since the ground isn't moving, the relative velocity is simply the velocity of the particle. `vrn` is a scalar that's used to store the component of the relative velocity in the direction of the collision unit normal vector. We compute `vrn` by taking the dot product of the relative velocity with the unit normal vector. `J` is a scalar that stores the impulse resulting from the collision. `Fi` is a vector that stores the impact force as derived from the impulse

J. Finally, `hasCollision` is a flag that's set based on whether or not a collision has been detected.

Now we'll look at the details within `CheckForCollisions`. The first task is to initialize the impact force vector, `vImpactForces`, to 0. Next, we make the first collision check by determining if the y-position of the particle is less than the ground plane height plus the particles radius. If it is, then we know a collision may have occurred. (`_GROUND_PLANE` represents the y coordinate of the ground plane, which we have set to 100.) If a collision may have occurred, then we make the next check—to determine if the particle is moving toward the ground plane.

To perform this second check, we compute the unit normal vector, relative velocity, and relative velocity component in the collision normal direct as described earlier. If the relative velocity in the normal direction is negative (i.e., if $vr \cdot n < 0$), then a collision has occurred. If either of these checks is `false`, then a collision has not occurred and the function exits, returning `false`.

The interesting stuff happens if the second check passes. This is where we have to determine the impact force that will cause the particle to bounce off the ground plane. Here's the specific code that computes the impact force:

```
J = -(vr*n) * (_RESTITUTION + 1) * p->fMass;
Fi = n;
Fi *= J/_TSTEP;
p->vImpactForces += Fi;

p->vPosition.y = _GROUND_PLANE + p->fRadius;
p->vPosition.x = (_GROUND_PLANE + p->fRadius -
    p->vPreviousPosition.y) /
    (p->vPosition.y - p->vPreviousPosition.y) *
    (p->vPosition.x - p->vPreviousPosition.x) +
    p->vPreviousPosition.x;

hasCollision = true;
```

We compute the impulse using the formulas presented in [Chapter 5](#). J is a scalar equal to the negative of the relative velocity in the normal direction times the coefficient of restitution plus 1 times the particle mass. Recall that the coefficient of restitution, `_RESTITUTION`, governs how elastic or inelastic the collision is, or in other words, how much energy is transferred back to the particle during the impact. We have this value set to 0.6, but it is tunable depending on what effect you're trying to achieve. A value of 1 makes the particles very bouncy, while a value of, say, 0.1 makes them sort of stick to the ground upon impact.

Now, to compute the impact force, \mathbf{F}_i , that will act on the particle during the next time step, making it bounce off the ground, we set \mathbf{F}_i equal to the collision normal vector. The magnitude of the impact force is equal to the impulse, \mathbf{J} , divided by the time step in seconds. The line $\mathbf{F}_i *= \mathbf{J}/_Timestep$; takes care of calculating the final impact force.

To keep the particle from penetrating the ground, we reposition it so that it's just resting on the ground. The y-position is easy to compute as the ground plane elevation plus the radius of the particle. We compute the x-position by linearly interpolating between the particle's previous position and its current position using the newly computed y-position. This effectively backs up the particle along the line of action of its velocity to the point where it is just touching the ground plane.

When you run the simulation now, you'll see the particles fall, drifting a bit from left to right until they hit the ground plane. Once they hit, they'll bounce on the ground, eventually coming to rest. Their specific behavior in this regard depends on what drag coefficient you use and what coefficient of restitution you use. If you have wind applied, when the particles do come to rest, vertically, they should still drift to the right as though they are sliding on the ground plane.

Particle-to-Obstacle Collisions

To make things really interesting, we'll now add those circular obstacles you saw in [Figure 8-1](#) through [Figure 8-4](#). The particles will be able to hit them and bounce off or even settle down into crevasses made by overlapping obstacles. The obstacles are simply static particles. We'll define them as particles and initialize them but then skip them when integrating the equations of motion of the dynamic particles. Here's the declaration for the `Obstacles` array:

```
Particle     Obstacles[_NUM_OBSTACLES];
```

Initializing the obstacles is a matter of assigning them positions and a common radius and a mass. The few lines of code shown next were added to the main program's `Initialize` function to randomly position obstacles in the lower, middle portion of the window above the ground plane. [Figure 8-1](#) through [Figure 8-4](#) illustrate how they are distributed.

```
bool Initialize(void)
{
    .
    .
    .

    for(i=0; i<_NUM_OBSTACLES; i++)
    {
        Obstacles[i].vPosition.x = GetRandomNumber(_WINWIDTH/2 -
            _OBSTACLE_RADIUS*10,
            _WINWIDTH/2 +
            _OBSTACLE_RADIUS*10, false);
    }
}
```

```

        Obstacles[i].vPosition.y = GetRandomNumber(_GROUND_PLANE +
            _OBSTACLE_RADIUS, _WINHEIGHT/2 -
            _OBSTACLE_RADIUS*4, false);
        Obstacles[i].fRadius = _OBSTACLE_RADIUS;
        Obstacles[i].fMass = 100;
    }
    .
    .
}

```

Drawing the obstacles is easy since they are `Particle` types with a `Draw` method that already draws circular shapes. We created `DrawObstacles` to iterate through the `Obstacles` array, calling the `Draw` method of each obstacle.

```

void DrawObstacles(void)
{
    int    i;

    for(i=0; i<_NUM_OBSTACLES; i++)
    {
        Obstacles[i].Draw();
    }

}

```

`DrawObstacles` is then called from `UpdateSimulation`:

```

void  UpdateSimulation(void)
{
    .
    .

// initialize the back buffer
if(FrameCounter >= _RENDER_FRAME_COUNT)
{
    ClearBackBuffer();
    // Draw ground plane
    DrawLine(0, _WINHEIGHT - _GROUND_PLANE,
        _WINWIDTH, _WINHEIGHT - _GROUND_PLANE,
        3, RGB(0,0,0));

    DrawObstacles();
}
.
.
.
}

```

The last bit of code we need to add to have fully functioning collisions with obstacles involves adding more collision detection and handling code to the `CheckForCollisions` function. Before we look at `CheckForCollisions`, let's consider colliding circles in general to gain a better understanding of what the new code will do.

Figure 8-6 illustrates two circles colliding. We aim to detect whether or not these circles are colliding by checking the distance between their centers. If the distance between the two centers is greater than the sum of the radii of the circles, then the particles are not colliding. The topmost illustration in **Figure 8-6** shows the distance, d , between centers and the distance, s , between the edges of the circles; s is the gap between the two. Another way to think about this is that if s is positive, then there's no collision. Referring to the middle illustration in **Figure 8-6**, if s is equal to 0, then the circles are in contact. If s is a negative number, as shown in the bottommost illustration, then the circles are penetrating.

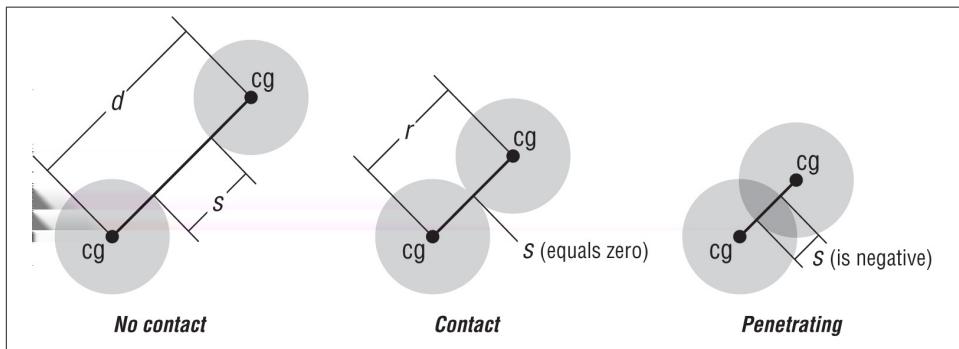


Figure 8-6. Collision states

We'll apply these principles for detecting colliding circles to detecting collisions between our particles and obstacles since they are both circles. **Figure 8-7** illustrates how our particle-to-obstacle collisions might look.

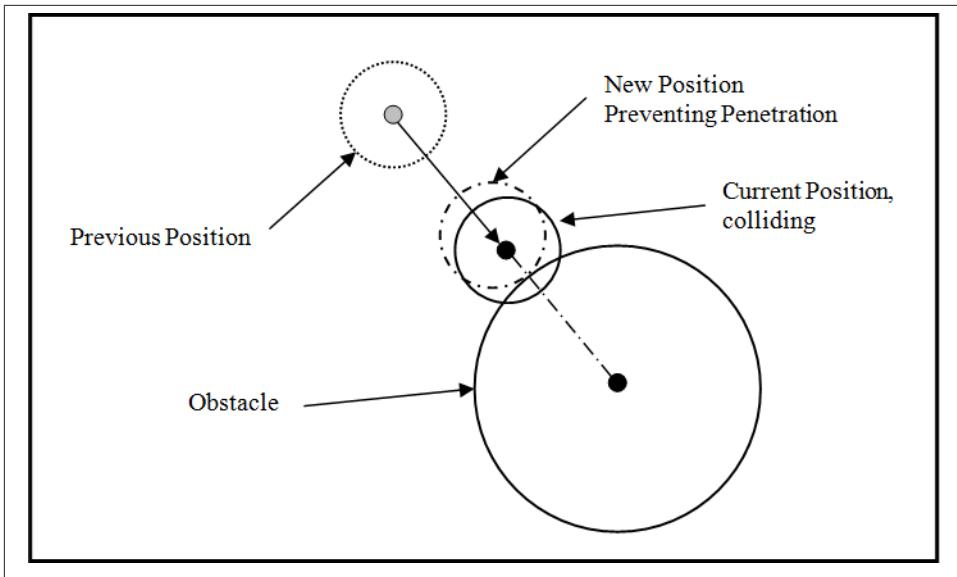


Figure 8-7. Particle-to-obstacle collision

We'll calculate s for each particle against each obstacle to determine contact or penetration. If we find either, then we'll perform the relative velocity check in the same manner we did for particle-to-ground collisions to see if the particle is moving toward the obstacle. If it is, then we have a collision and we'll back up the particle along the collision normal vector line of action, which is simply the line connecting the centers of the particle and the obstacle. We'll also compute the impact force like we did earlier and let the integrator take care of the rest.

OK, now let's look at the new code in `CheckForCollisions`:

```
bool CheckForCollisions(Particle* p)
{
    .
    .
    .

    // Check for collisions with obstacles
    float r;
    Vector d;
    float s;

    for(i=0; i<_NUM_OBSTACLES; i++)
    {
        r = p->fRadius + Obstacles[i].fRadius;
        d = p->vPosition - Obstacles[i].vPosition;
        s = d.Magnitude() - r;
```

```

if(s <= 0.0)
{
    d.Normalize();
    n = d;
    vr = p->vVelocity - Obstacles[i].vVelocity;
    vrn = vr*n;

    if(vrn < 0.0)
    {
        J = -(vr*n) * (_RESTITUTION + 1) /
            (1/p->fMass + 1/Obstacles[i].fMass);
        Fi = n;
        Fi *= J/_Timestep;
        p->vImpactForces += Fi;

        p->vPosition -= n*s;
        hasCollision = true;
    }
}
.
.
.
}

```

The new code is nearly the same as the code that checks for and handles particle-to-ground collisions. The only major differences are in how we compute the distance between the particle and the obstacle and how we adjust the colliding particle's position to prevent it from penetrating an obstacle, since the unit normal vector may not be straight up as it was before. The rest of the code is the same, so let's focus on the differences.

As explained earlier and illustrated in [Figure 8-6](#), we need to compute the separation, s , between the particle and the obstacle. So to get s , we declare a variable r and equate it to the sum of radii of the particle and the obstacle against which we're checking for a collision. We define d , a `Vector`, as the difference between the positions of the particle and obstacle. The magnitude of d minus r yields s .

If s is less than 0, then we make the relative velocity check. Now, in this case the collision normal vector is along the line connecting the centers of the two circles representing the particle and the obstacle. Well, that's just the vector d we already calculated. To get the unit normal vector, we simply normalize d . The relative velocity vector is simply the difference in velocities of the particle and the obstacle. Since the obstacles are static, the relative velocity is just the particle's velocity. But we calculated the relative velocity by taking the vector difference $vr = p->vVelocity - Obstacles[i].vVelocity$, because in a more advanced scenario, you might have moving obstacles.

Taking the dot product of the relative velocity vector, \mathbf{v}_r , with the unit normal vector yields the relative velocity in the collision normal direction. If that relative velocity is less than 0, then the particle and the object are colliding and the code goes on to calculate the impact force in a manner similar to that described earlier for the particle-to-ground collisions. The only difference here is that both the particle's and object's masses appear in the impulse formula. Earlier we assumed the ground was infinitely massive relative to the particle's mass, so the $1/m$ term for the ground went to 0, essentially dropping out of the equation. Refer back to [Chapter 5](#) to recall the impulse formulas.

Once the impact force is calculated, the code backs up the particle by a distance equal to s , the penetration, along the line of action of the collision normal vector, giving us what we desire (as shown in [Figure 8-7](#)).

Now, when you run this simulation you'll see the particles falling down, bouncing off the obstacles or flowing around them depending on the value you're using for coefficient of restitution, ultimately bouncing and coming to rest on the ground plane. If you have a wind speed greater than 0, then the particles will still drift along the ground plane from left to right.

Tuning

Tuning is an important part of developing any simulator. Tuning means different things to different people. For some, tuning is adjusting formulas and coefficients to make your simulation match some specific “right answer,” while to others tuning is adjusting parameters to make the simulation look and behave how you want it to, whether or not it’s technically the right answer. After all, the right answer for a game is that it’s fun and robust. Speaking of robustness, other folks view tuning in the sense of adjusting parameters to make the simulation stable. In reality, this is all tuning and you should think of it as a necessary part of developing your simulation. It’s the process by which you tweak, nudge, and adjust things to make the simulation do what you want it to do.

For example, you can use this same example simulation to model very springy rubber balls. To achieve this, you’ll probably adjust the coefficient of restitution toward a value approaching 1 and perhaps lower the drag coefficient. The particles will bounce all over the place with a lot of energy. If, on the other hand, you want to model something along the lines of mud, then you’ll lower the coefficient of restitution and increase the drag coefficient. There is no right or wrong combination of coefficient of restitution or drag coefficient to use, so long as you are pleased with the results.

Another aspect you might tune is the number of simulation frames per rendering frame. You may find the simulation calculations take so long that your resulting animations are too jerky because you aren’t updating the display enough. The converse may be true in other cases. An important parameter that plays into this is the time step size you take at each simulation iteration. If the step size is too small, you’ll perform a lot of simulation

steps, slowing the animation down. On the other hand, a small time step can keep the simulation numerically stable. Your chosen integrator plays a huge role here.

If you make your time step too large, the simulation may just blow up and not work. It will be numerically unstable. Even if it doesn't blow up, you might see weird results. For example, if the time step in the example simulation discussed in this chapter is too large, then particles may completely step over obstacles in a single time step, missing the collision that would otherwise have happened. (We'll show you in [Chapter 10](#) how to deal with that situation.)

In general, tuning is a necessary part of developing physics-based simulations, and we encourage you to experiment—trying different combinations of parameters to see what results you can achieve. You should even try combinations that may break the example in this chapter to see what happens and what you should try to avoid in a deployed game.

2D Rigid-Body Simulator

After reading [Chapter 8](#), you've learned the main ingredients that go into a simulator, specifically a particle simulator. In this chapter we'll look beyond particles at 2D rigid bodies. The main difference here is that rigid bodies rotate, and you must deal with an additional equation of motion—namely, the angular equation of motion relating a rigid body's angular acceleration and inertia to the sum of all moments (torques) acting on the rigid body. The fundamental elements of the simulator—the model, integrator, renderer, etc.—are the same as before; you just have to deal with rotation. In two dimensions, handling rotation is simple. Things get a bit more involved when handling rotation in three dimensions, and we'll treat that problem in [Chapter 11](#).

The example we'll take a close look at in this chapter is simple by design. We want to focus on the differences between the particle simulator and a 2D rigid-body simulator. In [Chapter 10](#), we'll extend this simple example to deal with multiple rigid bodies and collisions. That's where things really get interesting. For now, we'll consider a single rigid body, a virtual hovercraft, that moves around the screen under the influences of thrust forces that you can control with the keyboard. While simple, this example covers the most fundamental aspects of simulating 2D rigid bodies.

[Figure 9-1](#) shows our virtual hovercraft. The pointy end is the front, and the hovercraft will start off moving from the left side of the screen to the right. Using the arrow keys, you can increase or decrease its speed and make it turn left or right (port or starboard).

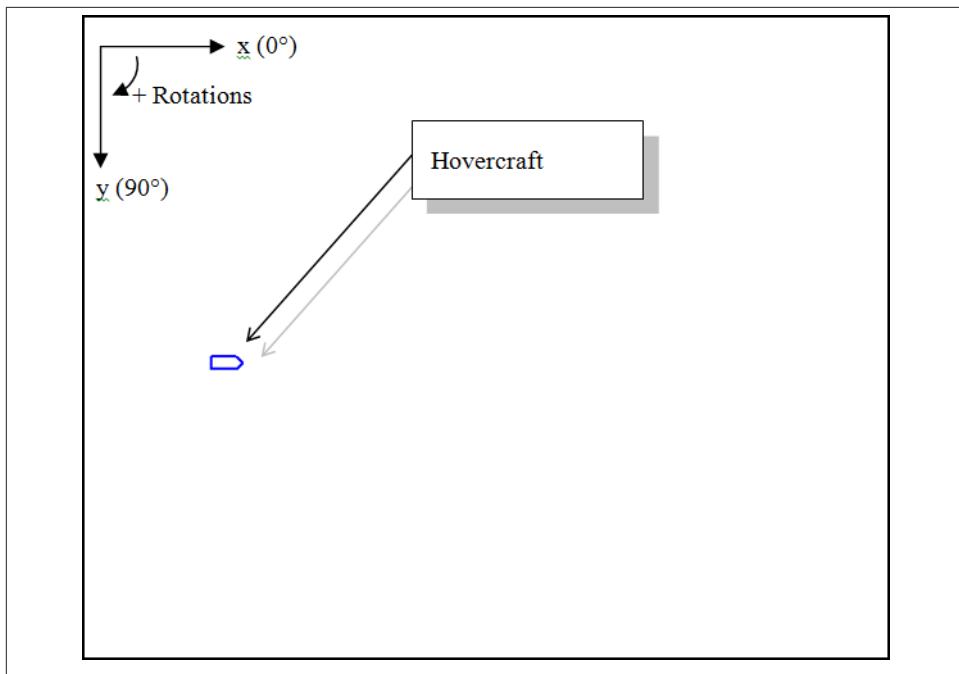


Figure 9-1. 2D rigid-body example

In this simulation, the world coordinate system has its x-axis pointing to the right, its y-axis pointing down toward the bottom of the screen, and the z-axis pointing into the screen. Even though this is a 2D example where all motion is confined to the x-y plane, you still need a z-axis about which the hovercraft will rotate. Also, the local, or body-fixed, coordinate system has its x-axis pointing toward the front of the hovercraft, its y-axis pointing to the starboard side, and its z-axis into the screen. The local coordinate system is fixed to the rigid body at its center of gravity location.

Model

The hovercraft modeled in this simulation is a simplified version of the hovercraft we'll model in [Chapter 17](#). You can refer to [Chapter 17](#) for more details on that model. For convenience we repeat some of the basic properties of the model here. [Figure 9-2](#) illustrates the main features of the model.

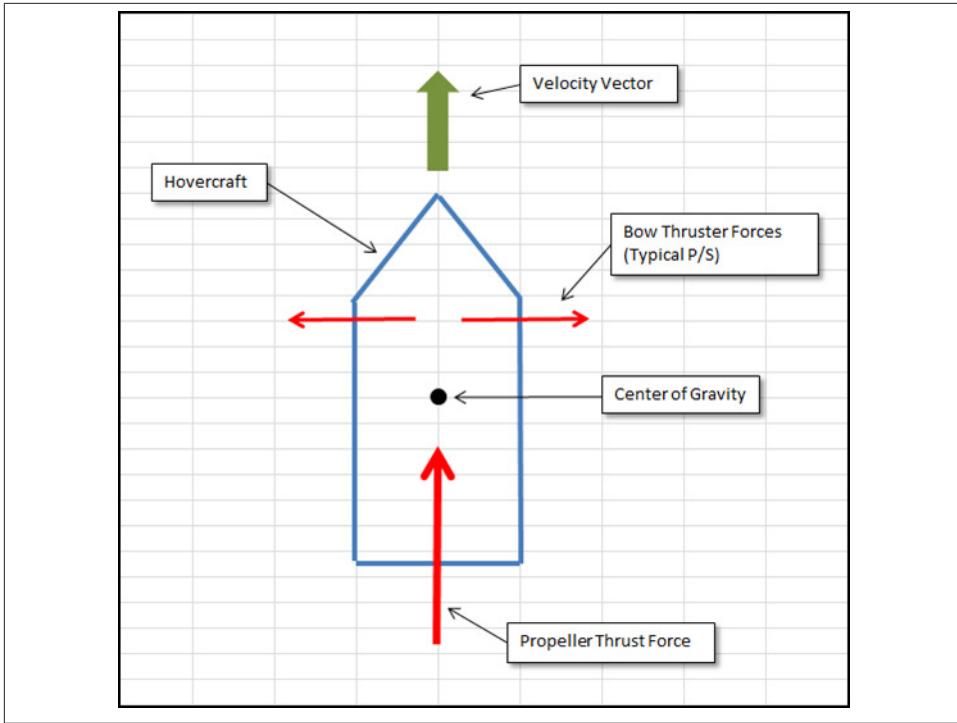


Figure 9-2. Simple hovercraft model

We're assuming this hovercraft operates over smooth land and is fitted with a single airscrew propeller, located toward the aft end of the craft, that provides forward thrust. For controllability, the craft is fitted with two bow thrusters, one to port and the other to starboard. These bow thrusters are used to steer the hovercraft.

We use a simplified drag model where the only drag component is due to aerodynamic drag on the entire craft with a constant projected area. This model is similar to the one used in [Chapter 8](#) for particle drag. A more rigorous model would consider the actual projected area of the craft as a function of the direction of relative velocity, as in the flight simulator example discussed in [Chapter 15](#), as well as the frictional drag between the bottom of the craft's skirt and the ground. We also assume that the center of drag—the point through which we can assume the drag force vector is applied—is located some distance aft of the center of gravity so as to give a little directional stability (that is, to counteract rotation). This serves the same function as the vertical tail fins on aircraft. Again, a more rigorous model would include the effects of rotation on aerodynamic drag, but we ignore that here.

In code, the first thing you need to do to represent this vehicle is define a rigid-body class that contains all of the information you'll need to track it and calculate the forces

and moments acting on it. This `RigidBody2D` class is very similar to the `Particle` class from [Chapter 8](#), but with some additions mostly dealing with rotation. Here's how we did it:

```
class RigidBody2D {
public:
    float    fMass;          // total mass (constant)
    float    fInertia;        // mass moment of inertia
    float    fInertiaInverse; // inverse of mass moment of inertia
    Vector   vPosition;      // position in earth coordinates
    Vector   vVelocity;      // velocity in earth coordinates
    Vector   vVelocityBody;  // velocity in body coordinates
    Vector   vAngularVelocity; // angular velocity in body coordinates

    float    fSpeed;         // speed
    float    fOrientation;   // orientation

    Vector   vForces;        // total force on body
    Vector   vMoment;        // total moment on body

    float    ThrustForce;    // Magnitude of the thrust force
    Vector   PThrust, SThrust; // bow thruster forces

    float    fWidth;         // bounding dimensions
    float    fLength;
    float    fHeight;

    Vector   CD; // location of center of drag in body coordinates
    Vector   CT; // location of center of propeller thrust in body coords.
    Vector   CPT; // location of port bow thruster thrust in body coords.
    Vector   CST; // location of starboard bow thruster thrust in body
                  // coords.

    float    ProjectedArea;  // projected area of the body

    RigidBody2D(void);
    void    CalcLoads(void);
    void    UpdateBodyEuler(double dt);
    void    SetThrusters(bool p, bool s);
    void    ModulateThrust(bool up);
};
```

The code comments briefly explain each property, and so far you've seen all these properties somewhere in this book, so we won't explain them again here. That said, notice that several of these properties are the same as those shown in the `Particle` class from [Chapter 8](#). These properties include `fMass`, `vPosition`, `vVelocity`, `fSpeed`, `vForces`, and `fRadius`. All of the other properties are new and required to handle the rotational motion aspects of rigid bodies.

The RigidBody2D constructor is straightforward, as shown next, and simply initializes all the properties to some arbitrarily tuned values we decided worked well. In [Chapter 17](#), you'll see how we model a more realistic hovercraft.

```
RigidBody2D::RigidBody2D(void)
{
    fMass = 100;
    fInertia = 500;
    fInertiaInverse = 1/fInertia;
    vPosition.x = 0;
    vPosition.y = 0;
    fWidth = 10;
    fLength = 20;
    fHeight = 5;
    fOrientation = 0;

    CD.x = -0.25*fLength;
    CD.y = 0.0f;
    CD.z = 0.0f;

    CT.x = -0.5*fLength;
    CT.y = 0.0f;
    CT.z = 0.0f;

    CPT.x = 0.5*fLength;
    CPT.y = -0.5*fWidth;
    CPT.z = 0.0f;

    CST.x = 0.5*fLength;
    CST.y = 0.5*fWidth;
    CST.z = 0.0f;

    ProjectedArea = (fLength + fWidth)/2 * fHeight; // an approximation
    ThrustForce = _THRUSTFORCE;
}
```

As in the particle simulator of [Chapter 8](#), you'll notice here that the Vector class is actually a *tuple* (that is, it has three components—x, y, and z). Since this is a 2D example, the z components will always be 0, except in the case of the angular velocity vector where only the z component will be used (since rotation occurs only about the z-axis). The class that we use in this example is discussed in [Appendix A](#). The reason we didn't write a separate 2D vector class, one with only x and y components, is because we'll extend this code to 3D later and wanted to get you accustomed to using the 3D vector class. Besides, it's pretty easy to create a 2D vector class from the 3D class by simply stripping out the z component.

As with the particle example of [Chapter 8](#), we need a CalcLoads method for the rigid body. As before, this method will compute all the loads acting on the rigid body, but now these loads include both forces and moments that will cause rotation. CalcLoads looks like this:

```

void    RigidBody2D::CalcLoads(void)
{
    Vector    Fb;           // stores the sum of forces
    Vector    Mb;           // stores the sum of moments
    Vector    Thrust;        // thrust vector

    // reset forces and moments:
    vForces.x = 0.0f;
    vForces.y = 0.0f;
    vForces.z = 0.0f;      // always zero in 2D

    vMoment.x = 0.0f;      // always zero in 2D
    vMoment.y = 0.0f;      // always zero in 2D
    vMoment.z = 0.0f;

    Fb.x = 0.0f;
    Fb.y = 0.0f;
    Fb.z = 0.0f;

    Mb.x = 0.0f;
    Mb.y = 0.0f;
    Mb.z = 0.0f;

    // Define the thrust vector, which acts through the craft's CG
    Thrust.x = 1.0f;
    Thrust.y = 0.0f;
    Thrust.z = 0.0f;      // zero in 2D
    Thrust *= ThrustForce;

    // Calculate forces and moments in body space:
    Vector    vLocalVelocity;
    float     fLocalSpeed;
    Vector    vDragVector;
    float     tmp;
    Vector    vResultant;
    Vector    vtmp;

    // Calculate the aerodynamic drag force:
    // Calculate local velocity:
    // The local velocity includes the velocity due to
    // linear motion of the craft,
    // plus the velocity at each element
    // due to the rotation of the craft.

    vtmp = vAngularVelocity^CD; // rotational part
    vLocalVelocity = vVelocityBody + vtmp;

    // Calculate local air speed
    fLocalSpeed = vLocalVelocity.Magnitude();

    // Find the direction in which drag will act.
    // Drag always acts in line with the relative

```

```

// velocity but in the opposing direction
if(fLocalSpeed > tol)
{
    vLocalVelocity.Normalize();
    vDragVector = -vLocalVelocity;

    // Determine the resultant force on the element.
    tmp = 0.5f * rho * fLocalSpeed*fLocalSpeed
        * ProjectedArea;
    vResultant = vDragVector * _LINEARDRAGCOEFFICIENT * tmp;

    // Keep a running total of these resultant forces
    Fb += vResultant;

    // Calculate the moment about the CG
    // and keep a running total of these moments

    vtmp = CD^vResultant;
    Mb += vtmp;
}

// Calculate the Port & Starboard bow thruster forces:
// Keep a running total of these resultant forces

Fb += PThrust;

// Calculate the moment about the CG of this element's force
// and keep a running total of these moments (total moment)
vtmp = CPT^PThrust;
Mb += vtmp;

// Keep a running total of these resultant forces (total force)
Fb += SThrust;

// Calculate the moment about the CG of this element's force
// and keep a running total of these moments (total moment)
vtmp = CST^SThrust;
Mb += vtmp;

// Now add the propulsion thrust
Fb += Thrust; // no moment since line of action is through CG

// Convert forces from model space to earth space
vForces = VRotate2D(fOrientation, Fb);

vMoment += Mb;
}

```

The first thing that `CalcLoads` does is initialize the force and moment variables that will contain the total of all forces and moments acting on the craft at any instant in time. Just as we must aggregate forces, we must also aggregate moments. The forces will be

used along with the linear equation of motion to compute the linear displacement of the rigid body, while the moments will be used with the angular equation of motion to compute the orientation of the body.

The function then goes on to define a vector representing the propeller thrust, `Thrust`. The propeller thrust vector acts in the positive (local) x-direction and has a magnitude defined by `ThrustForce`, which the user sets via the keyboard interface (we'll get to that later). Note that if `ThrustForce` is negative, then the thrust will actually be a reversing thrust instead of a forward thrust.

After defining the thrust vector, this function goes on to calculate the aerodynamic drag acting on the hovercraft. These calculations are very similar to those discussed in [Chapter 17](#). The first thing to do is determine the relative velocity at the center of drag, considering both linear and angular motion. You'll need the magnitude of the relative velocity vector when calculating the magnitude of the drag force, and you'll need the direction of the relative velocity vector to determine the direction of the drag force since it always opposes the velocity vector. The line `vtmp = vAngularVelocity^CD` computes the linear velocity at the drag center by taking the vector cross product of the angular velocity vector with the position vector of the drag center, `CD`. The result is stored in a temporary vector, `vtmp`, and then added vectorially to the body velocity vector, `vVelo` `cityBody`. The result of this vector addition is a velocity vector representing the velocity of the point defined by `CD`, including contributions from the body's linear and angular motion. We compute the actual drag force, which acts in line with but in a direction opposing the velocity vector, in a manner similar to that for particles, using a simple formula relating the drag force to the speed squared, density of air, projected area, and a drag coefficient. The following code performs this calculation:

```
vLocalVelocity.Normalize();
vDragVector = -vLocalVelocity;

// Determine the resultant force on the element.
tmp = 0.5f * rho * fLocalSpeed*fLocalSpeed
     * ProjectedArea;
vResultant = vDragVector * _LINEARDRAGCOEFFICIENT * tmp;
```

Note that the drag coefficient, `LINEARDRAGCOEFFICIENT`, is defined as follows:

```
#define LINEARDRAGCOEFFICIENT      1.25f
```

Once the drag force is determined, it gets aggregated in the total force vector as follows:

```
Fb += vResultant;
```

In addition to aggregating this force, we must aggregate the moment due to that force in the total moment vector as follows:

```
vtmp = CD^vResultant;
Mb += vtmp;
```

The first line computes the moment due to the drag force by taking the vector cross product of the position vector, to the center of drag, with the drag force vector. The second line adds this force to the variable, accumulating these moments.

With the drag calculation complete, `CalcLoads` proceeds to calculate the forces and moments due to the bow thrusters, which may be active or inactive at any given time.

```
Fb += PThrust;  
  
vtmp = CPT^PThrust;  
Mb += vtmp;
```

The first line aggregates the port bow thruster force into `Fb`. `PThrust` is a force vector computed in the `SetThrusters` method in response to your keyboard input. The next two lines compute and aggregate the moment due to the thruster force. A similar set of code lines follows, computing the force and moment due to the starboard bow thruster.

Next, the propeller thrust force is added to the running total of forces. Remember, since the propeller thrust force acts through the center of gravity, there is no moment to worry about. Thus, all we need is:

```
Fb += Thrust; // no moment since line of action is through CG
```

Finally, the total force is transformed from local coordinates to world coordinates via a vector rotation given the orientation of the hovercraft, and the total forces and moments are stored so they are available when it comes time to integrate the equations of motion at each time step.

As you can see, computing loads on a rigid body is a bit more complex than what you saw earlier when dealing with particles. This, of course, is due to the nature of rigid bodies being able to rotate. What's nice, though, is that all this new complexity is encapsulated in `CalcLoads`, and the rest of the simulator is pretty much the same as when we're dealing with particles.

Transforming Coordinates

Let's talk about transformation from local to world coordinates a bit more since you'll see this sort of transform again in a few places. When computing forces acting on the rigid body, we want those forces in a vector form relative to the coordinates that are fixed with respect to the hovercraft (e.g., relative to the body's center of gravity with the x-axis pointing toward the front of the body and the y-axis pointing toward the starboard side). This simplifies our calculations of forces and moments. However, when integrating the equation of motion to see how the body translates in world coordinates, we use the equations of motion in world coordinates, requiring us to represent the aggregate force in world coordinates. That's why we rotated the aggregate force at the end of the `CalcLoads` method.

In two dimensions, the coordinate transformation involves a little trigonometry as shown in the following `VRotate2D` function:

```
Vector VRotate2D( float angle, Vector u)
{
    float x,y;

    x = u.x * cos(DegreesToRadians(-angle)) +
        u.y * sin(DegreesToRadians(-angle));
    y = -u.x * sin(DegreesToRadians(-angle)) +
        u.y * cos(DegreesToRadians(-angle));

    return Vector( x, y, 0);
}
```

The angle here represents the orientation of the local, body fixed coordinate system with respect to the world coordinate system. When converting from local coordinates to world coordinates, use a positive angle; use a negative angle when going the other way. This is just the convention we've adopted so transformations from local coordinates to world coordinates are positive. You can see we actually take the negative of the `angle` parameter, so in reality you could do away with that negative, and then transformations from local coordinates to world coordinates would actually be negative. It's your preference. You'll see this function used a few more times in different situations before the end of this chapter.

Integrator

The `UpdateBodyEuler` method actually integrates the equations of motion for the rigid body. Since we're dealing with a rigid body, unlike a particle, we have two equations of motion: one for translation, and the other for rotation. The following code sample shows `UpdateBodyEuler`.

```
void RigidBody2D::UpdateBodyEuler(double dt)
{
    Vector a;
    Vector dv;
    Vector ds;
    float aa;
    float dav;
    float dr;

    // Calculate forces and moments:
    CalcLoads();

    // Integrate linear equation of motion:
    a = vForces / fMass;

    dv = a * dt;
    vVelocity += dv;
```

```

ds = vVelocity * dt;
vPosition += ds;

// Integrate angular equation of motion:
aa = vMoment.z / fInertia;

dav = aa * dt;

vAngularVelocity.z += dav;

dr = RadiansToDegrees(vAngularVelocity.z * dt);
fOrientation += dr;

// Misc. calculations:
fSpeed = vVelocity.Magnitude();
vVelocityBody = VRotate2D(-fOrientation, vVelocity);
}

```

As the name of this method implies, we've implemented Euler's method of integration as described in [Chapter 7](#). Integrating the linear equation of motion for a rigid body follows exactly the same steps we used for integrating the linear equation of motion for particles. All that's required is to divide the aggregate forces acting on a body by the mass of the body to get the body's acceleration. The line of code `a = vForces / fMass` does just this. Notice here that `a` is a `Vector`, as is `vForces`. `fMass` is a scalar, and the `/` operator defined in the `Vector` class takes care of dividing each component of the `vForces` vector by `fMass` and setting the corresponding components in `a`. The change in velocity, `dv`, is equal to acceleration times the change in time, `dt`. The body's new velocity is then computed by the line `vVelocity += dv`. Here again, `vVelocity` and `dv` are `Vectors` and the `+=` operator takes care of the vector arithmetic. This is the first actual integration for translation.

The second integration takes place in the next few lines, where we determine the body's displacement and new position by integrating its velocity. The line `ds = vVelocity * dt` determines the displacement, or change in the body's position, and the line `vPosition += ds` computes the new position by adding the displacement to the body's old position. That's it for translation.

The next order of business is to integrate the angular equation of motion to find the body's new orientation. The line `aa = vMoment.z / fInertia;` computes the body's angular acceleration by dividing the aggregate moment acting on the body by its mass moment of inertia. `aa` is a scalar, as is `fInertia` since this is a 2D problem. In 3D, things are a bit more complicated, and we'll get to that in [Chapter 11](#).

We compute the change in angular velocity, `dav`, a scalar, by multiplying `aa` by the time step size, `dt`. The new angular velocity is simply the old velocity plus the change: `vAngularVelocity.z += dav`. The change in orientation is equal to the new angular velocity multiplied by the time step: `vAngularVelocity.z * dt`. Notice that we convert the

change in orientation from radians to degrees here since we're keeping track of orientation in degrees. You don't really have to, so long as you're consistent.

The last line in `UpdateBodyEuler` computes the body's linear speed by transforming the magnitude of its velocity vector to local, body coordinates. Recall in `CalcLoads` that we require the body's velocity in body-fixed coordinates in order to compute the drag force on the body.

Rendering

In this simple example, rendering the virtual hovercraft is just a little more involved than rendering the particles in the example from [Chapter 8](#). All we do is draw a few connected lines using Windows API calls wrapped in our own functions to hide some of the Windows-specific code. The following code snippet is all we need to render the hovercraft:

```
void DrawCraft(RigidBody2D craft, COLORREF clr)
{
    Vector vList[5];
    double wd, lg;
    int i;
    Vector v1;

    wd = craft.fWidth;
    lg = craft.fLength;
    vList[0].x = lg/2;    vList[0].y = wd/2;
    vList[1].x = -lg/2;   vList[1].y = wd/2;
    vList[2].x = -lg/2;   vList[2].y = -wd/2;
    vList[3].x = lg/2;    vList[3].y = -wd/2;
    vList[4].x = lg/2*1.5; vList[4].y = 0;
    for(i=0; i<5; i++)
    {
        v1 = VRotate2D(craft.fOrientation, vList[i]);
        vList[i] = v1 + craft.vPosition;
    }

    DrawLine(vList[0].x, vList[0].y, vList[1].x, vList[1].y, 2, clr);
    DrawLine(vList[1].x, vList[1].y, vList[2].x, vList[2].y, 2, clr);
    DrawLine(vList[2].x, vList[2].y, vList[3].x, vList[3].y, 2, clr);
    DrawLine(vList[3].x, vList[3].y, vList[4].x, vList[4].y, 2, clr);
    DrawLine(vList[4].x, vList[4].y, vList[0].x, vList[0].y, 2, clr);
}
```

You can use your own rendering code here, of course, and all you really need to pay close attention to is transforming the coordinates for the outline of the hovercraft from body to world coordinates. This involves rotating the vertex coordinates from body-fixed space using the `VRotate2D` function and then adding the position of the center of gravity of the hovercraft to each transformed vertex. These lines take care of this coordinate transformation:

```

for(i=0; i<5; i++)
{
    v1 = VRotate2D(craft.fOrientation, vList[i]);
    vList[i] = v1 + craft.vPosition;
}

```

The Basic Simulator

The heart of this simulation is handled by the `RigidBody2D` class described earlier. However, we need to show you how that class is used in the context of the main program. This simulator is very similar to that shown in [Chapter 8](#) for particles, so if you've read that chapter already you can breeze through this section.

First, we define a few global variables as follows:

```

// Global Variables:
int           FrameCounter = 0;
RigidBody2D   Craft;

```

`FrameCounter` counts the number of time steps integrated before the graphics display is updated. How many time steps you allow the simulation to integrate before updating the display is a matter of tuning. You'll see how this is used momentarily when we discuss the `UpdateSimulation` function. `Craft` is a `RigidBody2D` type that will represent our virtual hovercraft.

For the most part, `Craft` is initialized in accordance with the `RigidBody2D` constructor shown earlier. However, its position is at the origin, so we make a call to the following `Initialize` function to locate the `Craft` in the middle of the screen vertically and on the left side. We set its orientation to 0 degrees so it points toward the right side of the screen:

```

bool   Initialize(void)
{
    Craft.vPosition.x = _WINWIDTH/10;
    Craft.vPosition.y = _WINHEIGHT/2;
    Craft.fOrientation = 0;

    return true;
}

```

OK, now let's consider `UpdateSimulation` as shown in the code snippet below. This function gets called every cycle through the program's main message loop and is responsible for making appropriate function calls to update the hovercraft's position and orientation, as well as rendering the scene. It also checks the states of the keyboard arrow keys and makes appropriate function calls:

```

void   UpdateSimulation(void)
{
    double dt = _Timestep;
    RECT   r;

```

```

Craft.SetThrusters(false, false);

if (IsKeyDown(VK_UP))
    Craft.ModulateThrust(true);

if (IsKeyDown(VK_DOWN))
    Craft.ModulateThrust(false);

if (IsKeyDown(VK_RIGHT))
    Craft.SetThrusters(true, false);

if (IsKeyDown(VK_LEFT))
    Craft.SetThrusters(false, true);

// update the simulation
Craft.UpdateBodyEuler(dt);

if(FrameCounter >= _RENDER_FRAME_COUNT)
{
    // update the display
    ClearBackBuffer();

    DrawCraft(Craft, RGB(0,0,255));

    CopyBackBufferToWindow();
    FrameCounter = 0;
} else
    FrameCounter++;

if(Craft.vPosition.x > _WINWIDTH) Craft.vPosition.x = 0;
if(Craft.vPosition.x < 0) Craft.vPosition.x = _WINWIDTH;
if(Craft.vPosition.y > _WINHEIGHT) Craft.vPosition.y = 0;
if(Craft.vPosition.y < 0) Craft.vPosition.y = _WINHEIGHT;
}

```

The local variable `dt` represents the small yet finite amount of time, in seconds, over which each integration step is taken. The global define `_Timestep` stores the time step, which we have set to 0.001 seconds. This value is subject to tuning.

The first action `UpdateSimulation` takes is to reset the states of the bow thrusters to inactive by calling the `SetThrusters` method as follows:

```
Craft.SetThrusters(false, false);
```

Next, the keyboard is polled using the function `IsKeyDown`. This is a wrapper function we created to encapsulate the necessary Windows API calls used to check key states. If the up arrow key is pressed, then the `RigidBody2D` method `ModulateThrust` is called, as shown here:

```
Craft.ModulateThrust(true);
```

If the down arrow key is pressed, then `ModulateThrust` is called, passing `false` instead of `true`.

`ModulateThrust` looks like this:

```
void RigidBody2D::ModulateThrust(bool up)
{
    double dT = up ? _DTHRUST:-_DTHRUST;

    ThrustForce += dT;

    if(ThrustForce > _MAXTHRUST) ThrustForce = _MAXTHRUST;
    if(ThrustForce < _MINTHRUST) ThrustForce = _MINTHRUST;
}
```

All it does is increment the propeller thrust force by a small amount, either increasing it or decreasing it, depending on the value of the `up` parameter.

Getting back to `UpdateSimulation`, we make a couple more calls to `IsKeyDown`, checking the states of the left and right arrow keys. If the left arrow key is down, then the `RigidBody2D` method `SetThrusters` is called, passing `false` as the first parameter and `true` as the second parameter. If the right arrow key is down, these parameter values are reversed. `SetThrusters` looks like this:

```
void RigidBody2D::SetThrusters(bool p, bool s)
{
    PThrust.x = 0;
    PThrust.y = 0;
    SThrust.x = 0;
    SThrust.y = 0;

    if(p)
        PThrust.y = _STEERINGFORCE;
    if(s)
        SThrust.y = -_STEERINGFORCE;
}
```

It resets the port and starboard bow thruster thrust vectors and then sets them according to the parameters passed in `SetThrusters`. If `p` is `true`, then a right turn is desired and a port thrust force, `PThrust`, is created, pointing toward the starboard side. This seems opposite of what you'd expect, but it is the port bow thruster that is fired, pushing the bow of the hovercraft toward the right (starboard) side. Similarly, if `s` is `true`, a thrust force is created that will push the bow of the hovercraft to the left (port) side.

Now with the thrust forces managed, `UpdateSimulation` makes the call:

```
Craft.UpdateBodyEuler(dt)
```

`UpdateBodyEuler` integrates the equations of motion as discussed earlier.

The next segment of code checks the value of the frame counter. If the frame counter has reached the defined number of frames (stored in `_RENDER_FRAME_COUNT`), then the

back buffer is cleared to prepare it for drawing upon and ultimately copying to the screen.

Finally, the last four lines of code wrap the hovercraft's position around the edges of the screen.

Tuning

You'll probably want to tune this example to run well on your computer since we didn't implement any profiling for processor speed. Moreover, you should tune the various parameters governing the behavior of the hovercraft to see how it responds. The way we have it set up now makes the hovercraft exhibit a soft sort of response to turning—that is, upon application of turning forces, the craft will tend to keep tracking in its original heading for a bit even while yawed. It will not respond like a car would turn. You can change this behavior, of course.

Some things we suggest you play with include the time step size and the various constants we've defined as follows:

```
#define _THRUSTFORCE      5.0f
#define _MAXTHRUST        10.0f
#define _MINTHRUST        0.0f
#define _DTHRUST          0.001f
#define _STEERINGFORCE    3.0f
#define _LINEARDRAGCOEFFICIENT 1.25f
```

`_THRUSTFORCE` is the initial magnitude of the propeller thrust force. `_MAXTHRUST` and `_MINTHRUST` set upper and lower bounds to this force, which is modulated by the user pressing the up and down arrow keys. `_DTHRUST` is the incremental change in thrust in response to the user pressing the up and down arrow keys. `_STEERINGFORCE` is the magnitude of the bow thruster forces. You should definitely play with this value to see how the behavior of the hovercraft changes. Finally, `_LINEARDRAGCOEFFICIENT` is the drag coefficient used to compute aerodynamic drag. This is another good value to play with to see how behavior is affected. Speaking of drag, the location of the center of drag that's initialized in the `RigidBody2D` constructor is a good parameter to change in order to understand how it affects the behavior of the hovercraft. It influences the craft's directional stability, which affects its turning radius—particularly at higher speeds.

Implementing Collision Response

In this chapter, we'll show you how to add a little excitement to the hovercraft example discussed in the preceding chapter. Specifically, we'll add another hovercraft and show you how to add collision response so that the hovercraft can crash into each other and bounce off like a couple of bumper cars. This is an important element for many types of games, so it's crucial that you understand the code that we'll present here. Now would be a good time to go back and review [Chapter 5](#) to refresh your memory on the fundamentals of rigid-body collision response since we'll use the principles and formulas discussed there to develop the collision response algorithms for the hovercraft simulation. In [Chapter 8](#) you saw how to implement linear collision response for particles, and now we'll show you how to handle angular effects.

To start simply, we'll first show you how to implement collision response as if the hovercraft were a couple of particles just like those in [Chapter 8](#). This approach uses only linear impulse and does not include angular effects, so the results will be somewhat unrealistic for these hovercraft; however, this approach is applicable to other types of problems that you may be interested in (for example, billiard ball collisions). Plus, taking this approach allows us to show you very clearly the distinction between linear and angular effects. Including angular effects will make the simulation much more realistic; when the hovercraft crash into each other, not only will they bounce off each other, but they will also spin.

Before diving into collisions, let's add another hovercraft to the example we started in [Chapter 9](#). Recall in that example, we had a single craft that you could control using the keyboard. Now, we'll add another hovercraft that simply moves under constant forward thrust. Later, when we add collision detection and response you'll be able to run into this new hovercraft to alter its course.

Referring back to the example from [Chapter 9](#), we need to add another craft as follows:

```
RigidBody2D    Craft2;
```

We're calling the new hovercraft, very creatively, `Craft2`. In the `Initialize` function, we must now add the following code:

```
bool Initialize(void)
{
.
.
.
Craft2.vPosition.x = _WINWIDTH/2;
Craft2.vPosition.y = _WINHEIGHT/2;
Craft2.fOrientation = 90;
.
.
.
```

This new code sample positions the second hovercraft in the middle of the screen and pointing toward the bottom.

There are a few required changes to `UpdateSimulation` as well. First, add `Craft2.UpdateBodyEuler(dt);` right after the line `Craft.UpdateBodyEuler(dt);`. Then, add `DrawCraft(Craft2, RGB(200, 200, 0));` after the similar line that draws the first `Craft`. `Craft2` will be drawn yellow to distinguish it from the first `Craft`. Finally, add the following lies at the end of `UpdateSimulation`:

```
if(Craft2.vPosition.x > _WINWIDTH) Craft2.vPosition.x = 0;
if(Craft2.vPosition.x < 0) Craft2.vPosition.x = _WINWIDTH;
if(Craft2.vPosition.y > _WINHEIGHT) Craft2.vPosition.y = 0;
if(Craft2.vPosition.y < 0) Craft2.vPosition.y = _WINHEIGHT;
```

Now, we can add the code to handle collision detection and response, allowing you to ram your hovercraft into the new one we just added.

Linear Collision Response

In this section, we'll show you how to implement simple collision response, assuming that the two hovercraft are particles. We're going to implement only bare-minimum collision detection in this simulation; however, regardless of the level of sophistication of your collision detection routines, there are very specific pieces of information that you must collect from your collision detection routine(s) in order for your physics-based collision response routines to work.

To revise the hovercraft example of the previous chapter to include simple collision response, you'll have to modify the `UpdateSimulation` function and add a couple more functions: `CheckForCollision` and `ApplyImpulse`.

Before showing you `CheckForCollision`, we want to explain what your collision detection function must do. First, it must let you know whether or not there is a collision occurring between the hovercraft. Secondly, it must let you know if the hovercraft are

penetrating each other. Thirdly, if the hovercraft are colliding, it must tell you what the collision normal vector is and what the relative velocity is between the colliding hovercraft.

To determine whether or not there is a collision, you need to consider two factors:

- Whether or not the objects are close enough, within numerical tolerances, to be considered in colliding contact
- What the relative normal velocity is between the objects

If the objects aren't close to each other, they obviously have not collided. If they are within your tolerance for contact, then they may be colliding; and if they are touching and overlapping such that they are moving inside each other, they are penetrating, as illustrated in [Figure 10-1](#). If your collision detection routine finds that the two objects are indeed close enough to be in colliding contact, then you have to do another check on the relative normal velocity to see if they are moving away from each other or toward each other. A collision occurs when the objects are in contact and the contact points are moving toward each other.

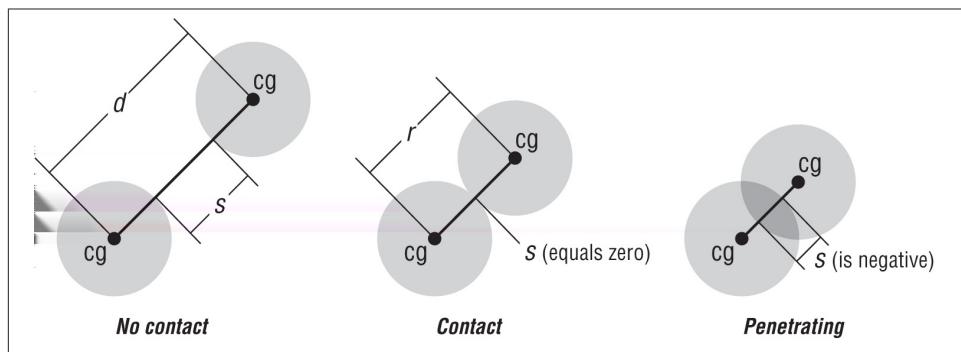


Figure 10-1. Collision nomenclature

Penetration is important because if your objects overlap during the simulation, the results won't look realistic—you'll have one hovercraft moving inside the other. What you have to do is detect this penetration condition and then back up your simulation, reduce the time step, and try again. You keep doing this until they are no longer penetrating or they are within tolerance to be considered colliding.

You need to determine the normal velocity vector of the collision in order to calculate the collision impulse that will be used to simulate their response to the collision. For simple cases, determining this normal vector is fairly straightforward. In the case of particles or spheres, the collision normal is simply along the line that connects the

centers of gravity of each colliding object; this is *central impact*, as discussed in [Chapter 5](#), and is the same as that used for the particle example in [Chapter 8](#).

Now take a look at the function we've prepared for this simulation to check for collisions:

```
int      CheckForCollision (pRigidBody2D body1, pRigidBody2D body2)
{
    Vector    d;
    float     r;
    int       retval = 0;
    float     s;
    Vector    v1, v2;
    float     Vrn;

    r = body1->ColRadius + body2->ColRadius;
    d = body1->vPosition - body2->vPosition;
    s = d.Magnitude() - r;

    d.Normalize();
    vCollisionNormal = d;

    v1 = body1->vVelocity;
    v2 = body2->vVelocity;
    vRelativeVelocity = v1 - v2;

    Vrn = vRelativeVelocity * vCollisionNormal;
    if((fabs(s) <= ctol) && (Vrn < 0.0))
    {
        retval = 1; // collision;
        CollisionBody1 = body1;
        CollisionBody2 = body2;
    } else if(s < -ctol)
    {
        retval = -1; // interpenetrating
    } else
        retval = 0; // no collision

    return retval;
}
```

This function uses a simple bounding circle check to determine whether or not the hovercraft are colliding. The first thing it does is calculate the distance, r , that represents the absolute minimum separation between these hovercraft when they are in contact. `ColRadius` is the radius of the bounding circle of the hovercraft. We must compute it for each hovercraft upon initialization as follows:

```
->ColRadius = SQRT(fLength*fLength + fWidth*fWidth);
```

Next, the distance separating the hovercraft at the time this function is called is determined and stored in the variable `d`. Since we're assuming that these hovercraft are particles, determining `d` is simply a matter of calculating the distance between the coordi-

nates of each craft's center of gravity. In terms of vectors, this is simply the position vector of one craft minus the position vector of the other.

Once the function has d and r , it needs to determine the actual amount of space, s , separating the hovercraft's bounding circles. After this separation is determined, the function normalizes the vector d . Since the vector d is along the line that separates the hovercraft's centers of gravity, normalizing it yields the collision normal vector that we need for our collision response calculations. The collision normal vector is saved in the global variable `vCollisionNormal`.

After calculating the collision normal, this function goes on to determine the relative velocity between the hovercraft. In vector form, this is simply the difference between the velocity vectors of each craft. Note that the velocity vectors used here must be in global coordinates, not body-fixed (local) coordinates. Since what's really needed to determine if a collision is made is the relative *normal* velocity, the function proceeds to take the vector dot product of the relative velocity and the collision normal vectors, saving the result in the variable `Vrn`.

At this point, all of the calculations are complete, and the only thing left to do is make the appropriate checks to determine if there is a collision, penetration, or no collision at all.

The first check is to see if the hovercraft are colliding. We determine this by comparing the absolute value of the separation between the hovercraft, s , with a distance tolerance, `ctol`. If the absolute value of s is less than `ctol`, a collision might be occurring. The second requirement is that the relative normal velocity be negative, which implies that the points of impact on the hovercraft are moving toward each other. If there is a collision, the function returns a 1 to indicate that collision response is necessary.

If the hovercraft are found not to be colliding, then we perform a second check to see if they've moved so close together that they are penetrating each other. In this case, if s is less than $-ctol$, the hovercraft are penetrating and the function returns a -1. If the hovercraft are not colliding and not penetrating, then the function simply returns a 0, indicating that no further action is required.

Before moving on, let's say a word or two about `ctol`—the collision tolerance distance. This value is subject to tuning. There's no single value that works well in all cases. You must consider the overall sizes of the objects potentially colliding, the step size you're using, and how far the colliding objects are from the viewer while being rendered (i.e., their scale). Basically, you should choose a value that makes collisions look correct, so that on the one hand objects do not appear to be penetrating each other, and on the other hand you do not report a collision when objects do not appear to be touching at all.

Take a look now at the other new function, `ApplyImpulse`:

```
void      ApplyImpulse(pRigidBody2D body1, pRigidBody2D body2)
{
    float j;

    j = (-(1+fCr) * (vRelativeVelocity*vCollisionNormal)) /
        ( (vCollisionNormal*vCollisionNormal) *
        (1/body1->fMass + 1/body2->fMass) );

    body1->vVelocity += (j * vCollisionNormal) / body1->fMass;
    body2->vVelocity -= (j * vCollisionNormal) / body2->fMass;
}
```

This is a simple but crucial function for collision response. What it does is calculate the linear collision impulse as a function of the colliding hovercraft's relative normal velocity, masses, and coefficient of restitution, using the formula that we showed you in [Chapter 5](#). Further, it applies this impulse to each hovercraft, effectively changing their velocities in response to the collision. Note that the impulse is applied to one hovercraft and then the negative impulse applied to the other.

With those two new functions complete, it's now time to revise `UpdateSimulation` to handle collision detection and response as the simulation steps through time. Here's what the new `UpdateSimulation` function looks like:

```
void      UpdateSimulation(float dt)
{
    float      dtime = dt;
    bool       tryAgain = true;
    int        check=0;
    RigidBody2D   craft1Copy, craft2Copy;
    bool       didPen = false;
    int        count = 0;

    Craft.SetThrusters(false, false);

    if (IsKeyDown(VK_UP))
        Craft.ModulateThrust(true);

    if (IsKeyDown(VK_DOWN))
        Craft.ModulateThrust(false);

    if (IsKeyDown(VK_RIGHT))
        Craft.SetThrusters(true, false);

    if (IsKeyDown(VK_LEFT))
        Craft.SetThrusters(false, true);

    while(tryAgain && dtime > tol)
```

```

{
    tryAgain = false;
    memcpy(&craft1Copy, &Craft, sizeof(RigidBody2D));
    memcpy(&craft2Copy, &Craft2, sizeof(RigidBody2D));

    Craft.UpdateBodyEuler(dtime);
    Craft2.UpdateBodyEuler(dtime);

    CollisionBody1 = 0;
    CollisionBody2 = 0;
    check = CheckForCollision(&craft1Copy, &craft2Copy);

    if(check == PENETRATING)
    {
        dtime = dtime/2;
        tryAgain = true;
        didPen = true;
    } else if(check == COLLISION)
    {
        if(CollisionBody1 != 0 && CollisionBody2 != 0)
            ApplyImpulse(CollisionBody1, CollisionBody2);
    }
}

if(!didPen)
{
    memcpy(&Craft, &craft1Copy, sizeof(RigidBody2D));
    memcpy(&Craft2, &craft2Copy, sizeof(RigidBody2D));
}
}

```

Obviously, this version is more complicated than the original version. There's one main reason for this: penetration could occur because the hovercraft can move far enough within a single time step to become overlapped. Visually, this situation is unappealing and unrealistic, so you should try to prevent it.

The first thing this function does is enter a `while` loop:

```

while(tryAgain && dtime > tol)
{
    .
    .
    .
}

```

This loop is used to back up the simulation if penetration has occurred on the initial time step. What happens is this: the function first tries to update the hovercraft and then checks to see if there is a collision. If there is a collision, then it gets handled by applying the impulse. If there is penetration, however, then you know the time step was too big and you have to try again. When this occurs, `tryAgain` is set to `true`, the time step is cut in half, and another attempt is made. The function stays in this loop as long as there

is penetration or until the time step has been reduced to a size small enough to force an exit to the loop. The purpose of this looping is to find the largest step size, less than or equal to dt , that can be taken and still avoid penetration. You either want a collision or no collision.

You might ask yourself when does small become too small in terms of time step? Too small is obviously when the time step approaches 0 and your entire simulation grinds to a halt. Therefore, you may want to put in some criteria to exit this loop before things slow down too much. This is all subject to tuning, by the way, and it also depends on the value you set for `ctol`. We can't stress enough the importance of tuning these parameters. Basically, you must strive for visual realism while keeping your frame rates up to required levels.

Looking inside this `while` loop reveals what's going on. First, `tryAgain` is set to `false`, optimistically assuming that there will be no penetration, and we make copies of the hovercraft's states, reflecting the last successful call to `UpdateSimulation`.

Next, we make the usual call to `UpdateBody` for each copy of the hovercraft. Then a call to the collision detection function, `CheckForCollision`, is made to see if `Craft` is colliding with or penetrating `Craft2`. If there is penetration, then `tryAgain` is set to `true`, `dtime` is cut in half, `didPen` is set to `true`, and the function takes another lap through the `while` loop. `didPen` is a flag that lets us know that a penetration condition did occur.

If there was a collision, the function handles it by applying the appropriate impulse:

```
if(CollisionBody1 != 0 && CollisionBody2 != 0)
    ApplyImpulse(CollisionBody1, CollisionBody2);
```

After getting through the `while` loop, the updated hovercraft states are saved and `UpdateSimulation` is complete.

The last bit of code you need to add includes a few new global variables and `defines`:

```
#define LINEARDRAGCOEFFICIENT 0.25f
#define COEFFICIENTOFRESTITUTION 0.5f
#define COLLISIONTOLERANCE 2.0f

Vector vCollisionNormal;
Vector vRelativeVelocity;
float fCr = COEFFICIENTOFRESTITUTION;
float const ctol = COLLISIONTOLERANCE;
```

The only one we haven't mentioned so far, although you've seen it in `ApplyImpulse`, is `fCr`, the coefficient of restitution. Here we have it set to 0.5, which means that the collisions are halfway between perfectly elastic and perfectly inelastic (refer back to our earlier discussions on coefficients of restitution in [Chapter 5](#) if you've forgotten these terms). This is one of those parameters that you'll have to tune to get the desired behavior.

While we're on the subject of tuning, we should mention that you'll also have to play with the linear drag coefficient used to calculate the drag force on the hovercraft. While this coefficient is used to simulate fluid dynamic drag, it also plays an important role in terms of numerical stability. You need some damping in your simulation so that your integrator does not blow up—that is, damping helps keep your simulation stable.

That's pretty much it as far as implementing basic collision response. If you run this example, you'll be able to drive the hovercraft into each other and bounce off accordingly. You can play around with the mass of each hovercraft and the coefficient of restitution to see how the craft behave when one is more massive than the other, or when the collision is somewhere between perfectly elastic and perfectly inelastic.

You may notice that the collision response in this example sometimes looks a little strange. Keep in mind that's because this collision response algorithm, so far, assumes that the hovercraft are round when in fact they are rectangular. This approach will work just fine for round objects like billiard balls, but to get the level of realism required for non-round rigid bodies you need to include angular effects. We'll show you how to do that in the next section.

Angular Effects

Including angular effects will yield more realistic collision responses for these rigid bodies, the hovercraft. To get this to work, you'll have to make several changes to `ApplyImpulse` and `CheckForCollision`; `UpdateSimulation` will remain unchanged. The more extensive changes are in `CheckForCollision`, so we'll discuss it first.

The new version of `CheckForCollision` will do more than a simple bounding circle check. Here, each hovercraft will be represented by a polygon with four edges and four vertices, and the types of contact that will be checked for are vertex-vertex and vertex-edge contact (see [Figure 10-2](#)).¹

1. Note that this function does not handle multiple contact points.

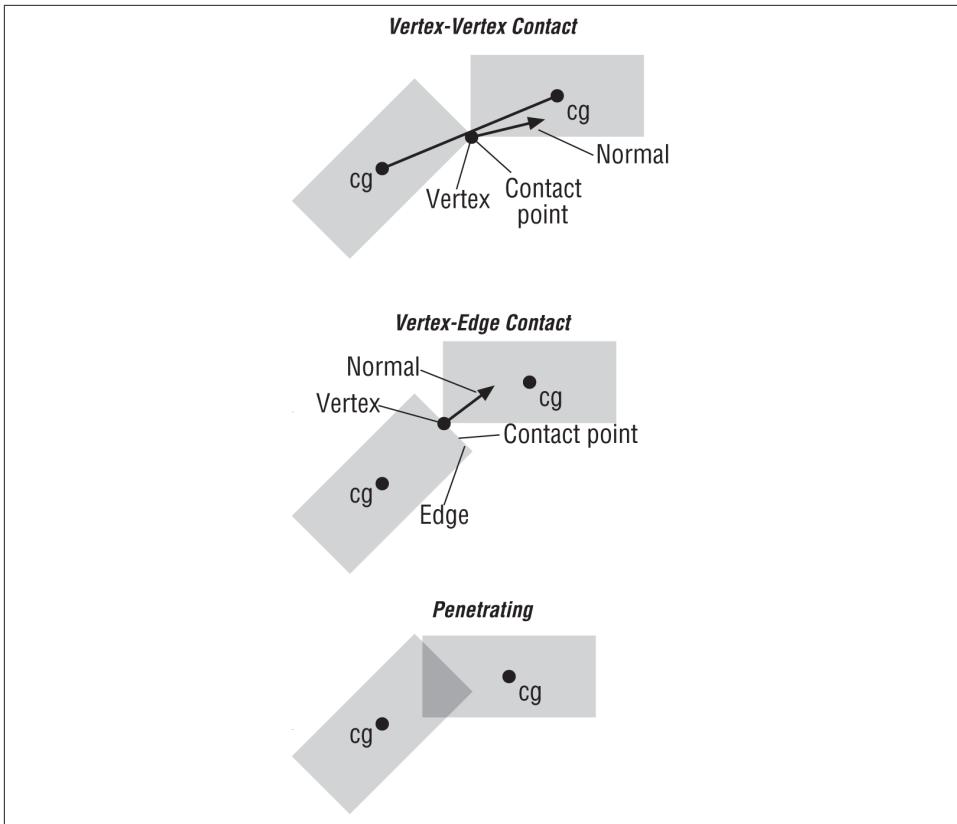


Figure 10-2. Types of collision

In addition to the tasks discussed in the last section, this new version of `CheckForCollision` must also determine the exact point of contact between the hovercraft. This is a very important distinction between this new version and the last. You need to know the point of contact because in order to affect the angular velocity, you must apply the impulse at the point of contact. In the last section, the normal to the contact point always passed through the center of gravity of the hovercraft because we assumed they were spheres; that's not the case here.

This now brings up the challenge of finding the collision normal. There are two cases to consider here. In edge-vertex collisions, the normal is always perpendicular to the edge that's involved in the collision. In vertex-vertex collisions, however, the normal is ambiguous, so we've resorted to taking the normal parallel to the line connecting the hovercraft's centers of gravity.

All of these considerations make `CheckForCollisions` a little more involved than in the previous section. The following code listing shows what we mean:

```

int      CheckForCollision(pRigidBody2D body1, pRigidBody2D body2)
{
    Vector    d;
    float     r;
    int       retval = 0;
    float     s;
    Vector    vList1[4], vList2[4];
    float     wd, lg;
    int       i,j;
    bool      haveNodeNode = false;
    bool      interpenetrating = false;
    bool      haveNodeEdge = false;
    Vector    v1, v2, u;
    Vector    edge, p, proj;
    float     dist, dot;
    float     Vrn;

    // First check to see if the bounding circles are colliding
    r = body1->fLength/2 + body2->fLength/2;
    d = body1->vPosition - body2->vPosition;
    s = d.Magnitude() - r;

    if(s <= ctol)
    {   // We have a possible collision, check further
        // build vertex lists for each hovercraft
        wd = body1->fWidth;
        lg = body1->fLength;
        vList1[0].y = wd/2;           vList1[0].x = lg/2;
        vList1[1].y = -wd/2;          vList1[1].x = lg/2;
        vList1[2].y = -wd/2;          vList1[2].x = -lg/2;
        vList1[3].y = wd/2;           vList1[3].x = -lg/2;

        for(i=0; i<4; i++)
        {
            VRotate2D(body1->fOrientation, vList1[i]);
            vList1[i] = vList1[i] + body1->vPosition;
        }

        wd = body2->fWidth;
        lg = body2->fLength;
        vList2[0].y = wd/2;           vList2[0].x = lg/2;
        vList2[1].y = -wd/2;          vList2[1].x = lg/2;
        vList2[2].y = -wd/2;          vList2[2].x = -lg/2;
        vList2[3].y = wd/2;           vList2[3].x = -lg/2;

        for(i=0; i<4; i++)
        {
            VRotate2D(body2->fOrientation, vList2[i]);
            vList2[i] = vList2[i] + body2->vPosition;
        }

        // Check for vertex-vertex collision
    }
}

```

```

for(i=0; i<4 && !haveNodeNode; i++)
{
    for(j=0; j<4 && !haveNodeNode; j++)
    {

        vCollisionPoint = vList1[i];
        body1->vCollisionPoint = vCollisionPoint -
            body1->vPosition;

        body2->vCollisionPoint = vCollisionPoint -
            body2->vPosition;

        vCollisionNormal = body1->vPosition -
            body2->vPosition;

        vCollisionNormal.Normalize();

        v1 = body1->vVelocityBody +
            (body1->vAngularVelocity^body1->vCollisionPoint);

        v2 = body2->vVelocityBody +
            (body2->vAngularVelocity^body2->vCollisionPoint);

        v1 = VRotate2D(body1->fOrientation, v1);
        v2 = VRotate2D(body2->fOrientation, v2);

        vRelativeVelocity = v1 - v2;
        Vrn = vRelativeVelocity * vCollisionNormal;

        if( ArePointsEqual(vList1[i],
                           vList2[j]) &&
            (Vrn < 0.0) )
            haveNodeNode = true;
    }
}

// Check for vertex-edge collision
if(!haveNodeNode)
{
    for(i=0; i<4 && !haveNodeEdge; i++)
    {
        for(j=0; j<3 && !haveNodeEdge; j++)
        {
            if(j==2)
                edge = vList2[0] - vList2[j];
            else
                edge = vList2[j+1] - vList2[j];
            u = edge;
            u.Normalize();

            p = vList1[i] - vList2[j];

```

```

    proj = (p * u) * u;

    d = p^u;
    dist = d.Magnitude();

    vCollisionPoint = vList1[i];
    body1->vCollisionPoint = vCollisionPoint -
        body1->vPosition;

    body2->vCollisionPoint = vCollisionPoint -
        body2->vPosition;

    vCollisionNormal = ((u^p)^u);
    vCollisionNormal.Normalize();

    v1 = body1->vVelocityBody +
        (body1->vAngularVelocity ^
         body1->vCollisionPoint);

    v2 = body2->vVelocityBody +
        (body2->vAngularVelocity ^
         body2->vCollisionPoint);

    v1 = VRotate2D(body1->fOrientation, v1);
    v2 = VRotate2D(body2->fOrientation, v2);

    vRelativeVelocity = (v1 - v2);
    Vrn = vRelativeVelocity * vCollisionNormal;

    if( (proj.Magnitude() > 0.0f) &&
        (proj.Magnitude() <= edge.Magnitude()) &&
        (dist <= ctol) &&
        (Vrn < 0.0) )
        haveNodeEdge = true;
    }
}
}

// Check for penetration
if(!haveNodeNode && !haveNodeEdge)
{
    for(i=0; i<4 && !interpenetrating; i++)
    {
        for(j=0; j<4 && !interpenetrating; j++)
        {
            if(j==3)
                edge = vList2[0] - vList2[j];
            else
                edge = vList2[j+1] - vList2[j];

            p = vList1[i] - vList2[j];
            dot = p * edge;

```

```

        if(dot < 0)
        {
            interpenetrating = true;
        }
    }

    if(interpenetrating)
    {
        retval = -1;
    } else if(haveNodeNode || haveNodeEdge)
    {
        retval = 1;
    } else
        retval = 0;

} else
{
    retval = 0;
}

return retval;
}

```

The first thing that `CheckForCollision` does is perform a quick bounding-circle check to see if there is a possible collision. If no collision is detected, the function simply exits, returning 0. This is the same bounding-circle check performed in the earlier version:

```

r = body1->fLength/2 + body2->fLength/2;
d = body1->vPosition - body2->vPosition;
s = d.Magnitude() - r;

if(s <= ctol)
{
    .
    .
    .
} else
    retval = 0;
}

```

If the bounding-circle check indicates the possibility of a collision, then `CheckForCollision` proceeds by setting up a couple of polygons, represented by vertex lists, for each hovercraft:

```

wd = body1->fWidth;
lg = body1->fLength;
vList1[0].y = wd/2;           vList1[0].x = lg/2;
vList1[1].y = -wd/2;          vList1[1].x = lg/2;
vList1[2].y = -wd/2;          vList1[2].x = -lg/2;
vList1[3].y = wd/2;           vList1[3].x = -lg/2;

```

```

    for(i=0; i<4; i++)
    {
        VRotate2D(body1->fOrientation, vList1[i]);
        vList1[i] = vList1[i] + body1->vPosition;
    }

    wd = body2->fWidth;
    lg = body2->fLength;
    vList2[0].y = wd/2;           vList2[0].x = lg/2;
    vList2[1].y = -wd/2;          vList2[1].x = lg/2;
    vList2[2].y = -wd/2;          vList2[2].x = -lg/2;
    vList2[3].y = wd/2;           vList2[3].x = -lg/2;
    for(i=0; i<4; i++)
    {
        VRotate2D(body2->fOrientation, vList2[i]);
        vList2[i] = vList2[i] + body2->vPosition;
    }
}

```

The vertex lists are initialized in unrotated body-fixed (local) coordinates based on the length and width of the hovercraft. The vertices are then rotated to reflect the orientation of each hovercraft. After that, the position of each hovercraft is added to each vertex to convert from local coordinates to global coordinates

Checking first for vertex-vertex collisions, the function iterates through each vertex in one list, comparing it with each vertex in the other list to see if the points are coincident.

```

// Check for vertex-vertex collision
for(i=0; i<4 && !haveNodeNode; i++)
{
    for(j=0; j<4 && !haveNodeNode; j++)
    {

        vCollisionPoint = vList1[i];
        body1->vCollisionPoint = vCollisionPoint -
                                    body1->vPosition;

        body2->vCollisionPoint = vCollisionPoint -
                                    body2->vPosition;

        vCollisionNormal = body1->vPosition -
                           body2->vPosition;

        vCollisionNormal.Normalize();

        v1 = body1->vVelocityBody +
             (body1->vAngularVelocity^body1->vCollisionPoint);

        v2 = body2->vVelocityBody +
             (body2->vAngularVelocity^body2->vCollisionPoint);

        v1 = VRotate2D(body1->fOrientation, v1);
        v2 = VRotate2D(body2->fOrientation, v2);
    }
}

```

```

    vRelativeVelocity = v1 - v2;
    Vrn = vRelativeVelocity * vCollisionNormal;

    if( ArePointsEqual(vList1[i],
                        vList2[j]) &&
        (Vrn < 0.0) )
        haveNodeNode = true;

    }
}

```

This comparison makes a call to another new function, `ArePointsEqual`:

```

if( ArePointsEqual(vList1[i],
                    vList2[j]) &&
    (Vrn < 0.0) )
    haveNodeNode = true;

```

`ArePointsEqual` simply checks to see if the points are within a specified distance from each other, as shown here:

```

bool      ArePointsEqual(Vector p1, Vector p2)
{
    // Points are equal if each component is within ctol of each other
    if( (fabs(p1.x - p2.x) <= ctol) &&
        (fabs(p1.y - p2.y) <= ctol) &&
        (fabs(p1.z - p2.z) <= ctol) )
        return true;
    else
        return false;
}

```

Within the nested for loops of the vertex-vertex check, we perform a number of important calculations to determine the collision normal vector and relative velocity that are required for collision response.

First, we calculate the collision point, which is simply the coordinates of a vertex that is involved in the collision. Note that this point will be in global coordinates, so it will have to be converted to local coordinates for each hovercraft in order to be useful for collision response. Here's how that's done:

```

vCollisionPoint = vList1[i];
body1->vCollisionPoint = vCollisionPoint -
                           body1->vPosition;

body2->vCollisionPoint = vCollisionPoint -
                           body2->vPosition;

```

The second calculation is aimed at determining the collision normal vector, which for vertex-vertex collisions we've assumed is along the line connecting the centers of gravity

of each hovercraft. The calculation is the same as that shown in the earlier version of `CheckForCollision`:

```
vCollisionNormal = body1->vPosition -
    body2->vPosition;

vCollisionNormal.Normalize();
```

The third and final calculation is aimed at determining the relative velocity between the points of impact. This is an important distinction from the earlier version, since the velocities of the points of impact on each body are functions of the linear and angular velocities of the hovercraft:

```
v1 = body1->vVelocityBody +
    (body1->vAngularVelocity^body1->vCollisionPoint);

v2 = body2->vVelocityBody +
    (body2->vAngularVelocity^body2->vCollisionPoint);

v1 = VRotate2D(body1->fOrientation, v1);
v2 = VRotate2D(body2->fOrientation, v2);

vRelativeVelocity = v1 - v2;
Vrn = vRelativeVelocity * vCollisionNormal;
```

Here, `v1` and `v2` represent the velocities of the points of collision relative to each hovercraft in local coordinates, which are then converted to global coordinates. Once we've obtained the relative velocity, `vRelativeVelocity`, we obtain the relative normal velocity, `Vrn`, by taking the dot product of the relative velocity with the collision normal vector.

If there is no vertex-vertex collision, `CheckForCollision` proceeds to check for vertex-edge collisions:

```
// Check for vertex-edge collision
if(!haveNodeNode)
{
    for(i=0; i<4 && !haveNodeEdge; i++)
    {
        for(j=0; j<3 && !haveNodeEdge; j++)
        {
            if(j==3)
                edge = vList2[0] - vList2[j];
            else
                edge = vList2[j+1] - vList2[j];
            u = edge;
            u.Normalize();

            p = vList1[i] - vList2[j];
            proj = (p * u) * u;

            d = p^u;
```

```

    dist = d.Magnitude();

    vCollisionPoint = vList1[i];
    body1->vCollisionPoint = vCollisionPoint -
        body1->vPosition;

    body2->vCollisionPoint = vCollisionPoint -
        body2->vPosition;

    vCollisionNormal = ((u^p)^u);
    vCollisionNormal.Normalize();

    v1 = body1->vVelocityBody +
        (body1->vAngularVelocity ^
         body1->vCollisionPoint);

    v2 = body2->vVelocityBody +
        (body2->vAngularVelocity ^
         body2->vCollisionPoint);

    v1 = VRotate2D(body1->fOrientation, v1);
    v2 = VRotate2D(body2->fOrientation, v2);

    vRelativeVelocity = (v1 - v2);
    Vrn = vRelativeVelocity * vCollisionNormal;

    if( (proj.Magnitude() > 0.0f) &&
        (proj.Magnitude() <= edge.Magnitude()) &&
        (dist <= ctol) &&
        (Vrn < 0.0) )
        haveNodeEdge = true;
    }
}
}

```

Here, the nested `for` loops check each vertex in one list to see if it is in contact with each edge built from the vertices in the other list. After building the edge under consideration, we save and normalize a copy of it to represent a unit vector pointing along the edge:

```

if(j==3)
    edge = vList2[0] - vList2[j];
else
    edge = vList2[j+1] - vList2[j];
u = edge;
u.Normalize();

```

Variable `u` represents that unit vector, and it will be used in subsequent calculations. The next set of calculations determines the location of the projection of the vertex under consideration onto the edge under consideration, as well as the minimum distance from the vertex to edge:

```

p = vList1[i] - vList2[j];
proj = (p * u) * u;

d = p^u;
dist = d.Magnitude();

```

Variable p is a vector from the first vertex on the edge to the vertex under consideration, and $proj$ is the distance from the first edge vertex, along the edge, to the point upon which the vertex projects. $dist$ is the minimum distance from the vertex to the edge. [Figure 10-3](#) illustrates this geometry.

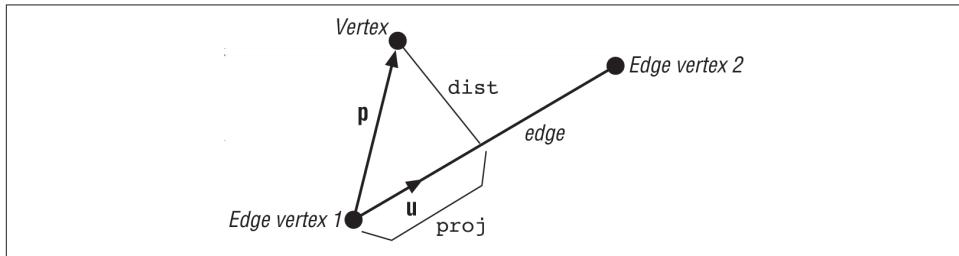


Figure 10-3. Vertex-edge check

If there is a collision, the global location of the point of impact is equal to the vertex under consideration, which we must convert to local coordinates for each hovercraft, as shown here:

```

vCollisionPoint = vList1[i];
body1->vCollisionPoint = vCollisionPoint -
                           body1->vPosition;

body2->vCollisionPoint = vCollisionPoint -
                           body2->vPosition;

```

Since, in this type of collision, the collision normal vector is perpendicular to the edge, you can determine it by taking the result of the cross product of u and p and crossing it with u as follows:

```

vCollisionNormal = ((u^p)^u);
vCollisionNormal.Normalize();

```

These calculations give you a unit length vector in the plane of vectors u and p and perpendicular to the edge.

Next, the relative velocity between the points of impact on each hovercraft is determined, just as in the vertex-vertex collision check:

```

v1 = body1->vVelocityBody +
      (body1->vAngularVelocity ^
       body1->vCollisionPoint);

```

```

v2 = body2->vVelocityBody +
      (body2->vAngularVelocity ^
       body2->vCollisionPoint);

v1 = VRotate2D(body1->fOrientation, v1);
v2 = VRotate2D(body2->fOrientation, v2);

vRelativeVelocity = (v1 - v2);
Vrn = vRelativeVelocity * vCollisionNormal;

```

In determining whether or not the vertex under consideration is in fact colliding with an edge, you have to check to see if the distance from the vertex is within your collision tolerance, and you also have to make sure the vertex actually projects onto the edge (that is, it does not project beyond the endpoints of the edge). Additionally, you need to make sure the relative normal velocity indicates that the points of contact are moving toward each other. Here's how this check looks:

```

if( (proj.Magnitude() > 0.0f) &&
    (proj.Magnitude() <= edge.Magnitude()) &&
    (dist <= ctol) &&
    (Vrn < 0.0 ) )
    haveNodeEdge = true;

```

After `CheckForCollision` checks for vertex-vertex and vertex-edge collisions, it goes on to check for penetration:

```

if(!haveNodeNode && !haveNodeEdge)
{
    for(i=0; i<4 && !interpenetrating; i++)
    {
        for(j=0; j<4 && !interpenetrating; j++)
        {
            if(j==3)
                edge = vList2[0] - vList2[j];
            else
                edge = vList2[j+1] - vList2[j];

            p = vList1[i] - vList2[j];
            dot = p * edge;
            if(dot < 0)
            {
                interpenetrating = true;
            }
        }
    }
}

```

This check is a standard point-in-polygon check using the vector dot product to determine if any vertex of one polygon lies within the bounds of the other polygon. After this check, the function simply returns the appropriate result. Here again, 0 indicates no collision or penetration, 1 indicates a collision, and -1 indicates penetration.

With `CheckForCollision` out of the way, turn your attention to `ApplyImpulse`, which also has to be revised to include angular effects. Specifically, you need to use the impulse formula that includes angular as well as linear effects (see [Chapter 5](#)), and you also have to apply the impulse to the hovercraft's angular velocities in addition to their linear velocities. Here's how the new `ApplyImpulse` function looks:

```
void      ApplyImpulse(pRigidBody2D body1, pRigidBody2D body2)
{
    float j;

    j = (-(1+fCr) * (vRelativeVelocity*vCollisionNormal)) /
        ( (1/body1->fMass + 1/body2->fMass) +
        (vCollisionNormal * (((body1->vCollisionPoint ^
        vCollisionNormal)/body1->fInertia)^body1->vCollisionPoint)) +
        (vCollisionNormal * (((body2->vCollisionPoint ^
        vCollisionNormal)/body2->fInertia)^body2->vCollisionPoint)) )
    );

    body1->vVelocity += (j * vCollisionNormal) / body1->fMass;
    body1->vAngularVelocity += (body1->vCollisionPoint ^
                                (j * vCollisionNormal)) /
                                body1->fInertia;

    body2->vVelocity -= (j * vCollisionNormal) / body2->fMass;
    body2->vAngularVelocity -= (body2->vCollisionPoint ^
                                (j * vCollisionNormal)) /
                                body2->fInertia;
}
```

Remember, the impulse is applied to one hovercraft while its negative is applied to the other.

That does it for this new version of the hovercraft simulation. If you run the program now, you'll see that you can crash the hovercraft into each other and they bounce and rotate accordingly. This makes for a much more realistic simulation than the simple, linear collision response approach of the last section. Here again, you can play with the mass of each hovercraft and the coefficient of restitution to see how these parameters affect the collision response between the hovercraft.

Rotation in 3D Rigid-Body Simulators

A fundamental difference between particles and rigid bodies is that we cannot ignore rotation of rigid bodies. This applies to both 2D and 3D rigid bodies. In two dimensions, it's quite easy to express the orientation of a rigid body; you need only a single scalar to represent the body's rotation about a single axis. In three dimensions, however, there are three primary coordinate axes about each of which a rigid body may rotate. Moreover, a rigid body in three dimensions may rotate about any arbitrary axis, not necessarily one of the coordinate axes.

In two dimensions, we say that a rigid body has only one rotational degree of freedom, whereas in three dimensions we say that a rigid body has three rotational degrees of freedom. This may lead you to infer that in three dimensions, you must have three scalar quantities to represent a body's rotation. Indeed, this is a minimum requirement, and you're probably already familiar with a set of angles that represent the orientation of a rigid body in 3D—namely, the three Euler angles (roll, pitch, and yaw) that we'll talk about in [Chapter 15](#).

These three angles—roll, pitch, and yaw—are very intuitive and easy for us to visualize. For example, in an airplane the nose pitches up or down, the plane rolls (or banks) left or right, and the yaw (or heading) changes to the left or right. Unfortunately, there's a problem with using these three Euler angles in rigid-body simulations. The problem is a numerical one that occurs when the pitch angle reaches plus or minus 90 degrees ($\pi/2$). When this happens, roll and yaw become ambiguous. Worse yet, the angular equations of motion written in terms of Euler angles contain terms involving the cosine of the pitch angle in the denominator, which means that when the pitch angle is plus or minus 90 degrees the equations become singular (i.e., there's division by 0). If this happens in your simulation, the results would be unpredictable to say the least. Given this problem with Euler angles, you must use some other means of keeping track of orientation in your simulation. We'll discuss two such means in this chapter—specifically, rotation matrices and quaternions.

Virtually every computer graphics book that we've read contains a chapter or section on using rotation matrices. Far fewer discuss quaternions, but if you're familiar with quaternions, it's probably in the same context as rotation matrices—that is, how they are used to rotate 3D points, objects, scenes, and points of view. In a simulation, however, you need to get a little more out of rotation matrices or quaternions and will use them in a different context than what you might be accustomed to. Specifically, you need to keep track of a body's orientation in space and, moreover, the change in orientation over time. So it's in this light that we'll discuss rotation matrices and quaternions. We'll try to be as concise as possible so as not to cloud the water with the proofs and derivations that you can find in the texts referred to in the [Bibliography](#).

Rotation Matrices

A rotation matrix is a 3×3 matrix that, when multiplied with a point or vector, results in the rotation of that point about some axis, yielding a new set of coordinates. You can rotate points about axes in one coordinate system or you can use rotation matrices to convert points from one coordinate system to another, where one is rotated relative to the other.

Rotating a vector by a rotation matrix is typically written as follows: if \mathbf{v} is a vector, and \mathbf{R} is a rotation matrix, then \mathbf{v}' is \mathbf{v} rotated by \mathbf{R} according to the formula:

as affected by the previous rotation before they can be correctly applied. In other words, you have to rotate \mathbf{R}_2 by \mathbf{R}_1 to get a new \mathbf{R}_2 before applying it. All this happens to work out in such a way that you reverse the order of multiplication of rotation matrices when they are defined in a rotating coordinate system.

Figure 11-1 shows a right-handed coordinate system that illustrates the directions of positive rotation about each coordinate axis.

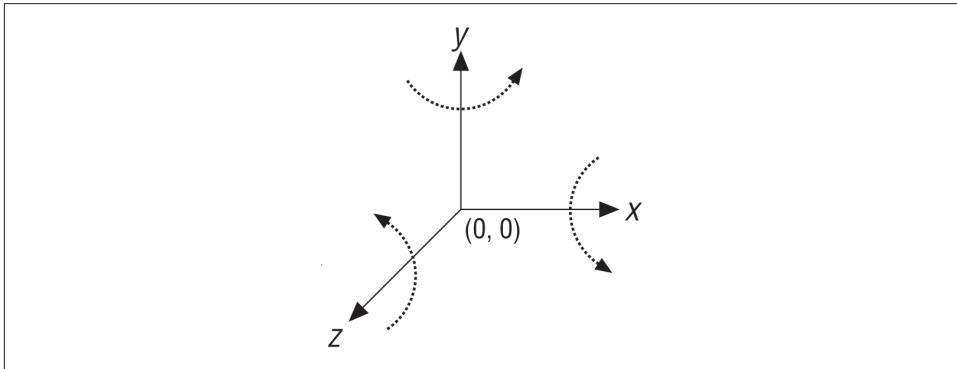


Figure 11-1. Right-handed coordinate system

Let's consider rotation around the z-axis where the point shown in **Figure 11-2** is rotated through an angle θ .

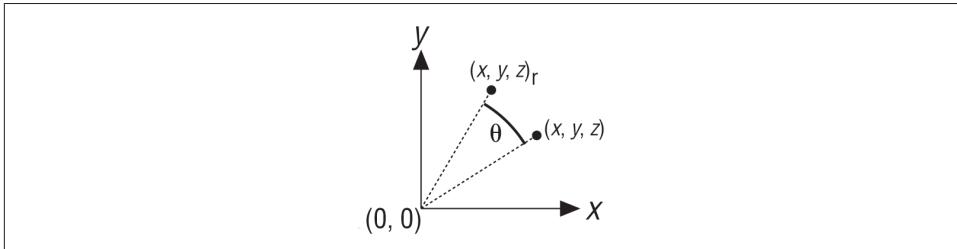


Figure 11-2. Rotation around the z-axis

The coordinates of the point before the rotation are (x, y, z) and after the rotation the coordinates are (x_r, y_r, z_r) . The rotated coordinates are related to the original coordinates and the rotation angle by the following:

Notice that since the point is rotating about the z-axis, its z coordinate remains unchanged. To write this in the vector-matrix notation, $\mathbf{v}' = \mathbf{R} \mathbf{v}$, let $\mathbf{v} = [x \ y \ z]$ and let \mathbf{R} be the matrix:

$$\begin{vmatrix} \cos(\theta_z) & -\sin(\theta_z) & 0 \\ \sin(\theta_z) & \cos(\theta_z) & 0 \\ 0 & 0 & 1 \end{vmatrix}$$

Here \mathbf{v}' will be the new, rotated vector, $\mathbf{v}' = [x_r \ y_r \ z_r]$.

Rotation about the x- and y-axes is similar to the z-axis; however, in those cases the x and y coordinates remain constant during rotations about each axis, respectively. Looking at rotation about each axis separately will yield three rotation matrices similar to the one we just showed you for rotation about the z-axis.

For rotation about the x-axis, the matrix is:

$$\begin{vmatrix} 1 & 0 & 0 \\ 0 & \cos(\theta_x) & -\sin(\theta_x) \\ 0 & \sin(\theta_x) & \cos(\theta_x) \end{vmatrix}$$

And for rotation about the y-axis, the matrix is:

$$\begin{vmatrix} \cos(\theta_y) & 0 & \sin(\theta_y) \\ 0 & 1 & 0 \\ -\sin(\theta_y) & 0 & \cos(\theta_y) \end{vmatrix}$$

These are the rotation matrices you typically see in computer graphics texts in the context of matrix transforms, such as translation, scaling, and rotation. You can combine all three of these matrices into a single rotation matrix to represent combinations of rotations about each coordinate axis, using matrix multiplication as mentioned earlier.

In rigid-body simulations, you can use a rotation matrix to represent the orientation of a rigid body. Another way to think of it is the rotation matrix, when applied to the unrotated rigid body aligned with the fixed global coordinate system, will rotate the rigid body's coordinates so as to resemble the body's current orientation at any given time. This leads to another important consideration when using rotation matrices to keep track of orientation in rigid-body simulations: the fact that the rotation matrix will be a function of time.

Once you set up your initial rotation matrix for the rigid body, you'll never directly calculate it again from orientation angles; instead, the forces and moments applied to the rigid body will change the body's angular velocity, likewise causing small changes

in orientation at each time step throughout the simulation. Thus, you can see that you must have a means of relating the rotation matrix to angular velocity so that you can update the orientation accordingly. The formula you need is as follows:

-
1. Two vectors are orthogonal if their dot product is 0.

proach that lets you keep the advantages rotation matrices have to offer, but at a cheaper price. That alternative, quaternions, is the subject of the next section.

Quaternions

Quaternions are somewhat of a mathematical oddity. They were developed over 100 years ago by William Hamilton through his work in complex (imaginary) math but have found very little practical use. A quaternion is a quantity, kind of like a vector, but made up of four components. It is typically written in the form:

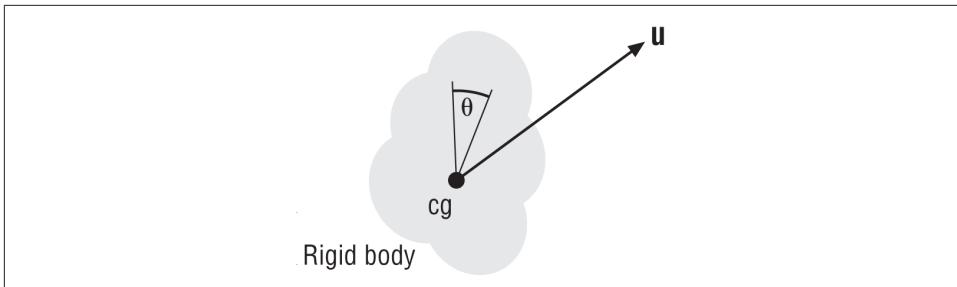


Figure 11-3. Quaternion rotation

You can readily see that quaternions, when used to represent rotation or orientation, require you to deal with only four parameters instead of nine, subject to the easily satisfied constraint that the quaternion be a unit quaternion.

The use of quaternions to represent orientation is similar to how you would use rotation matrices. First, you set up a quaternion that represents the initial orientation of the rigid body at time 0 (this is the only time you'll calculate the quaternion explicitly). Then you update the orientation to reflect the new orientation at a given instant in time using the angular velocities that are calculated for that instant. As you can see here, the differential equation relating an orientation quaternion to angular velocity is very similar to that for rotation matrices:

to rotating, body-fixed coordinates so that you can apply the forces to the body; or you might have to convert a body's velocity defined in global coordinates to body coordinates so that you can use the velocity in force calculations.

Quaternion Operations

As with vectors and matrices, quaternions have their own rules for the various operations that you'll need, such as multiplication, addition, subtraction, and so on. To make it easy on you, we've included sample code in [Appendix C](#) that implements all of the quaternion operations you'll need; however, we want to highlight a few of the more important ones here.

The `Quaternion` class is defined with a scalar component, n , and vector component, \mathbf{v} , where \mathbf{v} is the vector, $x\mathbf{i} + y\mathbf{j} + z\mathbf{k}$. The class has two constructors, one of which initializes the quaternion to 0, and the other of which initializes the elements to those passed to the constructor:

```
class Quaternion {
public:
    float     n;      // number (scalar) part
    Vector    v;      // vector part: v.x, v.y, v.z

    Quaternion(void);
    Quaternion(float e0, float e1, float e2, float e3);

    .
    .
    .

};
```

Magnitude

The `Magnitude` method returns the magnitude of the quaternion according to the following formula:

Conjugate: The \sim operator

The conjugate of the product of quaternions is equal to the product of the quaternion conjugates, but in reverse order:

```

inline     Quaternion operator*(Quaternion q1, Quaternion q2)
{
    return     Quaternion(q1.n*q2.n - q1.v.x*q2.v.x
                           - q1.v.y*q2.v.y - q1.v.z*q2.v.z,
                           q1.n*q2.v.x + q1.v.x*q2.n
                           + q1.v.y*q2.v.z - q1.v.z*q2.v.y,
                           q1.n*q2.v.y + q1.v.y*q2.n
                           + q1.v.z*q2.v.x - q1.v.x*q2.v.z,
                           q1.n*q2.v.z + q1.v.z*q2.n
                           + q1.v.x*q2.v.y - q1.v.y*q2.v.x);
}

```

Vector multiplication: The * operator

This operator multiplies the quaternion, q , by the vector v as though the vector v were a quaternion with its scalar component equal to 0. There are two forms of this operator depending on the order in which the quaternion and vector are encountered. Since v is assumed to be a quaternion with its scalar part equal to 0, the rules of multiplication follow those outlined earlier for quaternion multiplication:

```

inline     Quaternion operator*(Quaternion q, Vector v)
{
    return     Quaternion(   -(q.v.x*v.x + q.v.y*v.y + q.v.z*v.z),
                           q.n*v.x + q.v.y*v.z - q.v.z*v.y,
                           q.n*v.y + q.v.z*v.x - q.v.x*v.z,
                           q.n*v.z + q.v.x*v.y - q.v.y*v.x);
}
inline     Quaternion operator*(Vector v, Quaternion q)
{
    return     Quaternion(   -(q.v.x*v.x + q.v.y*v.y + q.v.z*v.z),
                           q.n*v.x + q.v.z*v.y - q.v.y*v.z,
                           q.n*v.y + q.v.x*v.z - q.v.z*v.x,
                           q.n*v.z + q.v.y*v.x - q.v.x*v.y);
}

```

MakeQFromEulerAngles

This function constructs a quaternion from a set of Euler angles.

For a given set of Euler angles, yaw (ψ), pitch (τ), and roll (ϕ), defining rotation about the z-axis, then the y-axis, and then the x-axis, you can construct the representative rotation quaternion. You do this by first constructing a quaternion for each Euler angle and then multiplying the three quaternions following the rules of quaternion multiplication. Here are the three quaternions representing each Euler rotation angle:

Each one of these quaternions is of unit length.²

Now you can multiply these quaternions to obtain a single one that represents the rotation, or orientation, defined by the three Euler angles:

2. You can verify this by recalling the trigonometric relation $\cos^2\theta + \sin^2\theta = 1$.

MakeEulerAnglesFromQ

This function extracts the three Euler angles from a given quaternion.

You can extract the three Euler angles from a quaternion by first converting the quaternion to a rotation matrix and then extracting the Euler angles from the rotation matrix. Let \mathbf{R} be a nine-element rotation matrix:

$$\mathbf{R} = \begin{vmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{vmatrix}$$

and let \mathbf{q} be a quaternion:

```

q00 = q.n * q.n;
q11 = q.v.x * q.v.x;
q22 = q.v.y * q.v.y;
q33 = q.v.z * q.v.z;

r11 = q00 + q11 - q22 - q33;
r21 = 2 * (q.v.x*q.v.y + q.n*q.v.z);
r31 = 2 * (q.v.x*q.v.z - q.n*q.v.y);
r32 = 2 * (q.v.y*q.v.z + q.n*q.v.x);
r33 = q00 - q11 - q22 + q33;

tmp = fabs(r31);
if(tmp > 0.999999)
{
    r12 = 2 * (q.v.x*q.v.y - q.n*q.v.z);
    r13 = 2 * (q.v.x*q.v.z + q.n*q.v.y);

    u.x = RadiansToDegrees(0.0f); //roll
    u.y = RadiansToDegrees((float) (-(pi/2) * r31/tmp)); // pitch
    u.z = RadiansToDegrees((float) atan2(-r12, -r31*r13)); // yaw
    return u;
}

u.x = RadiansToDegrees((float) atan2(r32, r33)); // roll
u.y = RadiansToDegrees((float) asin(-r31)); // pitch
u.z = RadiansToDegrees((float) atan2(r21, r11)); // yaw
return u;
}

```

Quaternions in 3D Simulators

The quaternion operations just presented are required when you are using quaternions to represent orientation in 3D simulations. All the 3D simulations discussed in this book use these quaternion operations, and in this section we'll highlight where they are used in the context of the airplane example presented in [Chapter 15](#).

When initializing the orientation of the airplane, you have to set its orientation quaternion to something corresponding to the Euler angles you desire. You do so as follows:

```
Airplane.qOrientation = MakeQFromEulerAngles(iRoll, iPitch, iYaw);
```

In this code sample, `Airplane` is a rigid-body class with the property `qOrientation`, which represents the orientation quaternion, which is a `Quaternion` class. `iRoll`, `iPitch`, and `iYaw` are the three Euler angles describing the orientation of the airplane.

If at any time you want to report the Euler angles—for example, in a heads-up display-like interface for the game player—you can use `MakeEulerAnglesFromQ`, as follows:

```

// get the Euler angles for our information
Vector u;

u = MakeEulerAnglesFromQ(Airplane.qOrientation);
Airplane.vEulerAngles.x = u.x;      // roll
Airplane.vEulerAngles.y = u.y;      // pitch
Airplane.vEulerAngles.z = u.z;      // yaw

```

Very often, it's more convenient to calculate loads on an object like the airplane using body-fixed coordinates. For example, when computing aerodynamic drag on the airplane, you'll want to know the relative air velocity over the aircraft in body-fixed coordinates. The resulting drag force will also be in body-fixed coordinates. However, when resolving all the loads on the aircraft to determine its motion in earth-fixed coordinates, you'll want to convert those forces from body-fixed coordinates to earth-fixed coordinates. You can use QVRotate to rotate any vector, such as a force vector, from one coordinate system to another. The following code sample shows how QVRotate is used to convert a force vector in body-fixed coordinates to the equivalent force in earth-fixed coordinates.

```

void    CalcAirplaneLoads(void)
{
    .
    .
    .

    // Convert forces from model space to earth space
    Airplane.vForces = QVRotate(Airplane.qOrientation, Fb);
    .
    .
    .

}

```

Throughout the simulation, you'll have to update the airplane's orientation by integrating the angular equations of motion. The first step in handling angular motion is to calculate the new angular velocity at a given time step based on the previously calculated moments acting on the airplane and its mass properties. We do this in body coordinates using the angular equation of motion:

Next, to enforce the constraint that this orientation quaternion be a *unit* quaternion, you must normalize the orientation quaternion. The following code sample illustrates these steps:

```
.  
. .  
  
// calculate the angular velocity of the airplane in body space:  
Airplane.vAngularVelocity += Airplane.mInertiaInverse *  
    (Airplane.vMoments -  
     (Airplane.vAngularVelocity^  
      (Airplane.mInertia *  
       Airplane.vAngularVelocity)))  
    * dt;  
  
// calculate the new rotation quaternion:  
Airplane.qOrientation += (Airplane.qOrientation *  
    Airplane.vAngularVelocity) *  
    (0.5f * dt);  
  
// now normalize the orientation quaternion:  
mag = Airplane.qOrientation.Magnitude();  
if (mag != 0)  
    Airplane.qOrientation /= mag;  
  
// calculate the velocity in body space:  
// (we'll need this to calculate lift and drag forces)  
Airplane.vVelocityBody = QVRotate(~Airplane.qOrientation,  
    Airplane.vVelocity);  
. .  
. .
```

Notice the last line of code in the preceding sample. That line converts the airplane's velocity vector from earth-fixed coordinates to body-fixed coordinates using `QVRotate`. Recall that it's more convenient to compute body forces in body-fixed coordinates. `QVRotate` allows you to work with vectors back and forth from body-fixed to earth-fixed coordinates.

3D Rigid-Body Simulator

In this chapter we'll show you how to make the leap from 2D to 3D by implementing a rigid-body simulation of an airplane. Specifically, this is a simulation of the hypothetical airplane model that we'll discuss extensively in [Chapter 15](#). This airplane is of typical configuration with its large wings forward, its elevators aft, a single vertical tail, and plain flaps fitted on the wings.

As with the 2D simulator in previous chapters, we'll concentrate on the code that implements the physics part of the simulator and not the platform-specific GUI aspects of the simulations.

As in 2D, there are four main elements to this 3D simulation—the model, integrator, user input, and rendering. Remember, the model refers to your idealization of the thing—an airplane, in this case—that you are trying to simulate, while the integrator refers to the method by which you integrate the differential equations of motion. These two elements take care of most of the physics of the simulation. The user input and rendering elements refer to how you'll allow the user to interact with and view your simulation.

In this simulation, the world coordinate system has its positive x-axis pointing into the screen, its positive y-axis pointing to the left of your screen, and the positive z-axis pointing up. Also, the local, or body-fixed, coordinate system has its positive x-axis pointing toward the front of the airplane, its positive y-axis pointing to the port side (left side), and its positive z-axis pointing up. Since this is a 3D simulation of an airplane, once you get it running, you'll be able to fly in any direction, looping, banking, diving, and climbing, or performing any other aerobatic maneuver you desire.

Model

One of the most important aspects of this simulation is the flight model. We'll spend all of [Chapter 15](#) discussing the physics behind this flight model, so we won't include that discussion here except to introduce a few key bits of code.

To implement the flight model, you first need to prepare a rigid-body structure to encapsulate all of the data required to completely define the state of the rigid body at any instant during the simulation. We've defined a structure called `RigidBody` for this purpose:

```
typedef struct _RigidBody {  
  
    float          fMass;           // total mass  
    Matrix3x3     mInertia;        // mass moment of inertia  
                                // in body coordinates  
  
    Matrix3x3     mInertiaInverse; // inverse of mass moment of inertia  
    Vector         vPosition;       // position in earth coordinates  
    Vector         vVelocity;       // velocity in earth coordinates  
    Vector         vVelocityBody;   // velocity in body coordinates  
    Vector         vAngularVelocity; // angular velocity in body coordinates  
    Vector         vEulerAngles;    // Euler angles in body coordinates  
    float          fSpeed;          // speed (magnitude of the velocity)  
    Quaternion     qOrientation;    // orientation in earth coordinates  
    Vector         vForces;          // total force on body  
    Vector         vMoments;        // total moment (torque) on body  
}; RigidBody, *pRigidBody;
```

You'll notice that it is very similar to the `RigidBody2D` structure that we used in the 2D hovercraft simulation. One significant difference, however, is that in the 2D case, orientation was a single `float` value, and now in 3D it's a quaternion of type `Quaternion`. We discussed the use of quaternions for tracking rigid-body orientation in the previous chapter, and [Appendix C](#) contains a complete definition of the `Quaternion` class.

The next step in defining the flight model is to prepare an initialization function to initialize the airplane at the start of the simulation. For this purpose, we've prepared a function called `InitializeAirplane`:

```
RigidBody     Airplane;      // global variable representing the airplane  
.  
.  
.  
  
void     InitializeAirplane(void)  
{  
    float iRoll, iPitch, iYaw;  
  
    // Set initial position  
    Airplane.vPosition.x = -5000.0f;  
    Airplane.vPosition.y = 0.0f;  
    Airplane.vPosition.z = 2000.0f;  
  
    // Set initial velocity  
    Airplane.vVelocity.x = 60.0f;  
    Airplane.vVelocity.y = 0.0f;  
    Airplane.vVelocity.z = 0.0f;
```

```

Airplane.fSpeed = 60.0f;

// Set initial angular velocity
Airplane.vAngularVelocity.x = 0.0f;
Airplane.vAngularVelocity.y = 0.0f;
Airplane.vAngularVelocity.z = 0.0f;

// Set the initial thrust, forces, and moments
Airplane.vForces.x = 500.0f;
Airplane.vForces.y = 0.0f;
Airplane.vForces.z = 0.0f;
ThrustForce = 500.0;

Airplane.vMoments.x = 0.0f;
Airplane.vMoments.y = 0.0f;
Airplane.vMoments.z = 0.0f;

// Zero the velocity in body space coordinates
Airplane.vVelocityBody.x = 0.0f;
Airplane.vVelocityBody.y = 0.0f;
Airplane.vVelocityBody.z = 0.0f;

// Set these to false at first,
// you can control later using the keyboard
Stalling = false;
Flaps = false;

// Set the initial orientation
iRoll = 0.0f;
iPitch = 0.0f;
iYaw = 0.0f;
Airplane.qOrientation = MakeQFromEulerAngles(iRoll, iPitch, iYaw);

// Now go ahead and calculate the plane's mass properties
CalcAirplaneMassProperties();
}

```

This function sets the initial location, speed, attitude, and thrust for the airplane and goes on to calculate its mass properties by making a call to `CalcAirplaneMassProperties`. You'll see much more of this function in [Chapter 15](#), so we won't show the whole thing here. We do want to point out a portion of the code that is distinctly different from what you do in a 2D simulation, and that's the calculation of the moment of inertia tensor:

```

void      CalcAirplaneMassProperties(void)
{
    .
    .

    // Now calculate the moments and products of inertia for the
    // combined elements.

```

```

// (This inertia matrix (tensor) is in body coordinates)
Ixx = 0;      Iyy = 0;      Izz = 0;
Ixz = 0;      Ixz = 0;      Iyz = 0;
for (i = 0; i< 8; i++)
{
    Ixx += Element[i].vLocalInertia.x + Element[i].fMass *
        (Element[i].vCGCoords.y*Element[i].vCGCoords.y +
         Element[i].vCGCoords.z*Element[i].vCGCoords.z);
    Iyy += Element[i].vLocalInertia.y + Element[i].fMass *
        (Element[i].vCGCoords.z*Element[i].vCGCoords.z +
         Element[i].vCGCoords.x*Element[i].vCGCoords.x);
    Izz += Element[i].vLocalInertia.z + Element[i].fMass *
        (Element[i].vCGCoords.x*Element[i].vCGCoords.x +
         Element[i].vCGCoords.y*Element[i].vCGCoords.y);
    Ixy += Element[i].fMass * (Element[i].vCGCoords.x *
        Element[i].vCGCoords.y);
    Ixz += Element[i].fMass * (Element[i].vCGCoords.x *
        Element[i].vCGCoords.z);
    Iyz += Element[i].fMass * (Element[i].vCGCoords.y *
        Element[i].vCGCoords.z);
}
// Finally set up the airplane's mass and its inertia matrix and take the
// inverse of the inertia matrix
Airplane.fMass = mass;
Airplane.mInertia.e11 = Ixx;
Airplane.mInertia.e12 = -Ixy;
Airplane.mInertia.e13 = -Ixz;
Airplane.mInertia.e21 = -Ixy;
Airplane.mInertia.e22 = Iyy;
Airplane.mInertia.e23 = -Iyz;
Airplane.mInertia.e31 = -Ixz;
Airplane.mInertia.e32 = -Iyz;
Airplane.mInertia.e33 = Izz;

Airplane.mInertiaInverse = Airplane.mInertia.Inverse();
}

```

The airplane is modeled by a number of elements, each representing a different part of the airplane’s structure—for example, the tail rudder, elevators, wings, and fuselage (see [Chapter 15](#) for more details). The code specified here takes the mass properties of each element and combines them, using the techniques discussed in [Chapter 7](#) to come up with the combined inertia tensor for the entire aircraft. The important distinction between these calculations in a 3D simulation and the 2D simulation is that here the inertia is a tensor and in 2D it is a single scalar.

`InitializeAirplane` is called at the very start of the program. We found it convenient to make the call right after the application’s main window is created.

The final part of the flight model has to do with calculating the forces and moments that act on the airplane at any given instant in time during the simulation. As in the 2D

hovercraft simulation, without this sort of function, the airplane will do nothing. For this purpose we've defined a function called `CalcAirplaneLoads`, which is called at every step through the simulation. This function relies on a couple of other functions—namely, `LiftCoefficient`, `DragCoefficient`, `RudderLiftCoefficient`, and `RudderDragCoefficient`. All of these functions are shown and discussed in detail in the section “[Modeling](#)” on page 305 in [Chapter 15](#).

For the most part, the code contained in `CalcAirplaneLoads` is similar to the code you've seen in the `CalcLoads` function of the hovercraft simulation. `CalcAirplaneLoads` is a little more involved since the airplane is modeled by a number of elements that contribute to the total lift and drag on the airplane. There's also another difference that we've noted here:

```
void      CalcAirplaneLoads(void)
{
    .
    .
    .

    // Convert forces from model space to earth space
    Airplane.vForces = QVRotate(Airplane.qOrientation, Fb);

    // Apply gravity (g is defined as -32.174 ft/s^2)
    Airplane.vForces.z += g * Airplane.fMass;

    .
    .
    .

}
```

Just about all of the forces acting on the airplane are first calculated in body-fixed coordinates and then converted to earth-fixed coordinates before the gravity force is applied. The coordinate conversion is effected through the use of the function `QVRotate`, which rotates the force vector based on the airplane's current orientation, represented by a quaternion.¹

Integration

Now that the code to define, initialize, and calculate loads on the airplane is complete, you need to develop the code to actually integrate the equations of motion so that the simulation can progress through time. The first thing you need to do is decide on the integration scheme that you want to use. In this example, we decided to go with the basic Euler's method. We've already discussed some better methods in [Chapter 7](#). We're going with Euler's method here because it's simple and we didn't want to make the code

1. `QVRotate` is defined in [Appendix C](#).

here overly complex, burying some key code that we need to point out to you. In practice, you're better off using one of the other methods we discuss in [Chapter 7](#) instead of Euler's method. With that said, we've prepared a function called `StepSimulation` that handles all of the integration necessary to actually propagate the simulation:

```
void      StepSimulation(float dt)
{
    // Take care of translation first:
    // (If this body were a particle, this is all you would need to do.)

    Vector Ae;

    // calculate all of the forces and moments on the airplane:
    CalcAirplaneLoads();

    // calculate the acceleration of the airplane in earth space:
    Ae = Airplane.vForces / Airplane.fMass;

    // calculate the velocity of the airplane in earth space:
    Airplane.vVelocity += Ae * dt;

    // calculate the position of the airplane in earth space:
    Airplane.vPosition += Airplane.vVelocity * dt;

    // Now handle the rotations:
    float mag;

    // calculate the angular velocity of the airplane in body space:
    Airplane.vAngularVelocity += Airplane.mInertiaInverse *
        (Airplane.vMoments -
        (Airplane.vAngularVelocity^
        (Airplane.mInertia *
        Airplane.vAngularVelocity)))
        * dt;

    // calculate the new rotation quaternion:
    Airplane.qOrientation += (Airplane.qOrientation *
        Airplane.vAngularVelocity) *
        (0.5f * dt);

    // now normalize the orientation quaternion:
    mag = Airplane.qOrientation.Magnitude();
    if (mag != 0)
        Airplane.qOrientation /= mag;

    // calculate the velocity in body space:
    // (we'll need this to calculate lift and drag forces)
    Airplane.vVelocityBody = QVRotate(~Airplane.qOrientation,
        Airplane.vVelocity);

    // calculate the air speed:
```

```

Airplane.fSpeed = Airplane.vVelocity.Magnitude();

// get the Euler angles for our information
Vector u;

u = MakeEulerAnglesFromQ(Airplane.qOrientation);
Airplane.vEulerAngles.x = u.x;      // roll
Airplane.vEulerAngles.y = u.y;      // pitch
Airplane.vEulerAngles.z = u.z;      // yaw

}

```

The very first thing that `StepSimulation` does is call `CalcAirplaneLoads` to calculate the loads acting on the airplane at the current instant in time. `StepSimulation` then goes on to calculate the linear acceleration of the airplane based on current loads. Next, the function goes on to integrate, using Euler's method, once to calculate the airplane's linear velocity and then a second time to calculate the airplane's position. As we've commented in the code, if you were simulating a particle this is all you would have to do; however, since this is not a particle, you need to handle angular motion.

The first step in handling angular motion is to calculate the new angular velocity at this time step, using Euler integration, based on the previously calculated moments acting on the airplane and its mass properties. We do this in body coordinates using the following equation of angular motion but rewritten to solve for $d\omega$:

As another convenience, we calculate the air speed, which is simply the magnitude of the linear velocity vector. This is used to report the air speed in the main window title bar.

Lastly, the three Euler angles—roll, pitch, and yaw—are extracted from the orientation quaternion so that they can also be reported in the main window title bar. The function to use here is `MakeEulerAnglesFromQ`, which is defined in [Appendix C](#).

Don't forget, `StepSimulation` must be called once per simulation cycle.

Flight Controls

At this point, the simulation still won't work very well because you have not implemented the flight controls. The flight controls allow you to interact with the airplane's various controls surfaces in order to actually fly the plane. We'll use the keyboard as the main input device for the flight controls. Remember, in a physics-based simulation such as this one, you don't directly control the motion of the airplane; you control only how various forces are applied to the airplane, which then, by integration over time, affect the airplane's motion.

For this simulation, the flight stick is simulated by the arrow keys. The down arrow pulls back on the stick, raising the nose; the up arrow pushes the stick forward, causing the nose to dive; the left arrow rolls the plane to the left (port side); and the right arrow rolls the plane to the right (starboard side). The X key applies left rudder action to cause the nose of the plane to yaw toward the left, while the C key applies right rudder action to cause the nose to yaw toward the right. Thrust is controlled by the A and Z keys. The A key increments the propeller thrust by 100 pounds, and the Z key decrements the thrust by 100 pounds. The minimum thrust is 0, while the maximum available thrust is 3,000 pounds. The F key activates the landing flaps to increase lift at low speed, while the D key deactivates the landing flaps.

We control pitch by deflecting the flaps on the aft elevators; for example, to pitch the nose up, we deflect the aft elevator flaps upward (that is, the trailing edge of the elevator is raised with respect to the leading edge). We control roll in this simulation by applying the flaps differentially; for example, to roll right, we deflect the right flap upward and the left flap downward. Finally, we control yaw by deflecting the vertical tail rudder; for example, to yaw left, we deflect the trailing edge of the tail rudder toward the left.

We've prepared several functions to handle the flight controls that should be called whenever the user is pressing one of the flight control keys. There are two functions for the propeller thrust:

```
void      IncThrust(void)
{
    ThrustForce += _DTHRUST;
    if(ThrustForce > _MAXTHRUST)
```

```

        ThrustForce = _MAXTHRUST;
    }

void      DecThrust(void)
{
    ThrustForce -= _DTHRUST;
    if(ThrustForce < 0)
        ThrustForce = 0;
}

```

`IncThrust` simply increases the thrust by `_DTHRUST` checking to make sure it does not exceed `_MAXTHRUST`. We've defined `_DTHRUST` and `_MAXTHRUST` as follows:

```

#define      _DTHRUST      100.0f
#define      _MAXTHRUST    3000.0f

```

`DecThrust`, on the other hand, decreases the thrust by `_DTHRUST` checking to make sure it does not fall below 0.

To control yaw, we've prepared three functions that manipulate the rudder:

```

void      LeftRudder(void)
{
    Element[6].fIncidence = 16;
}

void      RightRudder(void)
{
    Element[6].fIncidence = -16;
}

void      ZeroRudder(void)
{
    Element[6].fIncidence = 0;
}

```

`LeftRudder` changes the incidence angle of `Element[6]`, the vertical tail rudder, to 16 degrees, while `RightRudder` changes the incidence angle to -16 degrees. `ZeroRudder` centers the rudder at 0 degrees.

The ailerons, or flaps, are manipulated by these functions to control roll:

```

void      RollLeft(void)
{
    Element[0].iFlap = 1;
    Element[3].iFlap = -1;
}

void      RollRight(void)
{
    Element[0].iFlap = -1;
    Element[3].iFlap = 1;
}

```

```

void      ZeroAilerons(void)
{
    Element[0].iFlap = 0;
    Element[3].iFlap = 0;
}

```

`RollLeft` deflects the port aileron, located on the port wing section (`Element[0]`), upward, and the starboard aileron, located on the starboard wing section (`Element[3]`), downward. `RollRight` does just the opposite, and `ZeroAilerons` resets the flaps back to their undeflected positions.

We've defined yet another set of functions to control the aft elevators so as to control pitch:

```

void      PitchUp(void)
{
    Element[4].iFlap = 1;
    Element[5].iFlap = 1;
}

void      PitchDown(void)
{
    Element[4].iFlap = -1;
    Element[5].iFlap = -1;
}

void      ZeroElevators(void)
{
    Element[4].iFlap = 0;
    Element[5].iFlap = 0;
}

```

`Element[4]` and `Element[5]` are the elevators. `PitchUp` deflects their flaps upward, and `PitchDown` deflects their flaps downward. `ZeroElevators` resets their flaps back to their undeflected positions.

Finally, there are two more functions to control the landing flaps:

```

void      FlapsDown(void)
{
    Element[1].iFlap = -1;
    Element[2].iFlap = -1;
    Flaps = true;
}

void      ZeroFlaps(void)
{
    Element[1].iFlap = 0;
    Element[2].iFlap = 0;
}

```

```
    Flaps = false;  
}
```

The landing flaps are fitted on the inboard wings sections, port and starboard, which are `Element[1]` and `Element[2]`. `FlapsDown` deflects the flaps downward, while `ZeroFlaps` resets them back to their undeflected position.

As we said, these functions should be called when the user is pressing the flight control keys. Further, they need to be called before `StepSimulation` is called so that they can be included in the current time step's force and moment calculations. The sequence of calls should look something like this:

```
.  
. .  
  
ZeroRudder();  
ZeroAilerons();  
ZeroElevators();  
  
// pitch down  
if (IsKeyDown(VK_UP))  
    PitchDown();  
  
// pitch up  
if (IsKeyDown(VK_DOWN))  
    PitchUp();  
  
// roll left  
if (IsKeyDown(VK_LEFT))  
    RollLeft();  
  
// roll right  
if (IsKeyDown(VK_RIGHT))  
    RollRight();  
  
// Increase thrust  
if (IsKeyDown(0x41)) // A  
    IncThrust();  
  
// Decrease thrust  
if (IsKeyDown(0x5A)) // Z  
    DecThrust();  
  
// yaw left  
if (IsKeyDown(0x58)) // x  
    LeftRudder();  
  
// yaw right  
if (IsKeyDown(0x43)) // c  
    RightRudder();
```

```

// landing flaps down
if (IsKeyDown(0x46)) //f
    FlapsDown();

// landing flaps up
if (IsKeyDown(0x44)) // d
    ZeroFlaps();

StepSimulation(dt);
.
.
.
```

Before `StepSimulation` is called, we check each of the flight control keys to see if it is being pressed. If so, then the appropriate function is called.

The function `IsKeyDown`, which checks whether a certain key is pressed, looks like this in a Windows implementation:

```

BOOL IsKeyDown(short KeyCode)
{
    SHORT      retval;

    retval = GetAsyncKeyState(KeyCode);

    if (HIBYTE(retval))
        return TRUE;

    return FALSE;
}
```

The important thing to note here is that the keys are being checked asynchronously because it is possible that more than one key will be pressed at any given time, and they must be handled simultaneously instead of one at a time (as would be the case in the standard Windows message processing function).

The addition of flight control code pretty much completes the physics part of the simulation. So far, you have the model, the integrator, and the user input or flight control elements completed. All that remains is setting up the application's main window and actually drawing something that represents what you're simulating. We'll leave that part up to you, or you can look at the example we've included on the book's website to see what we did on a Windows machine.

CHAPTER 13

Connecting Objects

Simulating particles and rigid bodies is great fun, and with these simple entities you can achieve a wide variety of effects or simulate a wide variety of objects. In this chapter we'll take things a step further, showing you how to simulate connected particles and rigid bodies. Doing so opens a whole new realm of possibilities. In this book's first edition, David showed you how to use springs and particles to simulate cloth. [Chapter 17](#) in the first edition covers that, and the corresponding "Cloth Simulation" example on the book's website implements the model. As shown in [Figure 13-1](#), the flag model is simply a collection of particles initially laid out in a grid pattern connected by linear springs that are then rendered to look like cloth. The springs give structure to the particles, keeping them organized in a mesh that can be rendered while allowing them to move, emulating the movement of a flowing fabric.

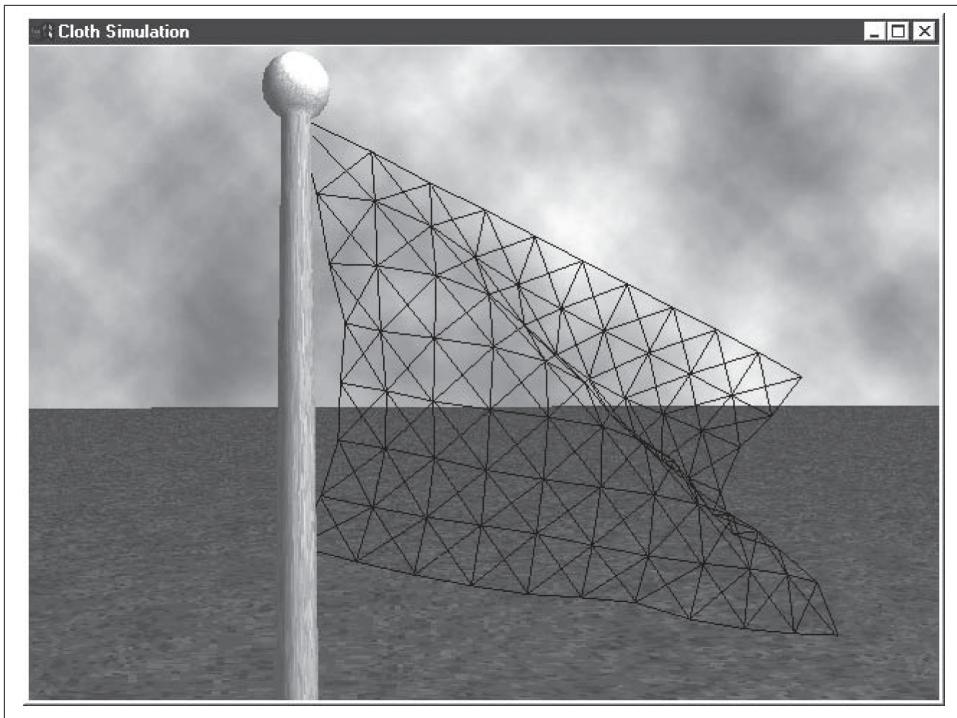


Figure 13-1. Network of particles and springs

Each line in the wireframe flag shown in Figure 13-1 represents a spring-damper element, while the nodes where these springs intersect represent the particles. We modeled the springs using the spring-damper formulas that we showed you back in Chapter 3. The (initially) horizontal and vertical springs provide the basic structure for the flag, while the diagonal springs are there to resist shear forces and lend further strength to the cloth. Without these shear springs, the cloth would be quite stretchy. Note that there are no particles located at the intersection of the diagonal springs.

In this chapter, we'll show you how to use those same techniques to simulate something like a hanging rope or vine. You can use these techniques to simulate all sorts of things besides cloth and rope or vines. For example, you can model the swing of a golf club if you can imagine one rigid body representing the arm and another representing the golf club. We'll get to that example in Chapter 19, but for now let's see how to model a hanging rope or vine and some other springy objects.

Application of linear springs is not the only method available to connect objects, but it has the advantages of being conceptually simple, easy to implement, and effective. One of the potential disadvantages is that you can run into numerical stability problems if the springs are too stiff. We'll talk more about these issues throughout this chapter. Also, the examples we'll cover are in 2D for simplicity, but the techniques apply in 3D, too.

Springs and Dampers

You learned in [Chapter 3](#) that springs are structural elements that, when connected between two objects, apply equal and opposite forces to each object. This spring force follows Hooke's law and is a function of the stretched or compressed length of the spring relative to the rest length of the spring and the spring constant. The spring constant is a quantity that relates the force exerted by the spring to its deflection:

body 1 minus the position of body 2. Similarly, \mathbf{v}_1 and \mathbf{v}_2 are the velocities of the connected points on bodies 1 and 2. The quantity $(\mathbf{v}_1 - \mathbf{v}_2)$ represents the relative velocity between the connected bodies.

It's fairly straightforward to connect particles with springs (and dampers); you need only specify the particles to which the spring is connected and compute the stretched or compressed length of the spring as the particles move relative to each other. The force generated by the spring is then applied equally (but in opposite directions) to the connected particles. This is a linear force.

For rigid bodies, things are a bit more complicated. First, not only do you have to specify to which body the spring is attached, but you must also specify the precise points on each object where the spring attaches. Then, in addition to the linear force applied by the spring to each body, you must also compute the resulting moment on each body causing each to rotate.

Connecting Particles

From swinging vines in Activision's *Pitfall* to barnacle tongues in Valve Corporation's *Half-Life*, dangling rope-like objects have appeared in video games in various incarnations since the very early days of video gaming. Some implementations, such as those in the 1982 versions of *Pitfall*, are implemented rather simply and unrealistically, while others, such as barnacle tongues, are implemented more realistically in how they dangle and swing. Whether it's a vine, rope, chain, or tongue, you can use particles and springs to simulate realistic rope-like behavior. We'll show you how in the following simple example.

Rope

You know from your real-life experience that ropes are flexible, although some are more flexible than others. Ropes are elastic and stretch to varying extents. They drape when suspended by their two ends. They bend when swinging or when collapsing on the ground. We can capture all these behaviors using simple particles connected with springs. [Figure 13-2](#) illustrates the rope example we'll cover here.

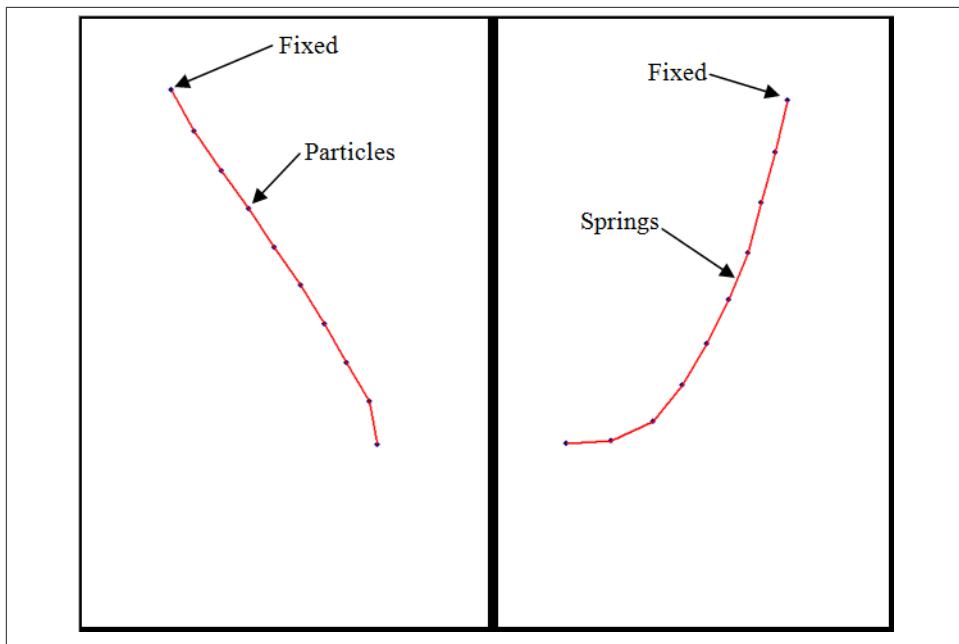


Figure 13-2. Swinging rope

The example consists of a rope comprising 10 particles and 9 springs. At the start of the simulation, the rope, originally extended straight out to the right, falls under the influence of gravity, swinging left and right until it comes to rest (hanging straight down). The dots represent particles and the lines represent springs. The topmost particle is fixed, and the illustration on the left in [Figure 13-2](#) shows the rope swinging down from right to left while the illustration on the right shows the rope swinging back from left to right.

This example uses all the same code and techniques presented in [Chapter 7](#) through [Chapter 9](#) for simulating particles and rigid bodies. Really, the only difference is that we have to compute a new force—the spring force on each object. But before we do that, we have to define and initialize the springs.

Spring structure and variables

The following code sample shows the spring data structure we set up to store each spring's information:

```
typedef struct _Spring {
    int      End1;
    int      End2;
    float   k;
    float   d;
```

```
    float      InitialLength;  
} Spring, *pSpring;
```

Specifically, this information includes:

End1

A reference to the first particle to which the spring is connected

End2

A reference to the second particle to which the spring is connected

k

The spring constant

d

The damping constant

InitialLength

The unstretched length of the spring

This structure is appropriate for connecting particles. We'll make a slight modification to this structure later, when we get to the example where we're connecting rigid bodies.

There are defines and variables unique to this example that must be set up as follows:

```
#define      _NUM_OBJECTS      10  
#define      _NUM_SPRINGS     9  
#define      _SPRING_K        1000  
#define      _SPRING_D        100  
  
Particle      Objects[_NUM_OBJECTS];  
Spring       Springs[_NUM_SPRINGS];
```

As stated earlier, there are 10 particles (objects) and 9 springs in this simulation. The arrays `Objects` and `Springs` are used to keep track of them. We also set up a few defines representing the spring and damping constants. The values shown here are arbitrary, and you can change them to suit whatever behavior you desire. The higher the spring constant, the stiffer the springs; whereas the lower the spring constant, the stretchier the springs. Stretchy springs make your rope more elastic. Keep in mind while tuning these values that if you make the spring constant too high, you'll probably have to make the simulation time step smaller and/or use a robust integration scheme to avoid numerical instabilities.

The damping constant controls how quickly the springiness of the springs dampens out. You'll end up tuning this value to get the behavior you desire. A small value can make the rope seem jittery, while a large value will make the stretchiness appear smoother. Higher damping also helps alleviate numerical instabilities to some extent, although it's no substitute for a robust integration scheme.

Initialize the particles and springs

Initially, our particle rope is set up horizontally, as shown in [Figure 13-3](#), with the left-most particle, p_0 , fixed—that is, the particle p_0 will not move, and the remainder of the rope will pivot about p_0 . For convenience, all remaining particles are incrementally indexed from left to right.

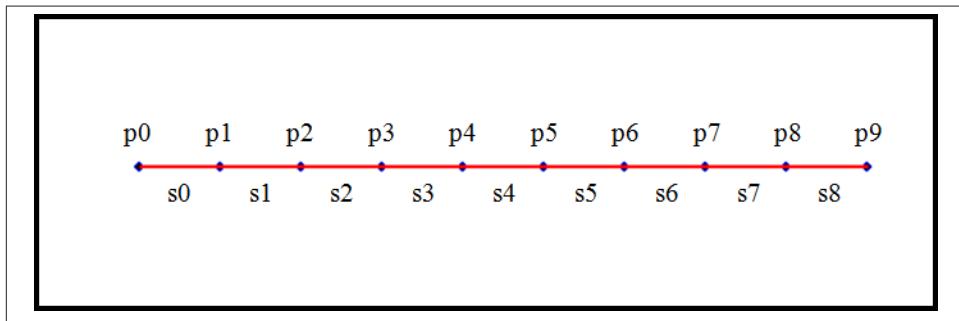


Figure 13-3. Particle rope setup

There are nine springs, which are indexed from left to right as illustrated in [Figure 13-3](#). Spring 0 connects particle 0 to particle 1, spring 1 connects particle 1 to particle 2, and so on. The following code sample shows how all this is initialized:

```
bool Initialize(void)
{
    Vector r;
    int i;

    Objects[0].bLocked = true;

    // Initialize particle locations from left to right.
    for(i=0; i<_NUM_OBJECTS; i++)
    {
        Objects[i].vPosition.x = _WINWIDTH/2 + Objects[0].fLength * i;
        Objects[i].vPosition.y = _WINHEIGHT/8;
    }

    // Initialize springs connecting particles from left to right.
    for(i=0; i<_NUM_SPRINGS; i++)
    {
        Springs[i].End1 = i;
        Springs[i].End2 = i+1;
        r = Objects[i+1].vPosition - Objects[i].vPosition;

        Springs[i].InitialLength = r.Magnitude();
        Springs[i].k = _SPRING_K;
        Springs[i].d = _SPRING_D;
    }
}
```

```
        return true;
    }
```

First, the local variables `r` and `i` are declared. `r` will be used to compute the initial, unstretched length of the springs, and `i` will be used to index the `Objects` and `Springs` arrays. Second, `Objects[0]`, the one that is fixed, has its `bLocked` property set to `true`, indicating that it does not move (that is, it's locked).

Next, the particle positions are initialized starting from the first particle—the fixed one positioned at the middle of the screen—and proceeding to the rest of the particles, offsetting each to the right by an amount equal to property `fLength`. `fLength` is an arbitrary length that you can define, that represents the spacing of the particles and, subsequently, the initial length of the springs connecting the particles.

Finally, we set up the springs, connecting each particle to its neighbor on the right. Starting at the first spring, we set its end references to the index of the particle to its left and right in the properties `End1` and `End2`, respectively. These indices are simply `i` and `i+1`, as shown in the preceding code sample within the last `for` loop. The initial length vector of the spring is computed and stored in the vector `r`, where `r = Objects[i + 1].vPosition - Objects[i].vPosition`. The magnitude of this vector is the initial spring length, which is stored in the spring's property, `InitialLength`. This step isn't strictly necessary in this example since you already know that the property `fLength` discussed earlier is the initial length of each spring. However, we've done it this general way since you may not necessarily initialize the particle positions as we have simply done.

Update the simulation

Updating the particle positions at each simulation time step, under the influence of gravity and spring forces proceeds just like in the earlier examples of [Chapter 8](#) and [Chapter 9](#). Essentially, you must compute the forces on the particles, integrate the equations of motion, and redraw the scene. As usual in our examples, the function `UpdateSimulation` is called on to perform these tasks. For the current example, `UpdateSimulation` looks like this:

```
void UpdateSimulation(void)
{
    double dt = _Timestep;
    int i;
    double f, dl;
    Vector pt1, pt2;
    int j;
    Vector r;
    Vector F;
    Vector v1, v2, vr;

    // Initialize the spring forces on each object to zero.
    for(i=0; i<_NUM_OBJECTS; i++)
```

```

{
    Objects[i].vSprings.x = 0;
    Objects[i].vSprings.y = 0;
    Objects[i].vSprings.z = 0;
}

// Calculate all spring forces based on positions of connected objects.
for(i=0; i<_NUM_SPRINGS; i++)
{
    j = Springs[i].End1;
    pt1 = Objects[j].vPosition;
    v1 = Objects[j].vVelocity;

    j = Springs[i].End2;
    pt2 = Objects[j].vPosition;
    v2 = Objects[j].vVelocity;

    vr = v2 - v1;
    r = pt2 - pt1;
    dl = r.Magnitude() - Springs[i].InitialLength;
    f = Springs[i].k * dl; // - means compression, + means tension
    r.Normalize();
    F = (r*f) + (Springs[i].d*(vr*r))*r;

    j = Springs[i].End1;
    Objects[j].vSprings += F;

    j = Springs[i].End2;
    Objects[j].vSprings -= F;
}

.

.

.

// Integrate equations of motion as usual.
.

.

.

// Render the scene as usual.
.

.

.

}

```

As you can see, there are several local variables here. We'll explain each one as we get to the code where it's used. After the local variable declarations, this function's first task is to reset the aggregate spring forces on each particle to 0. Each particle stores the aggregate spring force in the property `vSprings`, which is a vector. In this example, each particle will have up to two springs acting on it at any given time.

The next block of code in the `for` loop computes the springs forces acting on each particle. There are several steps to this, so we'll go through each one. First the loop is set up to step through the list of springs. Recall that each spring is connected to two particles, so each step through the loop will compute a spring force and apply it to two separate particles.

Within the loop, the variable `j` is used as a convenience to temporarily store the index that refers to the `Object` to which the spring is attached. For each spring `j` is first set to the spring's `End1` property. A temporary variable, `pt1`, is then set equal to the position of the `Object` to which `j` refers. Another temporary variable, `v1`, is set to the velocity of the `Object` to which `j` refers. Next, `j` is set to the index of `End2`, the other `Object` to which the current spring is attached, and that object's position and velocity are stored in `pt2` and `v2`, respectively. This sort of temporary variable use isn't necessary, of course, but it makes the following lines of code that compute the spring force more readable in our opinion.

`vr` is a vector that stores the relative velocity between the two ends of the spring. We compute `vr` by subtracting `v1` from `v2`. Similarly, `r` is a vector that stores the relative distance between the two ends of the spring. We compute `r` by subtracting `pt1` from `pt2`. The magnitude of `r` represents the stretched or compressed length of the spring. The change in spring length is computed and stored in `dl` as follows:

```
dl = r.Magnitude() - Springs[i].InitialLength;
```

`dl` will be negative if the computed length is shorter than the initial length of the spring. This implies that the spring is in compression and should act to push the particles away from each other. A positive `dl` means the spring is in tension and should act to pull the particles toward each other. The line:

```
f = Springs[i].k * dl;
```

computes the corresponding spring force as a function of `dl` and the spring constant. Note that `f` is a scalar and we have not yet computed its line of action, although we know it acts along the line connecting the particles at `End1` and `End2`. That line is represented by `r`, which we computed earlier. And the spring force is just `f` times the unit vector along `r`. Since we're including damping, we have to use the spring-damper equation for the total force acting on each particle, which we call the vector `F`. `F` is computed as follows:

```
F = (r*f) + (Springs[i].d*(vr*r))*r;
```

The first term on the right side of the equals sign is the Hooke's law-based spring force, and the second term is the damping force. Note here that `r` is a unit vector previously computed using the line:

```
r.Normalize();
```

Finally, the spring force is applied to each particle connected by the spring. Remember, the force is equal in magnitude but opposite in direction for each particle. The lines:

```
j = Springs[i].End1;  
Objects[j].vSprings += F;
```

apply the spring force to the particle at the first end of the spring, whereas the lines:

```
j = Springs[i].End2;  
Objects[j].vSprings -= F;
```

apply the opposite spring force to the particle at the second end of the spring.

That's it for computing and applying the spring forces. The remainder of the code is business as usual, where we compute the force due to gravity and add it to the aggregate spring force for each particle and then integrate the equations of motion. Finally, we render the scene at each time step.

Connecting Rigid Bodies

As with particles, you can connect rigid bodies with springs to simulate some interesting things. For example, you may want to simulate something as simple as a linked chain, where each link is connected to the other in series. Or perhaps you want to simulate connected body parts to simulate rag doll physics or maybe a golfer's swing. All these require some means of connecting rigid bodies. In this section we'll show you how to use linear spring-dampers, the same we've discussed already, to connect rigid bodies. We'll start with a simple analog to the rope example discussed earlier. Instead of connecting particles with springs to simulate a dangling rope, we'll connect rigid links to simulate a dangling rope or chain. Later, we'll show you how linear springs can be used to restrain angular motion.

Links

In this example, each link is rigid in that it does not deform; however, the links are connected by springs in a way that allows the ensemble to swing, stretch, and bend in a manner similar to a hanging chain. [Figure 13-4](#) illustrates our swinging linked chain as it swings from right to left and then back toward the right.

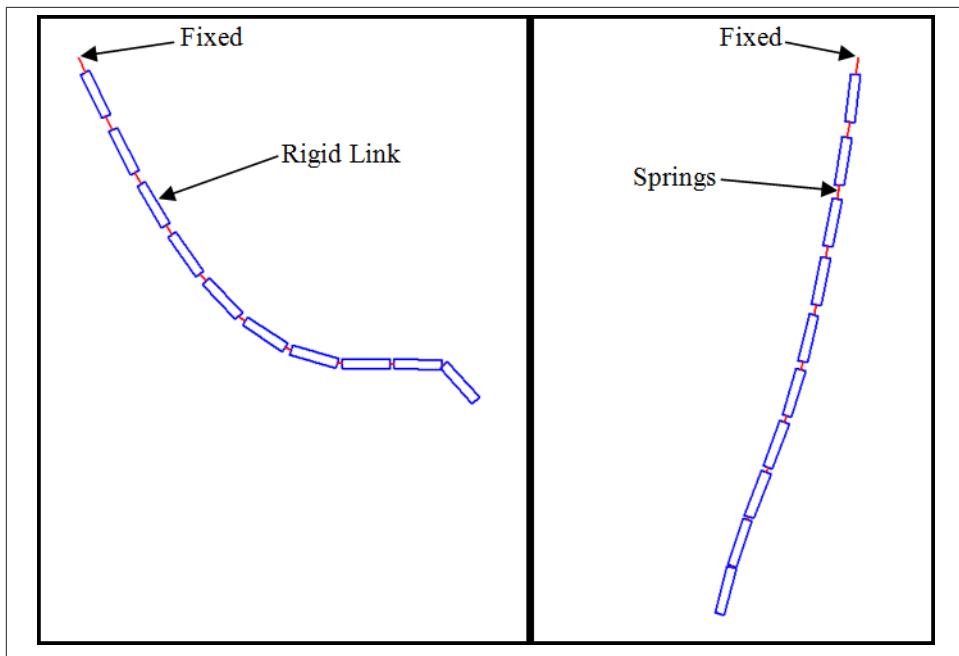


Figure 13-4. Swinging links

As in the rope example, the topmost link is connected to a fixed point by a spring, such that the linked chain pivots around and hangs from the fixed point. The rectangles represent each rigid link, with the lines connecting the rectangles representing springs.

To model this linked chain, we need only make a few changes to the rope example to address the fact that we're now dealing with rigid bodies that can rotate versus particles. This requires us to specify the point on each body to which the springs are attached, and in addition to computing the spring forces acting on each body, we must also compute the moments due to those forces. Aside from these spring force and moment computations, the remainder of the simulation is the same as those discussed in [Chapter 7](#) through [Chapter 12](#).

Basic structures and variables

We can use the `Spring` structure shown earlier in the rope example again here with one small modification. Basically, we need to change the type of the endpoint references, `End1` and `End2`, from integers to a new structure we'll call `EndPoint`. The new `Spring` structure looks like this:

```
typedef struct _Spring {
    EndPoint    End1;
    EndPoint    End2;
    float       k;
```

```

        float      d;
        float      InitialLength;
    } Spring, *pSpring;
}

```

The new `EndPoint` structure is as follows:

```

typedef struct _EndPointRef {
    int      ref;
    Vector   pt;
} EndPoint;

```

Here, `ref` is the index referring to the `Object` to which the spring is attached, and `pt` is the point in the attached `Object`'s local coordinate system to which the spring is attached. Notice from [Figure 13-4](#) that the first spring, the topmost one, is connected to a single object; the other end of it is connected to a fixed point in space. We'll use a `ref` of `-1` to indicate that a spring's endpoint is connected to a fixed point in space instead of an object.

As in the rope example, we have a few important `defines` and variables to set up:

```

#define      _NUM_OBJECTS     10
#define      _NUM_SPRINGS    10
#define      _SPRING_K        1000
#define      _SPRING_D        100

RigidBody2D      Objects[_NUM_OBJECTS];
Spring           Springs[_NUM_SPRINGS];

```

These are the same as before except now we have 10 springs instead of 9, and `Objects` is of type `RigidBody2D` instead of `Particle`.

The damping and spring constants play the same role here as they did in the rope example.

Initialize

Initially our linked chain is set up horizontally, just like the rope example, but with the link and spring indices shown in [Figure 13-5](#). Each rectangle represents a rigid link, and a spring attached to the left end of each link connects the link to its neighbor to the left. In the case of the first link, L_0 , the spring connects the left end of the link to a fixed point in space.

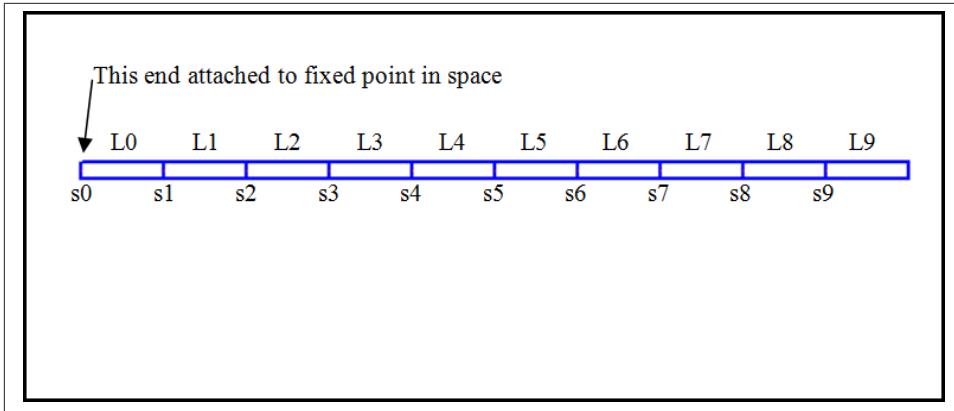


Figure 13-5. Linked-chain setup

The code for this setup is only a little more involved than that for the rope example; the additional complexity is due to having to deal with specific points on the rigid bodies to which each spring is attached. The following code sample contains the modified Initialize function:

```

bool Initialize(void)
{
    Vector r;
    Vector pt;
    int i;

    // Initialize objects for linked chain.
    for(i=0; i<_NUM_LINKS; i++)
    {
        Objects[i].vPosition.x = _WINWIDTH/2 + Objects[0].fLength * i;
        Objects[i].vPosition.y = _WINHEIGHT/8;
        Objects[i].fOrientation = 0;
    }

    // Connect end of the first object to a fixed point in space.
    Springs[0].End1.ref = -1;
    Springs[0].End1.pt.x = _WINWIDTH/2-Objects[0].fLength/2;
    Springs[0].End1.pt.y = _WINHEIGHT/8;

    Springs[0].End2.ref = 0;
    Springs[0].End2.pt.x = -Objects[0].fLength/2;
    Springs[0].End2.pt.y = 0;

    pt = VRotate2D(Objects[0].fOrientation, Springs[0].End2.pt)
        + Objects[0].vPosition;
    r = pt - Springs[0].End1.pt;

    Springs[0].InitialLength = r.Magnitude();
}

```

```

Springs[0].k = _SPRING_K;
Springs[0].d = _SPRING_D;

// Connect end of all remaining springs.
for(i=1; i<_NUM_LINKS; i++)
{
    Springs[i].End1.ref = i-1;
    Springs[i].End1.pt.x = Objects[i-1].fLength/2;
    Springs[i].End1.pt.y = 0;

    Springs[i].End2.ref = i;
    Springs[i].End2.pt.x = -Objects[i].fLength/2;
    Springs[i].End2.pt.y = 0;

    pt = VRotate2D(Objects[i].fOrientation, Springs[i].End2.pt)
        + Objects[i].vPosition;
    r = pt - (VRotate2D(Objects[i-1].fOrientation, Springs[i].End1.pt)
        + Objects[i-1].vPosition);

    Springs[i].InitialLength = r.Magnitude();
    Springs[i].k = _SPRING_K;
    Springs[i].d = _SPRING_D;
}

return true;
}

```

The local variables `r` and `i` are the same as before; however, there's a new variable, `pt`, that we use to temporarily store the coordinates of specific points when converting from one coordinate system to another. We'll see how this is done shortly.

After the local variables are declared, the `Object` positions are initialized, starting from the first `Object` positioned at the middle of the screen and proceeding to the rest of the `Objects`, offsetting each to the right by an amount equal to property `fLength`. Here, `fLength` is an arbitrary length representing the length of each rigid body, not the length of the springs connecting each rigid body. As you'll see momentarily, the initial length of all the springs in this example is 0.

You should be aware that the coordinates for each object computed here are the coordinates of the object's center of gravity, which in this example we defined as the middle of the rectangle representing each object. Since these are rigid bodies, not only must you specify their initial positions, but you must also specify their initial orientations as shown in the preceding code sample. The way we have this example set up, each object is initialized with an orientation of 0 degrees.

The next task is to set up the spring connecting the first link, the one on the left, to a fixed point in space. The following code handles this task:

```

// Connect end of the first object to a fixed point in space.
Springs[0].End1.ref = -1;

```

```

Springs[0].End1.pt.x = _WINWIDTH/2-Objects[0].fLength/2;
Springs[0].End1.pt.y = _WINHEIGHT/8;

Springs[0].End2.ref = 0;
Springs[0].End2.pt.x = -Objects[0].fLength/2;
Springs[0].End2.pt.y = 0;

pt = VRotate2D(Objects[0].fOrientation, Springs[0].End2.pt)
    + Objects[0].vPosition;
r = pt - Springs[0].End1.pt;

Springs[0].InitialLength = r.Magnitude();
Springs[0].k = _SPRING_K;
Springs[0].d = _SPRING_D;

```

The first spring, `Spring[0]`, has its first endpoint, `End1`, set to refer to `-1`, which, as explained earlier, means that this end of the spring is connected to some fixed point in space. The location of the point, stored in the `End1.pt` property, must be specified in global coordinates as shown previously.

Now the second end of the first spring is connected to the left end of the first link; therefore, `End2.ref` of the first spring is set to `0`, which is the index to the first `Object`. The point on `Object[0]` to which the spring is attached is the leftmost end on the centerline of the object; thus, its coordinates—relative to the object's center of gravity location and specified in local, body-fixed coordinates—are:

```

Springs[0].End2.pt.x = -Objects[0].fLength/2;
Springs[0].End2.pt.y = 0;

```

Now remember, the points on `Objects` to which springs are attached are specified in body-fixed, local coordinates of each referenced object, whereas any point fixed in space to which a spring is attached and not on an `Object` must be specified in global, earth-fixed coordinates. You have to keep these coordinates straight and make the appropriate rotations when computing spring lengths throughout the simulation. The code;

```

pt = VRotate2D(Objects[0].fOrientation, Springs[0].End2.pt)
    + Objects[0].vPosition;
r = pt - Springs[0].End1.pt;

```

illustrates how to do this. To compute the initial spring length, we need to compute the relative distance between the endpoints of the spring. In case of the first spring, `End1` was specified in global coordinates, but `End2` was specified in the local coordinate system of `Object[0]`. Therefore, we have to convert the coordinates of `End2` from local coordinates to global coordinates before calculating the relative distance between the ends. The preceding line, which calls the `VRotate2D` function you saw in earlier chapters, rotates the locally specified point, `End2.pt`, from local to global coordinates; it then adds the `Object`'s position to the result, arriving at a point, `pt`, in global coordinates coincident with the second endpoint of the spring. The relative distance, `r`, is the second endpoint, `pt`, minus the first endpoint, `End1.pt`.

Finally, we compute the initial length of the spring by taking the magnitude of r and storing the result in the spring's `InitialLength` property.

With the first spring out of the way, the `Initialize` function enters a loop to set up the remaining springs. Proceeding from left to right in [Figure 13-5](#), the first endpoint, `End1`, is connected to the right side of `Object[i-1]`, and the second endpoint, `End2`, is connected to the left side of `Object[i]`. Be aware that each endpoint of each spring is specified in different coordinate systems. The left end is in the coordinate system of `Object[i-1]`, while the right end is in the coordinate system of `Object[i]`. It may seem trivial during this setup, but when things start moving and rotating it is critically important to keep these coordinate systems straight. Doing so involves transforming each endpoint coordinate from the local system of the body to which it's attached to the global coordinate system. This is illustrated as follows:

```
pt = VRotate2D(Objects[i].fOrientation, Springs[i].End2.pt)
      + Objects[i].vPosition;
r = pt - (VRotate2D(Objects[i-1].fOrientation, Springs[i].End1.pt)
      + Objects[i-1].vPosition);
```

The first line converts the spring attachment point `End2` from the local coordinate system of `Object[i]` to global coordinates by performing a rotation and translation using functions you've already seen numerous times now. The result is temporarily stored in the local variable, `pt`. The second line converts the spring attachment point `End1` from the local coordinate system of `Object[i-1]` to global coordinates and then subtracts the result from `pt`, yielding a vector, `r`, representing the relative distance between the spring's endpoints. The magnitude of `r` is the spring's initial length. Performing these same calculations during the simulation will result in the spring's stretched or compressed length. That calculation is performed in `UpdateSimulation`.

Update

The function `UpdateSimulation` is substantially the same as that discussed in the rope example. There are a few differences that we'll highlight here. Again, these differences are due to the fact that we're now dealing with rigid bodies that rotate rather than simple particles. The following code sample shows the additions to `UpdateSimulation`. You can see there are a couple of new variables, `M` and `Fo`. `M` is used to temporarily store moments due to spring forces `Fo` in the local coordinates of each `Object`.

Just as the property `vSprings` was initialized to 0 at the start of `UpdateSimulation`, so too must we initialize `vMSprings` to 0. Recall, `vSprings` aggregates the spring forces acting on each `Object`. For rigid bodies that rotate, we'll use `vMSprings` to aggregate the moments on each `Object` resulting from those spring forces:

```
void UpdateSimulation(void)
{
    .
    .
}
```

```

.
Vector    M;
Vector    Fo;

// Initialize the spring forces and moments on each object to zero.
for(i=0; i<_NUM_OBJECTS; i++)
{
    .
    .
    .

    Objects[i].vMSprings.x = 0;
    Objects[i].vMSprings.y = 0;
    Objects[i].vMSprings.z = 0;
}

// Calculate all spring forces based on positions of connected objects
for(i=0; i<_NUM_SPRINGS; i++)
{
    if(Springs[i].End1.ref == -1)
    {
        pt1 = Springs[i].End1.pt;
        v1.x = v1.y = v1.z = 0; // point is not moving
    } else {
        j = Springs[i].End1.ref;
        pt1 = Objects[j].vPosition + VRotate2D(Objects[j].fOrientation,
                                                Springs[i].End1.pt);
        v1 = Objects[j].vVelocity + VRotate2D(Objects[j].fOrientation,
                                              Objects[j].vAngularVelocity^Springs[i].End1.pt);
    }

    if(Springs[i].End2.ref == -1)
    {
        pt2 = Springs[i].End2.pt;
        v2.x = v2.y = v2.z = 0;
    } else {
        j = Springs[i].End2.ref;
        pt2 = Objects[j].vPosition + VRotate2D(Objects[j].fOrientation,
                                                Springs[i].End2.pt);
        v2 = Objects[j].vVelocity + VRotate2D(Objects[j].fOrientation,
                                              Objects[j].vAngularVelocity^Springs[i].End2.pt);
    }

    // Compute spring-damper force.
    vr = v2 - v1;
    r = pt2 - pt1;
    dl = r.Magnitude() - Springs[i].InitialLength;
    f = Springs[i].k * dl;
    r.Normalize();
    F = (r*f) + (Springs[i].d*(vr*r))*r;

    // Aggregate the spring force on each connected object
}

```

```

j = Springs[i].End1.ref;
if(j != -1)
    Objects[j].vSprings += F;

j = Springs[i].End2.ref;
if(j != -1)
    Objects[j].vSprings -= F;

// convert force to first ref local coords
// Get local lever
// calc moment

// Compute and aggregate moments due to spring force
// on each connected object.
j = Springs[i].End1.ref;
if(j != -1)
{
    Fo = VRotate2D(-Objects[j].fOrientation, F);
    r = Springs[i].End1.pt;
    M = r^Fo;
    Objects[j].vMSprings += M;
}

j = Springs[i].End2.ref;
if(j!= -1)
{
    Fo = VRotate2D(-Objects[j].fOrientation, F);
    r = Springs[i].End2.pt;
    M = r^Fo;
    Objects[j].vMSprings -= M;
}
}

.

.

.

// Integrate equations of motion as usual.
.

.

.

// Render the scene as usual.
.

.

.

}

```

As in the rope example, `UpdateSimulation` steps through all the `Springs`, computing their stretched or compressed length, the relative velocity of each spring's endpoints, and the resulting spring forces. These calculations are a bit different in this current example because we have to handle rotation, as explained earlier.

Upon entering the `for` loop in the preceding code sample, `End1` of the current spring is checked to see if it's connected to a fixed point in space. If so, the temporary variable `pt1` stores the global coordinates of the endpoint, and the variable `v1` stores the velocity of the endpoint, which is 0. If the endpoint reference is a valid `Object`, then we compute the position of the endpoint, stored in `pt1`, just like we did in the `Initialize` function, using a coordinate transform as follows:

```
pt1 = Objects[j].vPosition + VRotate2D(Objects[j].fOrientation,  
                                      Springs[i].End1.pt);
```

We compute the velocity of that point as shown in [Chapter 9](#) by first computing the velocity of the point due to rotation in body-fixed coordinates, converting that to global coordinates, and then adding the result to the `Object`'s linear velocity. This is accomplished in the following code:

```
v1 = Objects[j].vVelocity + VRotate2D(Objects[j].fOrientation,  
                                      Objects[j].vAngularVelocity^Springs[i].End1.pt);
```

We then repeat these calculations for the second endpoint of the spring.

Once we've obtained the positions and velocities of the spring endpoints, we compute the spring-damper force in the same manner as in the rope example. The resulting spring forces are aggregated in the `vSprings` property of each object. Note that if the spring endpoint reference is a fixed point in space, we do not aggregate the force on that fixed point.

Since the `Objects` are rigid bodies here, we now have to compute the moment due to the spring force acting on each object. You must do this so that when we integrate the equations of motion, the objects rotate properly.

For the `Object` connected to `End1` of the current spring, the following lines compute the moment:

```
Fo = VRotate2D(-Objects[j].fOrientation, F);  
r = Springs[i].End1.pt;  
M = r^Fo;  
Objects[j].vMSprings += M;
```

`Fo` is a vector representing the spring force computed earlier on the current `Object` in the current `Object`'s local, body-fixed coordinate system. The line:

```
Fo = VRotate2D(-Objects[j].fOrientation, F);
```

transforms `F` from global to local coordinates of the current `Object`, `Object[j]`.

`r` is set to the local, body-fixed coordinates of the spring attachment point for the current `Object`, and we compute the resulting moment by taking the vector cross product of `r` with `Fo`. The result is stored in the vector variable `M`, which gets aggregated in the `Object` property `vMSprings`. We then perform these same sorts of calculations for the `Object` connected to the other end of the spring.

After these calculations, the rest of `UpdateSimulation` is the same as that shown earlier; the function integrates the equations of motion and renders the scene.

Upon running this simulation, you'll see the linked chain swing down and to the left and then back and forth until the motion dampens out. You'll also notice there's some stretch to the springs between the objects that appears to increase as you look from the lower link to the upper link. This is indeed a non-uniform stretch in the springs, which makes sense when you consider that the upper spring has more weight, thus more force, pulling down on it than does the lower spring.

As in this rope example, you can tune the spring and damping constants to minimize the spring stretch if that gap created by the stretched spring bothers you. You must keep in mind numerical stability if your springs are too stiff, and here again, you must implement a robust integrator.

Rotational Restraint

So far we've used springs only to attach objects in a way that keeps the attachment points together but allows the objects to rotate about the attachment point. This is a so-called *pinned joint*. If you want a fixed joint that minimizes the amount of rotation between the connected objects, you can add another spring to restrain the connected objects' rotation.

Figure 13-6 illustrates an example comprising two rigid objects connected at their ends, forming a ninety-degree angle. The uppermost end of the first object is connected to a fixed point in space as in our rope and linked-chain examples. Under gravity, the assembly would rotate and swing around this fixed point. However, unlike the linked-chain example, the extra spring prevents the lower link from pivoting around the other end of the first link, as illustrated in **Figure 13-7**.

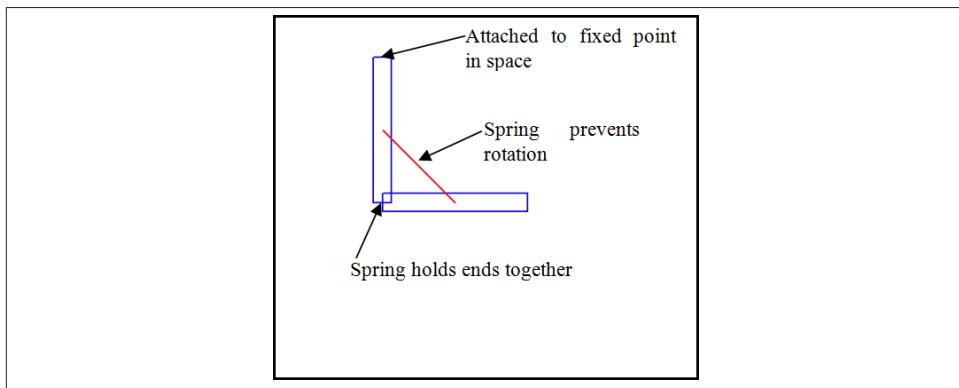


Figure 13-6. Rotation restraint setup

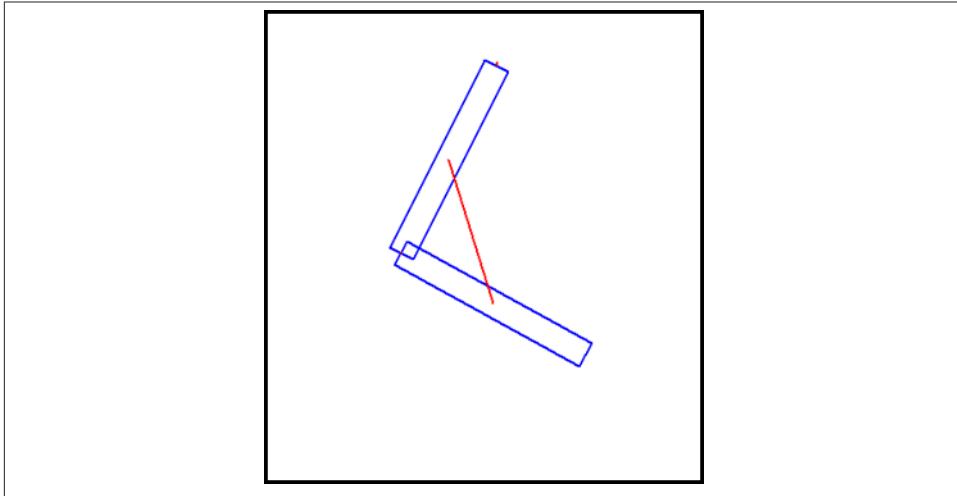


Figure 13-7. Rotation restraint in action

The setup for this example is relatively straightforward and consists of setting the initial positions and orientations of two rigid bodies and connecting three springs.

This example's `Initialize` function is as follows:

```
bool Initialize(void)
{
    Vector r;
    Vector pt;
    int i;

    // Position objects
    Objects[0].vPosition.x = _WINWIDTH/2;
    Objects[0].vPosition.y = _WINHEIGHT/8+Objects[0].fLength/2;
    Objects[0].fOrientation = 90;

    Objects[1].vPosition.x = _WINWIDTH/2+Objects[1].fLength/2;
    Objects[1].vPosition.y = _WINHEIGHT/8+Objects[0].fLength;
    Objects[1].fOrientation = 0;

    // Connect end of the first object to the earth:
    Springs[0].End1.ref = -1;
    Springs[0].End1.pt.x = _WINWIDTH/2;
    Springs[0].End1.pt.y = _WINHEIGHT/8;

    Springs[0].End2.ref = 0;
    Springs[0].End2.pt.x = -Objects[0].fLength/2;
    Springs[0].End2.pt.y = 0;

    pt = VRotate2D(Objects[0].fOrientation, Springs[0].End2.pt) +
```

```

        Objects[0].vPosition;
r = pt - Springs[0].End1.pt;

Springs[0].InitialLength = r.Magnitude();
Springs[0].k = _SPRING_K;
Springs[0].d = _SPRING_D;

// Connect other end of first object to end of second object
i = 1;
Springs[i].End1.ref = i-1;
Springs[i].End1.pt.x = Objects[i-1].fLength/2;
Springs[i].End1.pt.y = 0;

Springs[i].End2.ref = i;
Springs[i].End2.pt.x = -Objects[i].fLength/2;
Springs[i].End2.pt.y = 0;

pt = VRotate2D(Objects[i].fOrientation, Springs[i].End2.pt) +
    Objects[i].vPosition;
r = pt - (VRotate2D(Objects[i-1].fOrientation, Springs[i].End1.pt) +
    Objects[i-1].vPosition);

Springs[i].InitialLength = r.Magnitude();
Springs[i].k = _SPRING_K;
Springs[i].d = _SPRING_D;

// Connect CG of objects to each other
Springs[2].End1.ref = 0;
Springs[2].End1.pt.x = 0;
Springs[2].End1.pt.y = 0;

Springs[2].End2.ref = 1;
Springs[2].End2.pt.x = 0;
Springs[2].End2.pt.y = 0;

r = Objects[1].vPosition - Objects[0].vPosition;

Springs[2].InitialLength = r.Magnitude();
Springs[2].k = _SPRING_K;
Springs[2].d = _SPRING_D;
}

```

The two Objects are positioned with the lines:

```

Objects[0].vPosition.x = _WINWIDTH/2;
Objects[0].vPosition.y = _WINHEIGHT/8+Objects[0].fLength/2;
Objects[0].fOrientation = 90;

Objects[1].vPosition.x = _WINWIDTH/2+Objects[1].fLength/2;
Objects[1].vPosition.y = _WINHEIGHT/8+Objects[0].fLength;
Objects[1].fOrientation = 0;

```

Basically, the first Object, `Object[0]`, is located somewhere toward the top middle of the screen with an initial rotation of ninety degrees so that it stands vertically. The second Object, `Object[1]`, is positioned so that it lies horizontally with its left end coincident with the lower end of the first object. We'll put a spring there momentarily, but first, we'll connect a spring to the upper end of the first object to connect it to a fixed point. The following code takes care of that spring using the same techniques discussed earlier:

```
// Connect end of the first object to the earth:  
Springs[0].End1.ref = -1;  
Springs[0].End1.pt.x = _WINWIDTH/2;  
Springs[0].End1.pt.y = _WINHEIGHT/8;  
  
Springs[0].End2.ref = 0;  
Springs[0].End2.pt.x = -Objects[0].fLength/2;  
Springs[0].End2.pt.y = 0;  
  
pt = VRotate2D(Objects[0].fOrientation, Springs[0].End2.pt) +  
    Objects[0].vPosition;  
r = pt - Springs[0].End1.pt;  
  
Springs[0].InitialLength = r.Magnitude();  
Springs[0].k = _SPRING_K;  
Springs[0].d = _SPRING_D;
```

Now, we connect a spring at the corner formed by the two objects using the following code:

```
// Connect other end of first object to end of second object  
i = 1;  
Springs[i].End1.ref = i-1;  
Springs[i].End1.pt.x = Objects[i-1].fLength/2;  
Springs[i].End1.pt.y = 0;  
  
Springs[i].End2.ref = i;  
Springs[i].End2.pt.x = -Objects[i].fLength/2;  
Springs[i].End2.pt.y = 0;  
  
pt = VRotate2D(Objects[i].fOrientation, Springs[i].End2.pt) +  
    Objects[i].vPosition;  
r = pt - (VRotate2D(Objects[i-1].fOrientation, Springs[i].End1.pt) +  
          Objects[i-1].vPosition);  
  
Springs[i].InitialLength = r.Magnitude();  
Springs[i].k = _SPRING_K;  
Springs[i].d = _SPRING_D;
```

If we stop here, the simulation will behave just like the linked-chain example, albeit we'll have a very short chain. So, to prevent rotation at the corner, we'll add another spring connecting the centers of gravity of the objects. You can use other points if you desire; we chose the centers of gravity for convenience. The following code adds this rotational restraint spring:

```
// Connect CG of objects to each other
Springs[2].End1.ref = 0;
Springs[2].End1.pt.x = 0;
Springs[2].End1.pt.y = 0;

Springs[2].End2.ref = 1;
Springs[2].End2.pt.x = 0;
Springs[2].End2.pt.y = 0;

r = Objects[1].vPosition - Objects[0].vPosition;

Springs[2].InitialLength = r.Magnitude();
Springs[2].k = _SPRING_K;
Springs[2].d = _SPRING_D;
```

The rest of this simulation is the same as in the linked-chain example. There are no other code modifications required. It's all in the setup.

Now, if you want to allow some amount of rotation or flexibility in the joint, you can do so by tuning the spring constant for the rotation restraint spring. Using linear springs creatively, you can model all sorts of joints very simply.

CHAPTER 14

Physics Engines

A *physics engine* is the part of your game that contains all the code required for whatever you’re trying to simulate using physics-based techniques. For many game programmers, a physics engine is a real-time, rigid-body simulator such as the sort we’ve discussed earlier in this book. The open source and licensable physics engines available to you are typically of the rigid-body-simulator variety. Some physics engines are rather generic and are useful for general rigid bodies and particles; others include various connectors and constraints, enabling ragdoll simulation. Still others focus on soft bodies and fluids. Fewer actually focus on the physics of some specific thing, like a car or a boat. A simple Internet search on the phrase “game physics engine” will generate many links to potential options for your use. That said, you could always write your own physics engine.

Building Your Own Physics Engine

We’re advocates of using physics where you need it. Sure, you can write a general-purpose physics engine for a game, but if you’re creating a game that doesn’t require a general-purpose physics engine, then don’t write one. That may sound obvious, but sometimes we are compelled to do more than what we need just so we can say we did it. Aside from the effort involved, a general-purpose physics engine will probably be less efficient than a purpose-built physics engine. By *purpose-built*, we mean designing the physics engine specifically to suit what you’re trying to simulate. For example, a general-purpose physics engine would surely include particles, rigid bodies, connectors, other force effectors, and who knows what else—fluids, perhaps—and be fully 3D. But if you’re writing a 2D side-scrolling game for a smartphone, you certainly won’t need 3D with the associated complexities involved in dealing with rotation and collisions in 3D; and if your game simply involves throwing a ball of fuzz at some arbitrary junk, then you may not even need to deal with rigid bodies at all. We’re being somewhat facetious here, but the point is, unless you *must* write a general-purpose physics engine—say, if you plan to license it as a middleware product or use it in a variety of game

types—then don't write one. Instead, write one specifically optimized for the game you're working on.

Let's consider a few examples. Let's say you're writing a 3D first-person shooter and you want to use physics to simulate how wooden barrels and crates blow apart when shot. Typically, such an effect would show pieces of wood flying off in different directions while falling under the influence of gravity. You could simulate such an effect in 3D using rigid bodies and you wouldn't even need to consider collisions, unless you wanted the pieces to bounce off each other or other objects. Ignoring these aspects greatly simplifies the underlying physics engine. Consider another example. Let's say you're working on a game involving flying an airplane. You can use physics to simulate the flight dynamics, as we explain in this book, without the need for particles, connectors, or even collision response.

The point of all this discussion is that you should consider which aspects of your game will really benefit from physics and write your physics engine to deal specifically with those aspects.

Another thing to consider is whether or not you need real-time physics. You might expect, after reading the available game physics literature, that your game must include real-time simulations if it is to incorporate physics. However, there are many ways to include physics in a game without having to solve the physics via real-time simulations. We show you an example in [Chapter 19](#) whereby a golf swing is simulated in order to determine club head velocity at the time of club-ball impact. In this case, given specific initial parameters, we can solve the swing quickly, almost instantaneously, to determine the club speed, which can then be used as an initial condition for the ball flight. The ball's flight can be solved quickly as well and not necessarily in real time. It really depends on how you want to present the result to the player. If your game involves following the flight of the ball as it soars through the air, then you might want to simulate its flight in real time so you can realistically move the ball and camera. If, however, you simply want to show where the ball ends up, then you need not perform the simulation in real time. For such a simple problem, you can solve for the final ball location quicker than real time. Sometimes, the action you're simulating may happen so fast in real life that you'll want to slow it down for your game. Following a golf ball's flight in real time might have the camera moving so fast that your player won't be able to enjoy the beautifully rendered bird's-eye view of the course. In this case, you rapidly solve the flight path, save the data, and then animate the scene at a more enjoyable pace of your choosing.

We don't want to come across as trying to talk you out of writing a physics engine if you so choose. The point of our discussion so far is that you simply don't have to write a generic, real-time physics engine in order to use physics in your games. You have other options as we've just explained.

Assuming that, after all these considerations, you need to write a physics engine, then we have the following to offer.

A physics engine is just one component of a game engine. The other components include the graphics engine, audio engine, AI engine, and whatever other engines you may require or whatever other components of a game you may elevate to the status of engine. Whatever the case, the physics engine handles the physics. Depending on whom you talk to, you'll get different ideas on what composes a physics engine. Some will say that the heart of the physics engine is the collision detection module. Well, what if your game doesn't require collision detection, yet it still uses physics to simulate certain behaviors or features? Then collision detection certainly cannot be the heart of your physics engine. Some programmers will certainly take issue with these statements. To them, a physics engine simulates rigid-body motion using Newtonian dynamics while taking care of collision detection and response. To us, a significant component of a physics engine is the model—that is, the idealization of the thing you're trying to simulate in a realistic manner. You cannot realistically simulate the flight characteristics of a specific aircraft by treating it as a generic rigid body. You have to develop a representative model of that aircraft including very specific features; otherwise, it's a hack (which, by the way, we recognize as a valid and long-established approach).

Earlier, in [Chapter 7](#) and [Chapter 13](#), we showed you several example simulations. While simple, these examples include many of the required components of a generic physics engine. There are the particle and rigid-body classes that encapsulate generic object properties and behaviors, physics models that govern object behaviors, collision detection and response systems, and a numerical integrator. Additionally, those examples include interfacing the physics code with user input and visual feedback. These examples also show the basic flow from user input to physics solver to visual feedback.

In summary, the major components of a generic physics engine include:

- Physics models
- Simulated objects manager
- Collision detection engine or interface thereto
- Collision response module
- Force effectors
- Numerical integrator
- Game engine interface

Physics Models

Physics models are the idealizations of the things you're simulating. If your physics engine is a generic rigid-body simulator used to simulate an assortment of solid objects your players can knock around, throw, shoot, and generally interact with in a basic manner, then the physics model will probably be very generic. It's probably safe to as-

sume that each object will be subject to gravity's pull, thus mass will be an important attribute. Size will also be important, not only because you'll need to know how big things are when checking for collisions and handling other interactions, but also because size is related to the distribution of the object's mass. More precisely, each object will have *mass moment of inertia* attributes. The objects will most likely also have some ascribed coefficient of restitution that will be used during collision response handling. Additionally, you might ascribe some friction coefficients that may be used during collision response or in situations where the objects may slide along a floor. As the objects will likely find themselves airborne at some point, you'll probably also include a drag coefficient for each object. All of these parameters will help you differentiate massive objects from lighter ones or compact objects from voluminous ones.

If your simulation involves more than generic rigid bodies, then your physics will be more specific and perhaps far more elaborate. A great example of a more complicated model is flight simulation. No matter how good your generic rigid-body model, it won't fly like any specific aircraft if it flies at all. You must develop a model that captures flight aerodynamics specific to the aircraft you're simulating. [Chapter 15](#) shows how to put together a model for an aircraft that can be used in a real-time flight simulation.

The other chapters in Part IV of this book are meant to give you a taste of modeling aspects for a variety of things you might simulate in a game. Just as you cannot simulate an aircraft with a generic rigid-body model, you cannot simulate a ship with an aircraft model, nor can you simulate a golf ball with a ship model. The point is that you must spend some time designing your physics model specific to what you're going to simulate in your game. Time spent here is just as important to creating a realistic physics engine as time spent on designing a robust integration scheme or collision detection system. We can't overstate the importance of the physical model. The model is what defines the behavior of the thing you're simulating.

Simulated Objects Manager

Your simulated objects manager will be responsible for instantiating, initializing, and disposing of objects. It will also be responsible for maintaining links between object physics and other attributes such as geometry, for example, if in a 3D simulation you use the same polyhedron to render an object and for collision detection and response.

You must have some means of managing the objects in your simulation. One can imagine many different approaches to managing these objects, and unless your simulation uses just a handful of objects or fewer, essentially what you need is a list of objects of whatever class you've defined. You've seen in previous chapters' examples where we use simple arrays of `RigidBody` type objects or `Particle` type objects. If all the objects in your simulation are the same, then you need only a single class capturing all their behavior. However, for more diversity, you should use a list of various classes with each class encapsulating the code required to implement its own physical model. This is

particularly important with respect to the forces acting on the model. For example, you could have some objects representing projectiles with others representing aircraft. These different classes will share some common code (for example, collision detection); however, the way forces are computed on each will vary due to the differences in how they are modeled. With such an approach, each class must have code that implements its particular model. During integration, the entire list will be traversed, calling the force aggregation method for each object, and the particular class will handle the details suitable for the type of object.

In some simple cases, you need not use different object classes if the types of objects you plan to use are not too different. For example, it would be fairly straightforward to implement a single class capable of handling both particles and rigid bodies. The object class could include an object type property used to denote whether the object is a particle or a rigid body, and then the class methods would call the appropriate code. Again, this will work satisfactorily for simple objects with few differences. If you want to simulate more than two types of objects or if they are very different, you're probably better off using different classes specific to each object being simulated.

However you structure your classes or lists, the flow of processing your objects will generally be the same. Every physics *tick*—that is, every time step in the physics simulation—you must check for object collisions, resolve those collisions, aggregate the usual forces on each object, integrate the equations of motion for each object, and then update each object's state.

As we said, this is the general flow at every physics tick, or time step, which may not be the same as your rendering steps. For example, for accuracy in your simulation you may have to take small steps around a millisecond or so. You wouldn't want to update the graphics every millisecond when you need only about a third as many graphics updates per second. Thus, your objects manager will have to be integrated with your overall game engine, and your game engine must be responsible for making sure the physics and graphics are updated appropriately.

Collision Detection

If collisions are an important part of your game, then a robust collision detection system is required.

Your collision detection system is distinct from the collision response system or module, though the two go hand in hand. Collision detection is the computational geometry problem of determining if objects collide and, if so, what points are making contact. These points are sometimes called the *contact manifold*. They're just the points that are touching, which could be a line or surface, though for simplicity usually the point, end points, or points defining the contact surface boundary are all that are included in the contact manifold.

The collision detection system's role is very specific: determine which objects are colliding, what points on each object are involved in the collision, and the velocities of those points. It sounds straightforward, but actual implementation can get quite complex. There are situations where fast-moving objects may go right through other objects, especially thin ones, over a single time step, making the collision detection system miss the collision if it relies solely on checking the separation distance between objects and their relative velocity (i.e., it detects a collision if the objects are within some collision tolerance and are also moving toward each other). A robust collision detection system will capture this situation and respond accordingly. In [Chapter 8](#) we simply check if particles moved past the ground over the course of a single time step, for example, and then reset their position to that of the ground plane level. We can handle many situations using such simple techniques, especially when dealing with objects passing through floors or walls; however, other situations may require more complex algorithms to predict if a collision will occur sometime in the near future depending on how fast objects are moving relative to each other. This latter case is called *continuous* collision detection, and it is covered in many Internet, book, and technical paper sources. Many commercial and open source physics engines advertise their capability to handle continuous collision detection.

Another challenge associated with collision detection is the fact that it can be very time-consuming if you have a large or even moderate number of objects in your simulation. There are various techniques to deal with this. First, the game space is partitioned in some coarse grid-like manner, and this grid is used to organize objects depending on which cell they occupy. Then, in the second phase of collision detection, only those objects occupying adjacent cells are checked against each other to see if they are colliding. Without this grid partitioning, pairwise checks of every object against every other object would be very computationally expensive. The second phase of collision detection is often a broad approach using bounding spheres or bounding boxes, which may be axis- or body-aligned. If the bounding spheres or boxes of each object are found to collide, then the objects likewise may be colliding and further checks will be required; otherwise, we can infer that the objects are not colliding. In the case of a potential collision, these further checks become more complex depending on the geometry of the objects. This phase generally involves polygon- and vertex-level checks; there are well-established techniques for performing such checks that we won't get into here. Again, there's a wealth of literature on collision detection available online.

Collision Response

Once the collision detection system does its job, it's time for the collision response system to deal with the colliding objects. Earlier in this book, we showed you how to implement an impulse-moment collision response method. Recall that this method assumes that at the instant of collision, the most significant forces acting on the objects are the collision forces, so all other forces can be ignored for that instant. The method then com-

putes the resulting velocities of the objects after colliding and instantly changes their velocities accordingly. To perform the required calculations, the collision response system requires the objects colliding, of course, the collision points, and the velocities of those points. Also, each object must have some associated mass and coefficient of restitution, which are likewise used to compute the resulting velocities of the objects after the collision.

In practice, the collision response system works hand in hand with the collision detection system, particularly when dealing with objects that may be penetrating each other. As we mentioned earlier, in many cases an object penetrating another, such as an object penetrating a wall or floor, can simply be moved so that it is just touching the wall or floor. Other cases may be more complicated, and iterative algorithms are used to resolve the penetration. For example, if penetration is detected, then the simulation may back up to the previous time step and take a short time step to see if penetration still occurs. If it does not, the simulation proceeds; otherwise, the simulation takes an even smaller time step. This process repeats until penetration does not occur. This works fine in many cases; however, sometimes the penetration is never resolved and the simulation could get stuck taking smaller and smaller time steps. This failure to resolve could be attributed to objects that are just outside the collision distance tolerance at one instant, and due to numerical errors, exceedingly small time steps are required to stay outside of the distance tolerance. Some programmers simply put an iteration limit in the code to prevent the simulation from getting stuck, but the consequences in every situation may be unpredictable. The continuous collision detection approach we mentioned earlier avoids this sort of problem by predicting the future collision or penetration and dealing with it ahead of time. Whatever the approach, there will be some back and forth and data exchange between your collision detection and response systems to avoid excessive penetration situations.

Additionally, there are situations when an object may come to rest in contact with another—for example, a box resting on the floor. There are many ways to deal with such contact situations, one of which is to just allow the impulse-momentum approach to deal with it. This works just fine in many cases; however, sometimes the objects in resting contact will jitter with the impulse-momentum approach. One resolution to this jittering problem is to put those objects to sleep—that is, if they are found to be colliding, but their relative velocities are smaller than some tolerance, they are put to sleep. A related but somewhat more complicated approach is to compute the contact normal between the object and the floor and set that velocity to 0. This serves as a constraint, preventing the object from penetrating the floor while still allowing it to slide along the floor.

Force Effectors

Force effectors apply direct or indirect force on the objects in your simulations. Your physics engine may include several. For example, if your engine allows users to move

objects around with the mouse, then you'll need some virtualization of the force applied by the user via the mouse or a finger on a touch screen. This is an example of a direct force. Another direct force effector could be a virtual jet engine. If you associate that virtual engine, which produces some thrust force, with some object, then the associated object will behave as though it were pushed around by the jet.

Some examples of indirect force effectors include gravity and wind. Gravity applies force on objects by virtue of their mass, but it is typically modeled as body acceleration and not an explicit force. Wind can be viewed as exerting a pressure force on an object, and that force will be a function of the object's size and drag coefficients.

You can imagine all sorts of force effectors, from ones similar to those just described to perhaps some otherworldly ones. Whatever you imagine, you must remember that a force has magnitude, direction, and some central point of application. If you put a jet engine on the side of a box, the box will not only translate but will spin as well. Wind creates a force that has a center of pressure, which is the point through which you can assume the total wind force acts. The direction of the force and point of application are important for capturing both translation and rotation. As an example, consider the hovercraft we modeled in [Chapter 9](#) that included two bow thrusters for steering and a propeller for forward motion. Each of these direct force effectors—the bow thrusters and the propeller—is applied at specific locations on the hovercraft. The bow thrusters are located toward the bow and point sideways in order to create spin, thus allowing some steering. The propeller is located on the center line of the hovercraft, which passes through the hovercraft's center of gravity so that it does not create spin and instead simply pushes the craft forward. There's another force effector in that model— aerodynamic drag, which is an indirect force effector. The drag force is applied at a point aft of the center of gravity so that it creates some torque, or moment, which in this model helps keep the hovercraft pointed straight; it provides some directional stability.

Whatever force effectors you contrive, they all must be aggregated for each object and dealt with in your numerical integrator. Thus, your integrator must have some means of accessing all the force effector information required to accurately simulate their effect on each associated object.

Numerical Integrator

The integrator is responsible for solving the equations of motion for each object. We showed you how to do this earlier, in [Chapter 7](#) through [Chapter 13](#). In your generic physics engine you'll iterate through all the objects in the simulation to compute their new velocities, positions, and orientations at every time step. To make these calculations, your integrator must have access to the force effectors associated with each object. The forces will be used to compute accelerations, which will then be integrated to compute velocities, and those in turn will be used to compute positions and orientations.

You can handle aggregating the forces in a few ways. You could aggregate all the forces on all objects prior to looping through the objects integrating the equations of motion, but this would require looping through all the objects twice. Alternatively, since you're looping through the object list in order to integrate each object's equations of motion, you can simply aggregate each object's forces during the integration step. A complication arises when object pairs apply forces to each other. A linear spring and damper, for example, connected between two objects, apply equal and opposite forces to each object. The force is a function of the relative distance between the objects (the spring component) and their relative velocity (the damping component). So, if during a given time step you aggregate the forces on one of the objects in the pair, the resulting force will be a function of the current relative position and velocities of the objects. Integrating that object will give it a new position and velocity. Then, when you get to the other object in the pair, if you recomputed the spring force, it will be a function of the new relative position and velocity between the objects that includes the new displacement and velocity of the previously updated object but the old displacement and velocity of the current object. This is inconsistent with Newton's law of equal and opposite forces. You can resolve this problem by storing the force computed for the first object and applying it to the second one without recomputing the spring force for the second object.

PART III

Physical Modeling

Part III focuses on physical modeling. The aim of this part is to provide you with valuable physical insight so you can make better judgments on what to include in your models and what you can safely leave out without sacrificing physical realism. We cannot and do not attempt to cover all the possible things you might want to simulate. Instead, we cover several *typical* things you may try to simulate in a game—such as aircraft, boats, and sports balls, among others—in order to give you some insight into their physical nature and into some of the choices you must make when developing suitable models.

CHAPTER 15

Aircraft

If you are going to write a flight simulation game, one of the most important aspects of your game engine will be your flight model. Yes, your 3D graphics, user interface, story, avionics system simulation, and coding are all important, but what really defines the behavior of the aircraft that you are simulating is your flight model. Basically, this is your simplified version of the physics of aircraft flight—that is, your assumptions, approximations, and all the formulas you'll use to calculate mass, inertia, and lift and drag forces and moments.

There are four major forces that act on an airplane in flight: gravity, lift, thrust, and drag. Gravity, of course, is the force that tends to pull the aircraft to the ground, while lift is the force generated by the wings (or lifting surfaces) of the aircraft to counteract gravity and enable the plane to stay aloft. The thrust force generated by the aircraft's propulsor (jet engine or propeller) increases the aircraft's velocity and enables the lifting surfaces to generate lift. Finally, drag counteracts the thrust force, tending to impede the aircraft's motion. [Figure 15-1](#) illustrates these forces.

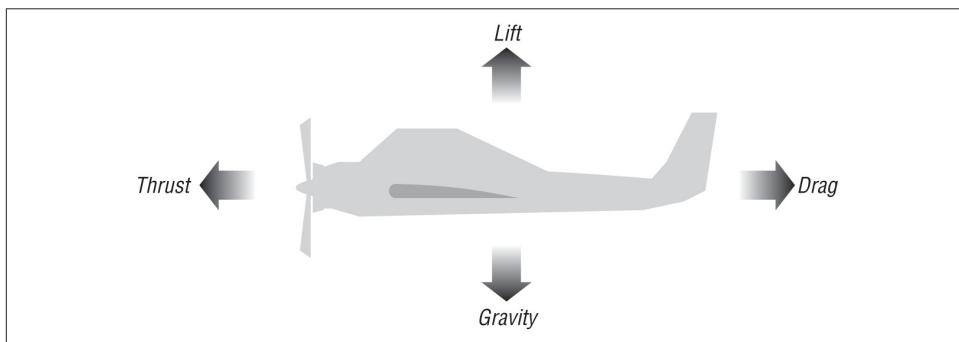


Figure 15-1. Forces on aircraft in flight

We've already discussed the force due to gravity in earlier chapters, so we won't address it again in this chapter except to say that the total of all lift forces must be greater than or equal to the gravitational force if an aircraft is to maintain flight.

To address the other three forces acting on an aircraft, we'll refer to a simplified, generic model of an airplane and use it as an illustrative example. There are far too many aircraft types and configurations to treat them all in this short chapter. Moreover, the subject of aerodynamics is too broad and complex. Therefore, the model that we'll look at will be of a typical subsonic configuration, as shown in [Figure 15-2](#).

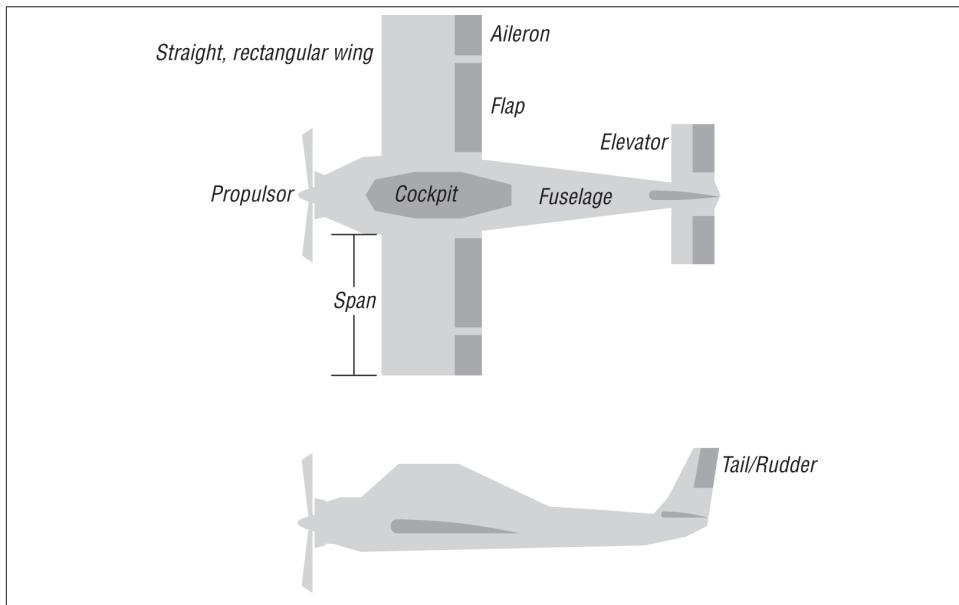


Figure 15-2. Model configuration

In this configuration the main lifting surfaces (the large wings) are located forward on the aircraft, with relatively smaller lifting surfaces located toward the tail. This is the basic arrangement of most aircraft in existence today.

We'll have to make some assumptions in order to make even this simplified model manageable. Further, we'll rely on empirical data and formulas for the calculation of lift and drag forces.

Geometry

Before getting into lift, drag, and thrust, we need to go over some basic geometry and terms to make sure we are speaking the same language. Familiarity with these terms

will also help you quickly find what you are looking for when searching through the references that we'll provide later.

First, take another look at the arrangement of our model aircraft in [Figure 15-2](#). The main body of the aircraft, the part usually occupied by cargo and people, is called the *fuselage*. The *wings* are the large rectangular lifting surfaces protruding from the fuselage near the forward end. The longer dimension of the wing is called its *span*, while its shorter dimension is called its *chord length*, or simply *chord*. The ratio of span squared to wing area is called the *aspect ratio*, and for rectangular wings this reduces to the ratio of span-to-chord.

In our model, the *ailerons* are located on the outboard ends of the wings. The *flaps* are also located on the wings inboard of the ailerons. The small wing-like surfaces located near the tail are called *elevators*. And the vertical flap located on the aft end of the tail is the *rudder*. We'll talk more about what these *control surfaces* do later.

Taking a close look at a cross section of the wing helps to define a few more terms, as shown in [Figure 15-3](#).

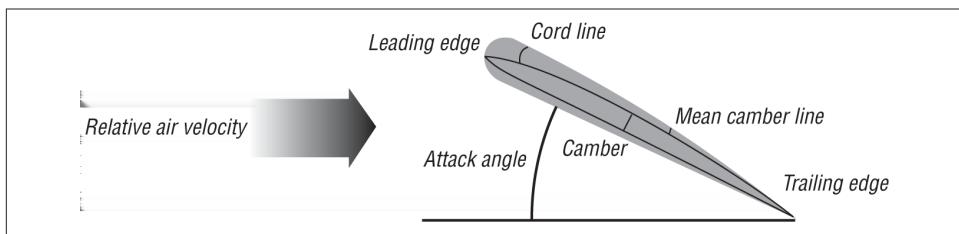


Figure 15-3. Airfoil section

The airfoil shown in [Figure 15-3](#) is a typical *cambered* airfoil. Camber represents the curvature of the airfoil. If you draw a straight line from the trailing edge to the leading edge, you end up with what's called the *chord line*. Now if you divide the airfoil into a number of cross sections, like slices in a loaf of bread, going from the trailing edge to the leading edge, and then draw a curved line passing through the midpoint of each section's thickness, you end up with the *mean camber line*. The maximum difference between the mean camber line and the chord line is a measure of the camber of the airfoil. The angle measured between the direction of travel of the airfoil (the relative velocity vector of the airfoil as it passes through the air) and the chord line is called the absolute *angle of attack*.

When an aircraft is in flight, it may rotate about any axis. It is standard practice to always refer to an aircraft's rotations about three axes relative to the pilot. Thus, these axes—the *pitch* axis, the *roll* axis, and the *yaw* axis—are fixed to the aircraft, so to speak, irrespective of its actual orientation in three-dimensional space.

The pitch axis runs transversely across the aircraft—that is, in the port-starboard direction.¹ Pitch rotation is when the nose of the aircraft is raised or lowered from the pilot's perspective. The roll axis runs longitudinally through the center of the aircraft. Roll motions (rotations) about this axis result in the wing tips being raised or lowered on either side of the pilot. Finally, the yaw axis is a vertical axis about which the nose of the aircraft rotates in the left-to-right (or right-to-left) direction with respect to the pilot. These rotations are illustrated in Figure 15-4.

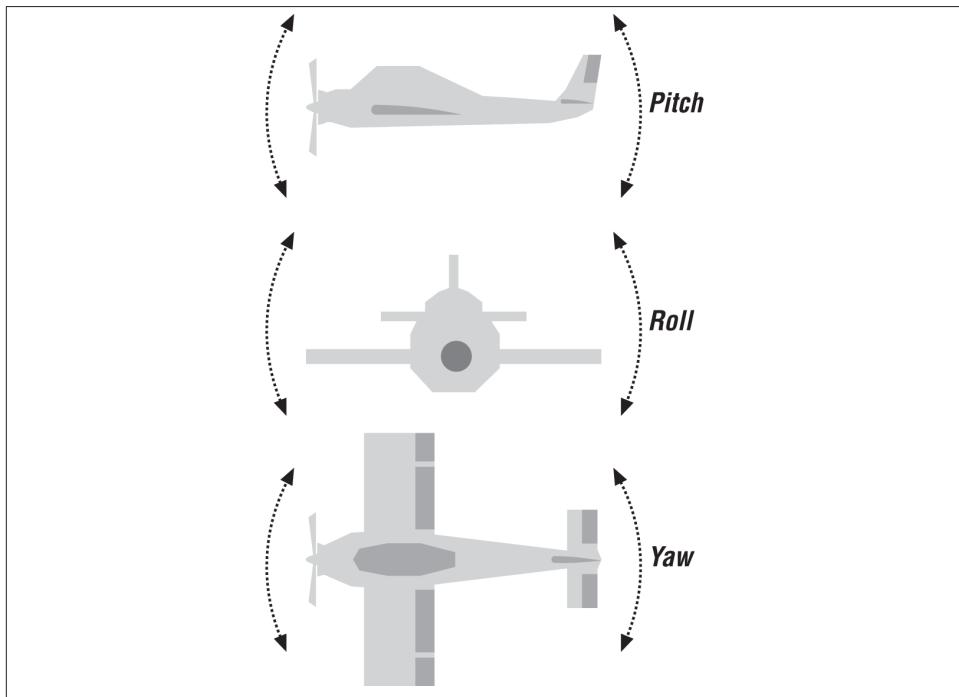


Figure 15-4. Aircraft rotations

1. Port is to the pilot's left and starboard is to the pilot's right when he or she is sitting in the cockpit facing forward.

Lift and Drag

When an airfoil moves through a fluid such as air, lift is produced. The mechanisms by which this occurs are similar to those in the case of the Magnus lift force, discussed earlier in [Chapter 6](#), in that Bernoulli's law is still in effect. However, this time, instead of rotation it's the airfoil's shape and angle of attack that affect the flow of air so as to create lift.

[Figure 15-5](#) shows an airfoil section moving through air at a speed V . V is the relative velocity between the foil and the undisturbed air ahead of the foil. As the air hits and moves around the foil, it splits at the forward stagnation point located near the foil leading edge such that air flows both over and under the foil. The air that flows under the foil gets deflected downward, while the air that flows over the foil speeds up as it goes around the leading edge and over the surface of the foil. The air then flows smoothly off the trailing edge; this is the so-called Kutta condition. Ideally, the boundary layer remains "attached" to the foil without separating as in the case of the sphere discussed in [Chapter 6](#).

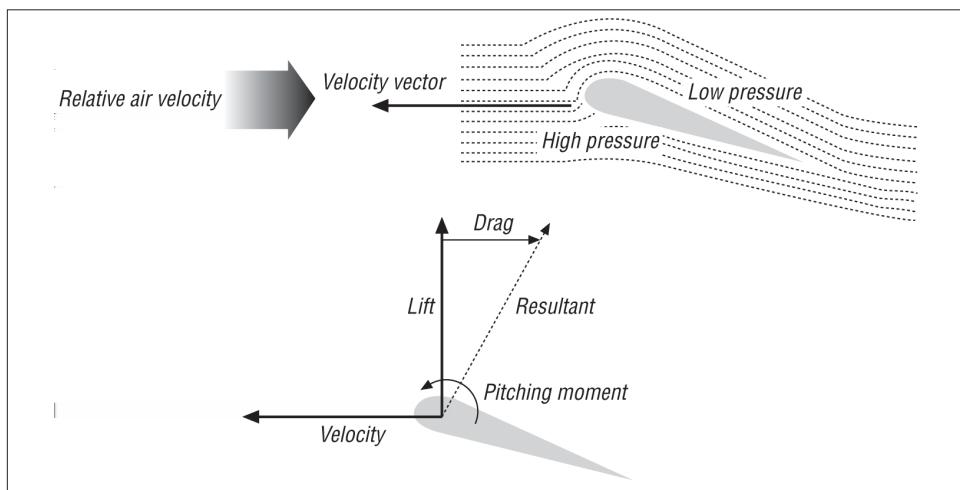


Figure 15-5. Airfoil moving through air

The relatively fast-moving air above the foil results in a region of low pressure above the foil (remember Bernoulli's equation that shows pressure is inversely proportional to velocity in fluid flow). The air hitting and moving along the underside of the foil creates a region of relatively high pressure. The combined effect of this flow pattern is to create regions of relatively low and high pressure above and below the airfoil. It's this pressure differential that gives rise to the lift force. By definition, the lift force is perpendicular to the line of flight—that is, the velocity vector.

Note that the airfoil does not have to be cambered in order to generate lift; a flat plate oriented at an angle of attack relative to the airflow will also generate lift. Likewise, an airfoil does not have to have an angle of attack either. Cambered airfoils can generate lift at 0, or even negative, angles of attack. Thus, in general, the total lift force on an airfoil is composed of two components: the lift due to camber and the lift due to attack angle.

Theoretically, the thickness of an airfoil does not contribute to lift. You can, after all, have a thin curved wing as in the case of wings made from fabric (such as those used for hang gliders). In practice, thickness is utilized for structural reasons. Further, thickness at the leading edge can help delay stall (more on this in a moment).

The pressure differential between the upper and lower surfaces of the airfoil also gives rise to a drag force that acts in line with, but opposing, the velocity vector. The lift and drag forces are perpendicular to each other and lie in the plane defined by the velocity vector and the vector normal (perpendicular) to the airfoil chord line. When combined, these two force components, lift and drag, yield the resultant force acting on the airfoil in flight. This is illustrated in [Figure 15-5](#).

Both lift and drag are functions of air density, speed, viscosity, surface area, aspect ratio, and angle of attack. Traditionally, the lift and drag properties of a given foil design are expressed in terms of nondimensional coefficients C_L and C_D , respectively:

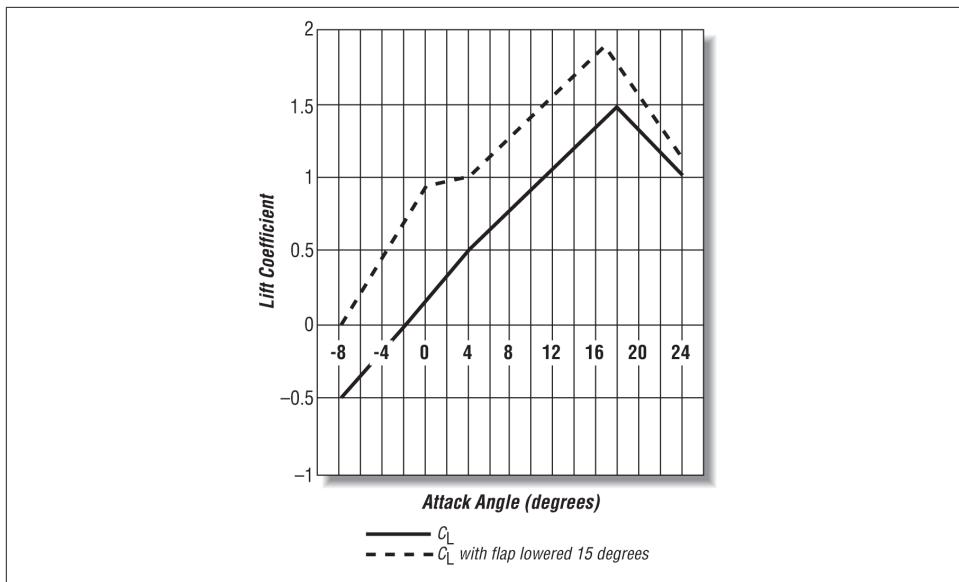


Figure 15-6. Typical C_L versus angle of attack

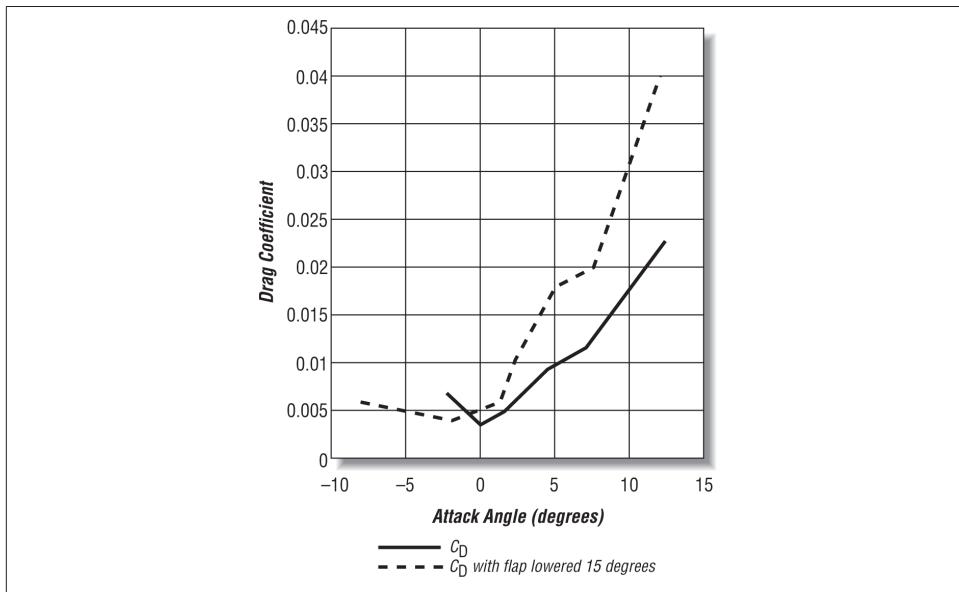


Figure 15-7. Typical C_D versus angle of attack

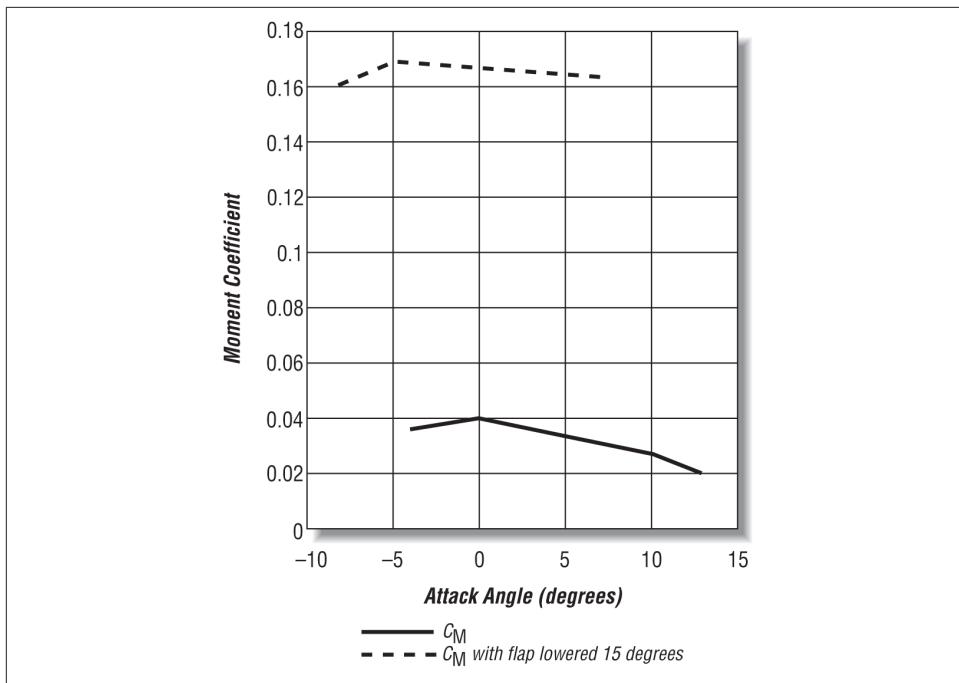


Figure 15-8. Typical C_M versus attack angle

The most widely known family of foil section designs and test data is the NACA foil sections. *Theory of Wing Sections* by Ira H. Abbott and Albert E. Von Doenhoff (Dover) contains a wealth of lift and drag data for practical airfoil designs (see the [Bibliography](#) for a complete reference to this work).²

In practice, the flow of air around a wing is not strictly two-dimensional—that is, flowing uniformly over each parallel cross section of the wing—and there exists a span-wise flow of air along the wing. The flow is said to be three-dimensional. The more three-dimensional the flow, the less efficient the wing.³ This effect is reduced on longer, high-aspect-ratio wings (and wings with end plates where the effective aspect ratio is increased); thus, high-aspect-ratio wings are comparatively more efficient.

To account for the effect of aspect ratio, wing sections of various aspect ratios for a given foil design are usually tested so as to produce a family of lift and drag curves versus attack angle. There are other geometrical factors that affect the flow around wings; for

2. *Theory of Wing Sections* includes standard foil section geometry and performance data, including the well-known NACA family of foil sections. The appendixes to *Theory of Wing Sections* have all the data you need to collect lift and drag coefficient data for various airfoil designs, including those with flaps.
3. Lifting efficiency can be expressed in terms of lift-to-drag ratio. The higher the lift-to-drag ratio, the more efficient the wing or foil section.

a rigorous treatment of these, we refer you to the *Theory of Wing Sections* and *Fluid-Dynamic Lift*.⁴

Turning back to [Figure 15-6](#), you'll notice that the drag coefficient increases sharply with attack angle. This is reasonable, as you would expect the wing to produce the most drag when oriented flat against or perpendicular to the flow of air.

A look at the lift coefficient curve, which initially increases linearly with attack angle, shows that at some attack angle the lift coefficient reaches a maximum value. This angle is called the *critical attack angle*. For angles beyond the critical, the lift coefficient drops off rapidly and the airfoil (or wing) will *stall* and cease to produce lift. This is bad. When an aircraft stalls in the air, it will begin to drop rapidly until the pilot corrects the stall situation by, for example, reducing pitch and increasing thrust. When stall occurs, the air no longer flows smoothly over the trailing edge, and the corresponding high angle of attack results in flow separation (as illustrated in [Figure 15-9](#)). This loss in lift is also accompanied by an increase in drag.

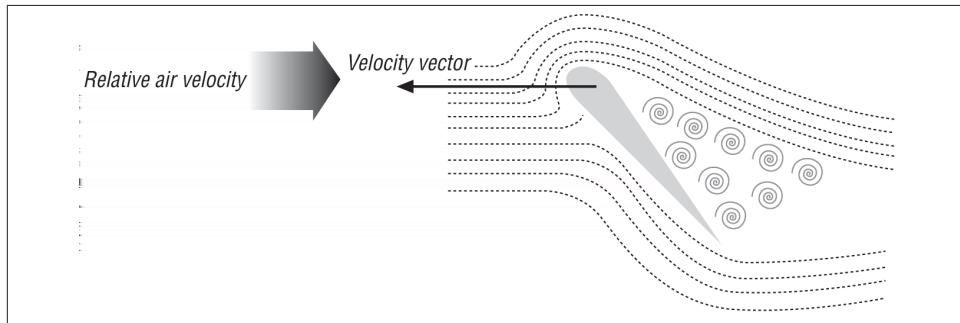


Figure 15-9. Stalled airfoil

Theoretically, the resultant force acting on an airfoil acts through a point located at one-quarter the chord length aft of the leading edge. This is called the *quarter-chord* point. In reality, the resultant force line of action will vary depending on attack angle, pressure distribution, and speed, among other factors. However, in practice it is reasonable to assume that the line of action passes through the quarter-chord point for typical operational conditions. To account for the difference between the actual line of action of the resultant and the quarter-chord point, we must consider the pitching moment about the quarter-chord point. This pitching moment tends to tilt the leading edge of the foil down. In some cases this moment is relatively small compared to the other moments

4. *Fluid-Dynamic Lift*, by Sighard F. Hoerner and Henry V. Borst, and *Fluid-Dynamic Drag*, by Sighard F. Hoerner (both self-published by Hoerner), contain tons of practical charts, tables, and formulas for virtually every aspect of aircraft aerodynamics. They even include material appropriate for high-speed boats and automobiles.

acting on the aircraft, and it may be neglected.⁵ An exception may be when the foil has deflected flaps.

Flaps are control devices used to alter the shape of the foil so as to change its lift characteristics. [Figure 15-6](#) also shows typical lift, drag, and moment coefficients for an airfoil fitted with a plain flap deflected downward at 15°.⁶ Notice the significant increase in lift, drag, and pitch moment when the flap is deflected. *Theory of Wing Sections* also provides data for flapped airfoils for flap angles between -15° and 60°.

Other Forces

The most notable force that we've yet to discuss is thrust—the propulsion force. Thrust provides forward motion; without it, the aircraft's wings can't generate lift and the aircraft won't fly. Thrust, whether generated by a propeller or a jet engine, is usually expressed in pounds, and a common ratio used to compare the relative merits of aircraft powering is the *thrust-to-weight ratio*. This ratio is the maximum thrust deliverable by the propulsion plant divided by the aircraft's total weight. When the thrust-to-weight ratio is greater than one, the aircraft is capable of overcoming gravity in a vertical climb. This is more like a rocket than a traditional airplane. Most normal planes are not capable of this, but many military planes do have thrust-to-weight ratios of greater than one. However, airplane engines rely on oxygen in the atmosphere to combust their fuel with and to produce the force that propels them forward. As the plane climbs higher, the engines will have less oxygen and produce less thrust. The thrust-to-weight ratio will fall, and eventually the plane will again need lift from the wings to maintain its altitude. Even when the plane is climbing vertically like a rocket, the wings still generate lift, and in this case try to pull the airplane away from a vertical trajectory.

Besides gravity, thrust, wing lift, and wing drag, there are other forces that act on an aircraft in flight. These are drag forces (and lift in some cases) on the various components of the aircraft besides the wings. For example, the fuselage contributes to the overall drag acting on the aircraft. Additionally, anything sticking out of the fuselage will contribute to the overall drag. If it's not a wing, anything sticking out of the fuselage is typically called an *appendage*. Some examples of appendages are the aircraft landing gear, canopy, bombs, missiles, fuel pods, and air intakes.

Typically, drag data for fuselages and appendages is expressed in terms of a drag coefficient similar to that discussed in [Chapter 6](#), where experimentally determined drag forces are nondimensionalized by projected frontal area (S), density (ρ), and velocity

5. Aircraft designers must always consider this pitching moment when designing the aircraft's structure, as this moment tends to want to twist the wings off the fuselage.
6. There's a large variety of flap designs besides the plain trailing-edge flap discussed here. Flaps are typically referred to in the literature as *high lift devices*, and the references we'll provide in this chapter give rough descriptions of the most common designs.

squared (V^2). This means that the experimentally measured drag force is divided by the quantity $(1/2) \rho V^2 S$ to get the dimensionless drag coefficient. Depending on the object under consideration, the drag coefficient data will be presented as a function of some important geometric parameter, such as attack angle in the case of airfoils, or length-to-height ratio in the case of canopies. Here again, Hoerner's *Fluid-Dynamic Drag* is an excellent source of practical data for all sorts of fuselage shapes and appendages.

For example, when an aircraft's landing gear is down, the wheels (as well as associated mechanical gear) contribute to the overall drag force on the aircraft. Hoerner reports drag coefficients based on the frontal area of some small-plane landing-gear designs to be in the range of 0.25 to 0.55. By comparison, drag coefficients for typical external storage pods (such as for fuel), which are usually streamlined, can range from 0.06 to 0.26.

Another component of the total drag force acting on aircraft in flight is due to skin friction. Aircraft wings, fuselages, and appendages are not completely smooth. Weld seams, rivets, and even paint cause surface imperfections that increase frictional drag. As in the case of the sphere data presented in [Chapter 6](#). This frictional drag is dependent on the nature of the flow around the part of the aircraft under consideration—that is, whether the flow is laminar or turbulent. This implies that frictional drag coefficients for specific surfaces will generally be a function of the Reynolds number.

In a rigorous analysis of a specific aircraft's flight, you'd of course want to consider all these additional drag components. If you're interested in seeing the nitty-gritty details of such calculations, we suggest you take a look at [Chapter 14](#) of *Fluid-Dynamic Drag*, where Hoerner gives a detailed example calculation of the total drag force on a fighter aircraft.

Control

The flaps located on the inboard trailing edge of the wing in our model are used to alter the chord and camber of the wing section to increase lift at a given speed. Flaps are used primarily to increase lift during slow speed flight, such as when taking off or landing. When landing, flaps are typically deployed at a high downward angle (downward flap deflections are considered positive) on the order to 30° to 60°. This increases both the lift and drag of the wings. During landing, this increase in drag also assists in slowing the aircraft to a suitable landing speed. During takeoff, this increase in drag works against you in that it necessitates higher thrust to get up to speed; thus, flaps may not be deployed to as great an angle as when you are landing.

Ailerons control or induce roll motion by producing differential lift between the port and starboard wing sections. The basic aileron is nothing more than a pair of trailing-edge flaps fitted to the tips of the wings. These flaps move opposite each other, one deflecting upward and the other downward, to create a lift differential between the port

and starboard wings. This lift differential, separated by the distance between the ailerons, creates a torque that rolls the aircraft. To roll the aircraft to the port side (the pilot's left), the starboard aileron would be deflected in a downward direction while the port aileron would be deflected in an upward direction relative to the pilot. Likewise, the opposite deflections of the ailerons would induce a roll to the starboard side. In a real aircraft, the pilot controls the ailerons by moving the flight stick to either the left or right.

Elevators, the tail "wings," are used to control the pitch of the aircraft. (Elevators can be flaps, as shown in [Figure 15-2](#), or the entire tail wing can rotate as on the Lockheed Martin F-16.) When the elevators are deflected such that their trailing edge goes down with respect to the pilot, a nose-down pitch rotation will be induced; that is, the tail of the aircraft will tend to rise relative to its nose, and the aircraft will dive. In an actual aircraft, the pilot achieves this by pushing the flight stick forward. When elevators are deflected such that their trailing edge goes up, a nose-up pitch rotation will be induced.

Elevators are very important for *trimming* (adjusting the pitch of) the aircraft. Generally, the aircraft's center of gravity is located above the mean quarter-chord line of the aircraft wings such that the center of gravity is in line with the main lift force. However, as we explained earlier, the lift force does not always pass through the quarter-chord point. Further, the aircraft's center of gravity may very well change during flight—for example, as fuel is burned off and when ordnance is released. By controlling the elevators, the pilot is able to adjust the *attitude* of the aircraft such that all of the forces balance and the aircraft flies at the desired orientation (pitch angle).

Finally, the rudder is used to control yaw. The pilot uses foot pedals to control the rudder; pushing the left (port) pedal yaws left and pushing the right pedals yaws right (starboard). The rudder is useful for fine-tuning the alignment of the aircraft for approach on landing or when sighting up a target. Typically, large rudder action tends to also induce roll motion that must be compensated for by proper use of the ailerons.

In some cases the rudder consists of a flap on the trailing edge of the vertical tail, while in other cases there is no rudder flap and the entire vertical tail rotates. In both cases, the vertical tail, which also provides directional stability, will usually have a symmetric airfoil shape; that is, its mean camber line will be coincident with its chord line. When the aircraft is flying straight and level, the tail will not generate lift since it is symmetric and its attack angle will be 0. However, if the plane sideslips (yaws relative to its flight direction), then the tail will be at an angle of attack and will generate lift, tending to push the plane back to its original orientation.

Modeling

Although we've yet to cover a lot of the material required to implement a real-time flight simulator, we'd like to go ahead and outline some of the steps necessary to calculate the lift and drag forces on your model aircraft:

1. Discretize the lifting surfaces into a number of smaller wing sections.
2. Collect geometric and foil performance data.
3. Calculate the relative air velocity over each wing section.
4. Calculate the attack angle for each wing section.
5. Determine the appropriate lift and drag coefficients and calculate lift and drag forces.

The first step is relatively straightforward in that you need to divide the aircraft into smaller sections where each section is approximately uniform in characteristics. Performing this step for the model shown in [Figure 15-2](#), you might divide the wing into four sections—one for each wing section that's fitted with an aileron and one for each section that's fitted with a flap. You could also use two sections to model the elevators, one port and one starboard, and another section to model the tail/rudder. Finally, you could lump the entire fuselage together as one additional section or further subdivide it into smaller sections depending on how detailed you want to get.

If you're going to model your aircraft as a rigid body, you'll have to account for all of the forces and moments acting on the aircraft while it is in flight. Since the aircraft is composed of a number of different components, each contributing to the total lift and drag, you'll have to break up your calculations into a number of smaller chunks and then sum all contributions to get the resultant lift and drag forces. You can then use these resultant forces along with thrust and gravity in the equations of motion for your aircraft. You can, of course, refine your model further by adding more components for such items as the cockpit canopy, landing gear, external fuel pods, bombs, etc. The level of detail to which you go depends on the degree of accuracy you're going for. If you are trying to mimic the flight performance of a specific aircraft, then you need to sharpen your pencil.

Once you've defined each section, you must now prepare the appropriate geometric and performance data. For example, for the wings and other lifting surfaces you'll need to determine each section's initial incidence angle (its fixed pitch or attack angle relative to the aircraft reference system), span, chord length, aspect ratio, planform area, and quarter-chord location relative to the aircraft's center of gravity. You'll also have to prepare a table of lift and drag coefficients versus attack angle appropriate for the section under consideration. Since this data is usually presented in graphical form, you'll have to pull data from the charts to build your lookup table for use in your game. Finally,

you'll need to calculate the unit normal vector perpendicular to the plane of each wing section. (You'll need this later when calculating angle of attack.)

These first two steps need only be performed once at the beginning of your game or simulation since the data will remain constant (unless your plane changes shape or its center of gravity shifts during your simulation).

The third step involves calculating the relative velocity between the air and each component so you can calculate lift and drag forces. At first glance, this might seem trivial since the aircraft will be traveling at an air speed that will be known to you during your simulation. However, you must also remember that the aircraft is a rigid body and in addition to the linear velocity of its center of gravity, you must account for its rotational velocity.

Back in [Chapter 2](#), we gave you a formula to calculate the relative velocity of any point on a rigid body that was undergoing both linear and rotational motion:

As a simple example, consider wing panel 1, which is the starboard aileron wing section. Assume that the wing is set at an initial incidence angle of 3.5° and that the plane is traveling at a speed of 38.58 m/s in level flight at low altitude with a pitch angle of 4.5° . This wing section has a chord length of 1.585 m and the span of this section is 1.829 m. Using the lift and drag data presented in [Figure 15-6](#), calculate the lift and drag on this wing section, assuming the ailerons are not deflected and that the density of air is 1.221 kg/m³.

The first step is to calculate the angle of attack, which is 8° , based on the information provided. Now, looking at [Figure 15-6](#), you can find the airfoil lift and drag coefficients to be 0.92 and 0.013, respectively.

Next, you'll need to calculate the planform area of this section, which is simply its chord times its span. This yields 2.899 m². Now you have enough information to calculate lift and drag as follows:

ticular covered some aspects of this flight simulation; therefore, some of the code to follow will be familiar to you. In this present chapter, though, we're going to focus on a few specific functions that implement the flight model. These functions are contained in the source file *Physics.cpp*.

The first function we want you to look at is `CalcAirplaneMassProperties`:

```
//-----//  
// This model uses a set of eight discrete elements to represent the  
// airplane. The elements are described below:  
//  
//      Element 0: Outboard; port (left) wing section fitted with ailerons  
//      Element 1: Inboard; port wing section fitted with landing flaps  
//      Element 2: Inboard; starboard (right) wing section fitted with  
//                  landing flaps  
//      Element 3: Outboard; starboard wing section fitted with ailerons  
//      Element 4: Port elevator fitted with flap  
//      Element 5: Starboard elevator fitted with flap  
//      Element 6: Vertical tail/rudder (no flap; the whole thing rotates)  
//      Element 7: The fuselage  
//  
// This function first sets up each element and then goes on to calculate  
// the combined weight, center of gravity, and inertia tensor for the plane.  
// Some other properties of each element are also calculated, which you'll  
// need when calculating the lift and drag forces on the plane.  
//-----//  
void      CalcAirplaneMassProperties(void)  
{  
    float      mass;  
    Vector     vMoment;  
    Vector     CG;  
    int        i;  
    float      Ixx, Iyy, Izz, Ixy, Ixz, Iyz;  
    float      in, di;  
  
    // Initialize the elements here  
    // Initially the coordinates of each element are referenced from  
    // a design coordinates system located at the very tail end of the plane,  
    // its baseline and center line. Later, these coordinates will be adjusted  
    // so that each element is referenced to the combined center of gravity of  
    // the airplane.  
    Element[0].fMass = 6.56f;  
    Element[0].vDCoords = Vector(14.5f,12.0f,2.5f);  
    Element[0].vLocalInertia = Vector(13.92f,10.50f,24.00f);  
    Element[0].fIncidence = -3.5f;  
    Element[0].fDihedral = 0.0f;  
    Element[0].fArea = 31.2f;  
    Element[0].iFlap = 0;  
  
    Element[1].fMass = 7.31f;  
    Element[1].vDCoords = Vector(14.5f,5.5f,2.5f);  
    Element[1].vLocalInertia = Vector(21.95f,12.22f,33.67f);
```

```

Element[1].fIncidence = -3.5f;
Element[1].fDihedral = 0.0f;
Element[1].fArea = 36.4f;
Element[1].iFlap = 0;

Element[2].fMass = 7.31f;
Element[2].vDCoords = Vector(14.5f,-5.5f,2.5f);
Element[2].vLocalInertia = Vector(21.95f,12.22f,33.67f);
Element[2].fIncidence = -3.5f;
Element[2].fDihedral = 0.0f;
Element[2].fArea = 36.4f;
Element[2].iFlap = 0;

Element[3].fMass = 6.56f;
Element[3].vDCoords = Vector(14.5f,-12.0f,2.5f);
Element[3].vLocalInertia = Vector(13.92f,10.50f,24.00f);
Element[3].fIncidence = -3.5f;
Element[3].fDihedral = 0.0f;
Element[3].fArea = 31.2f;
Element[3].iFlap = 0;

Element[4].fMass = 2.62f;
Element[4].vDCoords = Vector(3.03f,2.5f,3.0f);
Element[4].vLocalInertia = Vector(0.837f,0.385f,1.206f);
Element[4].fIncidence = 0.0f;
Element[4].fDihedral = 0.0f;
Element[4].fArea = 10.8f;
Element[4].iFlap = 0;

Element[5].fMass = 2.62f;
Element[5].vDCoords = Vector(3.03f,-2.5f,3.0f);
Element[5].vLocalInertia = Vector(0.837f,0.385f,1.206f);
Element[5].fIncidence = 0.0f;
Element[5].fDihedral = 0.0f;
Element[5].fArea = 10.8f;
Element[5].iFlap = 0;

Element[6].fMass = 2.93f;
Element[6].vDCoords = Vector(2.25f,0.0f,5.0f);
Element[6].vLocalInertia = Vector(1.262f,1.942f,0.718f);
Element[6].fIncidence = 0.0f;
Element[6].fDihedral = 90.0f;
Element[6].fArea = 12.0f;
Element[6].iFlap = 0;

Element[7].fMass = 31.8f;
Element[7].vDCoords = Vector(15.25f,0.0f,1.5f);
Element[7].vLocalInertia = Vector(66.30f,861.9f,861.9f);
Element[7].fIncidence = 0.0f;
Element[7].fDihedral = 0.0f;
Element[7].fArea = 84.0f;
Element[7].iFlap = 0;

```

```

// Calculate the vector normal (perpendicular) to each lifting surface.
// This is required when you are calculating the relative air velocity for
// lift and drag calculations.
for (i = 0; i< 8; i++)
{
    in = DegreesToRadians(Element[i].fIncidence);
    di = DegreesToRadians(Element[i].fDihedral);
    Element[i].vNormal = Vector((float)sin(in), (float)(cos(in)*sin(di)),
                                (float)(cos(in)*cos(di)));
    Element[i].vNormal.Normalize();
}

// Calculate total mass
mass = 0;
for (i = 0; i< 8; i++)
    mass += Element[i].fMass;

// Calculate combined center of gravity location
vMoment = Vector(0.0f, 0.0f, 0.0f);
for (i = 0; i< 8; i++)
{
    vMoment += Element[i].fMass*Element[i].vDCoords;
}
CG = vMoment/mass;

// Calculate coordinates of each element with respect to the combined CG
for (i = 0; i< 8; i++)
{
    Element[i].vCGCoords = Element[i].vDCoords - CG;
}

// Now calculate the moments and products of inertia for the
// combined elements.
// (This inertia matrix (tensor) is in body coordinates)
Ixx = 0;      Iyy = 0;      Izz = 0;
Ixxy = 0;     Ixzx = 0;     Ixyz = 0;
for (i = 0; i< 8; i++)
{
    Ixx += Element[i].vLocalInertia.x + Element[i].fMass *
           (Element[i].vCGCoords.y*Element[i].vCGCoords.y +
            Element[i].vCGCoords.z*Element[i].vCGCoords.z);
    Iyy += Element[i].vLocalInertia.y + Element[i].fMass *
           (Element[i].vCGCoords.z*Element[i].vCGCoords.z +
            Element[i].vCGCoords.x*Element[i].vCGCoords.x);
    Izz += Element[i].vLocalInertia.z + Element[i].fMass *
           (Element[i].vCGCoords.x*Element[i].vCGCoords.x +
            Element[i].vCGCoords.y*Element[i].vCGCoords.y);
    Ixy += Element[i].fMass * (Element[i].vCGCoords.x *
                               Element[i].vCGCoords.y);
    Ixz += Element[i].fMass * (Element[i].vCGCoords.x *
                               Element[i].vCGCoords.z);
}

```

```

        Iyz += Element[i].fMass * (Element[i].vCGCoords.y *
                                     Element[i].vCGCoords.z);
    }

    // Finally, set up the airplane's mass and its inertia matrix and take the
    // inverse of the inertia matrix.
    Airplane.fMass = mass;
    Airplane.mInertia.e11 = Ixx;
    Airplane.mInertia.e12 = -Ixy;
    Airplane.mInertia.e13 = -Ixz;
    Airplane.mInertia.e21 = -Ixy;
    Airplane.mInertia.e22 = Iyy;
    Airplane.mInertia.e23 = -Iyz;
    Airplane.mInertia.e31 = -Ixz;
    Airplane.mInertia.e32 = -Iyz;
    Airplane.mInertia.e33 = Izz;

    Airplane.mInertiaInverse = Airplane.mInertia.Inverse();
}

```

Among other things, this function essentially completes step 1 (and part of step 2) of our modeling method: discretize the airplane into a number of smaller pieces, each with its own mass and lift and drag properties. For this model we chose to use eight pieces, or elements, to describe the aircraft. Our comments at the beginning of the function explain what each element represents.

The very first thing this function does is initialize the elements with the properties that we've defined to approximate the aircraft. Each element is given a mass, a set of *design coordinates* to its center of mass, a set of moments of inertia about each element's center of mass, an initial incidence angle, a planform area, and a *dihedral angle*.

The design coordinates are the coordinates of the element with respect to an origin located at the very tip of the aircraft's tail, on its centerline and at its baseline. The x-axis of this system points toward the nose of the aircraft, while the y-axis points toward the port side. The z-axis points up. You have to set up your elements in this design coordinate system first because you don't yet know the location of the whole aircraft's center of mass, which is the combined center of mass of all of the elements. Ultimately, you want each element referenced from the combined center of mass because it's the center of mass that you'll track during the simulation.

The dihedral angle is the angle about the x-axis at which the element is initially set. For our model, all of the elements have a 0 dihedral angle; that is, they are horizontal, except for the tail rudder, which has a 90° dihedral since it is oriented vertically.

After we've set up the elements, the first calculation that this function performs is to find the unit normal vector to each element's surface based on the element's incidence and dihedral angles. You need this direction vector to help calculate the angle of attack between the airflow and the element.

The next calculation is the total mass calculation, which is simply the sum of all element masses. Immediately following that, we determine the combined center of gravity location using the technique we discussed in [Chapter 1](#). The coordinates to the combined center of gravity are referenced to the design coordinate system. You need to subtract this coordinate from the design coordinate of each element in order to determine each element's coordinates relative to the combined center of gravity. After that, you're all set with the exception of the combined moment of inertia tensor, which we already discussed in [Chapter 11](#) and [Chapter 12](#).

Step 2 of our modeling method says you need to collect the airfoil performance data. For the example program, we used a cambered airfoil with plain flaps to model the wings and elevators, and we used a symmetric airfoil without flaps to model the tail rudder. We didn't use flaps for the tail rudder since we just made the whole thing rotate about a vertical axis to provide rudder action.

For the wings, we set up two functions to handle the lift and drag coefficients:

```

//-----//
// Given the attack angle and the status of the flaps, this function
// returns the appropriate lift coefficient for a cambered airfoil with
// a plain trailing-edge flap (+/- 15 degree deflection).
//-----//
float LiftCoefficient(float angle, int flaps)
{
    float clf0[9] = {-0.54f, -0.2f, 0.2f, 0.57f, 0.92f, 1.21f, 1.43f, 1.4f,
                      1.0f};
    float clfd[9] = {0.0f, 0.45f, 0.85f, 1.02f, 1.39f, 1.65f, 1.75f, 1.38f,
                      1.17f};
    float clfu[9] = {-0.74f, -0.4f, 0.0f, 0.27f, 0.63f, 0.92f, 1.03f, 1.1f,
                      0.78f};
    float a[9]      = {-8.0f, -4.0f, 0.0f, 4.0f, 8.0f, 12.0f, 16.0f, 20.0f,
                      24.0f};

    float cl;
    int i;

    cl = 0;
    for (i=0; i<8; i++)
    {
        if( (a[i] <= angle) && (a[i+1] > angle) )
        {
            switch(flaps)
            {
                case 0:// flaps not deflected
                    cl = clf0[i] - (a[i] - angle) * (clf0[i] - clf0[i+1]) /
                        (a[i] - a[i+1]);
                    break;
                case -1: // flaps down
                    cl = clfd[i] - (a[i] - angle) * (clfd[i] - clfd[i+1]) /
                        (a[i] - a[i+1]);
                    break;
            }
        }
    }
}

```

```

        case 1: // flaps up
            cl = clfu[i] - (a[i] - angle) * (clfu[i] - clfu[i+1]) /
                (a[i] - a[i+1]);
            break;
        }
        break;
    }
}

return cl;
}

//-----//
// Given the attack angle and the status of the flaps, this function
// returns the appropriate drag coefficient for a cambered airfoil with
// a plain trailing-edge flap (+/- 15 degree deflection).
//-----//
float DragCoefficient(float angle, int flaps)
{
    float cdf0[9] = {0.01f, 0.0074f, 0.004f, 0.009f, 0.013f, 0.023f, 0.05f,
                     0.12f, 0.21f};
    float cdfd[9] = {0.0065f, 0.0043f, 0.0055f, 0.0153f, 0.0221f, 0.0391f, 0.1f,
                     0.195f, 0.3f};
    float cdfu[9] = {0.005f, 0.0043f, 0.0055f, 0.02601f, 0.03757f, 0.06647f,
                     0.13f, 0.18f, 0.25f};
    float a[9]      = {-8.0f, -4.0f, 0.0f, 4.0f, 8.0f, 12.0f, 16.0f, 20.0f,
                     24.0f};
    float cd;
    int   i;

    cd = 0.5;
    for (i=0; i<8; i++)
    {
        if( (a[i] <= angle) && (a[i+1] > angle) )
        {
            switch(flaps)
            {
                case 0:// flaps not deflected
                    cd = cdf0[i] - (a[i] - angle) * (cdf0[i] - cdf0[i+1]) /
                        (a[i] - a[i+1]);
                    break;
                case -1: // flaps down
                    cd = cdfd[i] - (a[i] - angle) * (cdfd[i] - cdfd[i+1]) /
                        (a[i] - a[i+1]);
                    break;
                case 1: // flaps up
                    cd = cdfu[i] - (a[i] - angle) * (cdfu[i] - cdfu[i+1]) /
                        (a[i] - a[i+1]);
                    break;
            }
            break;
        }
    }
}

```

```

    }

    return cd;
}

}

```

Each of these functions takes the angle of attack as a parameter along with a flag used to indicate the state of the flaps—that is, whether the flaps are in neutral position, deflected downward, or deflected upward. Notice that the lift and drag coefficient data is given for a set of discrete attack angles, thus we use linear interpolation to determine the coefficients for attack angles that fall between the discrete angles.

The functions for determining the tail rudder lift and drag coefficients are similar to those shown here for the wings, with the only differences being the coefficients themselves and the fact that the tail rudder does not include flaps. Here are the functions:

```

//-----//
// Given the attack angle, this function returns the proper lift coefficient
// for a symmetric (no camber) airfoil without flaps.
//-----//
float      RudderLiftCoefficient(float angle)
{
    float clf0[7] = {0.16f, 0.456f, 0.736f, 0.968f, 1.144f, 1.12f, 0.8f};
    float a[7]      = {0.0f, 4.0f, 8.0f, 12.0f, 16.0f, 20.0f, 24.0f};
    float cl;
    int     i;
    float   aa = (float) fabs(angle);

    cl = 0;
    for (i=0; i<8; i++)
    {
        if( (a[i] <= aa) && (a[i+1] > aa) )
        {
            cl = clf0[i] - (a[i] - aa) * (clf0[i] - clf0[i+1]) /
                (a[i] - a[i+1]);
            if (angle < 0) cl = -cl;
            break;
        }
    }
    return cl;
}

//-----//
// Given the attack angle, this function returns the proper drag coefficient
// for a symmetric (no camber) airfoil without flaps.
//-----//
float      RudderDragCoefficient(float angle)
{
    float cdf0[7] = {0.0032f, 0.0072f, 0.0104f, 0.0184f, 0.04f, 0.096f, 0.168f};
    float a[7]      = {0.0f, 4.0f, 8.0f, 12.0f, 16.0f, 20.0f, 24.0f};
    float cd;
    int     i;

```

```

float      aa = (float) fabs(angle);

cd = 0.5;
for (i=0; i<8; i++)
{
    if( (a[i] <= aa) && (a[i+1] > aa) )
    {
        cd = cdf0[i] - (a[i] - aa) * (cdf0[i] - cdf0[i+1]) /
            (a[i] - a[i+1]);
        break;
    }
}
return cd;
}

```

With steps 1 and 2 out of the way, steps 3, 4, and 5 are handled in a single function called `CalcAirplaneLoads`:

```

//-----//
// This function calculates all of the forces and moments acting on the
// plane at any given time.
//-----//
void      CalcAirplaneLoads(void)
{
    Vector      Fb, Mb;

    // reset forces and moments:
    Airplane.vForces.x = 0.0f;
    Airplane.vForces.y = 0.0f;
    Airplane.vForces.z = 0.0f;

    Airplane.vMoments.x = 0.0f;
    Airplane.vMoments.y = 0.0f;
    Airplane.vMoments.z = 0.0f;

    Fb.x = 0.0f;      Mb.x = 0.0f;
    Fb.y = 0.0f;      Mb.y = 0.0f;
    Fb.z = 0.0f;      Mb.z = 0.0f;

    // Define the thrust vector, which acts through the plane's CG
    Thrust.x = 1.0f;
    Thrust.y = 0.0f;
    Thrust.z = 0.0f;
    Thrust *= ThrustForce;

    // Calculate forces and moments in body space:
    Vector      vLocalVelocity;
    float       fLocalSpeed;
    Vector      vDragVector;
    Vector      vLiftVector;
    float       fAttackAngle;
    float       tmp;
    Vector      vResultant;

```

```

int      i;
Vector   vtmp;

Stalling = false;

for(i=0; i<7; i++) // loop through the seven lifting elements
                     // skipping the fuselage
{
    if (i == 6) // The tail/rudder is a special case since it can rotate;
    {           // thus, you have to recalculate the normal vector.
        float in, di;
        in = DegreesToRadians(Element[i].fIncidence); // incidence angle
        di = DegreesToRadians(Element[i].fDihedral); // dihedral angle
        Element[i].vNormal = Vector( (float)sin(in),
                                      (float)(cos(in)*sin(di)),
                                      (float)(cos(in)*cos(di)));
        Element[i].vNormal.Normalize();
    }

    // Calculate local velocity at element
    // The local velocity includes the velocity due to linear
    // motion of the airplane,
    // plus the velocity at each element due to the
    // rotation of the airplane.

    // Here's the rotational part
    vtmp = Airplane.vAngularVelocity^Element[i].vCGCoords;

    vLocalVelocity = Airplane.vVelocityBody + vtmp;

    // Calculate local air speed
    fLocalSpeed = vLocalVelocity.Magnitude();

    // Find the direction in which drag will act.
    // Drag always acts inline with the relative
    // velocity but in the opposing direction
    if(fLocalSpeed > 1.)
        vDragVector = -vLocalVelocity/fLocalSpeed;

    // Find the direction in which lift will act.
    // Lift is always perpendicular to the drag vector
    vLiftVector = (vDragVector^Element[i].vNormal)^vDragVector;
    tmp = vLiftVector.Magnitude();
    vLiftVector.Normalize();

    // Find the angle of attack.
    // The attack angle is the angle between the lift vector and the
    // element normal vector. Note, the sine of the attack angle
    // is equal to the cosine of the angle between the drag vector and
    // the normal vector.
    tmp = vDragVector*Element[i].vNormal;
    if(tmp > 1.) tmp = 1;
}

```

```

if(tmp < -1) tmp = -1;
fAttackAngle = RadiansToDegrees((float) asin(tmp));

// Determine the resultant force (lift and drag) on the element.
tmp = 0.5f * rho * fLocalSpeed*fLocalSpeed * Element[i].fArea;
if (i == 6) // Tail/rudder
{
    vResultant = (vLiftVector*RudderLiftCoefficient(fAttackAngle) +
                  vDragVector*RudderDragCoefficient(fAttackAngle))
                  * tmp;
} else
    vResultant = (vLiftVector*LiftCoefficient(fAttackAngle,
                                              Element[i].iFlap) +
                  vDragVector*DragCoefficient(fAttackAngle,
                                              Element[i].iFlap) ) * tmp;

// Check for stall.
// We can easily determine stall by noting when the coefficient
// of lift is 0. In reality, stall warning devices give warnings well
// before the lift goes to 0 to give the pilot time to correct.
if (i<=0)
{
    if (LiftCoefficient(fAttackAngle, Element[i].iFlap) == 0)
        Stalling = true;
}

// Keep a running total of these resultant forces (total force)
Fb += vResultant;

// Calculate the moment about the CG of this element's force
// and keep a running total of these moments (total moment)
vtmp = Element[i].vCGCoords^vResultant;
Mb += vtmp;
}

// Now add the thrust
Fb += Thrust;

// Convert forces from model space to earth space
Airplane.vForces = QVRotate(Airplane.qOrientation, Fb);

// Apply gravity (g is defined as -32.174 ft/s^2)
Airplane.vForces.z += g * Airplane.fMass;

Airplane.vMoments += Mb;
}

```

The first thing this function does is reset the variables that hold the total force and moment acting on the aircraft. Next, the thrust vector is set up. This is trivial in this example since we're assuming that the thrust vector always points in the plus x-axis direction (toward the nose) and passes through the aircraft center of gravity (so it does not create a moment).

After calculating the thrust vector, the function loops over the model elements to calculate the lift and drag forces on each element. We've skipped the fuselage in this model; however, if you want to account for its drag in your model, this is the place to add the drag calculation.

Going into the loop, the first thing the function does is check to see if the current element is element number six, the tail rudder. If it is, then the rudder's normal vector is recalculated based on the current incidence angle. The incidence angle for the rudder is altered when you press the X or C keys to apply rudder action.

The next calculation is to determine the relative velocity between the air and the element under consideration. As we stated earlier, this relative velocity consists of the linear velocity as the airplane moves through the air plus the velocity of each element due to the airplane's rotation. Once you've obtained this vector, you calculate the relative air speed by taking the magnitude of the relative velocity vector.

The next step is to determine the direction in which drag will act. Since drag opposes motion, it acts inline with, but opposite to, the relative velocity vector; thus, all you need to do is take the negative of the relative velocity vector and normalize the result (divide it by its magnitude) to obtain the drag direction vector. Since this vector was normalized, its length is equal to 1 (unity), so you can multiply it by the drag force that will be calculated later to get the drag force vector.

After obtaining the drag direction vector, this function uses it to determine the lift direction vector. The lift force vector is always perpendicular to the drag force vector, so to calculate its direction you first take the cross product of the drag direction vector with the element normal vector and then cross the result with the drag direction vector again. Here again, the function normalizes the lift direction vector.

Now that the lift and drag direction vectors have been obtained, the function proceeds to calculate the angle of attack for the current element. The attack angle is the angle between the lift vector and the element normal. You can calculate the angle by taking the inverse cosine of the vector dot product of the lift direction vector with the element normal vector. Since the drag vector is perpendicular to the lift vector, you can get the same result by taking the inverse sine of the vector dot product of the drag direction vector with the element normal vector.

Now with all the lift and drag vector stuff out of the way, the function goes on to calculate the resultant force acting on the element. The resultant force vector is simply the vector sum of the lift and drag force vectors. Notice that this is where the lift and drag coefficient functions are called and where the empirical lift and drag formulas previously discussed are applied.

After calculating the resultant force, the function checks to see if the calculated lift coefficient is 0. If it is, then the stall flag is set to warn us that the plane is in a stalled situation.

Finally, the resultant force is accumulated in the total force vector variable, and we calculate the moment by taking the cross product of the element coordinate vector with the resultant force. The resulting moment is accumulated in the total moment vector variable. After exiting the loop, the function adds the thrust vector to the total force.

So far, all of these forces and moments have been referenced in the body-fixed-coordinate system. The only thing left to do now is apply the gravity force, but this force acts in the negative y-axis direction in the earth-fixed-coordinate system. To apply the gravity force, the function must first rotate the body force vector from body space to earth space coordinates. We used a quaternion rotation technique in this example, which we already discussed in [Chapter 11](#) and [Chapter 12](#).

That's pretty much it for the flight model. We encourage you to play with the flight model in this program. Go ahead and tweak the element properties and watch to see what happens. Even though this is a rough model, the flight results look quite realistic.

CHAPTER 16

Ships and Boats

The physics of ships is a vast subject. While the same principles govern canoes and super tankers, the difference between the two scales is not trivial. Our goal in this chapter will be to explain some of the fundamental physical principles to allow you to develop realistic simulations. The typical displacement-type ship lends itself well to illustrating these principles; however, many of these principles also apply to other objects submerged or partially submerged in a fluid, such as submarines and air balloons. Remember, air is considered a fluid when we are considering buoyancy.

While surface ships or ships that operate on the water's surface (at the air water interface) are similar to fully submerged objects like submarines or air balloons in that they all experience buoyancy, there are some very distinct differences in their physical nature that we'll highlight in this chapter. These differences affect their behavior, so it is important to be aware of them if you intend to simulate such objects.

Ships have an entire language of their own, so we'll be spending a lot of time just getting the vocabulary right. This will allow you to do further research on any topics that are of particular interest. There are many ways to classify ships and boats, but in regards to the physics governing them, there are three basic types. *Displacement* vessels, *semi-displacement* vessels, and *planing* vessels are named after the forces that keep the boat afloat while it is at cruising speed. When not moving, all vessels are in *displacement mode*.

The term *displacement* in this context means that the ship is supported solely by buoyancy—that is, without dynamic or aerostatic lift as you would see on a high-speed racing boat or a hovercraft. The word *displacement* itself refers to the volume of water displaced or “pushed” out of the way by the ship as it sits floating in the water. We'll discuss this more in the next section.

A *planing* vessel is one that is not supported by buoyancy, but by hydrodynamic lift. This includes the everyday speedboats that most boaters own. When the boat isn't mov-

ing, it just floats in the water, bobbing up and down. However, when the boat begins traveling at high speed, the force of the water hitting the bottom of the boat causes the boat to rise up. This is known as planing, and it greatly reduces the resistance of the vessel. Semi-displacement vessels are those that straddle the two categories, with some support coming from buoyancy and some coming from planing forces. Before we continue discussing this, let's go over some vocabulary.

The *hull* of the ship is the watertight part of the ship that actually displaces the water. Everything in or on the ship is contained within the hull, which is partially submerged in the water. The length of the ship is the distance measured from the bow to the stern. In practice, there are several lengths used to denote the length of a ship, but here we'll refer to the *overall length* of the hull. The *bow* is the front of the ship, while the *stern* is the aft part. When you are on the ship facing the bow, the *port* side is to your left and the *starboard* side is to your right. The overall height of the hull is called the *depth*, and its width is called *breadth* or *beam*. When a ship is floating in the water, the distance from the water surface to the bottom of the hull is called the *draft*. **Figure 16-1** illustrates these terms.

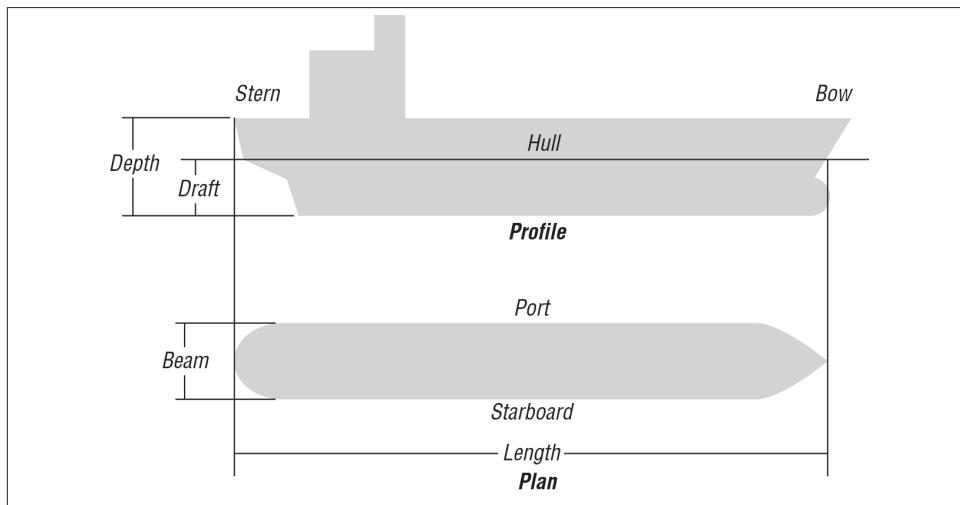


Figure 16-1. Ship geometry

Given that ship design is a diverse subject, we'll limit ourselves to discussing those aspects of ships that make for realistic models. These subjects include stability and sinking, resistance characteristics, propulsion, and maneuverability. Most of these subjects cannot be fully simulated in real time, so we'll show you some general rules that ships follow instead of full numerical simulation.

Stability and Sinking

If you have boats in your video game, the first step to making them realistic physically is allowing them to sink if they become damaged. To understand why boats sink and how they do so, you must first understand *stability*.

Stability

Most boats are least stable about their longitudinal axis—that is, they are easier to heel port and starboard than they are to flip end over end. If the vessel heels over so far that it is upside down, this is called capsizing. This is how most boats sink due to wind, waves, or in some cases of side damage. One of the most famous examples of a sinking ship, the *Titanic*, shows that when a boat is sinking from damage, it can sink end over end, sometimes with the ship breaking in two. We'll discuss both here so that you can animate realistic sinking in your simulation.

In [Chapter 3](#) we introduced the concept of buoyancy and stated that the force on a submerged object due to buoyancy is a function of the submerged volume of the object. Archimedes's principle states that the weight of an object floating in a fluid is equal to the weight of the volume of fluid displaced by the object. This is an important principle. It says that a ship of a given weight must have sufficient volume to displace enough water, an amount equal to the weight of the ship, in order for it to float. Further, this principle provides a clever way of determining the weight of a ship: simply measure or calculate the amount of water displaced by the ship and you can calculate the weight of the ship. In the marine field, displacement is synonymous with the weight of the ship.

As discussed in [Chapter 3](#), we can calculate the buoyant force on any object by using the following formula:

of the underwater portion of the hull. For example, if the ship rolls to the starboard side, then the center of buoyancy shifts out toward the starboard side. When this happens, the lines of action of the weight of the ship and the buoyant force are no longer in line, which results in a moment (torque) that acts on the ship. This torque is equal to the perpendicular distance between the lines of action of the forces times the weight of the ship.

Now here's where we get to the floating upright part that we mentioned earlier. When a ship rolls, for example, you don't want it to keep rolling until it capsizes. Instead, you want it to gently return itself to the upright position after whatever force caused it to roll—the wind, for example—has been removed. In short, you want the ship to be stable. For a ship to be stable, the line of action of the buoyant force must cross the vessel's centerline at a point, called the *metacenter*, above the center of gravity. When this happens, the moment developed when the ship rolls tends to restore the ship to the upright position. If the metacenter is located below the center of gravity, then the moment developed would tend to capsize the ship. The distance between the center of gravity and the metacenter is called *GM*. This is also known as the *stability index*, as a positive value means the floating body is stable and a negative GM means the body is unstable. **Figure 16-2** illustrates these two scenarios.

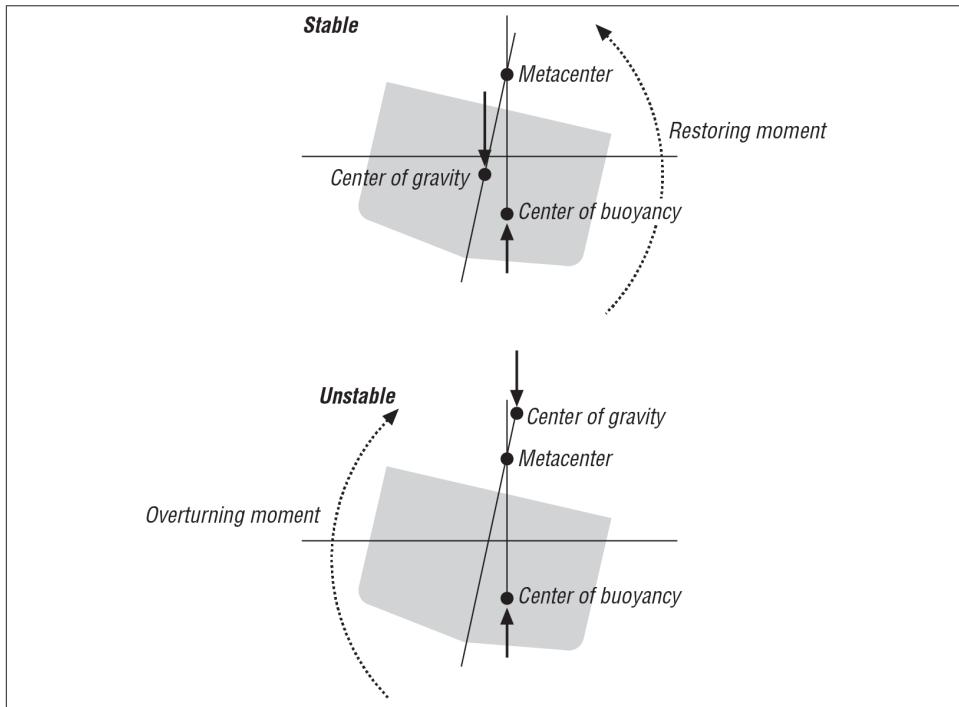


Figure 16-2. Ship stability

If you're a sailor, then you know how important it is to keep the center of gravity of your boat low. This helps increase the height of the metacenter above the center of gravity, and thus helps with stability.

In the case of fully submerged objects, like submarines, the situation is different. The buoyant force still acts through the geometric centroid of the object, but for stability, the center of buoyancy must be located above the center of gravity. This way, when the object rotates, the lines of action of the weight of the object and the buoyant force are separated and form a moment that tends to restore the object to its upright position. If it's the other way around, then the object would be unstable, like trying to balance one bowling ball on top of another. In this case, the slightest disturbance would upset the balance and the object would flip upside-down such that the center of gravity is located below the center of buoyancy.

Sinking

In general, boats protect their stability by compartmentalizing the hull into several watertight sections, fittingly called *compartments*. This way, if the side of a vessel hits an iceberg, only the compartment damaged will flood with seawater. If enough compartments are damaged, the vessel will not have enough buoyancy to support its weight and it will sink. The end with the flooded compartments will sink first, causing a large angle about the transverse axis. This is what happened to the *Titanic*. In fact, in that ship's case, the angle, called *trim*, was so large that the stern was lifted out of the water. The hull could not support the weight of the stern section that was no longer being supported by buoyancy, and the structure ripped in two.

It should be noted that ships can sink in the matter of minutes, or it can take hours. For instance, the *Titanic* took about three hours to sink. The *Lusitania* sank in 18 minutes. The time it takes depends heavily on the type of damage and the construction of the vessel. We don't suggest trying to get players to wait three hours for their game to end; however, it is possible to continue fighting/propelling a vessel that is terminally damaged. In many cases where terminal damage is suspected, captains endeavor to ground their vessels to prevent the ship from actually going under.

If side damage occurs, especially in high wind and waves, then it could be that the vessel can still have enough buoyancy to float, but no longer enough stability to remain upright. As damage usually occurs only on one side of a vessel, the center of buoyancy will no longer be on centerline. This means that the restoring moment in one direction is diminished by whatever amount the center has moved to that side. A big wave comes along and pushes the vessel over to the point where the righting arm is no longer positive. The vessel will flip 180 degrees with the bottom pointed skyward but will still float (capsizing). Once rolled over, the remaining compartments will tend to fill with water as vents or other openings fail over time. In the case of recreational boats, they are usually

only a single compartment. If they capsize, they will sink readily; indeed, this is the way that most small boats sink.

As we mentioned before, accurately computing all degrees of freedom for a nontrivial-shaped body in real time would be difficult to accomplish with today's computer hardware. In general, you want to follow a few high-level rules:

- The higher the center of gravity, the more likely it is that the boat will tip over.
- Large vessels are always compartmentalized. Damage should be limited to the watertight compartment in which it occurred.
- The vessel will heel or trim in the direction of damage. If damage occurs on the starboard side, the boat will heel to starboard. If the damage occurs in the bow, the boat will list forward.
- A boat will remain floating as long as the undamaged compartments have a volume in cubic meters of at least the weight of the hull in metric tons divided by 1.025.
- After being damaged, even if a vessel has enough undamaged volume to remain afloat, it doesn't necessarily mean it will float upright.
- Sinking almost never occurs as quickly as depicted in video games; however, capsizing can occur rapidly and is probably a more realistic way to model a stability failure.

Ship Motions

Closely related to ship stability is the subject of ship motions. Knowing how vessels work in a random set of waves will greatly help you to increase realism in your games. The most important aspect of this is coupled motions, which we will talk about shortly. First, some more vocabulary! As discussed before, there are six degrees of motion any rigid body is capable of; for boats, some of these have special names and are described next and illustrated in [Figure 16-3](#).

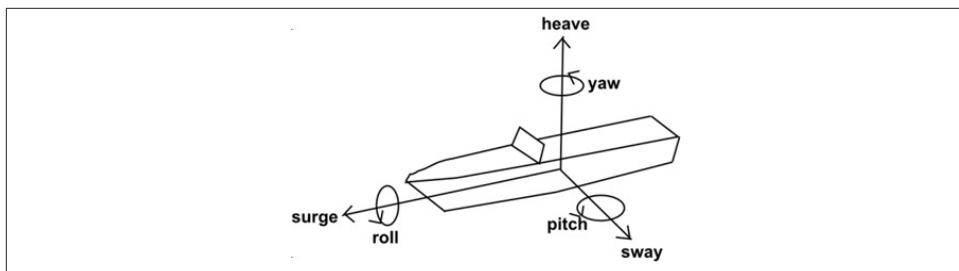


Figure 16-3. Floating-body degrees of freedom

Roll, pitch, and yaw are the terms also used for airplanes. The translation degrees of freedom are called surge, heave, and sway. Surge, sway, and yaw are not that apparent when vessels are moving forward, so it is acceptable to limit your model to heave, pitch, and roll. Heave is the up-and-down motion of the boat caused by the change in elevation of the water's surface as a wave passes. If a vessel is stationary, it would be referred to as bobbing. Pitch is the rotation about the transverse axis of the vessel due to increased buoyancy on one end of the ship as a wave passes. This motion is most pronounced when the waves are traveling in the same direction (or 180 degrees) from the vessel. Roll is like pitch, but about the longitudinal axis.

Heave

As stated before, heave is displacement in the vertical direction from the static equilibrium draft. This degree of freedom is straightforward to model as a hydrostatic spring acting in the vertical direction. Assuming we have a barge that is 30 meters long and 10 meters wide, we'll develop an equation that can govern our heave simulation.

Commonly, a vessel's hydrostatics include something called tons per centimeter immersion (TPCM)—that is, for every centimeter you press the boat down, a certain number of tons of buoyancy force is created. For our barge, this is a relatively straightforward calculation.

Given that the water plane area is a constant 300 square meters, 1 centimeter of immersion would result in a volume of 3 cubic meters. As 1 cubic meter of saltwater weighs 1,027 kg, 3 cubic meters would be 3,081 kg, and (assuming this boat is on Earth), would result in a buoyant force of $3,081 \text{ kg} \times 9.81 \text{ m/s}^2$, or 30.2 kN. Therefore, 30.2 kN per cm would make a good starting value for a spring constant to model the heave response of this vessel in waves.

Roll

For us to simulate realistic roll motions, it is important that the ship take time to complete the motion. This time is called the *roll period*. This defines the angular velocity that a ship rolled to one side will experience when it recovers. We can estimate it by the following equation:

roll periods are known as “tender” and lag behind the waves. These vessels generally heel farther over but are more comfortable for passengers.

Pitch

Likewise, there is a pitch period that measures the speed at which the vessel responds to a wave. This is highly dependent on the length of a ship and can be estimated as follows:

We'll describe each of these components and give you some empirical formulas in just a moment. First, however, we want to qualify the material to follow by saying it is very general in nature and applicable only when little detail is known about the complete geometry of the particular ship under consideration. In the practice of ship design, these formulas would be used only in the very early stages of the design process to approximate resistance. That said, they are very useful for getting in the ballpark, so to speak, and (sometimes more importantly) in performing parametric studies to see the effects of changes in major parameters.

The first resistance component is the frictional drag on the underwater surface of the hull as it moves through the water. This is the same as the frictional drag that we discussed in [Chapter 3](#). However, for ships there's a convenient set of empirical formulas that you can use to calculate this force:

test data where results from the model test are extrapolated to approximate drag on the full-size ship.

Just like pressure drag, wave drag is difficult to compute, and we usually rely on model testing in practice. Wave drag is due to the energy transfer, or momentum transfer, from the ship to the fluid, or in other words, it's a function of the work done by the ship on the surrounding fluid to generate the waves. The visible presence of wave resistance is evident in the large bow wave that builds up at the front of the ship as well as the wave system that originates at the stern of the ship as it moves through the water. These waves affect the pressure distribution around the ship and thus affect the pressure drag, which makes it difficult for us to separate the wave drag component from pressure drag when performing an analysis.

When scale model tests are performed, pressure drag and wave drag are usually lumped together in what's known as *residual resistance*. Analogous to the coefficient of frictional drag, you can determine a coefficient of residual resistance, such that:

-
1. These methods are quite involved and there are far too many to discuss here, so we've included some references in the [Bibliography](#) for you.

In lieu of enough information to calculate the projected transverse area of the ship, you can approximate it by:

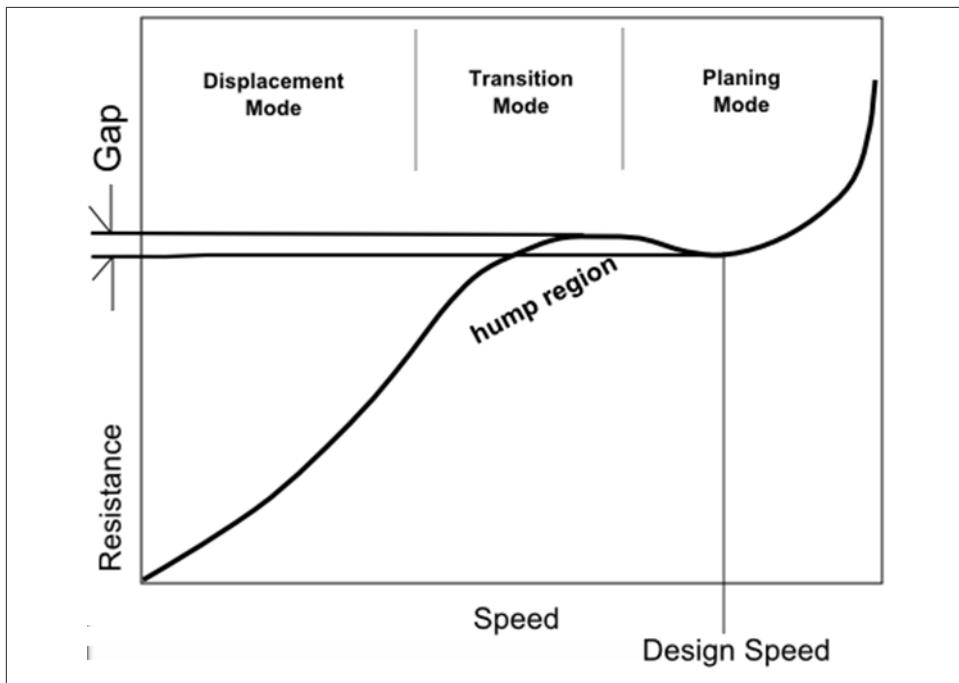


Figure 16-5. Planing craft resistance versus speed

Notice the hump region. This is where the vessel trims aft in the transition mode and there is a rise in the resistance. There have been embarrassing cases where a vessel, although having enough power to make the design speed once over the humps, lacked the power to make the transition.

Virtual mass

The concept of *virtual mass* is important for calculating the acceleration of a ship in a real-time simulator. Virtual mass is equal to the mass of the ship plus the mass of the water that is accelerated with the ship.

Back in [Chapter 3](#) we told you about the viscous boundary layer, and we said that the relative velocity (relative to the moving body) of the fluid particles near the moving body's surface is 0 at the body surface and increases to the free stream velocity as distance from the body surface increases. Essentially, some of the fluid sticks to the body as it moves and is accelerated with the body. Since the velocity of the fluid varies within the boundary layer, so does the acceleration. The *added mass*, the mass of water that gets accelerated, is a weighted integration of the entire mass of fluid that is affected by the body's acceleration.

For a ship, the viscous boundary layer can be quite thick, up to several feet near the end of the ship depending on its length, and the mass of water that gets accelerated is significant. Therefore, when doing any sort of analysis that involves the acceleration of the ship, you need to consider added mass, too. The calculation of added mass is beyond the scope of this book. We should also point out that, unlike mass, added mass is a tensor—that is, it depends on the direction of acceleration. Further, added mass applies to both linear and angular motion.

Added mass is typically expressed in terms of an added mass coefficient, which equals the added mass divided by the mass of the ship. Some methods actually integrate over the actual hull surface, while others approximate the hull as an ellipsoid with proportions matching the ship's. Using this approximation, the ellipsoid's length corresponds to the ship's length while its width corresponds to the ship's breadth. For longitudinal motion (that is, linear motion along an axis parallel to the ship's length), the added mass coefficient varies nearly linearly from 0.0 at a breadth-to-length ratio of 0 (the ship is infinitely thin) up to 0.5 at a breadth-to-length ratio of 1 (a sphere).

When the added mass coefficient is expressed as a percentage of the ship's mass, virtual mass can be calculated as $mv = m(1 + xa)$, where m is mass, and xa is the added mass coefficient—for example, 0.2 for 20%. For typical displacement ship proportions, the longitudinal added mass ranges from about 4% to 15% of the mass of the ship. Conservative estimates generally use 20%.

Guidance speeds

To provide some guidance, [Table 16-1](#) provides common ship types and appropriate speed ranges. This will help guide you in properly simulating the resistance of your vessel.

Table 16-1. Some vessels and their speeds

Vessel type	Speed (knots)	Horsepower (hp)
Aircraft carrier	31.5	260,000
Cruiser	30	80,000
Oil tanker	15–20	20,000–60,000
Containership	21	100,000
200-foot yacht	15.5	4,000
35-foot recreational boat	30	420
35-foot speedboat	70	1,200
40-foot sailboat	8.5	N/A

Note that at a certain speed, for non-planing hulls, there is a theoretical limit to how fast a boat can go. This speed is called the *hull speed*. At the hull speed the bow and stern waves reinforce each other, and there is a rise in wave-making resistance. This can be a

barrier for some fuller hulls. Note that the speed for the 40-foot sailboat is the hull speed of a 40-foot full-formed (not slender) hull. We can calculate the hull speed with the following formula:

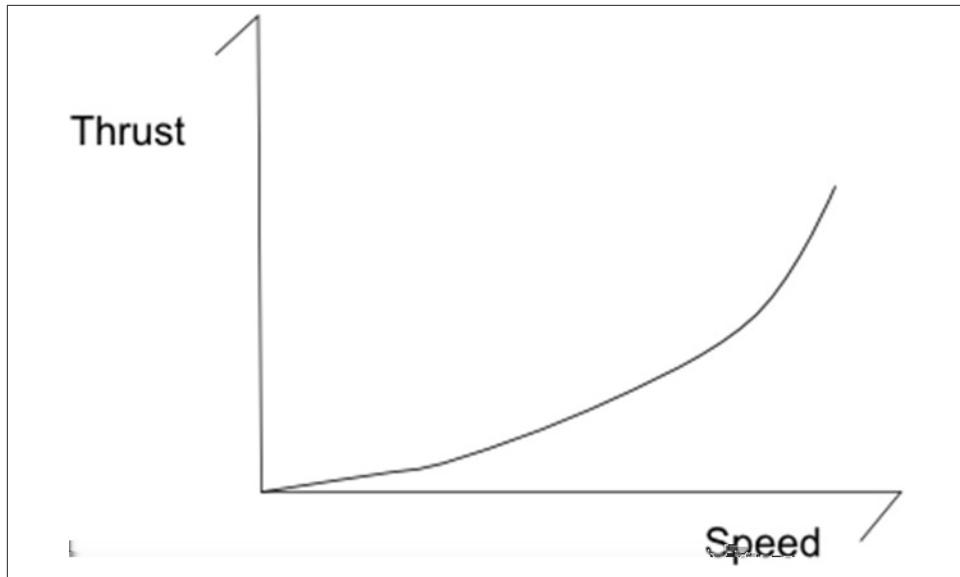


Figure 16-6. Speed versus required thrust

An important physical phenomenon concerning propulsion that you may want to incorporate is *cavitation*. Cavitation occurs when a propeller is moving fast enough that the low-pressure side of the blade starts spontaneously creating vapor bubbles. These bubbles exist for a short while, and then as the propeller turns, the static pressure changes. This higher static pressure causes the bubbles to collapse violently. This collapse is so fast and furious that it can cause metal erosion at a high rate. It will eat away at a propeller until it is no longer producing thrust. It is also very noisy. That is why submarine propellers are shaped very differently than other ships' propellers. They seek to limit cavitations so they're not heard by enemy vessels. The damage caused by cavitation also creates a speed limit on RPMs for a propeller. Cavitation is a real-life phenomenon you can exploit to penalize the player for driving around at high speed all the time.

Maneuverability

Another aspect of ships and boats that is often oversimplified is maneuverability. Maneuverability is also a very complex topic whose numerical simulation is beyond the current realm of real-time simulation. However, with some simplifications and assumptions, it can be more accurately modeled than if you do not know the underlying framework. Almost all vessels maneuver by way of two methods: rudders or thrust vectoring. The users generally won't care about the differences, so you can model both by angling the thrust vector off-center.

Rudders and Thrust Vectoring

Although rudders and thrust vectoring have the same result, there are some important differences. A thrust vectoring system, like a jet boat, can steer only when the vessel is producing thrust. A rudder, on the other hand, works only when the vessel has forward speed. If the boat isn't moving forward with enough speed, then the rudder can't produce a turning moment.

If you keep in mind those differences, you can model both systems the same way. The most important thing to keep in mind when modeling larger ships in your games is that they take significant time to respond to control inputs. [Figure 16-7](#) tracks the heading of a ship over time during what is called the *10/10 maneuver*.

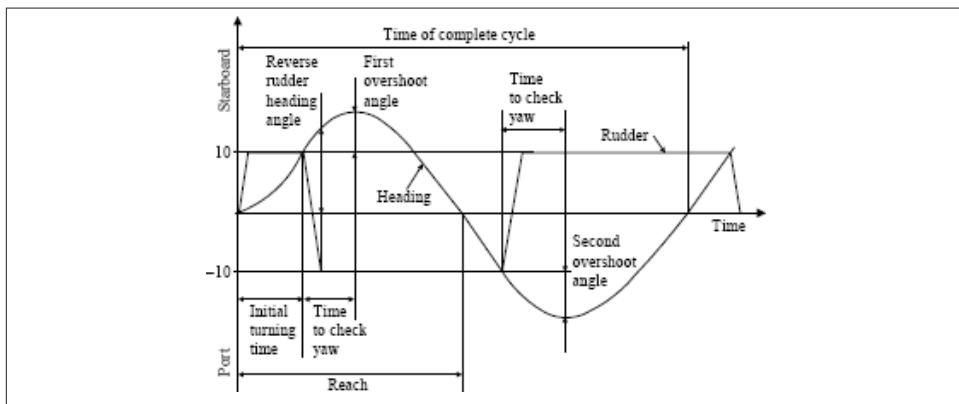


Figure 16-7. 10/10 zig-zag test

A vessel is moving in a straight line, and the rudder is put over 10 degrees in one direction. Once the vessel's heading changes 10 degrees, the rudder is moved to the opposite side at the same angle. The initial turning time is the time it took for the vessel to change its heading 10 degrees. As large ships have enormous momentum, they will continue to turn even though the rudder is in the opposite direction. The maximum deviation from the original heading minus the 10 degrees at which the rudder was flipped is called the *overshoot angle*. The size of this angle is one measure of how slow the vessel is to respond to the helm. For larger ships, this can be between 15 degrees when light and 45 degrees when loaded with cargo.

The time to check yaw is the time in seconds it takes for the overshoot angle to be achieved and the vessel to start changing its heading again. This is repeated for the other side to detect any bias a vessel may have for turning in a particular direction. The moral of the story is that for anything other than a high-speed small craft, boats and ships can take a significant amount of time to respond to the helm. Your simulation should strive to reproduce a turning ability that matches [Figure 16-7](#) for extra realism.

A special kind of thrust vectoring is called *throttle steering*. Imagine that a boat has two engines. If one is run in forward and the other in reverse, the vessel will turn quite rapidly. For a twin-engine vessel operating in close quarters, the rudders are often centered and the vessel maneuvered solely by altering the throttle settings of the two props.

Propeller walk

Another interesting maneuvering phenomenon that is closely related to thrust vectoring is called *propeller walk*, or *prop walk*. This is especially important for vessels with only one propeller moving in tight spaces. The cause of propeller walk is related to the fact that most propellers are installed at an angle to the horizon. This angle causes the thrust to be greater when the blades are moving down than when the blades are moving upward. In a propeller that turns clockwise, this creates a push to the right.

In forward gear the rudder is very effective at countering the propeller walk, but in reverse the rudder is much less effective, making the propeller walk much more noticeable. This can add a significant amount of realism when you are simulating vessels in docking maneuvers.

CHAPTER 17

Cars and Hovercraft

What cars and hovercraft have in common is that they operate in an essentially 2D manner. Unless they have jumped a ramp, both vehicles remain on the ground or water plane. In this chapter we'll discuss the forces behind each vehicle's method of travel and discuss how to accurately model them in your simulations.

Cars

In the following sections we want to discuss certain aspects of the physics behind automobile motion. Like the previous four chapters, the purpose of this chapter is to explain, by example, certain physical phenomena. We also want to give you a basic understanding of the mechanics involved in automobile motion in case you want to simulate one in your games. In keeping with the theme of this book, we'll be talking about mechanics in the sense of rigid-body motion, and not in the sense of how an internal combustion engine works, or how power is transferred through the transmission system to the wheels, etc. Those are all internal to the car as a rigid body, and we'll focus on the external forces. We will, however, discuss how the torque applied to the drive wheel is translated to a force that pushes the car along.

Resistance

Before we talk about why cars move forward, let's talk about what slows them down. When a car drives down a road, it experiences two main components of resistance that try to slow it down. The first component is aerodynamic drag, and the second is called *rolling resistance*. The total resistance felt by the car is the sum of these two components:

The aerodynamic drag is primarily skin friction and pressure drag similar to that experienced by the projectiles discussed in [Chapter 6](#), and the planes and boats discussed in earlier chapters. Here again, you can use the familiar drag formula:

work done over time, its units are, for example, foot-pounds per second. Usually power in the context of car engine output is expressed in units of *horsepower*, where 1 horsepower equals 550 ft-lbs/s.

To calculate the horsepower required to overcome total resistance at a given speed, you simply use this formula:

Here d_s is the skidding distance, g the acceleration due to gravity, μ the coefficient of friction between the tires and road, V the initial speed of the car, and φ the inclination of the roadway (where a positive angle means uphill and a negative angle means down-hill). Note that this equation does not take into account any aerodynamic drag that will help slow the car down.

The coefficient of friction will vary depending on the condition of the tires and surface of the road, but for rubber on pavement the dynamic friction coefficient is typically around 0.4, while the static coefficient is around 0.55.

When calculating the actual frictional force between the tire and road, say in a real-time simulation, you'll use the same formula that we showed you in [Chapter 3](#):

When you turn the steering wheel in a car, the tires produce a centripetal force toward the center of the curve via friction with the surface of the road. It follows that the maximum static frictional force between the tires and the road must exceed the required centripetal force. Mathematically, this takes on the following inequality:

If the front wheels slide, the car understeers and the arc is larger than the driver intended. This is commonly caused by traveling too fast through a corner and trying to take the corner too tightly. However, if the driver breaks hard or even just lets off the gas, there will be a weight shift forward. This will keep the forward wheels from slipping but if too aggressive will cause the rear wheels to slip as weight is transferred away from them. This causes the car to turn more than the driver intended, and can even result in a spin. These two conditions limit the speed at which a car can complete a turn and also the amount of deceleration the car can handle once the turn is initiated.

To increase the limit speed, v_{limit} we must increase the normal force. We could do so by increasing the car's mass, but this would have negative effects on the car's ability to accelerate or decelerate. A better solution is the use of aerodynamic features to create what is called *downforce*. You may have seen racecars or even street cars with large wing-like features called spoilers on their trunks. Formula 1 cars also have wing-like appendages on the front of the car. These are like the wings of an airplane but inverted so that instead of pulling the vehicle up, they actually push the vehicle down. These wings, therefore, increase the normal force, and their effects are proportional to speed so that more downforce is available at higher speeds...just when you need it. In fact, some very fast cars create so much downforce that if the road were inverted, the car would remain glued to the pavement and be able to drive upside-down.

The trade-off for this increased cornering limit speed is increased drag caused by the airfoils. This limits the top speed of the vehicle in areas of road that have no corners. It is good practice to allow users to select an angle of attack for their vehicle's airfoils. This forces them to choose between the option of higher top speed with slower cornering or faster cornering with lower top speed.

You may notice that on some raceways, the corners are not flat. This is called roadway bank or *superelevation*. In a corner where the car would normally skid, the superelevation helps keep the car in the turn because as the car is inclined, a force component develops that acts toward the center of curvature of the turn (see [Figure 17-2](#)).

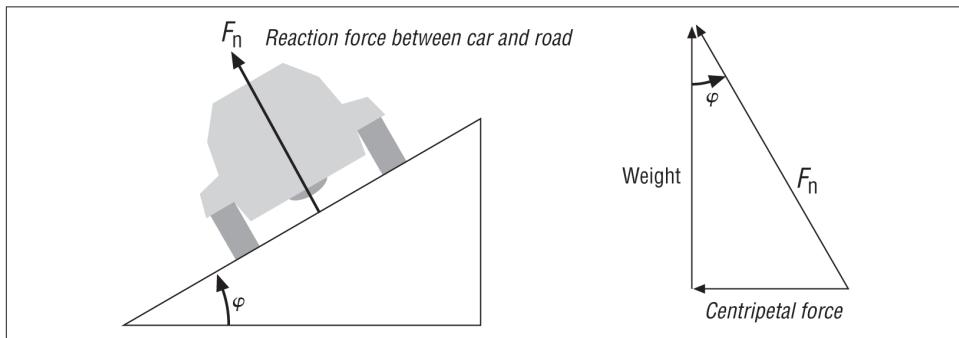


Figure 17-2. Superelevation

The following simple formula relates the superelevation angle of a roadway to the speed of the car and the coefficient of friction between the tires and road:

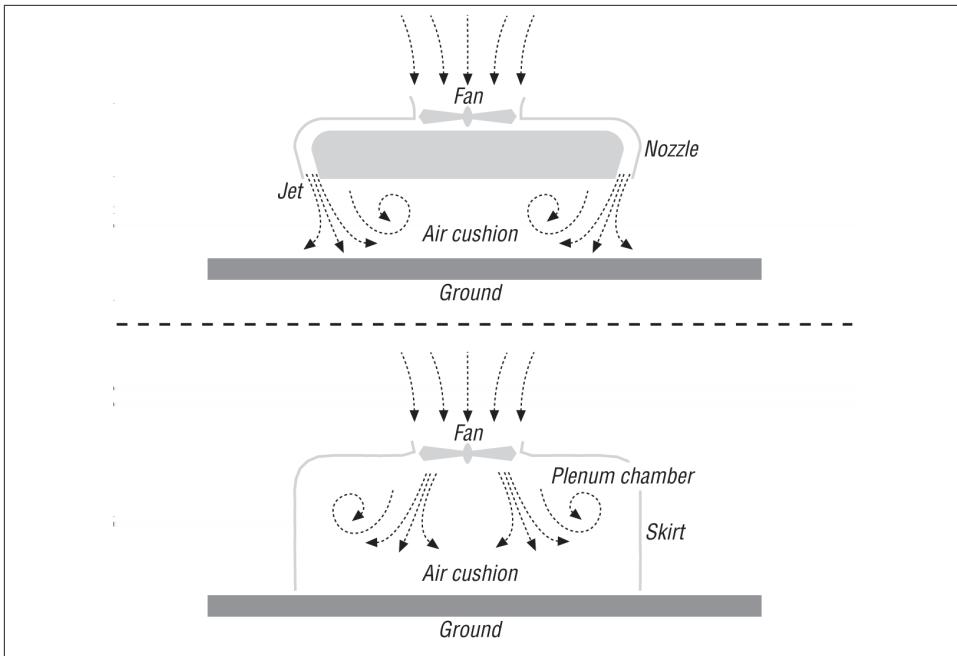


Figure 17-3. Hovercraft configurations

This approach proved impractical because hover heights were very limited and made the clearance between the hard structure of the craft and the ground (or water) too small to overcome all but the smallest obstacles. The solution to this problem was to fit a flexible *skirt* around the craft to contain the air cushion in what's called the *plenum chamber* (see Figure 17-3). This approach extended the clearance between the ground and the hard structure of the craft significantly even though the gap between the bottom of the skirt and the ground was very small. This is the basic configuration of most hovercraft in operation today, although there are all sorts of skirt designs. Some of these skirts are simple curtains, while others are sophisticated pressurized bag and finger arrangements. The end result is that hovercraft fitted with skirts can clear relatively large obstacles without damage to their hard structure, and the skirt simply distorts and conforms to the terrain over which the craft operates.

The actual calculation of the aerostatic lift force is fairly complicated because the pressure distribution within the air cushion is nonuniform and because you must also take into account the performance of the lift fan system. There are theories available to treat both the annular jet and plenum chamber configurations, but they are beyond the scope of this book. Besides, for a game simulation, what's important is that you realize that the lift force must equal the weight of the craft in order for it to maintain equilibrium in hovering flight.

Ideally, the ability of a hovercraft to eliminate contact with the ground (or water) over which it operates means that it can travel relatively fast since it no longer experiences contact drag forces. Notice we said *ideally*. In reality, hovercraft often pitch and roll, causing parts of the skirt to drag, and any obstacle that comes into contact with the skirt will cause more drag. At any rate, while eliminating ground contact is good for speed, it's not so good for maneuverability.

Hovercraft are notoriously difficult to control since they glide across the ground. They tend to continue on their original trajectory even after you try to turn them. Currently, there are several means employed in various configurations for directional control. Some hovercraft use vertical tail rudders much like an airplane, while others actually vector their propulsion thrust. Still others use bow thrusters, which offer very good control. All of these means are fairly easy to model in a simulation; they are all simply forces acting on the craft at some distance from its center of gravity so as to create a yawing moment. The 2D simulation that we walked you through in [Chapter 9](#) shows how to handle bow thrusters. You can handle vertical tail rudders as we showed you in [Chapter 15](#).

Resistance

Let's take a look now at some of the drag forces acting on a hovercraft during flight. To do this, we'll handle operation over land separately from operation over water since there are some specific differences in the drag forces experienced by the hovercraft.

When a hovercraft is operating over smooth land, the total drag acting against the hovercraft is aerodynamic in nature. This assumes that drag induced by dragging the skirt or hitting obstacles is ignored. The three components of aerodynamic drag are:

- Skin friction and viscous pressure drag on the body of the craft
- Induced drag when the craft is pitched
- Momentum drag

In equation form, the total drag is as follows:

Here ρ is the mass density of air, V the speed of the hovercraft, S_p the projected frontal area of the craft normal to the direction of V , and C_d the drag coefficient. Typical values of C_d for craft in operation today range from 0.25 to 0.4.

The next drag component, the induced drag, is a result of the craft assuming a pitched attitude when moving. When the bow of the craft pitches up by an angle τ , there will be a component of the aerostatic lift vector that acts in a direction opposing V . This component is approximately equal to the weight of the craft times the tangent of the pitch angle:

When a hovercraft operates over water, its air cushion creates a depression in the water surface due to cushion pressure (see [Figure 17-4](#)). At zero to low speeds, the weight of this displaced volume of water is equal to the weight of the craft, just as if the craft were floating in the water supported by buoyancy. As the craft starts to move forward, it tends to pitch up by the bow. When that happens, the surface of the water in the depressed region is approximately parallel to the bottom of the craft. As speed increases, the depression is reduced and the pitch angle tends to decrease.

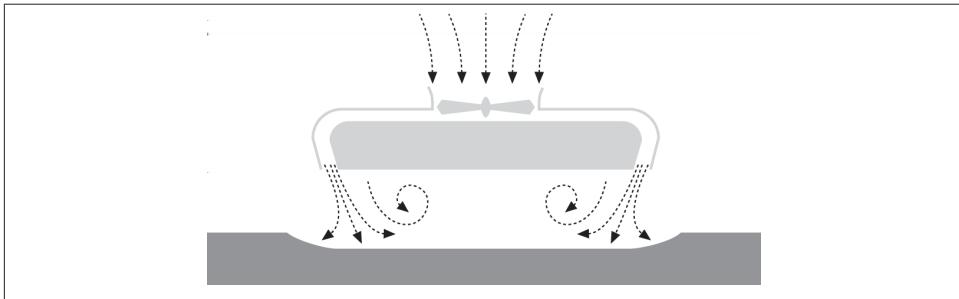


Figure 17-4. Hovercraft over water

Wave drag is a result of this depression and is equal to the horizontal components of pressure forces acting on the water surface in the depressed region. As it turns out, for small pitch angles and at low speeds, wave drag is on the same order of magnitude as the induced drag:

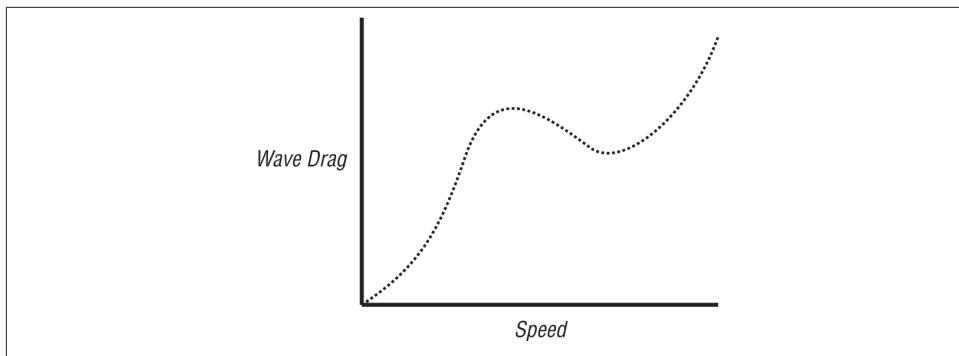


Figure 17-5. Wave drag

There are several theoretical treatments of wave drag in the literature that aim to predict the speed at which this hump occurs along with its magnitude. These theories indicate that the hump depends on the planform geometry of the hovercraft, and it tends to occur at speeds in the range of $\sqrt{gL} / 2$ to \sqrt{gL} , where g is the acceleration due to gravity and L is the length of the air cushion. In practice, the characteristics of a particular hovercraft's wave drag are usually best determined through scale model testing.

The so-called wetted drag is a function of several things:

- The fact that parts of the hull and skirt tend to hit the water surface during flight
- The impact of spray on the hull and skirt
- The increase in weight as the hovercraft gets wet and sometimes takes on water

Wetted drag is difficult to predict, and in practice model tests are relied on to determine its magnitude for a particular design. It's important to note, however, that this tends to be a significant drag component, sometimes accounting for as much as 30% of the total drag force.

Steering

In [Chapter 9](#), the hovercraft was steered using a bow thruster that pushed transversely forward of the center of gravity. In reality, most hovercraft are steered by vectoring the thrust of the propulsion fan via rudders attached directly aft of the fans. This can be modeled by angling the propulsive thrust.

The most important characteristic to remember about steering hovercraft is that they will not turn like a car or boat. The hovercraft, because it has lower friction with its environment, will take longer to turn and tends to continue in the direction it was heading while rotating. Once rotated, the thrust acts along a new vector. One possible maneuver in a hovercraft is to quickly rotate the vessel and then shut down the pro-

pulsive thrust. This would allow you to travel in one direction and point in another for as long as your momentum carries you. This could be very useful in strafing enemies or just racking up style points.

CHAPTER 18

Guns and Explosions

One of the most widely successful video game genres is the venerable first-person shooter. Ever since the breakthrough games of *Wolfenstein 3D* and *Doom*, the first-person shooter has received the lion's share of research and development budgets. It is amazing that the physics of aiming a gun and of a bullet traveling through the air are rarely modeled accurately. In general, game designers treat guns like laser beams so that wherever you point them, the bullet goes in an infinitely straight line. In this chapter we'll discuss how to more accurately model both aiming and the trajectory of bullets, which is known as ballistics.

Projectile Motion

There are actually four subtopics of ballistics. *Internal ballistics* is the study of what happens to the bullet inside the barrel of the gun; *transitional ballistics* is the study of what happens as the bullet exits the barrel. Once the bullet has fully exited the barrel, it is in the realm of *external ballistics*. At this point the only acceleration is that of gravity, and the same forces discussed in [Chapter 6](#) take over. The last topic is *terminal ballistics*, which is the study of what happens when the bullet hits its target. The last two topics are the ones we'll discuss here. The other phases are more important to firearm manufacturers and not so much to the shooter. If you don't recall the material in [Chapter 6](#), we highly recommend that you review it before continuing.

While we aren't very concerned about what happens in the barrel of the gun, there are a few tidbits we do need to know about the system. The first is where the barrel is pointed. This is referred to as the gun's *aim*, and is almost universally controlled by where the mouse is on the screen.

The next is the initial velocity of the bullet. The bullet here refers to the actual metal projectile that leaves the barrel; the thing that you load into the gun is called a *round* and contains a casing, gunpowder, a primer, and, of course, the bullet. The initial velocity is usually measured just after the bullet leaves the muzzle (the end of the barrel) and is appropriately called the *muzzle velocity*. Every kind of ammunition is tested at the factory and given a muzzle velocity. You can add realism to your game by giving different ammunition different muzzle velocities. This way, a handgun round won't have the same range as a rifle round. Ammunition also comes with a bullet weight measured in either grams or grains. One grain is equal to 0.0648 grams and is an old unit based on the weight of a single seed of wheat!

Last, but not least, we need some approximation for the way air resistance will affect the flight of a bullet. This is where things start to get interesting for people studying ballistics, but we'll stay away from exotic aerodynamics and use our existing drag model. First, we should review the current state of first-person-shooter physics.

Firearms in games present a unique problem to the game developer. If you have ever been to the firing range, you know that in reality it takes a good deal of practice and concentration to hit a target reliably. Considering that target shooting is hard enough to be an Olympic sport under very controlled circumstances, the ability for in-game characters to spring from cover and shoot five enemies with five bullets is somewhat superhuman. We have all played games where you find yourself shooting a target very far away, and the procedure is as simple as pointing the crosshair where you want the bullet to go and clicking the mouse button. In reality, the skill needed to get a bullet weighing a few grams to hit something a few hundred meters away is so complicated it is amazing that anyone does it with regularity. For those developers wishing to actually model firearm performance in their game, there is a double-edged sword to consider.

The physics of what happens to the bullet in its flight are not simple to boil down. However, the behavior of the bullet as it flies downrange is important in the practical art of marksmanship. There has been considerable work done to find a way to compare ammunition so that a hunter or marksman can predict the performance of a particular ammunition. The result is a pseudophysical factor called the *ballistic coefficient* (BC). The BC is a ratio that determines the ability of a particular bullet to retain its downrange velocity compared to some standard bullet. The most common form is that of the G1 reference projectile. However, this number has limited use, as it does not take into account modern bullet shapes that provide very low drag. There are updated models whose designations are G2, G3, and ECT. If you are interested in the details of highly accurate ballistic modeling, there are a few free programs that can provide you in-depth models such as Remington's Shoot! and the GNU Ballistics program. Given that most first-person shooters don't yet include wind effects or bullet drop, we'll limit ourselves to a simplified method of turning the existing parameters in the [Chapter 6](#) projectile algorithm.

Taking Aim

When discussing aim, we'll primarily be talking about rifles or carbines here, as handguns are not usually used for long-distance shooting. Similarly, shotguns, given that they fire many small projectiles, are generally not carefully aimed but instead pointed. Both of these weapons are what are known as *point-blank weapons*. Rifles have a point-blank range, which we will discuss, but for handguns and shotguns this really refers to the fact that at the ranges where these weapons are effective, you can reasonably expect to hit where you point the gun. That is not to say that these weapons don't need to be aimed to be effective, but in the fast-paced combat central to most first-person shooters it would be tedious to have the player use the sights on a handgun to effectively hit anything. Instead, most games use a “shoot from the hip” or free aim model, where the gun is not even in line with the camera. The careful programmer might still check to see if the target is within the effective range of the bullet before counting it as a hit even when auto-aiming in this manner. The effective range is the distance the bullet can travel before hitting the ground. Determining this is a straightforward application of the equations in [Chapter 2](#) and [Chapter 6](#).

To discuss firearms, we've adapted the code from the Cannon2 example in [Chapter 6](#) to run in a Java program called Marksman. In our example, the player is looking through a scope at a target of 1 meter by 1 meter. We've provided him an adjustable level of zoom so that as the range increases, he can still see the target. The aiming point is shown as an empty circle, and the bullet hole as a black filled circle. The method we've used to determine where the user is aiming in the model world converts the pixel-based location of the mouse to a coordinate in the model world. This distance and the range are then used to find the angles required to aim the gun. The code to do this is shown next, where `alp` and `gmm` are the angles of inclination and bearing. These are measured from horizontal and the line of view:

```
alp = 90-Math.toDegrees(Math.atan(((200-aimY)*(targetH/(drawH))/range));  
gmm = Math.toDegrees(Math.atan(((200-aimX)*(targetH/(drawH))/range));
```

where `targetH/drawH` is just the ratio of the target height to the height of the target in pixels on the screen. This allows the mouse coordinate given in pixels to be converted to meters. The arctangent then converts the ratio of these distances to an angle for the gun. The constant 200 refers to the pixel coordinate system, which is 200 pixels away from the center of the target. If you were to adapt this for a full 3D rendering system, you could remove a lot of these conversions we have to do.

At the 10-meter range shown in [Figure 18-1](#), we are within point-blank range and the bullet hole lines up with the sights.



Figure 18-1. Point-blank range

In [Figure 18-2](#), with the target at 100 meters, we see that for some reason the bullet is not hitting the target where we have pointed the weapon. At 300 meters, the bullet hole isn't even on the target. You can see that by simply accounting for projective motion and idealized drag, we are already having trouble hitting the bull's eye. The process by which these differences are accounted for is called *zeroing the sights*.

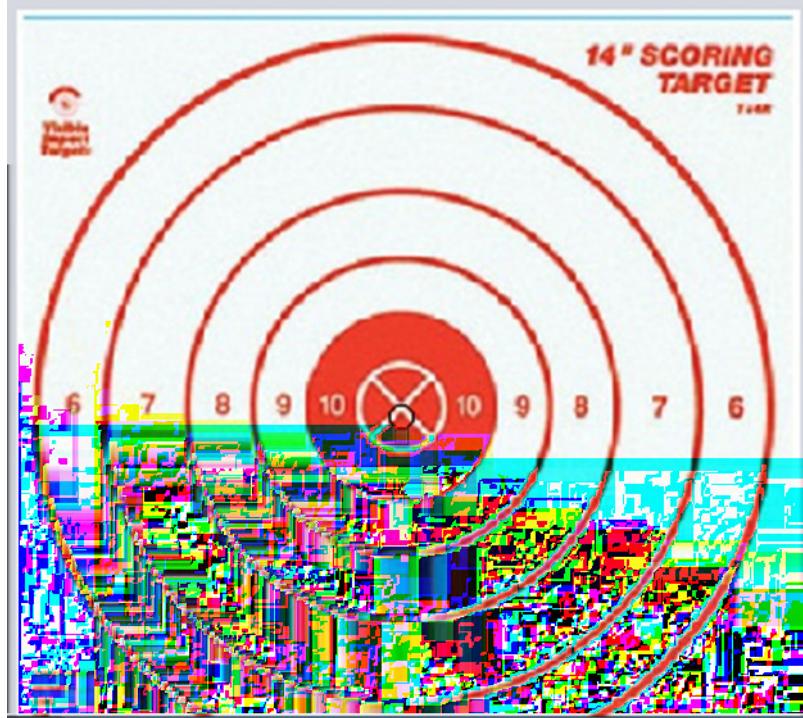


Figure 18-2. Hitting lower than expected

Zeroing the Sights

The idea of zeroing the sights is probably the most important thing to model if you want to have realistic gunplay in your game. As we mentioned before, when players are running from room to room, they probably don't want to be thinking about the wind and the range. However, for a hunting simulation or a sniper game, it may be appropriate to introduce this.

When a person is looking through a scope, her body and the rifle become a rigid body so that to change the aim of the weapon, she must rotate her entire body. This is convenient for us, because the player generally controls the shooter's position with his left hand via the keyboard and the direction of aim with the right hand. Other methods of aiming, briefly described previously as free aim, don't really have a counterpart in the real world, so we'll be limiting ourselves to this solid-body aiming.

Bullet drop: Gravity and air resistance

If you are aiming a rifle horizontally, you might expect that the bullet leaves the muzzle horizontally, and that gravity and air resistance cause it to drop from there. [Figure 18-3](#)

shows a rifle and scope combination that is mounted perfectly parallel. Ignoring all other factors for a minute, we see that the bullet will never hit where the scope is pointed. It will always be a few centimeters low.

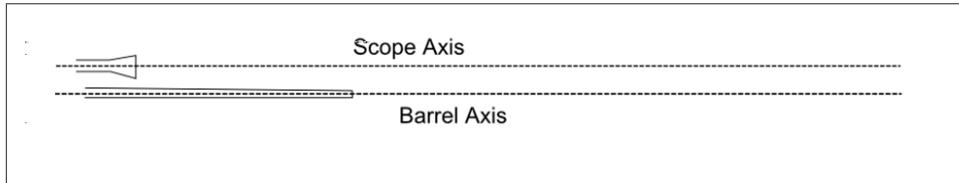


Figure 18-3. Zero-elevation scope

By adjusting the elevation control of the scope, we can make the rifle hit where the scope is pointed. (See [Figure 18-4](#).) The range at which the bullet will cross the line defined by the scope is called the *zero range*. If a target is at the zero range, you simply point the crosshairs and pull the trigger.

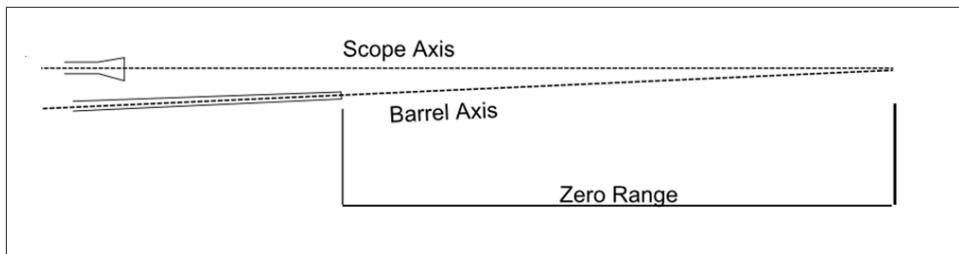


Figure 18-4. Scope with elevation

If we remove the target and graph the trajectory, it would look something like [Figure 18-5](#).

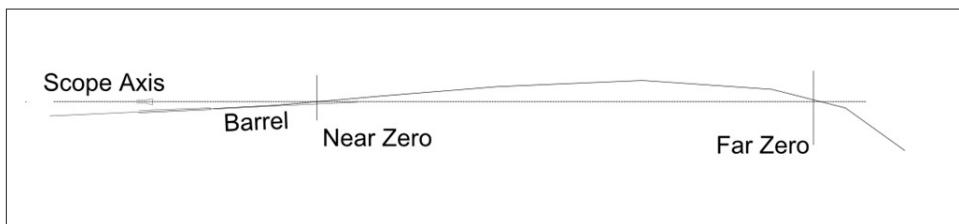


Figure 18-5. Ballistic trajectory

Here we can see there is a second point in which the bullet will cross the line made by the scope's crosshairs. This is called the *far zero*, or *second zero*. This is generally where the thing you will be shooting at will be. This diagram is often available from ammu-

nition suppliers. More often, they provide ballistic tables of downrange heights. It is important when using these charts that you remember they assume the scope to be horizontal and that the bullet starts with a negative height below the scope line. **Table 18-1** shows data from Remington's website and assumes the rifle was zeroed at 100 yards; it tells you how far the bullet is below the horizontal at every 50 yards past that 0. You can tune the drag coefficient in the projectile simulation to match these elevations.

Table 18-1. Long-range trajectory Remington Express .45-70 Govt

Range in yards	100	150	200	250	300	400	500
Drop in inches	0	-4.6	-13.8	-28.6	-50.1	-115.47	-219.1

From this table you can see that the bullet drop is over two feet at 250 yards. This means that if the target is at 250 yards and your scope is zeroed at 100 yards, then you would have to aim two feet above your target to hit it. That might be high enough that you can't even see the target in the scope anymore!

To counteract this problem, most rifles come with scopes that have elevation adjustments. This is a knob on the side of the scope that can be rotated to discrete settings called clicks. Most scopes use a 1/4 minute of angle adjustment per click although some use 1/8, 1/2, or even full minutes. A minute of angle is simply 1/60 of a degree. Therefore, when adjusting for elevation, the shooter can turn a knob on the scope, and as she hears the clicks, she knows that she has adjusted her scope however many minutes of angle. Now when she re-aims the crosshairs on the target, the barrel will have a slightly different angle than it did before, essentially aiming higher to accommodate the longer distance.

To achieve high accuracy in the field, the shooter would know that she zeroed her rifle to a certain range. Then, when attempting a shot, she would estimate the range to her target and adjust the scope however many clicks up or down. The biggest cause of error is an inaccurate estimate of range. Modern shooters often use laser range finders to determine exactly what elevation offset is required. In long-distance shooting situations, you can provide the range to the users and let them adjust the rifle's scope from the current zero range to a new zero range.

Most games today don't model even the effect of gravity on a bullet, so adding elevation adjustment to a sniper or hunting portion of your game will add much-needed accuracy.

Wind

Just like in the [Chapter 6](#) projectile example, our target shooting game's bullets are affected by wind. Just as before, a bullet's susceptibility to wind largely depends on its lateral drag coefficient. In our simulation, you can tune the bullet's susceptibility to the wind by adjusting the factor C_w . This, again, will apply only to rifles shooting at long ranges. At 20 meters, the wind will have little to do in determining where a bullet will hit. At 600 meters, it can cause the bullet to be off by a meter! The adjustment for wind

is similar to the adjustment for elevation. By turning a knob, you can adjust your scope or sights to be slightly to the right or left of the barrel's centerline. This angle, called gamma in our simulation, enables you to cancel out the effect of wind.

To deal with wind in the field, we require simple calculations that depend on the particular performance of the shooter's ammunition. It may work something like this: if the wind is blowing directly across your shot, the adjustment in inches is going to be half the wind speed in miles per hour. Of course, there are many other rules of thumb that differ for each caliber of ammunition, but for our simulation you can tune the wind drag coefficient to whatever value you want your shooter to encounter. He will have to spend some time at the range, just like a real shooter, figuring out how much the wind affects his shots. As the wind changes, he will have to adjust in real time and either aim to the left or right of the target or readjust the windage settings for the scope.

Breathing and Body Position

Although most games don't model gravity and wind when calculating bullet trajectories, many do attempt to regulate the accuracy with which you initially fire the bullet. Most commonly, game developers accomplish this by approximating the crosshairs with four lines that do not intersect. When fired, the bullet will land anywhere within the circle described by the inner endpoints of these four lines. Different weapons have different accuracies, and the lines can move in or out to reflect that. Usually the first shot is the most accurate, and once the weapon is fired you must "resight" the target, and this takes time. Therefore, shots fired in quick succession usually become less and less accurate.

In our simulation, we modeled a few things that affect accuracy in the real world and will give you some suggestions for other factors you could easily include. As our game was most interested in long-distance shooting via rifles, the most common source of error is breathing. As discussed before, when a shooter is looking through a scope on a rifle, she essentially becomes a fixed body. As she breathes, the rifle is essentially breathing too. When making difficult shots, it is very common for the shooter to take a breath and hold it while firing. In our simulation, we've modeled this with a `breathing` class that adjusts the point of aim up and down with time to mimic how a scope moves when the shooter is breathing deeply. This works via a timer started in our `initComponents()` function that fires every 100 ms. In the code that follows, you'll see that this leads to a breath every two seconds.

```
timer = new Timer(100, TargetPanel);
timer.start();
```

That function causes the aim point (`aimX, aimY`) to be moved independently of the cursor via the following algorithm:

```
if (direction == true) {
    breathHeight = breathHeight + 1;
    if (breathHeight == 5) {
```

```

        direction = false;
        breathHeight = breathHeight + 1;
    }
}

if (direction == false) {
    breathHeight = breathHeight - 1;
    if (breathHeight == -5) {
        direction = true;
    }
}

if (breathing) {
    aimY = aimY + breathHeight;
}

```

Here we are simply moving it up to some limit—in our case, 5 pixels—and then moving it back down. A better implementation would increase the unsteadiness as the user zooms in, as shakes are magnified also. There is no limit to the innovative functions you can use to move the aiming circle away from the cursor to simulate the reality of having to aim a gun. Yet this is certainly an area of first-person shooters that is lacking in variety.

When he is ready to fire, the user can left-click to hold his breath, the variable `breathing` becomes `false`, and the crosshairs will stop moving. This simple addition makes the game much more challenging and engaging. It should be noted that if the shooter holds his breath too long in real life, the aim will again become unsteady as his body reacts to not having fresh oxygen. Another improvement to this algorithm would be for the aim to become unstable after the left mouse button is pressed for some amount of time.

Many games also change the accuracy of a weapon depending on body position. There are three basic types of shooting positions: standing, kneeling, and prone. Standing is—you guessed it—standing up. Kneeling is some form of squatting rather than just kneeling on your knees. Prone is laid flat on the ground. Because the rifle is locked to your body, the less unstable your body is, the less unstable the aim. When standing, your body's muscles have to do a lot of work to remain upright. When kneeling, they do less so, and when prone, your muscles don't have to worry about keeping you standing at all.

You can add these parameters as random twitches in the aim and tune them to change the relative advantage of each position. However, prone should always be more stable than kneeling, and kneeling more stable than standing.

Recoil and Impact

Now that we've aimed, fired, and figured out where the bullet is at any given moment, let's talk about the last phase, terminal ballistics. To really understand what happens at

the end of a bullet's flight, let's revisit the beginning. Earlier we talked about recoil as the result of Newton's *conservation of momentum*. Everyone has seen a cheesy movie where the hero shoots the bad guy and the bullets cause the bad guy to be blown off his feet. There is a big problem with this! If the bullets were powerful enough to knock the person they hit off his feet, then the person shooting the gun would also be blown off his feet! In reality, the force felt by the person being shot is nearly the same as the force felt when the weapon recoils. For a 9 mm bullet weighing 7.45 g and leaving the barrel at 390 m/s, the gun will experience recoil such that its momentum is equal to the moment of the bullet.

One interesting way to incorporate recoil into a video game is in space. On Earth, a gun's recoil is pretty quickly transferred to the ground by friction between the player and the big bad earth. In space, the shooter has no planetary body to push against, so the recoil of the gun becomes the recoil of the gun/person system. Next time your character needs to move from one ship to another in a micro-gravity environment, you can make her spend some ammo to get herself moving.

Now, if you get shot you will probably fall down pretty quickly, but this has more to do with biology than physics. However, ignoring living targets, if you want to simulate the damage done by a bullet hitting something, it is more important to look at the bullet's kinetic energy. In fact, bullets and artillery shells are called *kinetic weapons*, as their primary means of destroying a target is by transferring their kinetic energy to the target. This is different than, say, a bomb that transfers its chemical energy into heat and kinetic energy after impact.

Explosions

Accurately modeling explosions involves multiphysics fluid simulations like the kind discussed in [Chapter 14](#) through [Chapter 16](#). One of our pet peeves is that video games usually have a collection of barrels lying around that, if shot once, explode violently enough to blow up nearby vehicles. While this makes for an easy out against multiple enemies, it is actually pretty hard to get everyday objects to blow up. Shooting a gas can with a handgun will almost never result in a fire, much less an explosion. Indeed, even shooting a propane tank with a rifle won't give you fireworks. It would take something like a tank of 1/4 propane mixed with 3/4 oxygen to explode, and those aren't usually lying around. Regardless, when we play video games we're often thankful that we have an occasional red barrel to shoot, so we'll review how to make the resulting explosion more accurate even if the ignition is improbable.

Particle Explosions

For most in-game explosions, it will be sufficient to implement a particle-type explosion that we covered way back in [Chapter 2](#). Now, in [Chapter 2](#) the particles were simply dots, but they don't have to be limited to such simple sprites. In some cases, such as sparks from a bullet hitting a metal container, it would be very accurate to model explosions as particles; however, by making our particles look like bits of cars, we can also make it appear that the car itself exploded. The reason this is easier is because the particle explosions don't have any angular motion. Although you can assign different parts of the cars to different particles, when they fly off due to the explosion, they won't be rotating. The good news is that a particle explosion will still give you a realistic distribution of fragments of something on the ground.

To talk about how to link bullets and particle explosions in detail, we'll consider something more physically accurate than a bullet blowing up a car. Let's consider a bullet hitting some loose gravel. This will generally cause the gravel to be thrown up into the air from the bullet collision. Instead of trying to calculate the complex collisions during impact, we'll generate a particle explosion based on the code in [Chapter 2](#):

```
void CreateParticleExplosion(int x, int y, int Vinit, int life,
                             float gravity, float angle)
{
    int i;
    int m;
    float f;

    Explosion.Active = TRUE;
    Explosion.x = x;
    Explosion.y = y;
    Explosion.V0 = Vinit;

    for(i=0; i<_MAXPARTICLES; i++)
    {
        Explosion.p[i].x = 0;
        Explosion.p[i].y = 0;
        Explosion.p[i].vi = tb_Rnd(Vinit/2, Vinit);

        if(angle < 999)
        {
            if(tb_Rnd(0,1) == 0)
                m = -1;
            else
                m = 1;
            Explosion.p[i].angle = -angle + m * tb_Rnd(0,10);
        } else
            Explosion.p[i].angle = tb_Rnd(0,360);

        f = (float) tb_Rnd(80, 100) / 100.0f;
    }
}
```

```

Explosion.p[i].life = tb_Round(life * f);
Explosion.p[i].r = 255;//tb_Rnd(225, 255);
Explosion.p[i].g = 255;//tb_Rnd(85, 115);
Explosion.p[i].b = 255;//tb_Rnd(15, 45);
Explosion.p[i].time = 0;
Explosion.p[i].Active = TRUE;
Explosion.p[i].gravity = gravity;
}

}

```

As you can see, the initial velocity V_0 controls the strength of the explosion. In [Chapter 2](#), we chose this value randomly. Now that we have a bullet flying through the air, we can make a better estimate of how strong of an explosion to create. As you recall from earlier in the chapter, a bullet has an energy associated with it at any time, t , in its flight. This energy is its kinetic energy and is equal to half the bullet mass times its velocity squared.

In our projectile simulation, it is simple to calculate this energy as the bullet flies through the air. It should be noted that a big bullet moving slowly is just as powerful as a smaller bullet moving quickly. Our upcoming code is going to assume that 100% of the kinetic energy is delivered to the target. This would not be true if a bullet shot straight through something. A way to visualize this is to imagine two targets, both hanging from the ceiling. One is made from paper and one is made from steel. When shot at, the steel target swings from its support, while the paper target stays still. This is because the bullet is passing straight through the paper and not transferring its kinetic energy to the target. To make things simple, we'll transfer all of the bullet's kinetic energy to the gravel. In equation form, this would look like:

```

BOOL          Active;           // indicates whether this particle
                                // is active or dead
float         mass;            //for calculating the particle's energy
} TParticle;

#define      _MAXPARTICLES 50
#define      _MASSOFPARTICLE .25

typedef struct _TParticleExplosion
{
    TParticle      p[_MAXPARTICLES]; // list of particles
                                // making up this effect
    int           x;   // initial x location
    int           y;   // initial y location
    float         KE;  //Available kinetic energy
    float
    BOOL          Active;        // indicates whether this effect is
                                //active or dead
} TParticleExplosion;

```

Notice that `V0` is no longer required, as the available kinetic energy will govern the strength of the explosion. Assuming that the bullet's kinetic energy is given as a variable `KEb`, our new `CreateParticleExplosion` function would look like the following:

```

void CreateParticleExplosion(int x, int y, int KEb, int life,
                             float gravity, float angle)
{
    int     i;
    int     m;
    float   f;

    Explosion.Active = TRUE;
    Explosion.x = x;
    Explosion.y = y;
    Explosion.KE = KEb;

    for(i=0; i<_MAXPARTICLES; i++)
    {
        Explosion.p[i].x = 0;
        Explosion.p[i].y = 0;
        Explosion.p[i].m = _MASSOFPARTICLE; //Mass of a single gravel
        Explosion.p[i].vi = tb_Rnd(0, sqrt(Explosion.KE/(_MASSOFPARTICLE*
                                              _MAXPARTICLES)));
        Explosion.KE = Explosion.KE - ((1/2)*(Explosion.p[i].m)*
                                         (Explosion.p[i].vi));

        if(angle < 999)
        {
            if(tb_Rnd(0,1) == 0)
                m = -1;
            else
                m = 1;
        }
    }
}

```

```

        Explosion.p[i].angle = -angle + m * tb_Rnd(0,10);
    } else
        Explosion.p[i].angle = tb_Rnd(0,360);

    f = (float) tb_Rnd(80, 100) / 100.0f;
    Explosion.p[i].life = tb_Round(life * f);
    Explosion.p[i].r = 255;//tb_Rnd(225, 255);
    Explosion.p[i].g = 255;//tb_Rnd(85, 115);
    Explosion.p[i].b = 255;//tb_Rnd(15, 45);
    Explosion.p[i].time = 0;
    Explosion.p[i].Active = TRUE;
    Explosion.p[i].gravity = gravity;
}

}

```

As you can see, we've altered the statements that set the initial velocity of the particles to be a random-number generator in a range anywhere from 0 to a velocity that would consume the entire explosion's kinetic energy. The next line reduces the available kinetic energy in the explosion by the amount just assigned to the particle. This way, you can be sure that the outgoing explosion is never more powerful than the input. A more interesting way to handle this would be to first initialize the particles with some given mass distribution and to assign the velocities not randomly, but with a normal distribution. Numerical recipes in C can help you accomplish this.

Even though the preceding code does not take into account some of the more subtle aspects of the transfer of kinetic energy, it will ensure that a small, slow-moving bullet produces a smaller explosion than a big, fast-moving one. This is something that is lacking in today's video games.

Polygon Explosions

While particle explosions are appropriate for small, uniform objects, they fail to give appropriate realism when something is blown into identifiable chunks. This is why in video games you rarely see a car explode and the door fly away to land next to you. Instead, games usually handle objects like this with a particle explosion that obscures the object while it is re-rendered in its now-exploded state with the missing pieces having been apparently blown to smithereens.

If you do want to model a full explosion of solid bodies, you can reuse the particle code for the translation aspects. Essentially the particles will now describe the center of gravity of each solid body. You will have to add in an initial angular velocity and let the simulation, as described in [Chapter 12](#), handle their motion after that initial angle.

While we don't have room to go over another example here, we'll talk a little about the input energy to such an explosion to help you bridge the gap. While we are on that subject, let's recall that a bullet just doesn't have the energy required to blow something apart. Even when you hit something with a tank-mounted gun, it really isn't the kinetic

energy of the bullet that blows apart the thing you hit, but some secondary explosion. In the case of a tank hitting another tank, the molten slag from the impact is usually peppered all over the inside of the tank, causing the fuel or ammunition to explode. That is where you get the big booms—it's the conversion of chemical energy to heat, light, and pressure!

The most common method of quantifying the chemical energy in weapons is called *TNT equivalency*. This is how much TNT it would take to cause the same explosion regardless of what you are actually exploding. Now, explosion modeling of, say, gasoline and air is pretty complex, so let's stick with TNT. A kilogram of TNT contains 4.184 Mega joules of energy; a 9 mm round has 400 J. You can see from that comparison why it is hard to blow something up by shooting at it, but easy to do with a block of TNT.

For the purposes of this discussion, let's say you have an open box (five polygon sides) into which your player just tossed a 1 kg block of TNT. When the TNT is detonated, you can give each polygon side an initial velocity (translational and angular) and let the kinematic equations take over. Those velocities can be based on two simple rules.

- The velocity vector can be defined by two points: the center of the block of TNT and the center of area of the polygon.
- The sum of all the kinetic energy must be less than the available chemical energy in the TNT. This can be prorated by the square of the distance from the polygon to the block of TNT.

The use of the center of area in our first rule will impart some rotation into our polygon, as it will cause a moment about the center of gravity unless the two coincide. If this is the case, as it would be for our box, aerodynamic drag and unevenness of the explosion will still cause rotation, so you should either model these explicitly or impart some rotational velocity manually.

Now that we have a velocity direction, we need to define its magnitude. The force on objects near an explosion is caused by the rapid expansion of gasses due to the heat generated by the detonation of the explosive. However, not all the chemical energy is transferred to the objects—a lot of it is converted into heat, light, and sound. Typically, only one-third of the available chemical energy is converted in the initial detonation. Let's call this the efficiency of the explosion, which we'll denote by ζ . Therefore, we can write the relationship between velocities of the polygons as follows:

divided up equally, you can see that lighter objects will have higher velocities, as you might expect. You can also adjust the amount of explosion energy available to each object by weighting the object's imparted energy by its distance from the explosion. This should also be tuned in the program, but in general, pressure from an explosion decreases with the cube of distance and exponentially with time.

If you want to model more complex explosion-structure interactions, there are many good references for how, say, buildings, react to bomb blasts. FEMA, as well as the Army and Navy, have several papers on the subject, such as *FEMA 426 Reference Manual to Mitigate Potential Terrorist Attacks Against Buildings*. The general concepts laid out in such documents can increase the realism of the building damage due to explosions.

CHAPTER 19

Sports

The topic of sports is nearly as vast as all of the subjects we've covered combined. There is a sport for everyone, and a sport that takes advantage of each of the physical models we've discussed so far. The topic ranges from games full of accessories, such as golf or polo, to running, where all you need are your own two feet.

One of the most attractive aspects of sports for the game programmer is that they take place in a limited physical space by design. Unlike a first-person shooter where the player will eventually reach an artificial boundary, in a sports game the player will not expect to be able to walk out of the court. Almost all sports have defined dimensions that are relatively easy to model. **Table 19-1** lists a few sports and their professional field dimensions.

Table 19-1. Various field dimensions

Sport	Field size
Soccer (football)	90–120 m long by 45–90 m wide
Football (including end zones)	109.7 m long by 48.8 m wide
Baseball	27.4 m between bases; 18.39 from pitcher's mound to home base; outfield varies
Basketball (international)	28 m by 15 m
Ice hockey (international)	61 m by 30 m

As you can see, other than baseball—where the shape of the outfield changes depending on what stadium you are in—modeling these field sizes is a rather straightforward exercise.

Additionally, the one thing that sports have in common is that they have a human actor. In this chapter we'll explore how the human action can be simulated as input for the other physical simulations we've discussed. Specifically, we'll show you an example of how to model a person swinging a golf club using accurate physiological models. This is called *biomechanics*. Before we get into that, another important thing to understand

when you're modeling sports is the limits of the human body. Although records are broken in every Olympics, no human being is able to jump 10 feet vertically into the air. Unless you are breaking the limits of biomechanics on purpose, doing so will decrease the realism of your game. The biomechanical statistics of what would be considered an outstanding athlete are given in [Table 19-2](#).

Table 19-2. Table of human performance

Physical attribute	Average value	Record value
Jump from standstill (vertical)	81 cm	155 cm
Running jump height (high jump)	1.83 m	2.45 m
Jump distance	5.0 m	8.95 m
Throwing speed	24.5 m/s	46.0 m/s
Running speed over 100 m	7.5 m/s	10 m/s
Running speed over 10,000 m	3.7 m/s	6.3 m/s

Almost all sports records are available online somewhere, so [Table 19-2](#) is by no means exhaustive. However, it is a good idea to use these values to limit your simulations of human actions in your video games. Obviously, part of the excitement of playing video games is to be able to jump higher and run faster than you otherwise could, but a good survey of biomechanics will at least let you know what is extraordinary and what is not. Now let's take a look at how we would model a human actor in a sports game.

Modeling a Golf Swing

Let's say you're writing a golf game and you want include a little realism. An obvious important element of the game is the golf swing. Another is the club-to-ball impact, and still another is the trajectory of the ball in flight. You can use the projectile motion modeling techniques discussed earlier in [Chapter 2](#), [Chapter 4](#), and [Chapter 6](#) to model the ball's flight, and the collision response techniques in [Chapter 5](#) to model the club-to-ball impact. But what about the golf swing?

Well, before we show you one way to model a golf swing, let's talk about why you would want to do so in the first place. To model club-to-ball impact, you need to know the club head velocity at the time of impact. That velocity is a function of the swing. The golfer raises the club through his backswing, torques his body, and brings the club head down in an arc while applying a torque with his wrists. As the club swings down, the wrist torque reverses, and the club whips through the downswing until the club head collides with the ball. (Or, in our case, collides with the ground!) Now, there are many subtle details we've omitted here with regard to technique and the physics, but you get the idea. At any rate, the swing determines the club head velocity at the moment of impact, which in turn determines the velocity of the ball after impact.

If you were writing a game for the Wii or some other platform that can capture a player's motion, then you can relate the player's swing motion to the initial torque applied to a virtual golf club, thus determining, through some model, the swing dynamics and resulting club head velocity.

Golfers take swing technique seriously and so do scientists who study golf swing dynamics. In an effort to understand what makes a good swing or how to improve a swing, there are many scientists out there actively studying the golf swing physics. As a result, there are many mathematical models of varying degrees of realism and complexity that aim to examine the golf swing. One example is the so-called *two-rod model* as described in Theodore P. Jorgensen's book *The Physics of Golf*. In his book, Dr. Jorgensen describes the two-rod model in detail, including assumptions and simplifications, and provides the resulting equations that must be solved to simulate a golf swing based on this model. He even provides empirical data used to validate the results of the mathematical model. As shown in [Figure 19-1](#), the two-rod model assumes that the golfer's arm is one rod that extends from the shoulders to the wrists. This is the *arm rod*. The club is represented by another rod that extends from the wrist end of the arm rod to the club head.

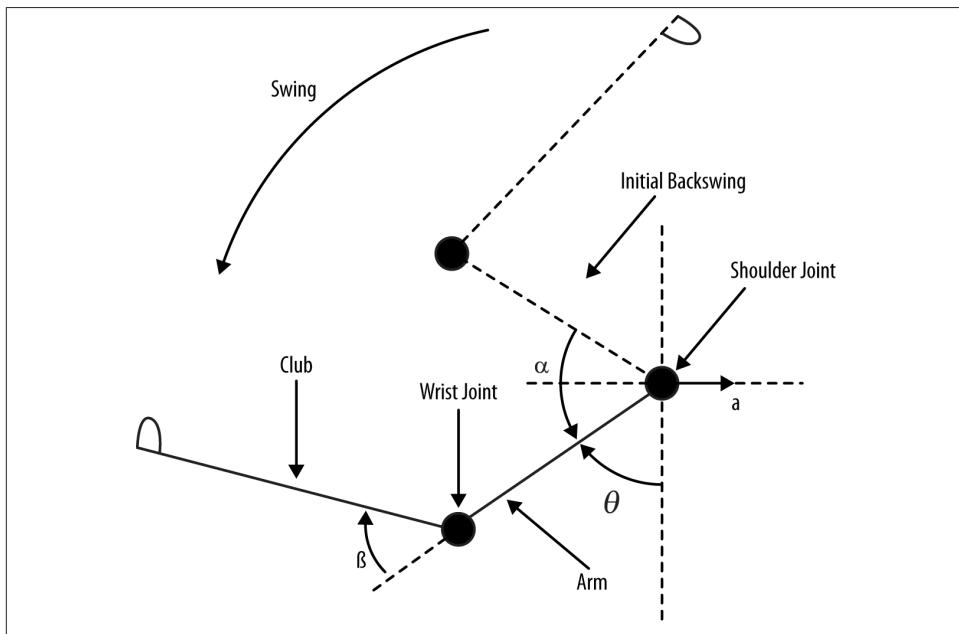


Figure 19-1. Two-rod model of golf swing

This model is essentially a double pendulum. More specifically, it is a driven double pendulum since the model assumes a torque applied at the shoulder end of the arm rod, and another torque applied at the wrist joint connecting the arm rod to the club rod.

We won't repeat Dr. Jorgensen's development of the model here; instead, we'll show you how to solve the resulting equations:

Those equations can be found in the Technical Appendix—Section 4 of Dr. Jorgensen's book—but are listed here for convenience.

Equation 1:

$$(J+I+M_c R^2 + 2RS \cdot \cos(\beta))\ddot{\alpha} - (I+RS \cdot \cos(\beta))\ddot{\beta} + \\ + (\dot{\beta}^2 - 2\dot{\alpha}\dot{\beta})RS \sin(\beta) - S[g \sin(\theta+\beta) - a \cdot \cos(\theta+\beta)] - \\ -(S_A + M_c \cdot R)(g \cdot \sin(\theta) - a \cdot \cos(\theta)) = Q_a$$

Equation 2:

$$I\ddot{\beta} - [I+RS \cdot \cos(\beta)]\ddot{a} + \dot{a}^2 RS \cdot \sin(\beta) + S[g \cdot \sin(\theta+\beta) - a \cdot \cos(\theta+\beta)] = Q_\beta$$

Table 19-3 explains what each symbol represents.

Table 19-3. Symbols used in golf swing model

Symbol	Meaning
J	Mass moment of inertia of the rod representing the arm. Units are kg·m ² .
I	Mass moment of inertia of the rod representing the club. Units are kg·m ² .
M _c	Mass of the club. Units are kg.
R	Length of the rod representing the arm. Units are m.
S	First moment of the rod representing the club about the wrist axis (where the club rod connects to the arm rod). Units are kg·m.
a	Angle swept by arm rod from initial backswing position. Units are radians.
β	Wrist-cock angle. Units are radians.
g	Acceleration due to gravity. Constant 9.8 m/s ² .
θ	Angle between arm rod and vertical axis. Units are radians.
a	Horizontal acceleration of the shoulder. Units are m/s ² .
S _A	First moment of the arm rod about the shoulder axis. Units are kg·m.
Q _a	Torque applied at the shoulder to the arm rod. Units are N·m.
Q _β	Torque applied at the wrist joint to the club rod. Units are N·m.

These equations represent a coupled system of nonlinear differential equations. They are coupled in that they both depend on the unknown quantities α and β . They are clearly differential equations, as they both include time derivatives of the unknown

quantities. And they are nonlinear because they include sines and cosines of one of the unknowns along with derivatives of the other unknown raised to some power greater than 1.

So, how do we solve these equations? Well, we can't do so in closed form and must resort to numerical means. There are a number of ways to proceed, but the approach we'll use is to first solve Equation 2 for $\ddot{\beta}$ and substitute the result into Equation 1. Then, we'll numerically integrate the result using a fourth-order Runge-Kutta scheme, as described in [Chapter 7](#).

More specifically, at each time step Equation 1 with $\ddot{\beta}$ replaced by the expression derived from Equation 1 will be solved for $\ddot{\alpha}$. Once $\ddot{\alpha}$ is found, we can find $\ddot{\beta}$ using the second equation previously solved for $\ddot{\beta}$. Next, we can integrate $\ddot{\alpha}$ and $\ddot{\beta}$ to find α and β . This process then repeats for each time step.

Again, this is only one method of solving these equations. Normally, when faced with a system of equations, practitioners use matrix schemes to solve the equations simultaneously. This is almost necessary for systems of equations that involve more than two equations. However, with just two equations, as we have here, we can avoid expensive matrix inversion computation by using the technique we just described.

Solving the Golf Swing Equations

Now we'll show you how to implement the solution we described in a simple console application. The example solves the two governing equations for α and β over time, the results of which can then be used to determine the club head velocity at any time instant using kinematic equations as described in [Chapter 2](#) (see the section “[Rigid-Body Kinematics](#)” on page 61). Alternatively, you can use the following equation, which Dr. Jorgensen gives for the club head velocity in his book:

$$V^2 = [R^2 + L^2 + 2RL \cdot \cos(\beta)]\ddot{\alpha}^2 + L^2\ddot{\beta}^2 - 2[L^2 + RL \cdot \cos(\beta)]\dot{\alpha}\dot{\beta}$$

We'll use Jorgensen's equation in this example.

Since the angles of interest are computed in units of radians, but we want to report them in units of degrees, we first create a few `defines` to make the conversions for us:

```
#define RADIANS(d) (d/180.0*3.14159)
#define DEGREES(r) (r*180.0/3.14159)
```

Next, we declare and initialize all of the variables. The initial values used here are some typical values that we assumed. You can change these values to simulate different swings:

```
// Variables
double alpha = 0.0;
double alpha_dot = 0.0;
double alpha_dotdot = 0.0;
double beta = RADIANS(120.0);
```

```

double beta_dot = 0.0;
double beta_dotdot = 0.0;

double J = 1.15; // kg m^2
double I = 0.08; // kg m^2
double Mc = 0.4; // kg
double R = 0.62; // m
double L = 1.1; // m
double S = 0.4*1.1*0.75; // kg m
double g = 9.8; // m/s^2
double gamma = RADIANS(135.0);
double theta = gamma - alpha;
double SA = 7.3*0.62*0.5; // kg m
double Qalpha = 100; // N m
double Qbeta = -10; // N m
double a = 0.1*g; // m/s^2
double dt = 0.0025; // s
double time = 0; // s
double Vc = 0;

```

Next we define two functions that we will use to compute the second time derivatives of α and β (i.e., $\ddot{\alpha}$ and $\ddot{\beta}$). These functions simply use Equations 1 and 2 solved for $\ddot{\alpha}$ and $\ddot{\beta}$, respectively.

`ComputeAlphaDotDot`, which solves for $\ddot{\alpha}$, is shown here:

```

double ComputeAlphaDotDot(void)
{
    double A, B, C, D, F, G;
    double num, denom;

    A = (J + I + Mc * R * R + 2 * R * S * cos(beta));
    B = -(I + R * S * cos(beta));
    F = Qalpha - (beta_dot * beta_dot - 2 * alpha_dot * beta_dot) * R * S *
        sin(beta) + S * (g * sin(theta + beta) - a * cos(theta + beta))
        + (SA + Mc * R) * (g * sin(theta) - a * cos(theta));

    C = B;
    D = I;
    G = Qbeta - alpha_dot * alpha_dot * R * S * sin(beta) -
        S * (g * sin(theta + beta) - a * cos(theta + beta));

    num = (F - (B * G / D));
    denom = (A - (B*C/D));
    return (F - (B * G / D)) / (A - (B*C/D));
}

```

The local variables A , B , C , D , F , and G are convenience variables used to organize the terms in Equation 1. This function returns the second derivative of α .

`ComputeBetaDotDot`, shown next, is very similar to `ComputeAlphaDotDot` but solves Equation 2 instead. This function returns the second derivative of β :

```

double ComputeBetaDotDot(void)
{
    double C, D, G;

    C = -(I + R * S * cos(beta));
    D = I;
    G = Qbeta - alpha_dot * alpha_dot * R * S * sin(beta) -
        S * (g * sin(theta + beta) - a * cos(theta + beta));

    return (G - C * alpha_dotdot) / D;
}

```

The solution to Equations 1 and 2 follows the Runge-Kutta scheme we showed you in [Chapter 7](#). Four intermediate steps are taken for each time step. The time step size is controlled by `dt`, which we've set to 0.0025s. If you simply used Euler's method, you'd have to reduce this step size quite a bit to obtain a stable solution. We implemented the solution in the main function of our console example. The code is as follows:

```

int _tmain(int argc, _TCHAR* argv[])
{
    double    a, at;
    double    b, bt;
    int       i;
    FILE*    fp;
    double    phi;
    double    Vc2;

    double   ak1, ak2, ak3, ak4;
    double   bk1, bk2, bk3, bk4;

    FILE*    fdebug;

    fp = fopen("results.txt", "w");
    fdebug = fopen("debug.txt", "w");

    for(i = 0; i<200; i++)
    {
        time += dt;

        if(time>=0.1)
        {
            Qbeta = 0;
        }

        // save results of previous time step
        a = alpha;
        b = beta;
        at = alpha_dot;
        bt = beta_dot;

        // integrate alpha'' and beta''

```

```

// The K1 Step:
alpha_dotdot = ComputeAlphaDotDot();
beta_dotdot = ComputeBetaDotDot();

ak1 = alpha_dotdot * dt;
bk1 = beta_dotdot * dt;

alpha_dot = at + ak1/2;
beta_dot = bt + bk1/2;

// The K2 Step:
alpha_dotdot = ComputeAlphaDotDot();
beta_dotdot = ComputeBetaDotDot();

ak2 = alpha_dotdot * dt;
bk2 = beta_dotdot * dt;

alpha_dot = at + ak2/2;
beta_dot = bt + bk2/2;

// The K3 Step:
alpha_dotdot = ComputeAlphaDotDot();
beta_dotdot = ComputeBetaDotDot();

ak3 = alpha_dotdot * dt;
bk3 = beta_dotdot * dt;

alpha_dot = at + ak3;
beta_dot = bt + bk3;

// The K3 Step:
alpha_dotdot = ComputeAlphaDotDot();
beta_dotdot = ComputeBetaDotDot();

ak4 = alpha_dotdot * dt;
bk4 = beta_dotdot * dt;

alpha_dot = at + (ak1 + 2*ak2 + 2*ak3 + ak4) / 6;
beta_dot = bt + (bk1 + 2*bk2 + 2*bk3 + bk4) / 6;

alpha = a + alpha_dot * dt;
beta = b + beta_dot * dt;

theta = gamma - alpha;
Vc2 = (R*R + L*L + 2 * R * L * cos(beta)) * ( alpha_dot * alpha_dot)
      + L*L * beta_dot * beta_dot
      - 2 * (L*L + R * L * cos(beta)) * alpha_dot * beta_dot;

Vc = sqrt(Vc2);

phi = theta + beta;
fprintf(fp, "%f, %f, %f, %f, %f\n", time, DEGREES(theta),

```

```

        DEGREES(alpha), DEGREES(beta), DEGREES(phi), Vc);

    fprintf(fdebug, "%f, %f, %f, %f, %f, %f\n", time, DEGREES(alpha),
            alpha_dot, alpha_dotdot, DEGREES(beta), beta_dot, beta_dotdot);
}

fclose(fp);
fclose(fdebug);
return 0;
}

```

Local variables `a`, `at`, `b`, and `bt` are used to temporarily store the previous time step's results for α and β and their first derivatives. `i` is a counter variable. `fp` is a file pointer that we'll use to write results out to a text file. `phi` is used to store the sum of $\theta + \beta$. And `Vc2` is the square of the club head velocity calculated according to Jorgensen's equation. The variables `ak1` through `ak4`, and `bk1` through `bk4`, are used to store intermediate results of the Runge-Kutta integration scheme. `fdebug` is a file pointer to a file we used for writing debugging information.

After the output and debug files are opened, the function enters a loop to perform the integration over 200 time steps. You can change the number of time steps as you see fit for your application. Keep in mind that the swing event, from start to striking the ball, takes place over a very short period of time—only fractions of a second long.

Upon entering the loop, you'll see some code that checks how much time has elapsed; if that time is greater than 0.1s, the wrist torque, `Qbeta`, is set to 0. This is a crude model of how the wrist torque that's initially applied is released, allowing the club to swing past the arm. Depending on the swing you're modeling, this torque could actually reverse direction, forcing the club past the arm even more. Dr. Jorgensen's book explains all this in detail, even giving experimental results.

Next, results of the previous time step are saved in the variables `a`, `b`, `at`, and `bt`. The first time step simply stores the initial values. Now, the integration starts for the first step, `k1` (see [Chapter 7](#)). Each of these steps involves computing $\ddot{\alpha}$ and $\ddot{\beta}$ using the functions `ComputeAlphaDotDot` and `ComputeBetaDotDot`. The `k1` results are then calculated and used to compute intermediate results for the first time derivatives of α and β . All four intermediate steps are carried out in a similar manner.

Finally, the current time step's results for `alpha_dot` and `beta_dot`, along with `alpha` and `beta`, are computed. Also, the square of the club head velocity, `Vc2`, is computed using Jorgensen's equation shown earlier; and the club head velocity, `Vc`, results from the square root of `Vc2`.

Results of interest are then written to the output and debug files. So, that's pretty much it. After the loop finishes, the files are closed and the application terminates.

If this were an actual game, you would use the club head velocity results along with the collision response method we showed in [Chapter 3](#) to determine the golf ball's trajectory. You could model the flight path of the golf ball using the methods we showed you in [Chapter 6](#).

Billiards

Now let's take a look at a different example. You may not think of billiards as a sport, but it is recognized internationally as a cue sport. Cue sports are a family of sports that include billiards, pool, snooker, and other related variations. For simplicity, we'll stick with the term *billiards*, although the topics presented apply to all cue sports.

Billiards is a good example of an activity that takes place over a limited physical space. Thus, when writing a billiards video game you need only concern yourself with a very finite space composed of well-established geometry. Billiard tables are typically 1.37 m × 2.74 m (4.5 ft × 9 ft), with some longer and some smaller depending on the game, style, and space available. Tables are typically cloth-covered slate. Balls vary in size between games and regions, with American-style pool balls measuring about 57 mm (2.25 inches) in diameter. Balls used to be made of wood, clay, or ivory, but nowadays they are plastic.

All these characteristics are important little details that you must consider if you're going to make a realistic billiards video game. The slate table and hard plastic balls have certain impact characteristics. The cloth-covered table provides some resistance to rolling. Side bumpers are not as hard as the slate table, thus yielding different impact characteristics. Fortunately, data on billiard tables and balls is readily available on the Web. And simulating billiards in a video game is fairly straightforward.

Billiards makes an interesting example because collisions are the heart of the game, and such an example also gives us an opportunity to demonstrate rolling contact. [Figure 19-2](#) and [Figure 19-3](#) illustrate the example we'll focus on. We have three object balls (the ones that get struck with the cue ball) set up in the middle of the table in a loose triangle configuration.

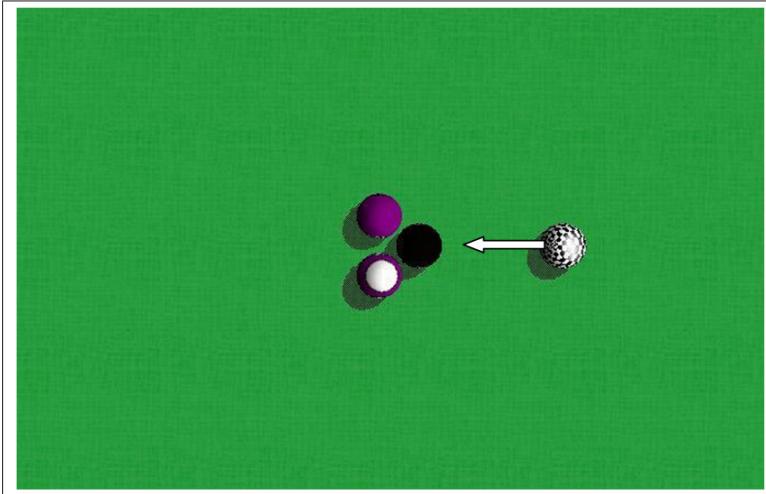


Figure 19-2. Elapsed time = 0.398s

The cue ball comes from the right at a set speed (see Figure 19-2) and then impacts the eight ball (see Figure 19-3).

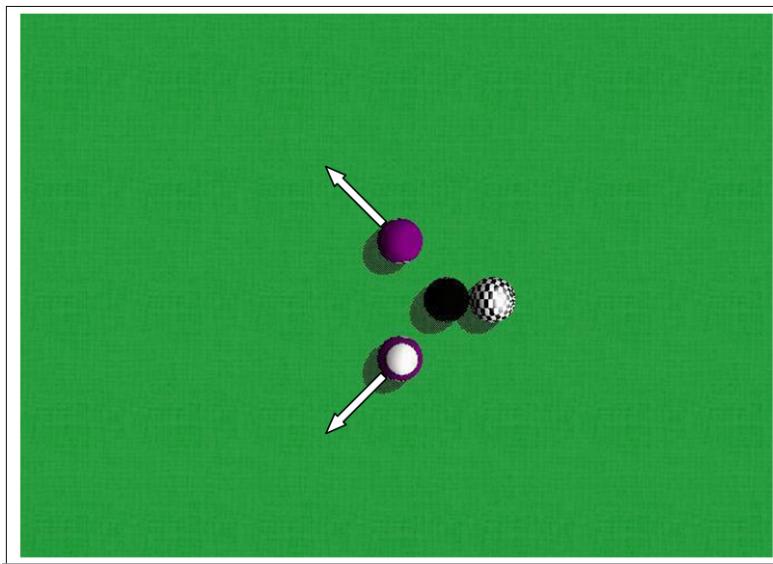


Figure 19-3. Elapsed time = 0.440s

After the initial impact between the cue ball and the eight ball, the eight ball moves to the left and impacts two more balls. These balls then shoot off diagonally. Most of the energy from the eight ball is transferred to the two other balls, so the eight ball quickly comes to rest while being kissed by the cue ball. The other two balls continue rolling away diagonally (see [Figure 19-4](#)).

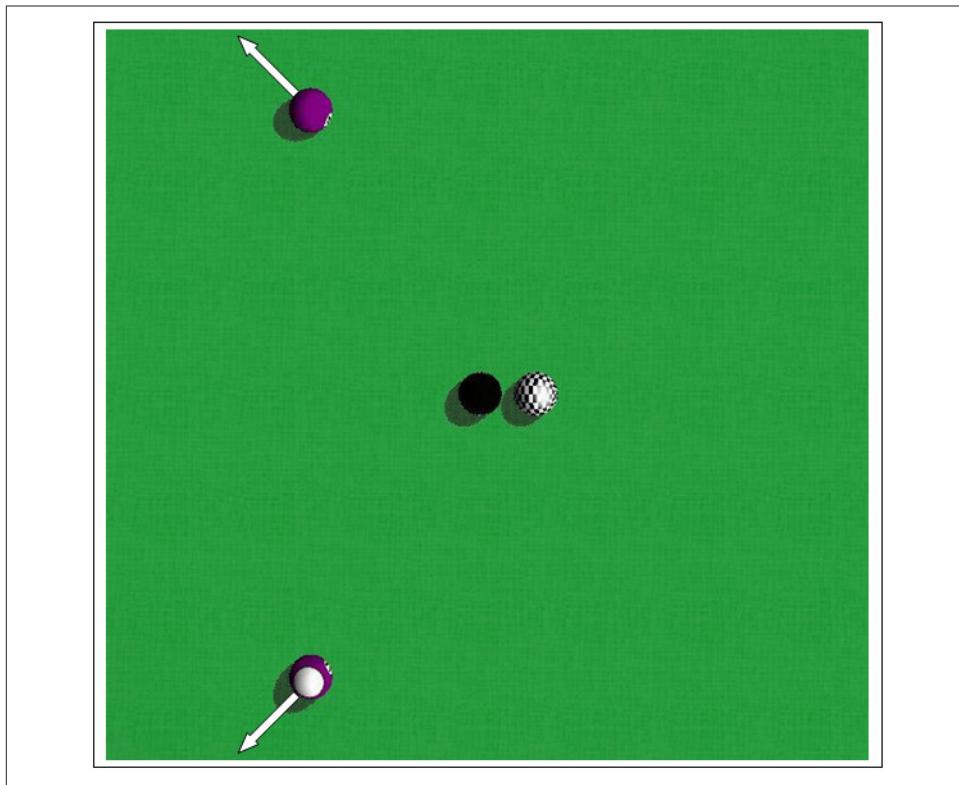


Figure 19-4. Elapsed time = 0.566s

In this example, we'll show you how to handle ball-ball collisions, ball-table collisions, ball-table contact, aerodynamic drag on the ball, rolling resistance, friction between balls at the time of impact, and friction between the balls and table.

Implementation

If you've read and studied the examples presented in [Chapter 7](#) through [Chapter 13](#), then the implementation of this billiards example will be very familiar to you; we use the same basic approach. During each simulation time step, we calculate all the forces acting on each ball; integrate the equations of motion, updating each ball's position and velocity; and then check for and deal with collisions.

The rigid-body class used in this example is very similar to that used for the airplane example in [Chapter 15](#). Even though the balls are compact and round, and it's tempting to treat them as particles, you must treat them as 3D rigid bodies in order to capture rolling and spinning, which are important elements of billiard ball dynamics. The rigid-body class adopted for this billiards example is as follows.

```
typedef struct _RigidBody {
    float      fMass;          // Total mass (constant)
    Matrix3x3  mInertia;       // Mass moment of inertia in body coordinates
    Matrix3x3  mInertiaInverse; // Inverse of mass moment of inertia matrix
    Vector     vPosition;      // Position in earth coordinates
    Vector     vVelocity;      // Velocity in earth coordinates
    Vector     vVelocityBody;   // Velocity in body coordinates
    Vector     vAcceleration;  // Acceleration of cg in earth space
    Vector     vAngularAcceleration; // Angular acceleration in body coordinates
    Vector     vAngularAccelerationGlobal; // Angular acceleration
                                         // in global coordinates
    Vector     vAngularVelocity; // Angular velocity in body coordinates
    Vector     vAngularVelocityGlobal; // Angular velocity in global coordinates
    Vector     vEulerAngles;     // Euler angles in body coordinates
    float      fSpeed;         // Speed (magnitude of the velocity)
    Quaternion qOrientation;   // Orientation in earth coordinates
    Vector     vForces;         // Total force on body
    Vector     vMoments;        // Total moment (torque) on body
    Matrix3x3  mIeInverse;     // Inverse of moment of inertia in earth coordinates
    float      fRadius;        // Ball radius
} RigidBody, *pRigidBody;
```

As you can see, this class looks very similar to the rigid-body classes we've used throughout this book, and in particular that used in the airplane example. All the usual suspects are here, and the comments in this code sample state what each class member represents. One particular property you have not seen yet is `fRadius`—this is simply the billiard ball's radius, which is used when we are checking for collisions and calculating drag forces.

As we discussed in [Chapter 14](#), since there are multiple objects in this simulation that may collide, we're going to iterate through all the objects, checking for collisions while storing the collision data. Since there are not that many objects in this simulation, we don't really need to partition the game space in order to optimize the collision detection checks (refer to [Chapter 14](#)). The data we need to store for each collision is included in the following `Collision` structure:

```
typedef struct _Collision {
    int      body1;
    int      body2;
    Vector   vCollisionNormal;
    Vector   vCollisionPoint;
    Vector   vRelativeVelocity;
    Vector   vRelativeAcceleration;
```

```

        Vector vCollisionTangent;
    } Collision, *pCollision;
}

```

The first two properties are indices to the two bodies involved in the collision. The next property, `vCollisionNormal`, stores the normal vector at the point of contact of the collision with the vector pointing outward from `body2`. The next property, `vCollisionPoint`, stores the coordinates of the point of contact in global coordinates. Since we're dealing with spheres (billiard balls), the collision manifold will always consist of a single point for each ball-ball or ball-table collision. The next two properties store the relative velocity and acceleration between the two bodies at the point of collision. The data is stored in `vRelativeVelocity` and `vRelativeAcceleration`, respectively. To capture friction at the point of contact, we need to know the tangent vector to the bodies at the point of contact. This tangent is stored in `vCollisionTangent`.

We set up several global `defines` to hold key data, allowing us to easily tune the simulation:

```

#define BALLDIAMETER      0.05715f
#define BALLWEIGHT         1.612f
#define GRAVITY            -9.87f
#define LINEARDRAGCOEFFICIENT 0.5f
#define ANGULARDRAGCOEFFICIENT 0.05f
#define FRICTIONFACTOR     0.5f
#define COEFFICIENTOFRESTITUTION 0.8f
#define COEFFICIENTOFRESTITUTIONGROUND 0.1f
#define FRICTIONCOEFFICIENTBALLS 0.1f
#define FRICTIONCOEFFICIENTGROUND 0.1f
#define ROLLINGRESISTANCECOEFFICIENT 0.025f

```

The first three `defines` represent the billiard ball diameter in meters, the ball weight in newtons, and the acceleration due to gravity in m/s^2 . The ball diameter and weight are typical values for American-style billiard balls (i.e., 2.25 inches and 5.8 oz on average).

The remaining `defines` are self-explanatory and represent nondimensional coefficients such as drag coefficients and coefficients of restitution. The values you see are what we came up with after tuning the simulation. You'll surely tune these yourself if you develop your own billiards game.

We use three important global variables for this simulation, as shown here:

```

RigidBody      Bodies[NUMBODIES];
Collision       Collisions[NUMBODIES*8];
int             NumCollisions = 0;

```

`Bodies` is an array of `RigidBody` types and represents the collection of the billiard balls. Here we've defined `NUMBODIES` as 4, so there are four billiard balls in this simulation. We've adopted the convention that the cue ball will always be `Bodies[0]`.

Initialization

At the beginning of the simulation, we have to initialize all four billiard balls. We use one function, `InitializeObjects`, for this task. It's a long function, but it's really simple. The code is shown on this and the following pages. `Bodies[0]` is the cue ball, and it is positioned 50 ball diameters along the negative x-axis away from the object balls. There's no magic to this number; we picked it arbitrarily. Now, we did deliberately set the z-position of the cue ball (all the balls, for that matter) to one-half the diameter so that the balls would be just touching the table at the start of the simulation.

To have the cue ball roll from right to left, we gave it an initial velocity of 7 m/s along the positive x-axis. With this initial velocity, the cue ball will begin sliding across the table for some short distance as it also starts to roll due to the friction between the ball and table. You can see in the upcoming code sample that all the other kinematic properties are set to 0 for the cue ball. For the object balls, all of their kinematic properties are set to 0.

We encourage you to experiment with different initial values for the cue ball's kinematic properties. For example, try setting the angular velocity about any of the coordinate axes to something other than 0. Doing so will allow you to see how a spinning ball may move slightly left or right depending on the spin. It's also fun to see how spin affects the object balls upon collision with the cue ball.

Aside from setting the positions and kinematic properties of the balls, `InitializeObjects` also initializes mass properties. We used the previously defined `BALLWEIGHT` divided by the acceleration due to gravity to determine the ball mass. For mass moment of inertia, we simply used the equations for a solid sphere we showed you way back in [Chapter 1](#):

```
void InitializeObjects(int configuration)
{
    float      iRoll, iPitch, iYaw;
    int       i;
    float      Ixx, Iyy, Izz;
    float      s;

    /////////////////////////////////
    // Initialize the cue ball:
    // Set initial position
    Bodies[0].vPosition.x = -BALLDIAMETER*50.0f;
    Bodies[0].vPosition.y = 0.0f;
    Bodies[0].vPosition.z = BALLDIAMETER/2.0f;

    // Set initial velocity
    s = 7.0;
    Bodies[0].vVelocity.x = s;
    Bodies[0].vVelocity.y = 0.0f;
    Bodies[0].vVelocity.z = 0.0f;
    Bodies[0].fSpeed = s;
```

```

// Set initial angular velocity
Bodies[0].vAngularVelocity.x = 0.0f; // rotate about long'l axis
Bodies[0].vAngularVelocity.y = 0.0f; // rotate about transverse axis
Bodies[0].vAngularVelocity.z = 0.0f; // rotate about vertical axis

Bodies[0].vAngularAcceleration.x = 0.0f;
Bodies[0].vAngularAcceleration.y = 0.0f;
Bodies[0].vAngularAcceleration.z = 0.0f;

Bodies[0].vAcceleration.x = 0.0f;
Bodies[0].vAcceleration.y = 0.0f;
Bodies[0].vAcceleration.z = 0.0f;

// Set the initial forces and moments
Bodies[0].vForces.x = 0.0f;
Bodies[0].vForces.y = 0.0f;
Bodies[0].vForces.z = 0.0f;

Bodies[0].vMoments.x = 0.0f;
Bodies[0].vMoments.y = 0.0f;
Bodies[0].vMoments.z = 0.0f;

// Zero the velocity in body space coordinates
Bodies[0].vVelocityBody.x = 0.0f;
Bodies[0].vVelocityBody.y = 0.0f;
Bodies[0].vVelocityBody.z = 0.0f;

// Set the initial orientation
iRoll = 0.0f;
iPitch = 0.0f;
iYaw = 0.0f;
Bodies[0].qOrientation = MakeQFromEulerAngles(iRoll, iPitch, iYaw);

// Set the mass properties
Bodies[0].fMass = BALLWEIGHT/(-g);

Ix = 2.0f * Bodies[0].fMass / 5.0f * (BALLDIAMETER/2*BALLDIAMETER/2);
Iz = Iy = Ix;

Bodies[0].mInertia.e11 = Ix;
Bodies[0].mInertia.e12 = 0;
Bodies[0].mInertia.e13 = 0;
Bodies[0].mInertia.e21 = 0;
Bodies[0].mInertia.e22 = Iy;
Bodies[0].mInertia.e23 = 0;
Bodies[0].mInertia.e31 = 0;
Bodies[0].mInertia.e32 = 0;
Bodies[0].mInertia.e33 = Iz;

Bodies[0].mInertiaInverse = Bodies[0].mInertia.Inverse();

```

```

Bodies[0].fRadius = BALLDIAMETER/2;

///////////////////////////////
// Initialize the other balls
for(i=1; i<NUMBODIES; i++)
{
    // Set initial position
    if(i==1)
    {
        Bodies[i].vPosition.x = 0.0;
        Bodies[i].vPosition.y = -(BALLDIAMETER/2.0f+0.25*BALLDIAMETER);
        Bodies[i].vPosition.z = BALLDIAMETER/2.0f;
    } else if(i==2) {
        Bodies[i].vPosition.x = 0.0;
        Bodies[i].vPosition.y = BALLDIAMETER/2.0f+0.25*BALLDIAMETER;
        Bodies[i].vPosition.z = BALLDIAMETER/2.0f;
    } else {
        Bodies[i].vPosition.x = -BALLDIAMETER;
        Bodies[i].vPosition.y = 0.0f;
        Bodies[i].vPosition.z = BALLDIAMETER/2.0f;
    }

    // Set initial velocity
    Bodies[i].vVelocity.x = 0.0f;
    Bodies[i].vVelocity.y = 0.0f;
    Bodies[i].vVelocity.z = 0.0f;
    Bodies[i].fSpeed = 0.0f;

    // Set initial angular velocity
    Bodies[i].vAngularVelocity.x = 0.0f;
    Bodies[i].vAngularVelocity.y = 0.0f;
    Bodies[i].vAngularVelocity.z = 0.0f;

    Bodies[i].vAngularAcceleration.x = 0.0f;
    Bodies[i].vAngularAcceleration.y = 0.0f;
    Bodies[i].vAngularAcceleration.z = 0.0f;

    Bodies[i].vAcceleration.x = 0.0f;
    Bodies[i].vAcceleration.y = 0.0f;
    Bodies[i].vAcceleration.z = 0.0f;

    // Set the initial forces and moments
    Bodies[i].vForces.x = 0.0f;
    Bodies[i].vForces.y = 0.0f;
    Bodies[i].vForces.z = 0.0f;

    Bodies[i].vMoments.x = 0.0f;
    Bodies[i].vMoments.y = 0.0f;
    Bodies[i].vMoments.z = 0.0f;

    // Zero the velocity in body space coordinates
}

```

```

        Bodies[i].vVelocityBody.x = 0.0f;
        Bodies[i].vVelocityBody.y = 0.0f;
        Bodies[i].vVelocityBody.z = 0.0f;

        // Set the initial orientation
        iRoll = 0.0f;
        iPitch = 0.0f;
        iYaw = 0.0f;
        Bodies[i].qOrientation = MakeQFromEulerAngles(iRoll, iPitch, iYaw);

        // Set the mass properties
        Bodies[i].fMass = BALLWEIGHT/(-g);
        Ixx = 2.0f * Bodies[i].fMass / 5.0f * (BALLDIAMETER*BALLDIAMETER);
        Izz = Iyy = Ixx;

        Bodies[i].mInertia.e11 = Ixx;
        Bodies[i].mInertia.e12 = 0;
        Bodies[i].mInertia.e13 = 0;
        Bodies[i].mInertia.e21 = 0;
        Bodies[i].mInertia.e22 = Iyy;
        Bodies[i].mInertia.e23 = 0;
        Bodies[i].mInertia.e31 = 0;
        Bodies[i].mInertia.e32 = 0;
        Bodies[i].mInertia.e33 = Izz;

        Bodies[i].mInertiaInverse = Bodies[i].mInertia.Inverse();
        Bodies[i].fRadius = BALLDIAMETER/2;
    }
}

```

Stepping the Simulation

During each time step, the simulation's main loop makes a call to `StepSimulation`. This function, shown next, is almost identical to the `StepSimulation` functions we covered in the other examples shown throughout this book, so there really are no surprises here. `StepSimulation` first makes a call to `CalcObjectForces`, which we'll discuss momentarily, and then proceeds to integrate the equations of motion for each ball. We use a basic Euler scheme here for simplicity. After integrating, `StepSimulation` makes a few function calls to deal with collisions. We'll cover those shortly.

```

void StepSimulation(float dtime)
{
    Vector     Ae;
    int         i;
    float      dt = dtime;
    int         check = NOCOLLISION;
    int         c = 0;

    // Calculate all of the forces and moments on the balls:
    CalcObjectForces();

```

```

// Integrate the equations of motion:
for(i=0; i<NUMBODIES; i++)
{
    // Calculate the acceleration in earth space:
    Ae = Bodies[i].vForces / Bodies[i].fMass;
    Bodies[i].vAcceleration = Ae;

    // Calculate the velocity in earth space:
    Bodies[i].vVelocity += Ae * dt;

    // Calculate the position in earth space:
    Bodies[i].vPosition += Bodies[i].vVelocity * dt;

    // Now handle the rotations:
    float      mag;

    Bodies[i].vAngularAcceleration = Bodies[i].mInertiaInverse *
        (Bodies[i].vMoments -
         (Bodies[i].vAngularVelocity^
          (Bodies[i].mInertia *
           Bodies[i].vAngularVelocity)));

    Bodies[i].vAngularVelocity += Bodies[i].vAngularAcceleration * dt;

    // Calculate the new rotation quaternion:
    Bodies[i].qOrientation += (Bodies[i].qOrientation *
                                Bodies[i].vAngularVelocity) *
                                (0.5f * dt);

    // Now normalize the orientation quaternion:
    mag = Bodies[i].qOrientation.Magnitude();
    if (mag != 0)
        Bodies[i].qOrientation /= mag;

    // Calculate the velocity in body space:
    Bodies[i].vVelocityBody = QVRotate(~Bodies[i].qOrientation,
                                         Bodies[i].vVelocity);

    // Get the angular velocity in global coords:
    Bodies[i].vAngularVelocityGlobal = QVRotate(Bodies[i].qOrientation,
                                                 Bodies[i].vAngularVelocity);

    // Get the angular acceleration in global coords:
    Bodies[i].vAngularAccelerationGlobal = QVRotate(Bodies[i].qOrientation,
                                                    Bodies[i].vAngularAcceleration);

    // Get the inverse inertia tensor in global coordinates
    Matrix3x3 R, RT;
    R = MakeMatrixFromQuaternion(Bodies[i].qOrientation);
    RT = R.Transpose();
    Bodies[i].mIeInverse = R * Bodies[i].mInertiaInverse * RT;
}

```

```

// Calculate the air speed:
Bodies[i].fSpeed = Bodies[i].vVelocity.Magnitude();

// Get the Euler angles for our information
Vector u;

u = MakeEulerAnglesFromQ(Bodies[i].qOrientation);
Bodies[i].vEulerAngles.x = u.x;      // roll
Bodies[i].vEulerAngles.y = u.y;      // pitch
Bodies[i].vEulerAngles.z = u.z;      // yaw
}

// Handle Collisions    :
check = CheckForCollisions();
if(check == COLLISION)
    ResolveCollisions();
}

```

Calculating Forces

The first function call made by `StepSimulation` is a call to `CalcObjectForces`, which is responsible for computing all the forces on each ball except collision forces. This is the same approach used in previous examples. The entire `CalcObjectForces` source code is included here:

```

void CalcObjectForces(void)
{
    Vector    Fb, Mb;
    Vector    vDragVector;
    Vector    vAngularDragVector;
    int       i, j;
    Vector    ContactForce;
    Vector    pt;
    int       check = NOCOLLISION;
    pCollision   pCollisionData;
    Vector    FrictionForce;
    Vector    fDir;
    double    speed;
    Vector    FRn, FRt;

    for(i=0; i<NUMBODIES; i++)
    {
        // Reset forces and moments:
        Bodies[i].vForces.x = 0.0f;
        Bodies[i].vForces.y = 0.0f;
        Bodies[i].vForces.z = 0.0f;

        Bodies[i].vMoments.x = 0.0f;
        Bodies[i].vMoments.y = 0.0f;
        Bodies[i].vMoments.z = 0.0f;
    }
}

```

```

Fb.x = 0.0f;      Mb.x = 0.0f;
Fb.y = 0.0f;      Mb.y = 0.0f;
Fb.z = 0.0f;      Mb.z = 0.0f;

// Do drag force:
vDragVector = -Bodies[i].vVelocityBody;
vDragVector.Normalize();
speed = Bodies[i].vVelocityBody.Magnitude();
Fb += vDragVector * ((1.0f/2.0f)*speed * speed * rho *
LINEARDRAGCOEFFICIENT * pow(Bodies[i].fRadius,2) *
Bodies[i].fRadius*pi);

vAngularDragVector = -Bodies[i].vAngularVelocity;
vAngularDragVector.Normalize();
Mb += vAngularDragVector * (Bodies[i].vAngularVelocity.Magnitude() *
Bodies[i].vAngularVelocity.Magnitude() * rho * ANGULARDRAGCOEFFICIENT
* 4 * pow(Bodies[i].fRadius,2)*pi);

// Convert forces from model space to earth space:
Bodies[i].vForces = QVRotate(Bodies[i].qOrientation, Fb);

// Apply gravity:
Bodies[i].vForces.z += GRAVITY * Bodies[i].fMass;

// Save the moments:
Bodies[i].vMoments += Mb;

// Handle contacts with ground plane:
Bodies[i].vAcceleration = Bodies[i].vForces / Bodies[i].fMass;
Bodies[i].vAngularAcceleration = Bodies[i].mInertiaInverse *
(Bodies[i].vMoments -
(Bodies[i].vAngularVelocity^
(Bodies[i].mInertia *
Bodies[i].vAngularVelocity))));

// Resolve ground plane contacts:
FlushCollisionData();
pCollisionData = Collisions;
NumCollisions = 0;
if(DOCONTACT)
    check = CheckGroundPlaneContacts(pCollisionData, i);
if((check == CONTACT) && DOCONTACT)
{
    j = 0;
    {
        assert(NumCollisions <= 1);

        ContactForce = (Bodies[i].fMass * (-Bodies[i].vAcceleration *
Collisions[j].vCollisionNormal)) *
Collisions[j].vCollisionNormal;

        if(DOFRICTION)

```

```

    {
        double vt = fabs(Collisions[j].vRelativeVelocity *
                          Collisions[j].vCollisionTangent);
        if(vt > VELOCITYTOLERANCE)
        {
            // Kinetic:
            FrictionForce = (ContactForce.Magnitude() *
                              FRICTIONCOEFFICIENTGROUND) *
                              Collisions[j].vCollisionTangent;
        } else {
            // Static:
            FrictionForce = (ContactForce.Magnitude() *
                              FRICTIONCOEFFICIENTGROUND * 2 *
                              vt/VELOCITYTOLERANCE) *
                              Collisions[j].vCollisionTangent;
        }
    } else
        FrictionForce.x = FrictionForce.y = FrictionForce.z = 0;

    // Do rolling resistance:
    if(Bodies[i].vAngularVelocity.Magnitude() > VELOCITYTOLERANCE)
    {
        FRn = ContactForce.Magnitude() *
              Collisions[j].vCollisionNormal;
        Collisions[j].vCollisionTangent.Normalize();
        Vector m = (Collisions[j].vCollisionTangent
                    *(ROLLINGRESISTANCECOEFFICIENT *
                     Bodies[i].fRadius))^(FRn);
        double mag = m.Magnitude();
        Vector a = Bodies[i].vAngularVelocity;
        a.Normalize();
        Bodies[i].vMoments += -a * mag;
    }

    // accumulate contact and friction forces and moments
    Bodies[i].vForces += ContactForce;
    Bodies[i].vForces += FrictionForce;

    ContactForce = QVRotate(~Bodies[i].qOrientation, ContactForce);
    FrictionForce = QVRotate(~Bodies[i].qOrientation,
                            FrictionForce);
    pt = Collisions[j].vCollisionPoint - Bodies[i].vPosition;
    pt = QVRotate(~Bodies[i].qOrientation, pt);
    Bodies[i].vMoments += pt^ContactForce;
    Bodies[i].vMoments += pt^FrictionForce;
}

}
}
}

```

As you can see, upon entering `CalcObjectForces` the code enters a loop that cycles through all the billiard ball objects, computing the forces acting on each. The first force computed is simple aerodynamic drag. Both linear and angular drag are computed. We compute the magnitude of the linear drag by multiplying the linear drag coefficient by $1/2\rho V^2 r 2\pi r$, where ρ is the density of air, V is the ball's linear speed, and r is the ball's radius. We compute the magnitude of the angular drag moment by multiplying the angular drag coefficient by $\omega \rho 4r^2 \pi$, where ω is angular speed. Since drag retards motion, the linear drag and angular drag vectors are simply the opposite of the linear and angular velocity vectors, respectively. Normalizing those vectors and then multiplying by the respective drag magnitudes yields the linear and angular drag force and moment vectors.

The next set of forces calculated in `CalcObjectForces` is the contact forces between the table top and each ball. There are three contact forces. One is the vertical force that keeps the balls from falling through the table, another is the friction force that arises as the balls slide along the table, and the third is rolling resistance. These forces arise only if the ball is in contact with the table. We'll address how to determine whether a ball is in contact with the table later in this chapter. For now, we'll assume there's contact and show you how to compute the contact forces.

To compute the vertical force required to keep the ball from falling through the table, we must first compute the ball's linear acceleration, which is equal to the sum of forces (excluding contact forces) acting on the ball divided by the ball's mass. Next, we take the negative dot product of that acceleration and the vector perpendicular to the table surface and multiply the result by the ball's mass. This yields the magnitude of the contact force, and to get the vector we multiply that magnitude by the unit vector perpendicular to the table's surface. The following two lines of code perform these calculations:

```
Bodies[i].vAcceleration = Bodies[i].vForces / Bodies[i].fMass;

ContactForce = (Bodies[i].fMass * (-Bodies[i].vAcceleration *
    Collisions[j].vCollisionNormal)) *
    Collisions[j].vCollisionNormal;
```

The `vCollisionNormal` vector is determined by `CheckGroundPlaneContacts`, which we'll cover later. As with collisions, `CheckGroundPlaneContacts` fills in a data structure containing the point of contact, relative velocity between the ball and table at the point of contact, and the contact normal and tangent vectors, among other data.

To compute the sliding friction force, we must first determine the tangential component of the relative velocity between the ball and table. If the ball is sliding or slipping as it rolls, then the relative tangential velocity will be greater than 0. If the ball is rolling without sliding, then the relative velocity will be 0. In either case, there will be a friction force; in the former case, we'll use the kinetic friction coefficient, and in the latter we'll use the static friction coefficient. Friction force is computed in the same way we showed you in [Chapter 3](#). The following lines of code perform all these calculations:

```

ContactForce = (Bodies[i].fMass * (-Bodies[i].vAcceleration *
    Collisions[j].vCollisionNormal)) *
    Collisions[j].vCollisionNormal;

double vt = fabs(Collisions[j].vRelativeVelocity *
    Collisions[j].vCollisionTangent);
if(vt > VELOCITYTOLERANCE)
{
    // Kinetic:
    FrictionForce = (ContactForce.Magnitude() *
        FRICTIONCOEFFICIENTGROUND) *
        Collisions[j].vCollisionTangent;
} else {
    // Static:
    FrictionForce = (ContactForce.Magnitude() *
        FRICTIONCOEFFICIENTGROUND * 2 * 
        vt/VELOCITYTOLERANCE) *
        Collisions[j].vCollisionTangent;
}

```

Keep in mind that these forces will create moments if they do not act through the ball's center of gravity. So, after computing and aggregating these forces, you must also resolve any moments created and aggregate those using the same formulas we've shown through this book. The following lines of code take care of these tasks:

```

// accumulate contact and friction forces and moments
Bodies[i].vForces += ContactForce;
Bodies[i].vForces += FrictionForce;

ContactForce = QVRotate(~Bodies[i].qOrientation, ContactForce);
FrictionForce = QVRotate(~Bodies[i].qOrientation,
    FrictionForce);
pt = Collisions[j].vCollisionPoint - Bodies[i].vPosition;
pt = QVRotate(~Bodies[i].qOrientation, pt);
Bodies[i].vMoments += pt^ContactForce;
Bodies[i].vMoments += pt^FrictionForce;

```

Rolling resistance arises by virtue of small deformations in the cloth-covered table creating a little divot that the ball must overcome as it rolls. This divot shifts the center of application of the contact force just a little bit in the direction of rolling. That small offset results in a moment when multiplied by the contact force. The resulting moment opposes rolling; otherwise, without some other resistance the ball would continue rolling unrealistically. The following code computes the rolling resistance:

```

// Do rolling resistance:
if(Bodies[i].vAngularVelocity.Magnitude() > VELOCITYTOLERANCE)
{
    FRn = ContactForce.Magnitude() *
        Collisions[j].vCollisionNormal;
    Collisions[j].vCollisionTangent.Normalize();
    Vector m = (Collisions[j].vCollisionTangent *
        *(ROLLINGRESISTANCECOEFFICIENT *

```

```

        Bodies[i].fRadius))^^FRn;
    double mag = m.Magnitude();
    Vector a = Bodies[i].vAngularVelocity;
    a.Normalize();
    Bodies[i].vMoments += -a * mag;
}

```

Handling Collisions

Earlier you saw where `StepSimulation` makes a few function calls to deal with collision checking and response. You also saw where `CalcObjectForces` makes a function call that checks for contacts. The functions that check for collisions or contacts make use of the `Collisions` array we showed you earlier. This array stores all the relevant information pertaining to collisions or contacts—the collision or contact manifold, normal and tangent vectors, relative velocity, etc.

The first function we'll consider is `CheckForCollisions`, which is called toward the end of `StepSimulation`. `CheckForCollisions` checks for ball-ball collisions; we have a separate function to check for ball-table collisions that we'll get to later. `CheckForCollisions` relies on concepts we've already discussed and showed you in earlier chapters, so we'll summarize its action here. Basically, two billiard balls are colliding if 1) they are headed toward each other, and 2) the distance separating their centers is less than or equal to the sum of their radii. If both of these criteria are met, then a collision is recorded and all relevant data is stored in the `Collisions` array:

```

int     CheckForCollisions(void)
{
    int      status = NOCOLLISION;
    int      i, j;
    Vector   d;
    pCollision pCollisionData;
    int      check = NOCOLLISION;
    float    r;
    float    s;
    Vector   tmp;

    FlushCollisionData();
    pCollisionData = Collisions;
    NumCollisions = 0;

    // check object collisions with each other
    for(i=0; i<NUMBODIES; i++)
    {
        for(j=0; j<NUMBODIES; j++)
            if((j!=i) && !CollisionRecordedAlready(i, j))
            {
                // do a bounding sphere check
                d = Bodies[i].vPosition - Bodies[j].vPosition;
                r = Bodies[i].fRadius + Bodies[j].fRadius;
                s = d.Magnitude() - r;

```

```

        if(s < COLLISIONTOLERANCE)
        {// possible collision
            Vector  pt1, pt2, vel1, vel2, n, Vr;
            float   Vrn;

            pt1 = (Bodies[i].vPosition + Bodies[j].vPosition)/2;
            tmp = pt2 = pt1;

            pt1 = pt1-Bodies[i].vPosition;
            pt2 = pt2-Bodies[j].vPosition;

            vel1 = Bodies[i].vVelocity +
                   (Bodies[i].vAngularVelocityGlobal^pt1);
            vel2 = Bodies[j].vVelocity +
                   (Bodies[j].vAngularVelocityGlobal^pt2);

            n = d;
            n.Normalize();

            Vr = (vel1 - vel2);
            Vrn = Vr * n;

            if(Vrn < -VELOCITYTOLERANCE)
            {
                // Have a collision so fill the data structure
                assert(NumCollisions < (NUMBODIES*8));
                if(NumCollisions < (NUMBODIES*8))
                {
                    pCollisionData->body1 = i;
                    pCollisionData->body2 = j;
                    pCollisionData->vCollisionNormal = n;
                    pCollisionData->vCollisionPoint = tmp;
                    pCollisionData->vRelativeVelocity = Vr;
                    pCollisionData->vCollisionTangent = (n^Vr)^n;
                    pCollisionData->vCollisionTangent.Normalize();

                    pCollisionData++;
                    NumCollisions++;
                    status = COLLISION;
                }
            }
        }

        for(i=0; i<NUMBODIES; i++)
        {
            check = NOCOLLISION;

            assert(NumCollisions < (NUMBODIES*8));
            check = CheckGroundPlaneCollisions(pCollisionData, i);
            if(check == COLLISION)
            {

```

```

        status = COLLISION;
        pCollisionData++;
        NumCollisions++;
    }
}

return status;
}

```

Since `CheckForCollisions` loops through all of the balls checking for collisions with every other ball, it is possible that a collision would be recorded twice. For example, the i th ball may be found to be colliding with the j th ball, and later the j th ball would also be found to be colliding with the i th ball. We don't want to record that information twice, so we use the following function to check if a collision between two particular balls is already recorded. If so, we skip re-recording the data:

```

bool CollisionRecordedAlready(int i, int j)
{
    int k;
    int b1, b2;

    for(k=0; k<NumCollisions; k++)
    {
        b1 = Collisions[k].body1;
        b2 = Collisions[k].body2;

        if( ((b1 == i) && (b2 == j)) ||
            ((b1 == j) && (b2 == i)) )
            return true;
    }

    return false;
}

```

Checking ball-table collisions is fairly straightforward as well. If 1) a ball is found to be headed toward the table with some velocity greater than 0 (or some small threshold), and 2) the ball's vertical position to its center is less than or equal to its radius, then we record a collision. `CheckGroundPlaneCollisions` handles this for us:

```

int     CheckGroundPlaneCollisions(pCollision CollisionData, int body1)
{
    Vector    tmp;
    Vector    vel1;
    Vector    pt1;
    Vector    Vr;
    float     Vrn;
    Vector    n;
    int       status = NOCOLLISION;

    if(Bodies[body1].vPosition.z <= (Bodies[body1].fRadius))
    {
        pt1 = Bodies[body1].vPosition;

```

```

    pt1.z = COLLISIONTOLERANCE;
    tmp = pt1;
    pt1 = pt1-Bodies[body1].vPosition;
    vel1 = Bodies[body1].vVelocity/*Body*/ +
        (Bodies[body1].vAngularVelocityGlobal^pt1);

    n.x = 0;
    n.y = 0;
    n.z = 1;

    Vr = vel1;
    Vrn = Vr * n;

    if(Vrn < -VELOCITYTOLERANCE)
    {
        // Have a collision so fill the data structure
        assert(NumCollisions < (NUMBODIES*8));
        if(NumCollisions < (NUMBODIES*8))
        {
            CollisionData->body1 = body1;
            CollisionData->body2 = -1;
            CollisionData->vCollisionNormal = n;
            CollisionData->vCollisionPoint = tmp;
            CollisionData->vRelativeVelocity = Vr;

            CollisionData->vCollisionTangent = (n^Vr)^n;
            CollisionData->vCollisionTangent.Reverse();

            CollisionData->vCollisionTangent.Normalize();
            status = COLLISION;
        }
    }
}

return status;
}

```

Resolving collisions, whether ball-ball or ball-table collisions, uses the same approach we've already shown you. Thus, we won't go over the code again, and will instead just show you the function that implements collision response:

```

void ResolveCollisions(void)
{
    int      i;
    double   j;
    Vector   pt1, pt2, vB1V, vB2V, vB1AV, vB2AV;
    float    fCr = COEFFICIENTOFRESTITUTION;
    int      b1, b2;
    float    Vrt;
    float    muB = FRICTIONCOEFFICIENTBALLS;
    float    muG = FRICTIONCOEFFICIENTGROUND;
    bool     dofriiction = DOFRICTION;

```

```

for(i=0; i<NumCollisions; i++)
{
    b1 = Collisions[i].body1;
    b2 = Collisions[i].body2;

    if( (b1 != -1) && (b1 != b2) )
    {
        if(b2 != -1) // not ground plane
        {
            pt1 = Collisions[i].vCollisionPoint - Bodies[b1].vPosition;
            pt2 = Collisions[i].vCollisionPoint - Bodies[b2].vPosition;

            // Calculate impulse:
            j = (-(1+fCr) * (Collisions[i].vRelativeVelocity *
                Collisions[i].vCollisionNormal)) /
                ((1/Bodies[b1].fMass + 1/Bodies[b2].fMass) +
                (Collisions[i].vCollisionNormal * ( (pt1 ^
                    Collisions[i].vCollisionNormal) *
                    Bodies[b1].mIeInverse )^pt1) ) +
                (Collisions[i].vCollisionNormal * ( (pt2 ^
                    Collisions[i].vCollisionNormal) *
                    Bodies[b2].mIeInverse )^pt2) );
        }

        Vrt = Collisions[i].vRelativeVelocity *
            Collisions[i].vCollisionTangent;

        if(fabs(Vrt) > 0.0 && dofriiction) {
            Bodies[b1].vVelocity +=
                ((j * Collisions[i].vCollisionNormal) +
                ((muB * j) * Collisions[i].vCollisionTangent)) /
                Bodies[b1].fMass;
            Bodies[b1].vAngularVelocityGlobal +=
                (pt1 ^ ((j * Collisions[i].vCollisionNormal) +
                ((muB * j) * Collisions[i].vCollisionTangent))) *
                Bodies[b1].mIeInverse;
            Bodies[b1].vAngularVelocity =
                QVRotate(~Bodies[b1].qOrientation,
                Bodies[b1].vAngularVelocityGlobal);

            Bodies[b2].vVelocity +=
                ((-j * Collisions[i].vCollisionNormal) + ((muB *
                j) * Collisions[i].vCollisionTangent)) /
                Bodies[b2].fMass;
            Bodies[b2].vAngularVelocityGlobal +=
                (pt2 ^ ((-j * Collisions[i].vCollisionNormal) +
                ((muB * j) * Collisions[i].vCollisionTangent))) *
                Bodies[b2].mIeInverse;

            Bodies[b2].vAngularVelocity =
                QVRotate(~Bodies[b2].qOrientation,
                Bodies[b2].vAngularVelocityGlobal);
        }
    }
}

```

```

} else {
    // Apply impulse:
    Bodies[b1].vVelocity +=
        (j * Collisions[i].vCollisionNormal) /
        Bodies[b1].fMass;
    Bodies[b1].vAngularVelocityGlobal +=
        (pt1 ^ (j * Collisions[i].vCollisionNormal)) *
        Bodies[b1].mIeInverse;
    Bodies[b1].vAngularVelocity =
        QVRotate(~Bodies[b1].qOrientation,
        Bodies[b1].vAngularVelocityGlobal);

    Bodies[b2].vVelocity -=
        (j * Collisions[i].vCollisionNormal) /
        Bodies[b2].fMass;
    Bodies[b2].vAngularVelocityGlobal -=
        (pt2 ^ (j * Collisions[i].vCollisionNormal)) *
        Bodies[b2].mIeInverse;
    Bodies[b2].vAngularVelocity =
        QVRotate(~Bodies[b2].qOrientation,
        Bodies[b2].vAngularVelocityGlobal);
}

} else { // Ground plane:
    fCr = COEFFICIENTOFRESTITUTIONGROUND;
    pt1 = Collisions[i].vCollisionPoint - Bodies[b1].vPosition;

    // Calculate impulse:
    j = (-(1+fCr) * (Collisions[i].vRelativeVelocity *
    Collisions[i].vCollisionNormal)) /
    ( (1/Bodies[b1].fMass) +
    (Collisions[i].vCollisionNormal *
    ( (pt1 ^ Collisions[i].vCollisionNormal) *
    Bodies[b1].mIeInverse )^pt1)));

    Vrt = Collisions[i].vRelativeVelocity *
    Collisions[i].vCollisionTangent;

    if(fabs(Vrt) > 0.0 && dofFriction) {
        Bodies[b1].vVelocity +=
            ( (j * Collisions[i].vCollisionNormal) + ((muG *
            j) * Collisions[i].vCollisionTangent) ) /
            Bodies[b1].fMass;
        Bodies[b1].vAngularVelocityGlobal +=
            (pt1 ^ ((j * Collisions[i].vCollisionNormal) +
            ((muG * j) * Collisions[i].vCollisionTangent))) *
            Bodies[b1].mIeInverse;
        Bodies[b1].vAngularVelocity =
            QVRotate(~Bodies[b1].qOrientation,
            Bodies[b1].vAngularVelocityGlobal);
    } else {

```

```

        // Apply impulse:
        Bodies[b1].vVelocity += 
            (j * Collisions[i].vCollisionNormal) /
            Bodies[b1].fMass;
        Bodies[b1].vAngularVelocityGlobal += 
            (pt1 ^ (j * Collisions[i].vCollisionNormal)) *
            Bodies[b1].mIeInverse;
        Bodies[b1].vAngularVelocity =
            QVRotate(~Bodies[b1].qOrientation,
            Bodies[b1].vAngularVelocityGlobal);
    }
}
}
}
}

```

The final function we need to show you is `CheckGroundPlaneContacts`. Recall that this function is called from `CalcObjectForces` in order to determine if a ball is in resting contact with the table. If the ball's vertical position is less than or equal to its radius plus some small tolerance, and if the ball's vertical velocity is 0 (or nearly so within some small tolerance), then we consider the ball in contact with the table. If there's a contact, the relevant data gets stored in the `Collisions` array and used to resolve the contact, not the collision, in `CalcObjectForces`:

```

int     CheckGroundPlaneContacts(pCollision CollisionData, int body1)
{
    Vector   v1[8];
    Vector   tmp;
    Vector   u, v;
    Vector   f[4];
    Vector   vel1;
    Vector   pt1;
    Vector   Vr;
    float    Vrn;
    Vector   n;
    int      status = NOCOLLISION;
    Vector   Ar;
    float    Arn;

    if(Bodies[body1].vPosition.z <= (Bodies[body1].fRadius + COLLISIONTOLERANCE))
    {
        pt1 = Bodies[body1].vPosition;
        pt1.z = COLLISIONTOLERANCE;
        tmp = pt1;
        pt1 = pt1-Bodies[body1].vPosition;
        vel1 = Bodies[body1].vVelocity/*Body*/ +
            (Bodies[body1].vAngularVelocityGlobal^pt1);

        n.x = 0;
        n.y = 0;
    }
}

```

```

n.z = 1;

Vr = vel1;
Vrn = Vr * n;

if(fabs(Vrn) <= VELOCITYTOLERANCE) // at rest
{
    // Check the relative acceleration:
    Ar = Bodies[body1].vAcceleration +
        (Bodies[body1].vAngularVelocityGlobal ^
        (Bodies[body1].vAngularVelocityGlobal^pt1)) +
        (Bodies[body1].vAngularAccelerationGlobal ^ pt1);

    Arn = Ar * n;

    if(Arn <= 0.0f)
    {
        // We have a contact so fill the data structure
        assert(NumCollisions < (NUMBODIES*8));
        if(NumCollisions < (NUMBODIES*8))
        {
            CollisionData->body1 = body1;
            CollisionData->body2 = -1;
            CollisionData->vCollisionNormal = n;
            CollisionData->vCollisionPoint = tmp;
            CollisionData->vRelativeVelocity = Vr;
            CollisionData->vRelativeAcceleration = Ar;

            CollisionData->vCollisionTangent = (n^Vr)^n;
            CollisionData->vCollisionTangent.Reverse();

            CollisionData->vCollisionTangent.Normalize();
            CollisionData++;
            NumCollisions++;
            status = CONTACT;
        }
    }
}

return status;
}

```

That's all there is to this billiards example. As you can see, we used substantially the same methods shown in other examples throughout this book to implement this example. About the only new information we've shown here is how to compute rolling resistance. With a little effort, you can combine the material presented in this example with the projectile motion material presented in [Chapter 6](#) to model all sorts of sports balls. Whether you're modeling a billiard ball bouncing off a table or a basketball bouncing off a backboard, the methods are the same. The only things that will change are the empirical coefficients you use to model each ball and surface. Have fun.

PART IV

Digital Physics

Part IV covers digital physics in a broad sense. This is an exciting topic, as it relates to the technologies associated with mobile platforms, such as smartphones like the iPhone, and groundbreaking game systems like the Wii. Chapters in this part of the book will explain the physics behind accelerometers, touch screens, GPS, and other gizmos, showing you how to leverage these elements in your games. We recognize that these topics are not what most game programmers typically think about when they think of game physics; however, these technologies play an increasingly important role in modern mobile games, and we feel it's important to explain their underlying physics in the hope that you'll be better able to leverage these technologies in your games.

CHAPTER 20

Touch Screens

It is hard to deny that we are currently moving toward a post-PC computing environment. The proliferation of smartphones, tablets, and other mobile computing platforms will have far-reaching implications for how people interact with computers. These form factors do not allow for the more traditional mouse and keyboard of input for games and therefore rely heavily on the use of touch screens. This chapter aims to give you some background on the different types of touch screens, how they work, and their technical limitations. Note that we will extend our particle simulator to work with the iPhone's capacitive touch screen; the final product is very similar to the mouse-driven version but provides a starting point for a touch-driven physics simulator.

While this chapter will primarily deal with the most two most common types of touch-sensitive screens, *resistive* and *capacitive*, the following section gives an overview of many different types. In the future we may see a move to more exotic devices, especially for large-format computing devices.

Types of Touch Screens

Resistive

Resistive touch screens are basically a giant network of tiny buttons. Some of them have $4,096 \times 4,096$ buttons in a single square inch! OK, so they are not quite just normal buttons, but they come close. Resistive touch screens have at least two layers of conductors with an air gap between them. As you press on the screen, you close the gap. Bam, circuit complete, button pressed. We will flesh out that simplified description shortly.

Capacitive

Also a topic we'll soon discuss in detail, capacitive touch screens are very common on today's smartphones. These touch screens operate by calculating the change in electrical capacitance at the four corners of the screen when your finger influences the capacitive nature of the circuits under the glass. The limitation is that whatever touches the screen must be electrically conductive. If you insulate your fingers with gloves, the screen will no longer be able to locate your touch. However, this can be solved with a few stitches of conductive thread.

Infrared and Optical Imaging

Infrared touch screens use arrays of infrared LED and photodetectors to detect and interpret an object breaking the path of a LED-photodetector pair. This uses line-scanning techniques and is a very robust design.

Optical imaging techniques are relative newcomers to the touch screen scene whose big advantage is that they are extremely scalable. They use imaging devices and light sources to detect where the screen is being touched by interpreting any shadows cast by an object through the thickness of the material.

Exotic: Dispersive Signal and Surface Acoustic Wave

Several other exotic touch screen technologies exist. We won't get into detail here, but 3M has a system that detects mechanical energy in glass caused by a touch. The amount of vibration energy that reaches each sensor determines the position.

Another example of exotic screen input, surface acoustic wave technology detects changes in the pattern of ultrasonic waves traveling along the surface of the screen.

Step-by-Step Physics

Resistive Touch Screens

Resistive touch screens are classified as a *passive* touch screen technology because the screen registers a touch without any active participation by the object touching the screen. This is their major benefit over *active* technologies, such as capacitive touch screens, as resistive screens can be activated by nonconductive objects like a pen or gloved finger. In the past, resistive screens were limited to a single input, and that's the type we'll describe, but they can be made to work with two or more simultaneous inputs, also known as multitouch.

One-dimensional resistive touch sensor

To ease ourselves into this discussion, we will begin by looking at a one-dimensional touch screen. Let's imagine we have built the machine described in [Figure 20-1](#).

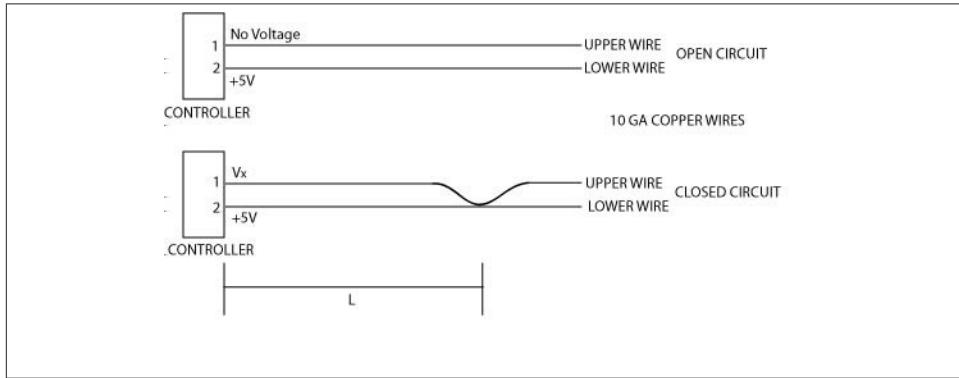


Figure 20-1. Linear resistive touch sensor

As you can see, our sensor has two states, an *open circuit state* and a *closed circuit state*. In the open circuit state, the controller is supplying a 5V signal to pin 2 and waiting for any return voltage on pin 1. With no touch to bring the wires together, the circuit is open. No voltage is present at pin 1, and therefore no touch is sensed. When the wires are touched, they are brought together and the circuit is closed. A voltage will then be present at pin 1. A touch event is registered.

This type of sensor, which looks only for the presence or absence of voltage without regard to its value, is called a *digital sensor*. It can detect only two states: on or off (1 or 0, respectively). OK, so it's not quite a touch screen yet; essentially at this point all we have is a simple button. Moving forward, let's say that we not only want to trigger an event when we press our button, but we also want to simultaneously input a value based on the location along the wire, L , that we pressed.

To accomplish this, the controller patiently waits for a voltage at pin 1. When it senses a voltage, that digital “on” switch causes the controller to then probe the voltage that is present, which we have labeled V_x . Now we get to the reason it is called a *resistive* touch sensor. Current, voltage, and resistance are all interrelated by Ohm's law. This physical relationship is expressed as:

constant. As the controller measures the voltage, V , at pin 1, we can now solve for resistance:

used at any given time. The general layout is given in [Figure 20-2](#). The squares containing the X and Y wires would actually be overlapped but are shown skewed here for clarity.

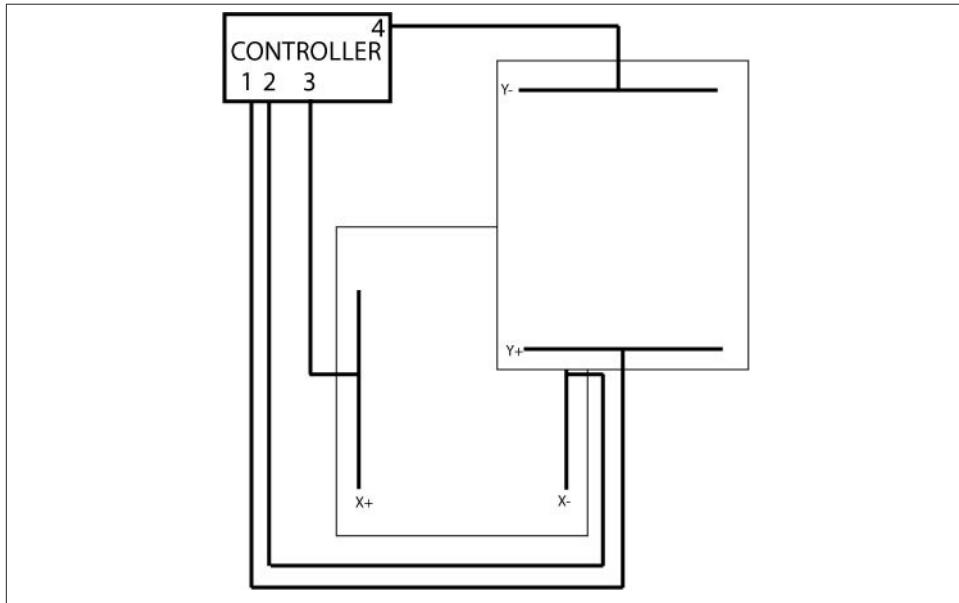


Figure 20-2. Four-wire touch screen

The reason for calling it a *four-wire* touch screen should now be obvious; however, remember that only three of the wires will be active at any time. The basic structure is shown in [Figure 20-3](#).

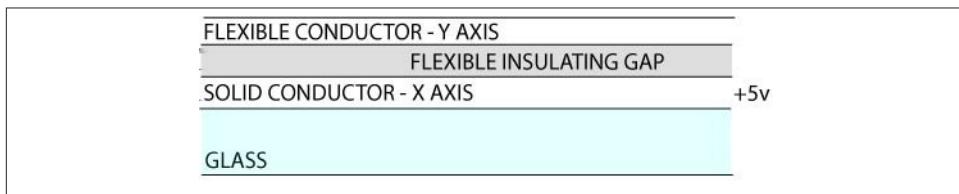


Figure 20-3. Four-wire touch screen profile

The first layer of the screen comprises a flexible conductor separated by an insulating gap. Under the gap lies a solid conductor. When a finger presses down on the outer layer of flexible conductor, it crosses the gap and makes contact with the solid conductor. The conductors are thin layers of *indium-tin oxide* (ITO) with silver bus bars on either end of the sheet, shown as black lines in [Figure 20-2](#).

To condense the description of its operation, we've outlined the three possible states in [Table 20-1](#).

Table 20-1. Possible states for four-wire touch screen

Activity	Pin 1	Pin 2	Pin 3	Pin 4
Waiting for touch detection	Open	Open	Digital input [pull up]	Ground
Read X position	Voltage probe	Ground	Voltage source	Open
Read Y position	Voltage source	Open	Voltage probe	Ground

Voltage probe means the chip is sensing the voltage on that pin, *voltage source* is the pin supplying a voltage to *ground*, and *open* means it is unused. The sequence of a touch event begins with pin 1 and pin 2 open. Pin 3 is configured to digital input with pullup signifying a voltage is supplied to the pin. When a finger presses on the outer layer and makes contact with the lower layer, pin 3 goes to ground. When the controller senses the voltage fall on pin 3, it moves to the second row and reads the X position.

To read the X position, the lower layer is energized from pin 3 to pin 2. The voltage source creates a gradient along the layer. Pin 1, connected to the upper layer, delivers a voltage to the controller when a touch pushes it down to make contact with the lower energized layer. The value of this voltage depends on where the contact is made in the gradient, much like the previous linear example. Once the X position is known, the controller moves to the next row and reads the Y position.

The method of obtaining the Y position is much the same but in reverse. The voltage supply is switched to pin 1, which develops a voltage gradient with pin 4. Then pin 3 is probed and the voltage corresponds to the distance along the voltage gradient. As the controller is capable of repeating the detect, read X, and read Y cycles approximately 500 times a second, the user is not aware that the screen doesn't actually register the X and Y coordinates at the same time.

While the four-wire resistive touch screen is the simplest two-dimensional touch sensor, there are issues with durability. The main drawback of this type of touch screen is that, because the layers must be separated by an insulating gap, at least one of the layers must be flexible. In the four-wire type, the constant flexing of the first conductive layer introduces microcracks in the coating, which lead to nonlinearities and reduce the accuracy. Other models of resistive touch screens overcome this issue with additional layers that remove the need for the flexible conductor. They have also been adapted to provide multitouch capability. We will discuss multitouch and how it works with capacitive touch screens in greater detail shortly.

Capacitive Touch Screens

A capacitive screen uses a piece of glass coated in a transparent conductor. When your finger or other conductor comes into contact with the screen, the electrostatic field is

disturbed, causing a change in the capacitance. To understand how capacitive screens work, let's quickly review capacitance in general.

A *capacitor* in its simplest form is two conductors, usually thin plates, separated by an insulator. If you apply a voltage across the two conductors, a current will flow and charge will build up. Once the voltage across the plates is equal to the supply voltage, the current will stop. The amount of charge built up in the plates is what we measure as the capacitance. Previously, we noted that one issue with resistive screens is that one part must always flex to close a insulating gap and complete a circuit. This repetitive action eventually leads to mechanical failure. A capacitor can be dynamically formed by any two conductors separated by an insulator. Noting that glass is a good insulator, it is easy to see that a finger separated from a conductor by glass can change the capacitance of a system. In this way, the finger or stylus doesn't have to cause any mechanical action, yet it can still effect changes to the sensors, which are then used to measure the location of the touch.

The methods of determining location based on capacitance on mobile devices are *self-capacitance* and *mutual capacitance*.

Self-capacitance

Anyone who has lived in a dry winter has felt the shock of a static electricity discharge. This zap is possible because the human body is a pretty good capacitor with a capacitance of about 22 pico-farads. This property is known as *body capacitance*. Self-capacitance screens take advantage of a physical property defined by the amount of electrical charge that must be added to an isolated conductor to raise its potential by one volt. When the fingers act as a conductor of the body's inherent capacitance, the sensors on the other side of the glass experience a rise in electric potential. Given that the sensors are on the other side of a good insulator, glass, there won't actually be any discharge of energy, unlike when you touch your metal car door and get "zapped." Self-capacitance in this manner produces a very strong signal but lacks the ability to accurately resolve multiple touches. Therefore, it is often used in conjunction with the next type of touch screen we'll discuss, mutual capacitance.

Mutual capacitance

The other form of capacitance-sensing screen, mutual capacitance, is formed by a grid of independent capacitors. A probing charge is sent over the rows or columns. As the capacitors charge and discharge, the system can sense the capacitance of each individual capacitor. As just discussed, the body is a good capacitor, and bringing part of it close to the capacitor grid changes the local electric field. Those capacitors that are under a finger or other conductor will read lower values than normal. Each capacitor can be scanned independently, enabling high resolution of where the touch event is occurring. Additionally, because they act independently of one another, it is possible to accurately register multiple touches. Think of this system as taking a picture of the capacitance on

the skin of the screen. Using algorithms similar to image processing and edge detection, this system can compute the extent of a touch event.

Example Program

Included in the source code accompanying this book is an example of the particle explosion program from [Chapter 8](#) that uses touch screen input instead of a mouse click.

The code for a Cocoa touch Objective-C event is as follows:

```
- (void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event
{
    UITouch* touch = [[event touchesForView:self] anyObject];
    firstTouch = [touch locationInView:self];
    self.status = YES;
    [self trigger];
}
```

where `firstTouch` is defined by `CGPoint firstTouch`; in the header file. The `CGPoint` is a Cocoa touch object capable of storing an (x,y) coordinate in the display view's coordinate system. We can then use `firstTouch.x` and `firstTouch.y` later in our program to give a location to the particle explosion.

As you can see, it is very similar to a mouse-based event. One big difference is that you could adapt the code to handle multitouch events. Computers recognize only one mouse cursor at a time, but with a touch screen you can register multiple clicks simultaneously.

Multitouch

In iOS you must first enable delivery of multiple touch events by setting the `multipleTouchEnabled` property of your view to YES; the default is NO. Next you must create a class to keep track of multiple `TParticleExplosion` structures. Then it is as simple as polling the position of the start points of multiple touches to trigger multiple explosions. The Objective C code to store the start points of multiple touch events would look like:

```
- (void)storeTouchPoints:(NSSet *)touches{
    if ([touches count] > 0) {
        for (UITouch *touch in touches) {
            CGPoint *point = (CGPoint *)
                CFDictionaryGetValue(touchBeginPoints, touch);
            if (point == NULL) {
                point = (CGPoint *)malloc(sizeof(CGPoint));
                CFDictionarySetValue(touchBeginPoints, touch, point);
            }
            *point = [touch locationInView:view.superview];
        }
    }
}
```

where `CFDictionaryRef` is an immutable dictionary object that allows the copying of the object and its key value. One last consideration for this example is that as you are now creating multiple physics simulations simultaneously, you may have to decrease the frequency of the time steps to allow the animations to proceed smoothly. Multitouch can become programmatically complex, but the physics are pretty simple. The event-handling guide for your particular development language should give detailed guidance on handling the event chain.

Other Considerations

One of the major advantages of touch screens is that their layout and actions are entirely software based. That is to say, if a certain button does not pertain to the current layout, it can be discarded and the freed space used for additional relevant controls. We will discuss other less obvious considerations in the following sections.

Haptic Feedback

The flip side of the advantage of not being locked into a set of physical buttons is that the user must rely almost entirely on vision to interact with the controls. At least one of the authors of this book uses a keyboard with no letters on it at all, instead relying totally on the physical position of the keys to determine which key to strike. This would be much harder without the tactile and audio cues to signify that the correct key has been pressed. Indeed, it is easy to tell when he is typing poorly because the backspace key is much louder than the rest!

The method of including physical feedback to assist a user in interaction with entirely virtual objects is known as *haptic feedback*. The first use of haptic feedback in games was limited to arcade games such as *Motocross*, in which the handlebars shook after an in-game impact. It is now considered standard on video game controllers, which vibrate to inform the user of some event.

In the realm of touch screens, haptic feedback is used to inform the user of a successful key strike or other touch-based event. Some touch screens even incorporate some movement of the entire screen when pressed. This feedback still doesn't allow touch typing, as it only dynamically responds and provides no static tactile feedback for different buttons.

Modeling Touch Screens in Games

Given their planar nature and the lack of inherent haptic feedback, touch screens can be an easy way to implement controls with which a character in a game can interact. The amount of physical modeling required to create a realistic in-game keyboard is pretty intense. Thus, there are very few examples of an in-game character having to sit down and type a code in to a terminal via a standard keyboard.

By using touch screens for in-play control of objects, you can avoid an additional physical model while retaining realism. It would also be interesting to see games use realistic touch screen interfaces so that a character would have to remove his gloves to use a capacitive screen. Lastly, the exotic screen technologies mentioned earlier provide many creative avenues of modeling those types of screens in games. For example, for screens measuring sound waves in the glass or other mechanical energy, low-grade explosions could be used to trigger these in-game input devices.

Difference from Mouse-Based Input

One important consideration for game developers in regards to touch screens is the difference from traditional mouse- and keyboard-based gaming. As console game developers have long been aware, it is hard to compete with the speed and accuracy of the mouse/keyboard combination. Many first-person shooters segregate their online gaming between controllers and mouse/keyboard setups, as the accuracy and speed of the mouse gives those players an unfair advantage. Upon using touch screens on many different gaming devices and mobile computing platforms, we feel that this advantage is even more pronounced.

A touch by a finger is an elliptical shape whose contact patch depends on the specific finger being used, the pressure applied, and the orientation of the finger. The user generally perceives the point of touch to be below where the actual center of contact is, so adjustments must be made. This is generally all handled automatically by the operating system so that a single touch point is computed and handed to the game via an API. However, this generic approach to computing touches must obviously sacrifice accuracy for universality so that it is not calibrated for one specific user.

Another inherent drawback to touch screens is the need to touch the screen. This means a large portion of your hand will be blocking the screen when you are controlling that element. One can imagine that in a first-person shooter, this would be a great disadvantage over someone who is playing with a keyboard and mouse.

Lastly, mouseover is not available to touch-screen-based input. Consider a game where you would trigger actions by merely moving a mouse cursor over an object. These actions could be distinct from clicking on the same object. However, with touch-screen-based input, that object would be obscured by whatever is triggering the screen, therefore rendering the mouseover action invisible to the user.

Custom Gestures

As a last note, another possibility for touch input to a game is the use of custom gestures. These allow the user to draw a shape on the screen that the program recognizes as a gesture. It can then execute arbitrary code based on that input. As this is more pattern recognition than physics, we won't cover it here, but we can recommend the book *Designing Gestural Interfaces* by Dan Saffer (O'Reilly) as a detailed look at this subject.

CHAPTER 21

Accelerometers

Accelerometers are a good introduction to a class of electronic components called *microelectromechanical systems* (MEMS). An accelerometer can either be one-axis, two-axis, or three-axis. This designates how many different directions it can simultaneously measure acceleration. Most gaming devices have three-axis accelerometers.

As far as game development is concerned, acceleration values are typically delivered to your program via an API with units in multiples of g . One g is equal to the acceleration caused by gravity on the Earth, or 9.8 m/s^2 . Let's pretend that we have a one-axis accelerometer and we orient it such that the axis is pointing toward the center of the earth. It would register 1g. Now, if we travel far away from any mass, such that there is no gravity, the accelerometer will read 0. If we then accelerate it such that in one second it goes from 0 m/s to 9.8 m/s, the accelerometer will read a steady 1g during that one-second interval. Indeed, it is impossible to tell the difference between acceleration due to gravity and acceleration due to changing velocity.

Real-life motion is generally nonsteady. Depending on your application's goals, you might have to apply different smoothing functions such as *high-pass* or *low-pass filters*. This amounts to *digital signal processing*, a topic that has consumed entire texts. One example we can recommend is *Digital Signal Processing: A Computer Science Perspective* by Jonathan Y. Stein (Wiley).

Also, many accelerometers have a method to set the *polling rate*, or the number of times per second that the program requests updates from the accelerometer. This is called *frequency* and is given in hertz (Hz). This parameter can be used to enhance the performance of the program when fine resolution of the acceleration over time is not needed.

When you accept input from an accelerometer—or do any other kind of signal processing—you have to accept that input won't come precisely when you want it. The operating systems normally used for gaming—Windows, OS X, Linux—are not real-

time environments. This means that although you set the polling rate at once a second, this guarantees only that the data will be delivered *no sooner than* once a second. If something distracts the operating system, such as the arrival of packets on the network, the signal you get from the accelerometer may be delayed.

Accelerometer Theory

The way in which MEMS measure accelerometers is more basic in principle than you may think. The major accomplishment is miniaturizing the technology until it can fit inside a computer chip! To clearly illustrate the basic principle, we will first show you the mechanics of it in the macro-scale version of a known mass and spring. Let's say you build something like the contraption shown in [Figure 21-1](#) and take it on an elevator in an area where there is no gravity. We'll worry about the effects of gravity in a minute.

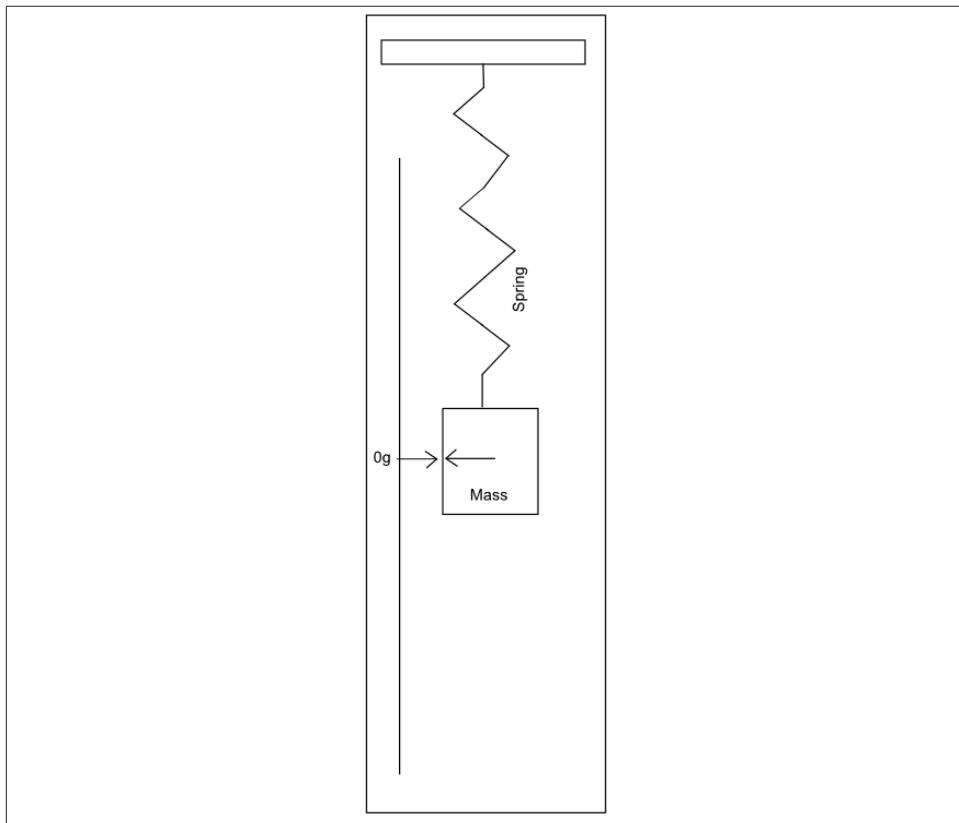


Figure 21-1. Simple accelerometer in absence of acceleration

As you can see, the device consists of a known mass at the end of a spring next to a measuring stick. When the elevator is not accelerating, the mass is at the 0 mark. When the elevator accelerates up or down, the mass at the end of the spring resists that acceleration and tends to stay at rest. This is Newton's first law in action. Inertial loading causes the spring to stretch or compress. If the elevator is accelerating upward, the mass will cause the spring to stretch downward. Recall from [Chapter 3](#) that the force acting on a spring is linearly dependent on the displacement of the mass via the equation:

MEMS Accelerometers

Micro-scale accelerometers are not that much different from the machine previously described but generally use a cantilevered beam instead of a spring. To track more than one axis, sometimes three discrete accelerometers are placed out of plane with respect to one another. Alternatively, more complex models use elements that can sense all three directions within a single integrated sensor. These generally give better results.

The only important difference from the aforementioned examples, besides MEMS being thousands of times smaller in scale than the mass and spring, is how to measure the deflection of the test mass. There are three common methods employed in accelerometers. For most game devices where extreme accuracy isn't required, the deflection is usually measured as a change in capacitance. This is somewhat the same way that capacitive touch screens work, as described in [Chapter 20](#), and is shown in [Figure 21-2](#).

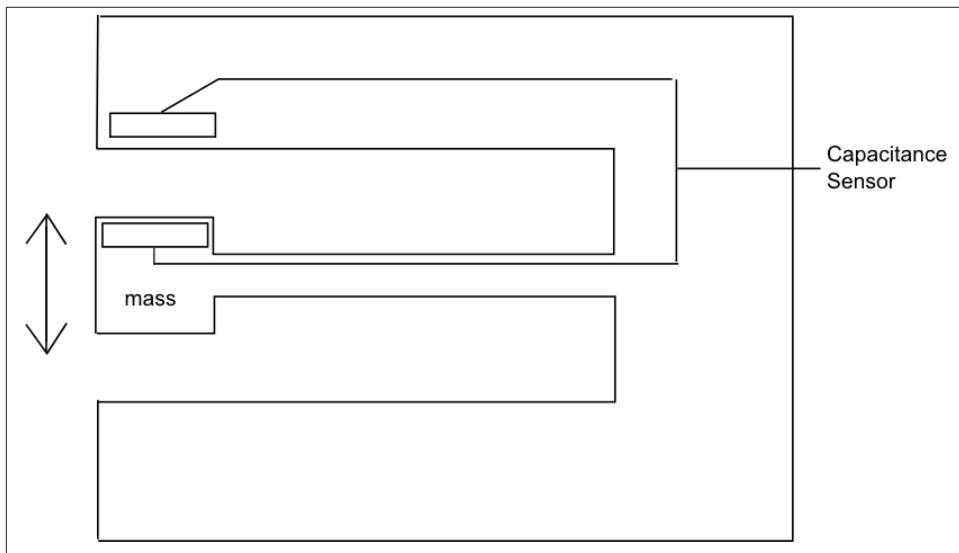


Figure 21-2. MEMS cantilever accelerometer

The beam deflects under the influence of the external accelerations of the test mass and brings two charged plates farther or closer together. This changes the capacitance of the system. This change can then be calibrated to the imposed acceleration.

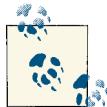
Other methods include integrating a piezoresistor in the beam itself so that the deflection of the beam changes the resistance of the circuit. Although this ultimately gives better results, these are harder to manufacture. For the most demanding applications, there are accelerometers using piezoelectric elements based on quartz crystals. These are very sensitive even during high-frequency changes in acceleration but are generally not used in sensing human-input motion.

Common Accelerometer Specifications

To help you better experiment with accelerometers, we've collected the specifications on a few of the most common accelerometers in use at the time of writing. The future may hold cheap accelerometers based on quantum tunneling that can provide almost limitless accuracy, but [Table 21-1](#) outlines what you'll generally be working with for now.

Table 21-1. Current common accelerometers

Device	Accelerometer chip	Sensor range	Sampling rate
iPhone/iPad/ Motorola Droid	LIS331D	$\pm 2g^*$	100 Hz or 400 Hz
Nintendo Wii	ADXL330	$\pm 3g$	x-/y-axis: 0.5 Hz to 1600 Hz z-axis: 0.5 Hz to 550 Hz
Sony Six Axis	Not published	$\pm 3g$	100 Hz



The chip LIS221D is actually capable of two modes. One mode is $\pm 2g$ and the other is $\pm 8g$. This is dynamically selectable according to the chip's datasheet; however, neither iOS nor Android allows developers to change the mode through the API.

The $2g$ limit for phones can cause problems when you're attempting to record motion. This limitation will be discussed later in this chapter. The larger range of Wii and Sony controllers demonstrate that they are dedicated to gaming where larger accelerations are expected.

Data Clipping

The human arm is capable of exceeding the $\pm 2g$ range of the iPhone's sensor easily. The values reported by the API will actually exceed $2g$ up to about $2.3g$. The accuracy of these values that exceed the specification is unknown. Regardless, they are probably at least as accurate as the option of trying to recreate the data, so if required they can be used. All values above this upper limit will be reported as the upper limit such that if you graphed the values, they would look like [Figure 21-3](#).

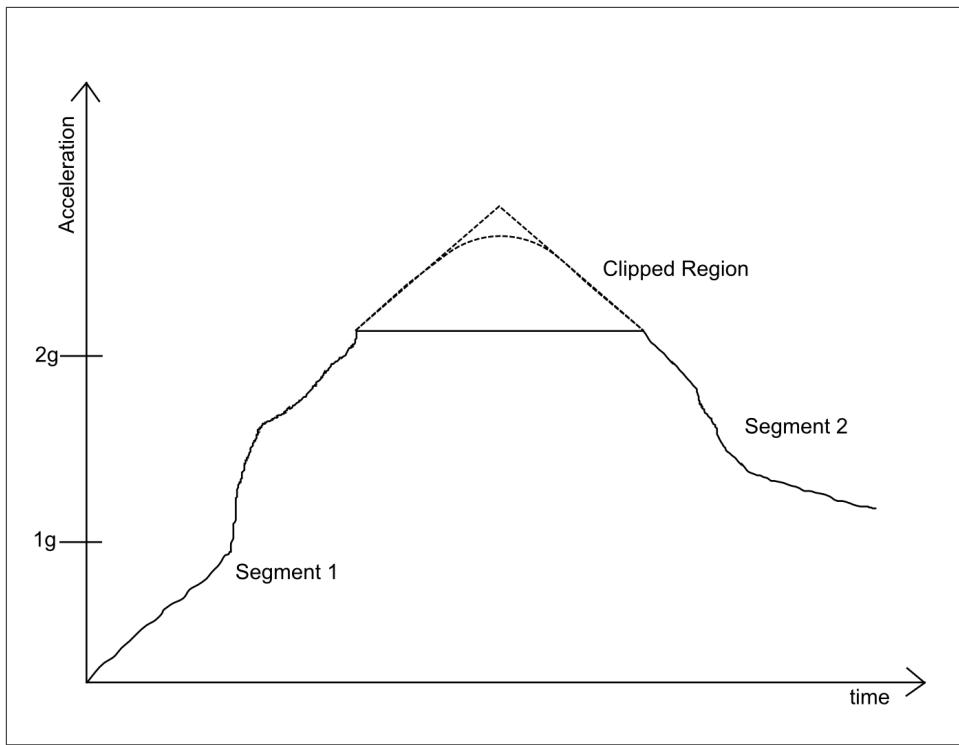


Figure 21-3. Acceleration graph showing clipping

There are several different ways to handle data clipping. One is to discard the data and alert the user that he has exceeded the available range. Another is to attempt to recreate the missing data. If you are recording the data for later processing, you can use both segment 1 and segment 2 to fit the curve between the point at which the data began to be clipped and the point in which meaningful data collection is resumed. This is highly application dependent, and the curve used to fit the data will have to be matched to the activity at hand. If you are recording the data for later processing, you can use both segment 1 and segment 2 to give your data.

If you are attempting to process the signal in real time, you'll have only segment 1 to work from. This could result in a discontinuity when meaningful data collection resumes, and you'll have to decide how to deal with that given the particulars of what you are doing with the data.

Sensing Orientation

Sensing rotation in three degrees of freedom amounts to sensing a rigid body's orientation and is a complex problem that cannot be fully resolved using only accelerometers.

Think about holding the device vertically. If you rotate the device about the axis described by the gravity vector, none of the accelerometers will measure any change in the force acting on their test masses. We can't measure that degree of freedom. To do so, we'd need to fix a gyroscope to the device, and even these run into problems when a body is free to rotate about all three axes. See [Chapter 11](#) for a discussion on Euler angles.

Now let's discuss what we can accomplish. First, [Figure 21-4](#) demonstrates the coordinate system we'll use; the actual coordinate system used will be determined by the manufacturer of your device, so make sure to check its documentation.

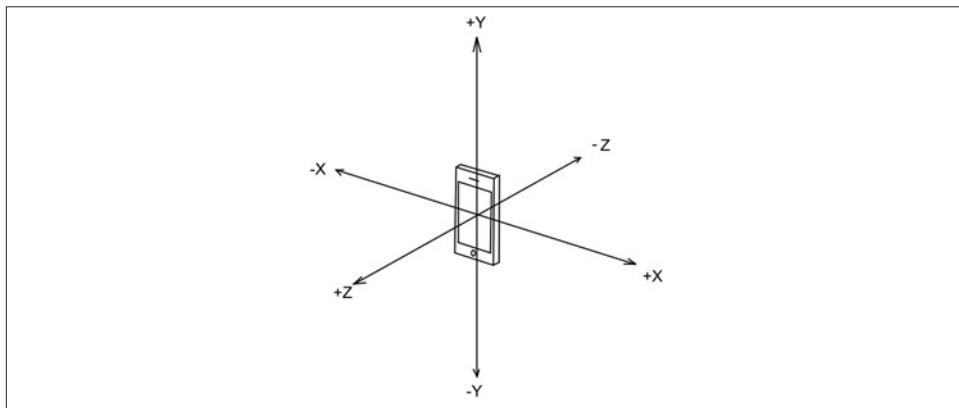


Figure 21-4. Accelerometer coordinate system

Now if we make some assumptions based on how a user will hold our device, we can determine some “gross” orientations. For illustration, [Table 21-2](#) gives some idea of what each value would be for each gross direction, assuming the coordinate system shown in [Figure 21-4](#).

Table 21-2. Gross acceleration values and orientations

Device orientation	X	Y	Z
Face down on table	0	0	1
Face up on table	0	0	-1
Horizontal on table, right side down	1	0	0
Horizontal on table, left side down	-1	0	0
Vertical on table, bottom down	0	-1	0
Vertical on table, top down	0	1	0

There are a few things to note here. First, if you were to hold the phone in these orientations with your hand, the accelerometer is sensitive enough to pick up small deviations

from true vertical. We are considering these the “gross” orientations such that these small deviations should be ignored.

Sensing Tilt

Although we can’t determine exactly what angle the user is holding the phone about all three axes, we can pick one axis, assume that it is pointing down, and then find the change in the angle from that assumption over time. For instance, if the phone is lying on a table, the average acceleration in the z-direction will be -1 , and in the other directions, 0 . Even if the user spins the phone, the values will remain as previously indicated and we cannot sense that rotation. However, if the user lifts one edge from the table—we’ll call this tilting it—then the accelerometer will register different values. Some of the acceleration due to gravity will act on the other two axes. By sensing this change, an accelerometer will allow us to determine at what angle the device is tilted.

Using Tilt to Control a Sprite

Here we will show you how to implement code for a simple game that asks the user to move an avatar to a target by tilting the phone. First, we will briefly show an example of determining the rotation about a single axis. Let’s assume we have an accelerometer rotated at some arbitrary angle, α , which is what our algorithm will solve for. As previously discussed, accelerometers generally report values as multiples of near earth gravity, g . For the following example, we are concerned only with the x- and y-axis values, a_x and a_y , respectively. If the device were in the “upright” position, then a_x would equal 0 and a_y would equal 1 . After rotating the device, we’d see different values that are related to our angle α by use of the arctangent function. In this case, because the *single-argument atan function* included in most programming languages doesn’t differentiate between diametrically opposed directions, it is beneficial to use the *two-argument function*. The relevant C code is as follows:

```
#define PI 3.14159

float find2dAngle(void){

    //LOCAL VARIABLES
    float alpha,
        double ax, ay;

    //POLL ACCELEROMETER FOR ACCELERATIONS, API SPECIFIC
    ax = getXacceleration();
    ay = getYacceleration();

    //FIND ANGLE
    alpha = atan2(ay,ax);

    if (alpha >= 0){


```

```

        alpha = alpha * (180/PI);
    else {
        alpha = alpha * (-180/PI) + 180;
    }

    return alpha;
}

```

This is pretty straightforward, but there are a few things to point out. First, the way in which your program will get results from the accelerometer will vary greatly between platforms, so we have encapsulated that API-specific code in a `getAcceleration()` function. In fact, most operating systems will be continuously polling the accelerometer in a separate thread, so you'll have to have a logical operator that tells your accelerometer object when you actually want to see those values passed to your program. Example Objective-C code for the accelerometer in the iPhone will be shown later. Secondly, you'll notice that we are using an `if` statement that changes the radians to degrees in such a way as to return proper 0°–360° answers. This avoids having to pay attention to the sign, as `atan2` returns only answers between 0° and 180°, using a negative value to represent the other half of the range. For example, an output of 0° means the device is vertical, an output of 90° means the device is rotated 90° to the left, and an output of 180° means the device is upside down.

Now let's extend this to two dimensions. This will tell us not only how far the phone is from vertical about one axis, but its inclination about the y-axis as well.

Two Degrees of Freedom

Now let's say that we want to develop a game in which we control a sprite moving in a 2D world. The user would hold the device as if it were lying on a table and look down from above. He or she would then tilt the phone out of that plane to get the sprite to move in the desired direction. The fraction of gravity that the accelerometer is now experiencing in the x- and y-directions will be inputs into our simulation.

The example will be demonstrated using Objective-C code for the iPhone, and we'll be using the Quartz2D graphics framework. If you aren't familiar with Objective-C, don't worry—we'll explain what we are doing in each step, and you can port that code to whatever language you are working in.

The first step will be to set up our accelerometer. In this case we are going to initialize it in our `tiltViewController.m` file so that we have:

```

- (void)viewDidLoad
{
    UIAccelerometer *accelerometer = [UIAccelerometer sharedAccelerometer];
    accelerometer.delegate = self;
    accelerometer.updateInterval = kPollingRate;
    [super viewDidLoad];
}

```

The important concept here is that we have defined a name for our accelerometer object, `accelerometer`, and we have set its `updateInterval` property to `kPollingRate`. This constant was defined in `tiltViewController.h` as `(1.0f / 60.0f)`, which corresponds to 60 Hz. In other words, this tells the operating system to update our program's accelerometer object 60 times a second. Also in `tiltViewController.m`, we write what happens when the accelerometer object gets updated via the accelerometer's `didAccelerate:` function as follows:

```
- (void)accelerometer:(UIAccelerometer *)accelerometer  
didAccelerate:(UIAcceleration *)acceleration{  
    [(SpriteView *)self.view setAcceleration:acceleration];  
    [(SpriteView *)self.view draw];  
}
```

This function is called every time the acceleration object is updated and does two things. First, it takes the acceleration data from the accelerometer and passes it to the `SpriteView` class, which we'll talk about in a second. Then it tells the `SpriteView` to go ahead and redraw itself.

The `SpriteView` class is where the action happens and consists of a header file, `SpriteView.h`, where we define the following global variables:

`UIImage *sprite`

A pointer to the image that will be used to represent our sprite on the screen.

`currentPos`

The position on the screen where we want the sprite to be drawn.

`prevPos`

The previous position of the sprite on the screen. We will use this to tell the `draw` function what parts of the screen need to be redrawn.

`UIAcceleration *acceleration`

A special Objective-C data type to hold data from the accelerometer.

`CGFloat xVelocity and CGFloat yVelocity`

Float variables to hold the current velocity in the x-direction and y-direction, respectively.

`CGFloat convertX and CGFloat convertY`

Float variables to hold the ratios for converting our physics engine's results in meters to pixels based on an assumed world size.

Additionally, we've defined the following global constants:

`g`

Near earth gravity value, set at 9.8 m/s^2 . This will convert the accelerometer's values from g to m/s^2 for use in calculating velocity. This can also be tuned to represent an

arbitrary acceleration instead of just using gravity as the force (e.g., percent of jet engine thrust).

`kWorldHeight` and `kWorldWidth`

These values are used to allow the programmer to change the assumed world dimensions. Higher values mean each pixel is a greater distance in meters. The world will always be scaled to fit on the screen, so a large world means the sprite will appear to move slower (a few pixels at a time) for a given acceleration. Note that our current code doesn't scale the sprite.

Now we'll show you how we use these variables in `SpriteView.m` to move our sprite on our screen as a result of the accelerometer values. First, we have some initialization to do, which takes place in the `initWithCoder:` method that runs the first time the view is loaded:

```
- (id)initWithCoder:(NSCoder *)coder {
    if((self = [super initWithCoder:coder])){
        self.sprite = [UIImage imageNamed:@"sprite.png"];
        self.currentPos = CGPointMake((self.bounds.size.width / 2.0f) +
(sprite.size.width / 2.0f), (self.bounds.size.height / 2.0f)+(sprite.size.height /2.0f));
        xVelocity = 0.0f;
        yVelcoity = 0.0f;

        convertX = self.bounds.size.width / kWorldWidth;
        convertY = self.bounds.size.height / kWorldHeight;

    }
    return self;
}
```

Most of this is pretty straightforward. We tell our program where to find the sprite image we've chosen and set its initial position to the center of the screen. We also set its initial velocity to 0 in both directions. We then go ahead and initialize our `convertX` and `convertY` variables based on the `self.bounds.size` property, which gives the bounds of the view in pixels. We'll show exactly how this affects our program later. Next, we'll write a custom `mutator` for the `CurrentPos` variable:

```
- (void)setCurrentPos:(CGPoint)newPos {
    prevPos = currentPos;
    currentPos = newPos;

    if(currentPos.x <0){
        currentPos.x = 0;
        xVelocity = 0.0f;
    }
    if(currentPos.y <0){
        currentPos.y = 0;
        yVelcoity = 0.0f;
    }
    if(currentPos.x > self.bounds.size.width - sprite.size.width){
```

```

        currentPos.x = self.bounds.size.width - sprite.size.width;
        xVelocity = 0.0f;
    }
    if(currentPos.y > self.bounds.size.height - sprite.size.height){
        currentPos.y = self.bounds.size.height - sprite.size.height;
        yVelocity = 0.0f;
    }

    CGRect curSpriteRect = CGRectMake(currentPos.x, currentPos.y,
    currentPos.x+sprite.size.width, currentPos.y+sprite.size.height);
    CGRect prevSpriteRect = CGRectMake(prevPos.x, prevPos.y,
    prevPos.x+sprite.size.width, currentPos.y+sprite.size.height);
    [self setNeedsDisplayInRect:CGRectUnion(curSpriteRect, prevSpriteRect)];
}

}

```

In case you are unfamiliar with Objective-C, when you define a class instance variable it will automatically define a mutator that simply updates the value of the variable to the value you are passing it. However, in the preceding example we are overriding that mutator to do some additional work. The first thing we do is to set the `prevPos` variable to the current position of the sprite and then update the `currentPos` with the value the mutator was given. However, our physics engine isn't going to include collision response with the screen boundaries, so we go on to check if the sprite has reached the screen edge. If so, we simply tell the program to leave it on the edge and to set the velocity in that direction to 0. Lastly, we define a couple of rectangles based on the new position of the sprite and the old position of the sprite. After we union those rectangles together, we tell the operating system to redraw the screen in that area with the `setNeedsDisplayInRect:` method. As you might recall, our `accelerometer` object is calling the `draw` method every time it updates, and it is in this method that we will put our physics engine:

```

- (void)draw {
    static NSDate *lastUpdateTime;

    if (lastUpdateTime != nil) {
        NSTimeInterval secondsSinceUpdate = -([lastUpdateTime
timeIntervalSinceNow]); //calculates interval in seconds from last update

        //Calculate displacement
        CGFloat deltaX = xVelocity * secondsSinceUpdate +
((acceleration.x*g*secondsSinceUpdate*secondsSinceUpdate)/2); // METERS
        CGFloat deltaY = yVelocity * secondsSinceUpdate +
((acceleration.y*g*secondsSinceUpdate*secondsSinceUpdate)/2); // METERS

        //Converts from meters to pixels based on defined World size
        deltaX = deltaX * convertX;
        deltaY = deltaY * convertY;

        //Calculate new velocity at new current position
        xVelocity = xVelocity + acceleration.x * g * secondsSinceUpdate; //assumes
acceleration was constant over last update interval
    }
}

```

```
    yVelocity = yVelocity - (acceleration.y * g * secondsSinceUpdate); //assumes
acceleration was constant over last update interval

    //Mutate currentPos which will update screen
    self.currentPos = CGPointMake(self.currentPos.x + deltaX,
self.currentPos.y + deltaY);

}

[lastUpdateTime release];
lastUpdateTime = [[NSDate alloc] init];

}
```

Previously, we discussed issues with timing when working with accelerometer data. In this case, Objective-C makes it very easy to get the correct elapsed time in seconds. We first define a static variable, `lastUpdateTime`, as an `NSDate` type. This type has a built-in function to give the time interval in seconds from now, which we assign to an `NSTimeInterval` variable. Skipping down to the last two lines, we are simply updating the last update time by releasing and reinitializing the variable. As it is static, it will remain even after the function returns. If you are using a lower-level language, you might have to write your own `timeIntervalSinceNow` function that takes into account the particular clock frequency of the system.

Now that we have our time interval in seconds, we can calculate our new position. Recall from [Chapter 2](#):

As you can see from the complete method description, the code of the y-direction is similar. Finally, we call the `currentPos` mutator to set the new position based on the change in displacements. Recall that this is a custom mutator that also tells the operating system to update the display. After the `draw` method is finished, the accelerometer waits 1/60 of a second and then calls it again. You could extend this program by adding in friction, fluid resistance, and collisions with the screen boundaries using the methods outlined in the other chapters of this book.

Gaming from One Place to Another

Once a tool meant to help the United States guide intercontinental ballistic missiles, the *Global Positioning System* (GPS) has evolved to be a part of our everyday lives. The current generation will never have known a world where getting lost was something that couldn't be fixed by trilaterating their position between satellites orbiting the planet. Although GPS has become commonplace in the navigational world, the proliferation of smartphones is just now opening the doors to GPS gaming. While this genre is just emerging, we'd like to give you an introduction to the physics behind GPS and the current applications in the gaming world.

Let's recall that positions near the earth's surface are generally given in the *geographic coordinate system*, more often described as latitude, longitude, and altitude. *Latitude* is a measure in degrees of how far north or south you are from the equator. *Longitude* is the measure in degrees of how far east or west you are from the prime meridian. A meridian is a line of constant latitude that runs from the North Pole to the South Pole. The prime meridian is arbitrarily defined as the meridian that passes through the Greenwich Observatory in the UK. *Altitude* is usually given as the measure of how far above or below sea level you are at the point described by latitude and longitude.

Location-Based Gaming

Before getting to the physics behind GPS, we'd like to take a moment to discuss how GPS is being implemented into games. Right now, this is an emerging market that is just starting to gain traction. There are several broad categories into which games fall. Another step beyond what the accelerometer did, GPS enables users to move computer games not only off the couch but also out into the world.

Geocaching and Reverse Geocaching

Geocaching is the oldest form of gaming involving GPS. It originated after selective availability was removed from GPS, making it more accurate, in the year 2000. In its most basic form, it is the process of hunting down a “cache” using provided GPS coordinates. The cache usually has a logbook and may contain other items such as coins with serial numbers that the finder can move to another cache and track online.

Because of the large amount of setup involved in implementing a geocaching game on a commercial scale, most implementations are community based. However, reverse geocaching has more promise for the gaming industry. In this variation there is nothing at the supplied coordinates, but traveling to them is required to execute some action. Think of it as carrying around a cache that cannot be unlocked until it is within range of some specific coordinate. This could be used to force users to travel in order to unlock a game item. For instance, perhaps to gain the ability to use a sword in a game, the user must travel to the nearest sporting goods store. The commercial possibility of corporate tie-ins is an obvious plus.

Mixed Reality

Mixed-reality games are similar to geocaching. They go beyond just using the coordinates of the user to trigger events, to using reality-based locals. A current example is Gbanga’s *Famiglia*. In this game your movement in the real world allows you to discover virtual establishments in the game world. This divorces it from the actual physical locations that your GPS is reporting but requires moving between locations in the real world to move your character in the virtual world. Popular right now is the FourSquare app on mobile devices. This is the simplest possible implementation of mixed-reality gaming. FourSquare allows a user to become the mayor of a place if she “checks in” at the locale more than anyone else.

Street Games

Street games are another step beyond mixed reality. These turn the environment around the user into a virtual game board. One example is the recent *Pac-Manhattan* multi-player game using GPS in smartphones to play a live version of *Pac-Man* in Washington Square Park. In general, the idea is to create a court for game play using the environment surrounding the user. The relationship between users is tracked in the virtual space of the game and provides the interactive elements.

What Time Is It?

The story of GPS really begins with a prize offered by the British government in 1717 for a simple way to determine your longitude. Awarded in 1773, the accepted solution was to compare local noon to the official noon sighted at the Greenwich Observatory. The difference between these two times would allow you to tell how far around the world you were from the observatory. Fast-forward three centuries, and we have satellites orbiting the earth, broadcasting time-stamped messages. By calculating the difference between the time the message was received and the time it was transmitted, we can calculate our distance from the satellite. In both cases you need an accurate way to keep or measure time. For sailors in the 1800s, the device was the newly invented chronometer. For us, it is the atomic clock.

Because the signals from a GPS satellite are moving at the speed of light, you need a very accurate clock to keep track of how long it took to travel to you. For instance, if the clock you are using to time when the signal arrives is 1 microsecond off, you will estimate a distance over 900 miles in error. On the supply side of the signal, each satellite has an atomic clock, and internal GPS time is accurate to about 14 nanoseconds. The problem is that you also need a very accurate clock in the receiver, and it would be pretty hard to fit an atomic clock into a phone economically. To get around this, the receiver must figure out the correct current time based on the signals from the satellites.

Two-Dimensional Mathematical Treatment

This section will give you a good idea of how GPS systems determine their location. This background will help you in many applications of geometry in games in general, but most GPS devices do the heavy lifting and report through an API your current latitude and longitude. Some APIs may include more information—for example, the current iOS API, called Core Location, gives the current latitude and longitude, the direction of travel, the distance traveled, and the distance in meters to a given coordinate. It also gives an estimate for the error associated with its position fix in meters.

One way to get your position via the kind of information that GPS provides is a technique called *trilateration*. We are going to give this problem a mathematical treatment in two dimensions. You could extend this to three dimensions by using spheres instead of circles.

To begin, we can list our unknowns: our x coordinate and y coordinate in space, and the error in our receiver's clock (or *bias*), b . In a two-dimensional plane, trilateration among three circles gives you an exact position; in three-dimensional space, four spheres are required to determine all three special coordinates. Note that if we included an assumption about being on the surface of some geometric shape, such as the earth,

we could reduce the number of unknowns. No such simplification is used here to provide you with the most general case.

In our example, we are somewhere on the surface of the two-dimensional earth, shown in [Figure 22-1](#) as a light gray solid disk. This disk is being orbited by several GPS satellites. The satellites' orbits are regular, and their positions at any time are tabulated in an almanac that is stored in the receiver. The time of transmission is encoded in the signal so that the givens are x_i , y_i , and t_i , with $i = 1, 2, 3$.

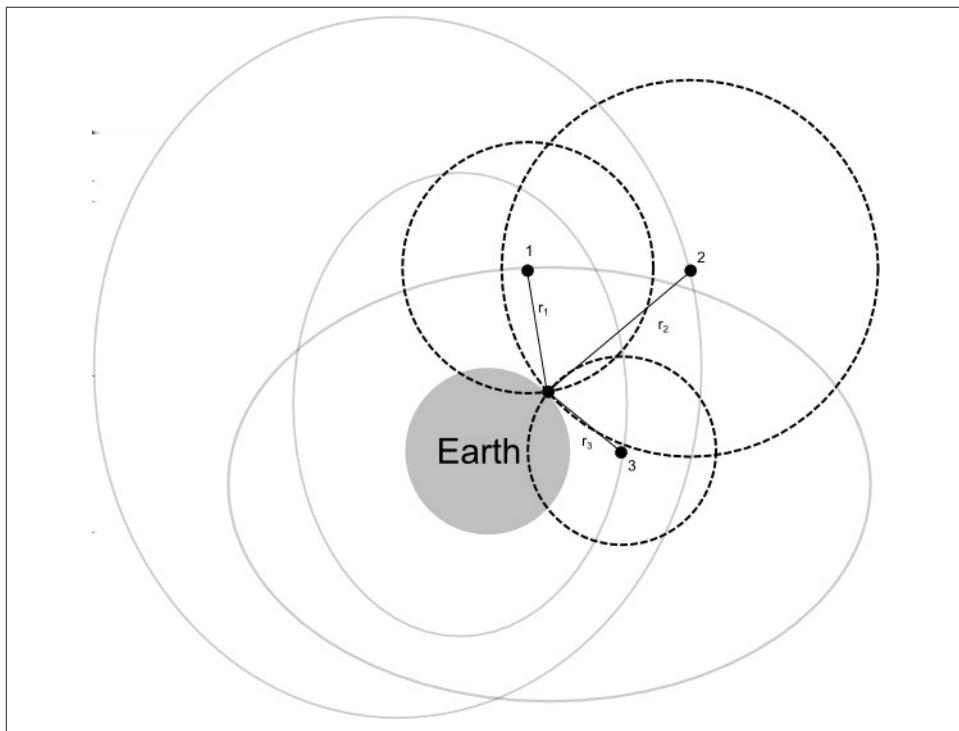


Figure 22-1. Trilateration in 2D

To make things easier for us, we are going to abandon the coordinate system of the earth and use the coordinate system defined by our three satellites. The origin will be at satellite 1, the x-axis going directly from satellite 1 straight to satellite 2 and the y-axis being perpendicular to that. This is shown in [Figure 22-2](#).

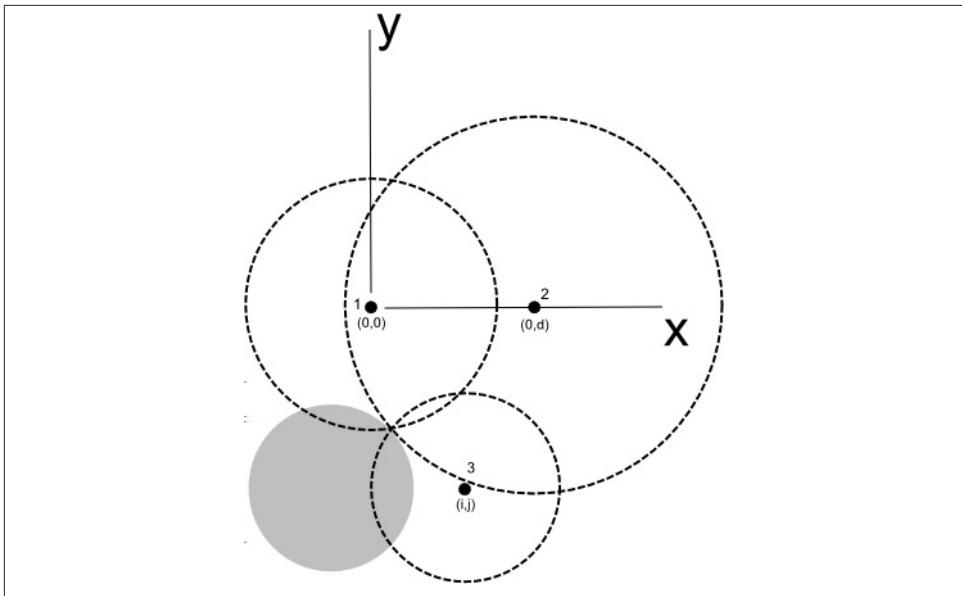


Figure 22-2. Satellite coordinate system

The equations of the three circles are therefore:

where d is the distance between the known locations of satellite 1 and satellite 2. We now substitute our x coordinate back into the first circle's equation:

solution that requires no iteration was developed by Stephen Bancroft. It is detailed in his paper “An Algebraic Solution of the GPS Equations” in the *IEEE Transactions on Aerospace and Electronics Systems* journal.

Besides clock errors, other errors are introduced by the atmosphere, signals bouncing off the ground and back to the receiver, relativistic effects (discussed in [Chapter 2](#)), and atomic clock drift. These are all accounted for in mathematical models applied to the raw position data. For instance, the GPS clocks lose about 7,214 nanoseconds every day due to their velocity according to special relativity. However, because they are higher up in the earth’s gravity well, they gain 45,850 nanoseconds every day according to general relativity. The net effect is found by adding these values together: they run 38,640 nanoseconds faster each day, which would cause about 10 kilometers inaccuracy to build each day they are in orbit. To account for this, the clocks in the GPS receivers are pre-adjusted from 10.23 MHz to 10.22999999543 MHz. The fact that we are giving you a number to 11 decimal places demonstrates the amount of accuracy the modern age enjoys in its time keeping.

Once the bias is taken care of and all the other possible errors adjusted for, the converged solution can be translated back into whatever coordinate system is convenient to give to the end user. Usually this is latitude, longitude, and altitude. Next, we will learn how to calculate different quantities based in the geographic coordinate system.

Location, Location, Location

Let’s take a minute to discuss distance between two latitude and longitude coordinates. You might be tempted to calculate it as the distance between two points. For very small distances, this approximation is probably accurate enough. However, because the earth is actually a sphere, over great distances the calculated route will be much shorter than the actual distance along the surface.

The shortest distance between two points on a sphere, especially in problems of navigation, is called a *great circle*. A great circle is the intersection of a sphere and a plane defined by the center point of the sphere, the origin, and the destination. The resulting course actually has a heading that constantly changes. On ships, this is avoided in favor of using a *rhumb line*, which is the shortest path of constant heading. This makes navigation easier at the expense of time. Airplanes, however, do follow great-circle routes to minimize fuel burn.

Distance

There are several ways of calculating the distance along a great circle. The one we will discuss here is the *haversine formula*. There are other methods like the *spherical law of cosines* and the *Vincenty formula*, but the haversine is more accurate for small distances

than the spherical law of cosines while remaining much simpler than the Vincenty formula.

The haversine formula for distance is:

```

//Calculate angular distance
float C = 2 * atan(sqrt(a)/sqrt(1-a));

//Find arclength
float distance = 6371 * C; //6371 is radius of earth in km
return distance;
}

```

One limitation of the preceding method is that if the two locations are nearly *antipodal*—that is, on opposite sides of the earth—then the haversine formula may have round-off issues that could result in errors on the order of 2 km. These, however, will be over a distance of 20,000 km. If extreme accuracy is required for nearly antipodal coordinates, you can fall back to the spherical law of cosines, which is best suited for large distances such as the antipodal case.

Great-Circle Heading

As discussed before, to follow the shortest path between two points on a sphere you must travel along a great circle. However, this requires that your heading be constantly changing with time. The formula to calculate your initial heading, or *forward azimuth*, is:

A negative angle involves starting at 0° and rotating in the decreasing-heading direction, but compasses aren't labeled with negative values! To fix this, the line that has the comment "fix range" is using a ternary operator to say that if the bearing is less than 0, return the value the compass would read. For example, if the bearing were -10° , then the compass bearing is $-10^\circ + 360^\circ = 350^\circ$. If the value is positive, then it just returns the same value.

To find the final bearing, we simply take the initial bearing going from the end point to the start point and then reverse it. The code is produced as follows:

```
float finalBearing (Coordinate2D startPoint, Coordinate2D endPoint){  
    //Convert location from degrees to radians  
    float lat1 = (M_PI/180.) * endPoint.lat;  
    float lon1 = (M_PI/180.) * endPoint.longi;  
    float lat2 = (M_PI/180.) * startPoint.lat;  
    float lon2 = (M_PI/180.) * startPoint.longi;  
  
    //Calculate deltas  
    float dLat = lat2 - lat1;  
    float dLon = lon2 - lon1;  
  
    //Calculate bearing in radians  
    float theta = atan2f( sin(dlon) * cos(lat2), cos(lat1)*sin(lat2)-sin(lat1)*cos(lat2)  
                        *cos(dlon));  
  
    //Convert to compass bearing  
    float bearing = theta * (180 / M_PI); //radians to degrees  
    bearing = ( bearing > 0 ? bearing : (360.0 + bearing)); //fix range  
    bearing = ((bearing + 180) % 360) //reverse heading  
    return bearing;  
}
```

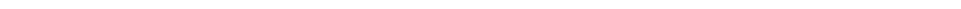
The difference here is that we have flipped `lat1`, `long1` and `lat2`, `long2` while converting the locations to radians. Also, before we return the bearing value, we reverse it by adding 180° degrees to it. The modulo operator (%) ensures that values over 360° are rolled over into compass coordinates. For example, if we calculate a bearing of 350° and add 180° to it, we get 530° degrees. If you start at 0° and go around 530° , you'll end up at 170° . The modulo operator will result in the bearing being calculated with this correct compass value.

Rhumb Line

As discussed before, it is sometimes preferable to take a longer path of constant heading, called a rhumb line, as compared to constantly changing your heading to follow a great circle path. The rhumb line will be longer than the great circle, and the distance you are from the great circle route at any moment is called the *cross track error*. To cross the Atlantic is about 5% longer if you follow a rhumb line. The extreme example of going from the East Coast of the United States to China is about 30% longer. However, such

large penalties are rarely encountered because ships have to alter course to avoid land! This makes “as the crow flies” examples unrealistic.

If your game is providing navigation information to anyone but pilots, it will probably be using rhumb lines. The following are the formulas used to calculate distance and bearing between two coordinates on a rhumb line. The easiest way to begin is to flatten the globe. In a Mercator projection, rhumb lines are straight. In fact, this makes graphically solving the problem very simple. You use a ruler. Mathematically, things get a bit more complicated. The following equation gives $\Delta\phi$, which is the difference in latitude after taking into account that we have stretched them in order to flatten the sphere:



```

//Calculate deltas
float dLat = lat2 - lat1;
float dLon = lon2 - lon1;

//find delta phi
float deltaPhi = log(tan(lat2/2+(M_PI)/4)/tan(lat1+M_PI/4))
float q=(deltaPhi==0 ? dLat/deltaPhi : cos(lat1)); //avoids division by 0

if (abs(dLon) > M_PI){
    dLon = (dLon>0 ? -(2*(M_PI-dLon)):(2*M_PI+dLon));
}

float D = sqrt(dLat*dLat + q*q*dLon*dLon)* 6371;
float theta = atan2f(dLon, deltaPhi);

//now convert to compass heading
float bearing = theta * (180 / M_PI); //radians to degrees
bearing = ( bearing > 0 ? bearing : (360.0 + bearing)); //fix range

return bearing;
}

```

There are a few things worth pointing out. First is that we are using a ternary function in the line commented by “avoids division by 0” to take care of the case when `deltaPhi` is equal to 0. If it is 0, `q` is set to `cos(lat1)`; if not, then it is set to `dLat/deltaPhi`. The `if` statement immediately following ensures that if `dLon` is greater than π (180°), hence putting us on a longer-than-required rhumb line, then we should correct the value to correspond to the shortest route. This is achieved via the ternary, which ensures that `dLon` is less than π and nonnegative. Lastly, we convert from a normalized radian answer to a compass direction.

Now that you have a good idea about how to calculate position and distance in the geographic coordinate system, you can use the earlier chapters to determine other quantities like speed and acceleration.

Pressure Sensors and Load Cells

Pressure sensors are an evolution of the simple button. A simple button has two states, on or off, which can be used to trigger simple atomic actions in a video game such as firing a gun or opening a door. However, simple buttons are not capable of informing the program how you, the user, hits that button. Did you hit it quickly? Did you barely touch it at all? The only thing the program can interpret is that you did in fact hit the button.

With pressure sensors, the program has the ability to discern *how* the user pressed the button. This information can be used as incremental input, such as the player raising a firearm before pressing the button harder to actually fire. Additionally, pressure sensors can be used to create novel forms of human-input devices. While pressure sensitivity is not uncommon in the more traditional console gaming markets, there is also a recent push to move the sensors into touch-screen devices like the Nintendo DS and cell phone gaming market. Pressure-sensitive touch screens are currently beyond state of the art, however, so we'll primarily discuss the traditional methods already in widespread adoption.

In addition to pressure sensors, some new gaming consoles use *load cells* to allow the player to use shifts in his or her body weight as input. The method by which this data is collected and how the center of gravity is determined will be discussed in this chapter. Lastly, some smartphones now include a *barometer*, a pressure sensor that measures the pressure of the atmosphere. What it is used for and the type of information it can provide will also be discussed.

Under Pressure

As discussed in [Chapter 3](#), *pressure* is a force applied over an area. Imagine a concrete block sitting on a steel plate. The weight of the block will be evenly distributed over the area of contact, creating a pressure on the steel plate. Gas and liquid can apply pressure as well. The weight of the air pressing down on us is what is known as *atmospheric pressure*.

Let's cover a quick example of how to calculate pressure just to illustrate the concepts involved. Pressure has many different units, but all of them can be equated to a force divided by an area. For this chapter we'll stick with Newtons per square meter, as this is easiest to visualize. The SI derived unit (a unit of measure made up of other fundamental units) is called a Pascal, which is just 1 N/m^2 .

Example Effects of High Pressure

In [Chapter 3](#), we discussed the concept of buoyancy and how it arises from hydrostatic pressure. Here, we'll show the tremendous forces that hydrostatic pressure can cause on a submerged object. Let's imagine we have a steel ball filled with normal atmospheric pressure at sea level, or about $101,000 \text{ N/m}^2$. While this seems like a lot, your body is used to dealing with this pressure, so you don't even notice it on a daily basis! Now we are going to take this ball and drop it into the Marianas trench, the deepest known part of the ocean. The water depth here is approximately 10,900 meters. The formula for calculating the pressure due to water (hydrostatic pressure) is:

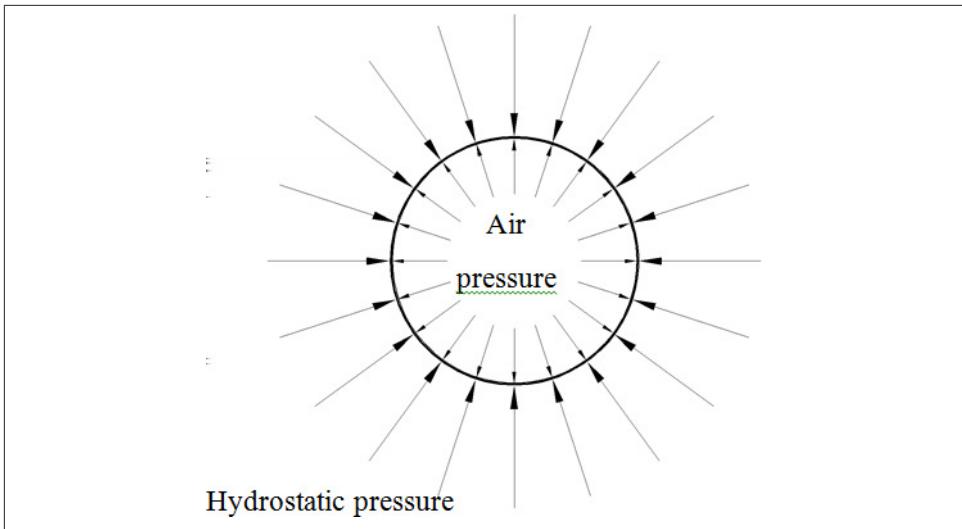


Figure 23-1. Pressure differential

It is clear that the water pressure acting on the sphere is much larger than the air pressure we trapped inside before sinking it. Also, note that pressure always acts normal to the surface. If you happen to apply a force to the vertex of an object, you'll have trouble modeling the right effect because a vertex does not have a well-defined normal. We can overcome this only by applying pressure to the faces of polygons or by averaging the direction of the pressure on either side of the vertex. Returning to our example, the net pressure differential on the steel ball is:

If the ball were open to the sea, then the pressure would act equally on each side of the steel boundary. Without a pressure differential, there would be no force to crush the ball; however, there would still be compression of the steel shell itself.

Button Mashing

While the preceding example highlights some important concepts about pressure in general, it is not usually the type of pressure used as input to a game. The most common types of pressure sensors you'll experience in video games are pressure-sensitive buttons that indirectly measure the amount of pressure the user is exerting on the button and convert this to a relative value. Both Sony and Microsoft have incorporated pressure-sensitive (also known as *analog*) buttons into their controllers for the PlayStation and Xbox series of consoles.

The method by which you can detect how hard a user is pressing a button varies from very simple to very complex. We'll focus on Sony's method, which is very elegant. A typical push button is just two contacts separated by an insulator, most commonly air. When the button is pushed, the upper contact moves down and touches the lower contact. This completes a circuit, causing a voltage spike, which the device interprets as a button press. This is another example of a digital sensor—it is either on or off. The buttons in Sony's controller work a bit differently. In State A in [Figure 23-2](#), we can see that the button is not yet pressed and an air gap exists between the solid conductor and the domed flexible conductor. In State B the button is depressed with minimal pressure, and the dome just barely makes contact. The button is now activated. If the user continues to press down harder on the button, the dome deflects and increases the area of contact with the fixed conductor; the larger the contact area, the greater the conductivity of the connection. This causes a rise in the current flowing in the circuit.

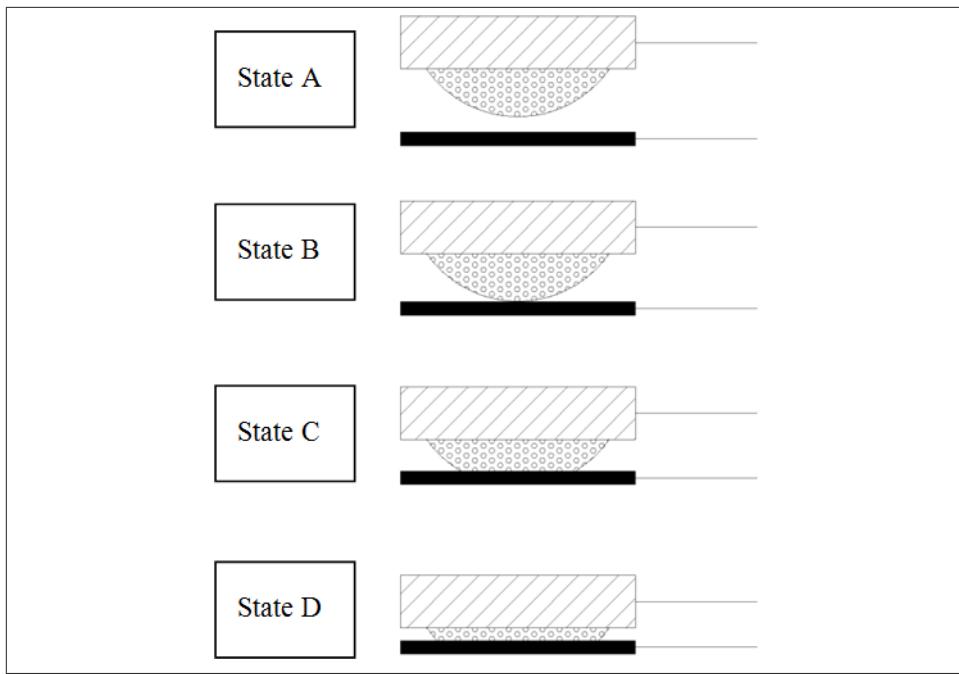


Figure 23-2. Pressure-sensitive button

By measuring this increase in current, the controller knows how far down the button is being pressed. In State D in [Figure 23-2](#), the button is at its limit of travel and the dome has deflected to its maximum contact area. The difference between this maximum and the minimum required to detect contact determines the absolute lowest and highest pressure the button is able to differentiate. For instance, let us assume that if the button were depressed completely, the current would register at I_{\max} . If the button were not pressed at all, of course, the current would be 0. If we call the current $I(t)$ for any time, t , we see that the ratio $I(t)/I_{\max}$ gives a nondimensional quantity for how far down the button is pressed. During this operation, the hardware converts the analog voltage to a digital representation suitable for input to a program. For the Sony example, this value is calculated by the hardware and passed as part of the data stream from the controller with hex values between 0x00 to 0xFF, or in other words, integers 0 to 255 in decimal. This means that each button's travel is divided into 255 parts that your program can register.

While 255 individual increments are beyond the human ability to control fingertip pressure, different ranges of pressure have practical uses in games. For example, you could program your button to raise a weapon with a half-press (0 to 127), bring the weapon to the shoulder with more pressure (127 to 250), and to fire when totally depressed (250 to 255). Of course, those values would have to be tuned for the desired

level of sensitivity. Another example would be to control the throttle on a car by using the values of 0 to 255 as thrust multipliers.

Another use of knowing a button's position would be tracking it over time. With a time history of position, you can differentiate to get velocity and again to get acceleration. This would allow the program to differentiate between a button that is either slowly depressed or quickly depressed. Most hardware doesn't help you here, so you'll have to store the values and calculate the velocities in whatever increments are appropriate for your program. As real-time velocity sensing might be taxing to the user as real-time input, the best use would be as input to something that the user doesn't have to control constantly. Imagine having to keep a button pressed down at the correct pressure for your gameplay for longer than a few minutes; I can feel my wrist cramping now. However, the pressure button is useful for many inputs. For example, how far a button is pressed down might be used to draw back the head of a putter, while the speed at which the button is released could be used to determine the speed at which the putter is brought back to the ball.

Load Cells

Beyond simple buttons, there are other novel ways to use pressure sensors to allow a user to interact with your games. For example, Nintendo's Wii uses a balance board peripheral based on load cells to detect a person's stance.



The original idea for the Nintendo balance board came to video game designer Shigeru Miyamoto after he was inspired by watching sumo wrestlers weigh themselves with each leg on a different scale. They are too heavy to use one scale!

Tiny scales

Load cells work differently than the pressure-sensitive buttons described previously, but like pressure-sensitive buttons, they come in different types, all of which measure the load pressing on them. The most common way, and the one used in the Nintendo balance board we'll discuss shortly, is through what is called a *strain gauge*.

A strain gauge, as you might be able to guess, does not measure force directly but instead measures how much strain the gauge is experiencing. Strain is a measure of how much a rigid body has deformed independent of its rigid-body motion. While there are several notions of strain in continuum mechanics, the one we are concerned with here is often referred to as *engineering strain*. This type of strain quantifies how much a structural element has deformed compared to its original, or *rest*, length. We normalize this by dividing the change in length over the rest length. By testing the material, one can develop a stress versus strain curve that relates how much stress it takes to cause a certain

amount of strain. Once the pressure that causes an amount of strain is known, it is possible to determine the amount of load. Now you might be wondering how the strain gauge measures the amount that the Wii's legs compress when you stand on them.

One of the most common electronic strain gauges is the *piezoresistive strain gauge*. The simplest example of a piezoresistive strain gauge would be a single wire. If you were to elongate a wire from its rest length, the cross-sectional area decreases. This causes a rise in the electrical resistance of the wire. After measuring the rest resistance, you can use the difference to determine how much the wire has elongated. Knowing the mechanical properties of the wire, you can also determine how much force it takes to stretch the wire.

To make strain gauges sensitive without having long linear wire elements, the conductive material is often arranged in a strain-sensitive pattern, as shown in [Figure 23-3](#). This looping back and forth of the conductor allows for great sensitivity without increasing the physical space the sensor occupies. Here the rest length would be 18 times longer than the physical length of the sensor.

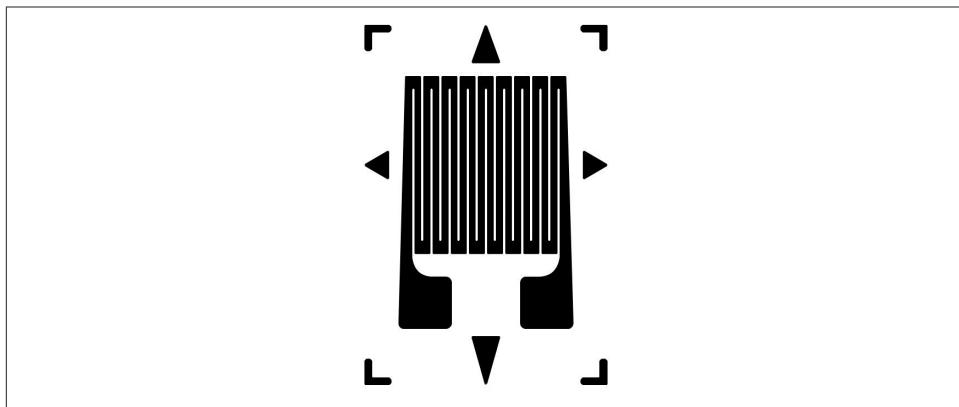


Figure 23-3. Typical strain-sensitive pattern

Center of gravity

The board has four legs, each of which houses a load sensor. The board uses strain gauges similar to those discussed earlier. These gauges elongate when a force is applied to them. The elongation changes the electrical resistance of the circuit of which the strips are a part, and this is reported back to the controller. [Figure 23-4](#) shows two sensor outputs. The first is with the user standing so that her center of gravity is over the center of the board. The second state shows what the board's sensors would measure after the user has shifted her center of gravity.

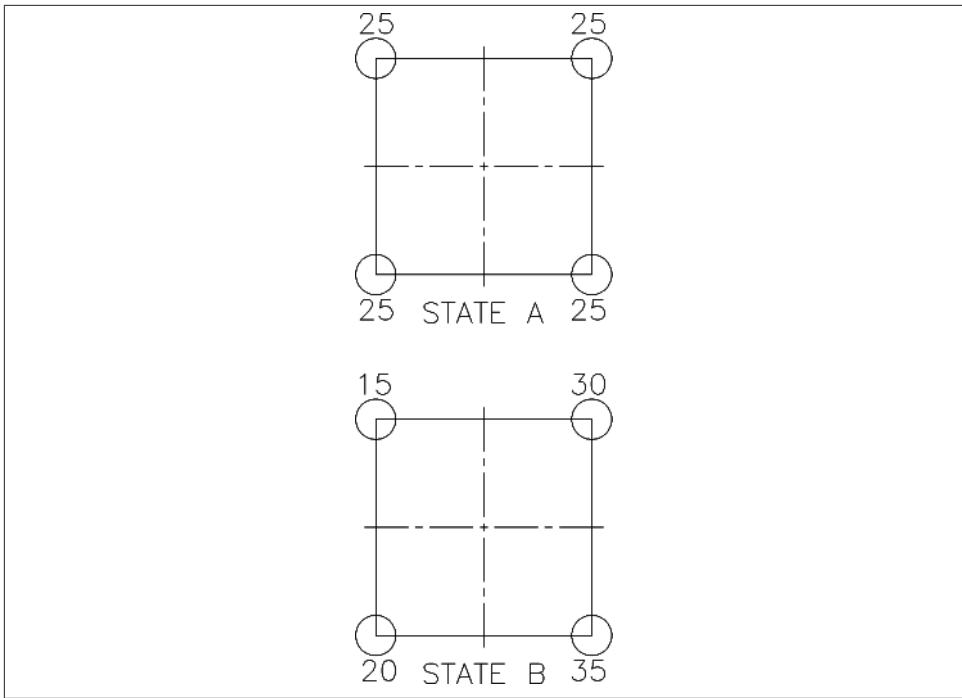


Figure 23-4. Balance board example

It is easy to intuitively recognize that the center of gravity must be over the center of the board in State A and toward the lower-right corner in State B. However, to get the exact location of the center of gravity in State B, we'll have to do a little more work. First things first: we have to define a coordinate system. This is shown in [Figure 23-5](#).

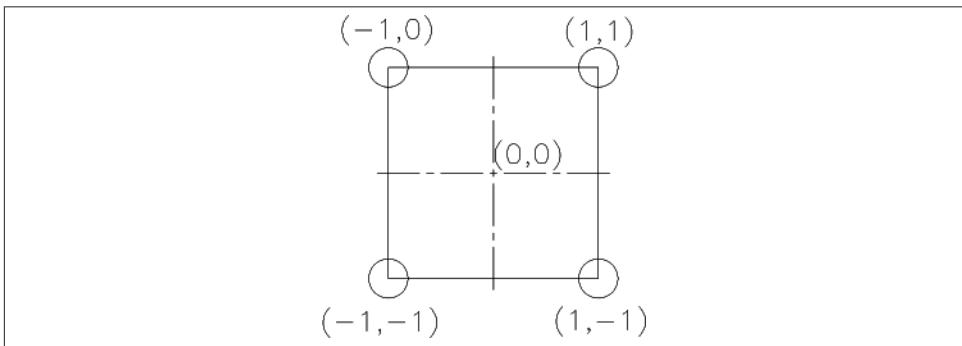


Figure 23-5. Balance board coordinate system

This coordinate system is arbitrary. If the board isn't a perfect square, such as with the Wii board, then the coordinates of the load cells must be changed accordingly. Now that we have defined the location of the center of the board and the position of the load cells, we can use a weighted average to compute the location of the user's center of gravity. The weight that we give each value will depend on how much of the user's weight is on each of the four corners. That weight will "pull" the center of gravity toward the location of the load cells as defined in our coordinate system. How much each load cell mathematically pulls the center of gravity will be based on the weight supported at that location. This is most easily determined via two tables, one for the x coordinate (Table 23-1) and one for the y coordinate (Table 23-2).

Table 23-1. x coordinate weighted average

Load cell	Weight	Arm	Weight × Arm
(1,1)	30	1	30
(-1,1)	15	-1	-15
(-1,-1)	20	-1	-20
(1,-1)	35	1	35
Total:	100		30
Average: $30/100 = 0.30$			

Table 23-1 takes the weight in each corner and multiplies that value by the value of its x coordinate. This is equivalent to a moment. Taking the sum of those moments (30) and dividing by the total weight gives the average value of 0.30, or .3 units to the right in our coordinate system. The y -axis is treated similarly.

Table 23-2. y coordinate weighted average

Load cell	Weight	Arm	Weight × Arm
(1,1)	30	1	30
(-1,1)	15	1	15
(-1,-1)	20	-1	-20
(1,-1)	35	-1	-35
Total:	100		-10
Average: $-10/100 = -0.10$			

Using a similar weighted average as shown in Table 23-2, we see that the user's center of gravity is -0.10, or 0.1 units behind the center. If we were using this to control an onscreen sprite, we could define a 2D direction vector based on this information.

In addition to just determining the center of gravity, you can use this information to make educated guesses on what else the user is doing to cause the load distributions. After computing the center of gravity, the Wii uses what Nintendo calls a *motion-identifying condition table* to guess what movements the user is making. The table cor-

relates the ratio of the sum of the load values to the body weight of the user and the position of the center of gravity to determine body orientation. For example, the Wii can tell if both of the user's feet are on the board, or if the user is accelerating part of his leg. The table provided in the Wii patent is reproduced in [Table 23-3](#).

Table 23-3. Motion-identifying condition table

Motion	Ratio of load value to body weight value	Position of the center of gravity
Right foot riding	25 to 75%	+0.01 to +1.0
Both feet riding	More than 95%	-0.7 to +0.7
Left foot riding	25 to 75%	-1.0 to -0.01
Left thigh lifting	More than 100%	+0.01 to +1.0
Right thigh lifting	More than 100%	-1.0 to -0.01
Both feet putting down	Less than 5%	Not considered

Barometers

Continuing our exploration of new user input methods, especially in the rapidly maturing mobile device gaming market, now we'll discuss an interesting inclusion in the latest smartphones: a barometer. Unlike buttons and balance boards, whose pressure sensors only indirectly handle pressure, barometers directly measure the fluid pressure that the atmosphere exerts on the sensor.

The sensors used in mobile phones today are piezoresistive microelectromechanical systems (MEMS) and are very accurate. As shown in [Figure 23-6](#), the sensors consist of a void machined into a piece of silicon. The diaphragm is then bonded to a stiff material such as steel or glass. As we are trying to measure *absolute pressure*, this bond is airtight. Using a material called *monocrystalline semiconductor silicon* to form the void ensures that the entire diaphragm acts much like a piezoresistive strain gauge.

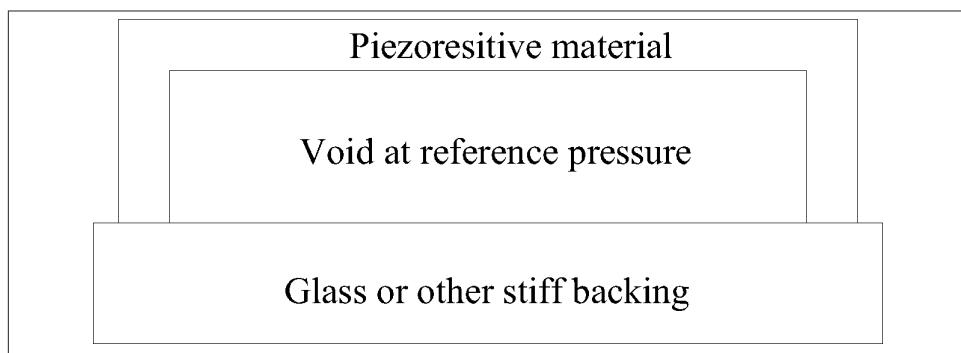


Figure 23-6. MEMS piezoresistive pressure sensor

Now we have a situation similar to the steel ball in the ocean, only this time it is a tiny silicon ball in the ocean of air surrounding the earth. When the sensor is moved deeper or shallower in the atmospheric ocean, the pressure on the outside of the diaphragm changes. This causes the pressure differential to change and a force to be exerted on the silicon diaphragm. This force causes a deflection that changes the resistance of the piezoresistive material and can therefore be measured by the sensor. This part will be taken care of by the hardware and the encoded value sent to the operating system.

To give you an example, in the Android operating system, the API has a public method, `getAltitude(float p0, float p)`, to determine altitude, in meters, between the sensor pressure and the pressure at sea level. It usually reads the current atmospheric pressure, `p`, from the sensor by listening to sensor manager callback interface method `abstract void onSensorChanged(SensorEvent event)`. Here the class `event` holds the sensor values, the accuracy of those values, a reference to the sensor itself, and a timestamp for when the event occurred. The pressure is reported in hectoPascals (hPa) or 100 N/m^2 . The sea-level pressure, `p0`, that this is compared to is either obtained from an online database or is set at the constant `SensorManager.PRESSURE_STANDARD_ATMOSPHERE`. As pressure at sea level changes with different weather conditions, we obtain higher accuracy by retrieving it from a nearby airport or other weather station via the Internet. To get the change in altitude between two points, you must repeat this process twice as follows:

```
float altitude_difference =
    getAltitude(SensorManager.PRESSURE_STANDARD_ATMOSPHERE,
                pressure_at_point2) -
    getAltitude(SensorManager.PRESSURE_STANDARD_ATMOSPHERE,
                pressure_at_point1);
```

At first it may seem strange for your cell phone to have a barometer in it; however, the barometer's ability to detect the air pressure allows you to make a good guess on your altitude. As shown in [Chapter 22](#), in order to determine your position via GPS you have to solve a four-dimensional set of linear equations. The time required to solve these equations can be dramatically decreased if you know approximately where you are to begin with. Currently, the position of which cell phone tower your phone is connected to is used as a starting point. Using a barometer allows the device to guess its altitude to further reduce the time to obtain a GPS fix.

While the sensor was included for a specific purpose, it can also be adapted as an input device. For instance, the Bosch BMP180 currently being included in devices is accurate to plus or minus one meter. In fact, Google Maps now provides indoor directions, including knowing what floor you are on in airports and shopping malls. This functionality could be used to aid in the location-based gaming discussed in [Chapter 22](#) by giving it greater resolution in the vertical dimension. It could also be used to determine if the user is holding the phone near her feet or her head, further augmenting the orientation

sensing discussed in [Chapter 21](#). Of course, it can also be used to help weather forecasting and allow you to have real-life changes in pressure affect in-game events.

CHAPTER 24

3D Display

For all the work we've done to make programs' graphics more realistic, the best we can do is project our realistic simulations onto a two-dimensional screen. Although graphics libraries such as Microsoft DirectX and OpenGL can provide photorealistic renderings in real time, they still lack the ability to truly immerse the user in the works you have so carefully created. Three-dimensional display is something that the entertainment industry has attempted to make standard for some time. In reality, almost all "three-dimensional" display technologies are what are technically called *stereoscopic* displays. These displays use the way in which your eyes perceive depth to trick your brain into thinking it is seeing a three-dimensional image while the display remains two-dimensional. In contrast, displays that actually involve creating a rendering in three dimensions are called *volumetric* displays. We'll cover these later as part of our effort to discuss emerging technologies.

Binocular Vision

The trick to displaying objects so that they appear to be three-dimensional depends on the method by which the human brain perceives the world around it. Indeed, animals that have two eyes engage in what is called *binocular vision*. Because each eye is in a slightly different position relative to the objects it is viewing, the left and right eye provide an image that is distinct to the brain. This is called *binocular disparity*. There are three possible results when the brain encounters these two different images: suppression, fusion, or summation. *Suppression* is when the brain ignores one of the images, *summation* is when the brain tries to perceive both images at the same time (double vision), and finally *fusion* is mixing the two images to create a depth of field. The process of binocular fusion is something our brain learns to do when we are first born.



Given the way that eyes focus light, we are born seeing the world upside down! After a few days, our brains instinctively flip the images over so that the motion of our hands matches the motion that we observe. There have even been tests where people wearing glasses that invert your vision will eventually see the images right side up. When they take off the glasses, everything looks upside down again until their brain has time to correct the image.

Binocular fusion is also a learned behavior of the brain. The visual cortex takes the independent visual information from each eye and fuses it into a single image. Your brain does this as a way of organically calculating the distance to objects so that you can efficiently interact with the three-dimensional world. The exact process by which your brain accomplishes this is an area of active research. In fact, researchers have found that two images need not have any geometrical disparity in order to be fused. That is, if you take the exact same photograph of the same object and the same angle, but with different lighting, the shadows being cast can also cause the brain to recreate the object in three dimensions.

Parallax is the distance an object moves between the left- and right-eye images. You can easily demonstrate it by holding your thumb six inches from your face and closing one of your eyes. Block some of the words on this page with your thumb. Now open that eye and close the other one. The words that were behind your thumb should now be visible. This is because your eyes are not in the same position, so the different angles provide slightly different pictures of the page. This distance your thumb appeared to move is the parallax at that distance from your eye.

Fusion is a little harder to achieve, but [Figure 24-1](#) provides an interesting example. The two circles are set a specific distance apart and show the top of a truncated cone coming out of the page. The top of the cone is offset compared to the bottom. This offset is in opposite directions, mimicking how your eyes would see it if you were directly over the cylinder.

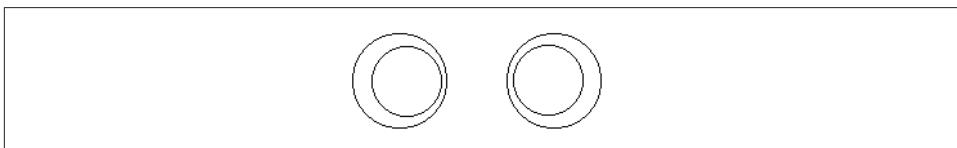


Figure 24-1. Cone stereopair

The best way to view the stereopair shown in [Figure 24-1](#) is to begin by looking above this book at a far-off object. Now lower your gaze without refocusing your eyes and stare between the two sets of circles. With some trying, your brain should be able to fuse the images so that there are now three sets of circles. The original two will be out of

focus, and the center set should appear to be three-dimensional. You can also get the sets to fuse by crossing your eyes; however, this is much less comfortable than using your eyes in their distance-viewing configuration.

Given that your brain is excellent at real-time pattern recognition, it can also compare visual information over time to get a sense of size and relative distance. This is called *movement parallax*, and it causes objects that are closer to you to appear to move faster when you move your head than objects that are farther away. For example, if you are driving in a car, you'll notice that the trees appear to move faster than the moon. This is because the trees are very close in comparison to the moon. Your brain uses this apparent speed disparity to help conclude that the moon is very far away indeed. In the next chapter we'll discuss how computer algorithms attempt this sort of pattern recognition.

In fact, according to *Flight Simulation* (edited by J. M. Rolfe and K. J. Staples; Cambridge University Press), the process of 3D visualization depends on the following eight major factors.

- Occlusion of one object by another
- Subtended visual angle of an object of known size
- Linear perspective (convergence of parallel edges)
- Vertical position (objects higher in the scene generally tend to be perceived as farther away)
- Haze, desaturation, and a shift to bluishness
- Change in size of textured pattern detail
- Stereopsis
- **Accommodation of the eyeball** (eyeball focus)

A standard 3D graphics library is capable of giving the appearance of three dimensions on the screen, just as any good painter on a canvas. Both standard 3D libraries and painters do their job by recreating the first six items in the preceding list. To further the illusion, 3D display technology simulates the seventh, stereopsis. Stereopsis is impression of depth generated by the fact that you have two eyeballs looking at slightly different angles. In short, the graphics library renders two different images, one for each eye, that have a parallax shift. These images are then delivered to each eye separately. The method by which the images are segregated varies from technology to technology. We will discuss these in a little bit.

The last item on the list, accommodation of the eyeball, is the process by which your eye changes shape to focus at different distances. By correlating the shape of your eye with the distance to the object, accommodation works as one of the pieces of information your brain uses to determine depth. As current 3D displays still use a 2D screen, the eyes are still focusing on the same plane regardless of the object's perceived depth; therefore, the eighth item in the list is not recreated. This is why most 3D displays still do not seem completely real. Some technologies, such as holograms and volumetric displays, allow for accommodation of the eyeball, but usually at the expense of some other factor. We'll touch on these beyond state-of-the-art technologies near the end of the chapter.

Stereoscopic Basics

There are some extra considerations when it comes using today's 3D display technologies to recreate the images that would usually be provided to the visual cortex by binocular vision. Normally two eyes create two images that the brain combines with a biological depth map. The earliest stereoscopic images were generated in the 1800s from two photographs taken from slightly different positions. The viewer would then look at the photos through what came to be known as a stereoscope. This device was essentially an early example of the View-Master that some of you might remember from childhood. While the principle of showing unique images to each eye is straightforward in this case, it doesn't allow group viewing and requires that the user have something pressed against his eyes. To make 3D display something that a group of people can all experience together and in some cases even without the aid of any headgear, we must look at some more sophisticated methods of segregating the right and left images.

The Left and Right Frustums

If you are familiar with computer graphics, the concept of the *viewing frustum* is not alien to you. If you aren't, we'll take a second to go over it, but it might be worthwhile to read about it in detail before you continue. The viewing frustum is the region of space in the model world that the camera can see from its given position in that world. In a normal 3D graphic rendering, the frustum is clipped by a near plane that represents the screen distance. In essence, you cannot render something closer to the user than the screen plane. If you remember things jumping out of the screen in the last 3D movie you saw, you can probably guess that when we are using stereoscopic rendering, this no longer holds true. A normal computer graphics frustum is shown in [Figure 24-2](#).

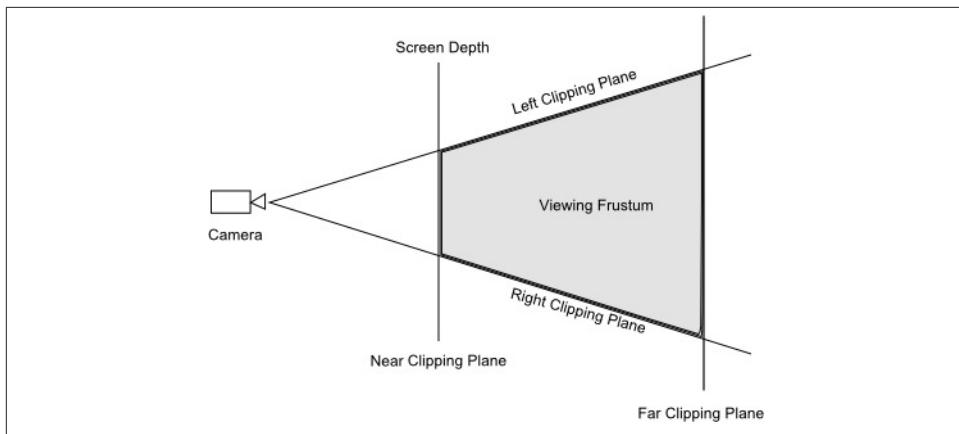


Figure 24-2. Normal viewing frustum

When using a stereoscopic 3D display library, we no longer have a single viewing frustum. Instead we have two cameras that are horizontally displaced from the 2D camera, as shown in [Figure 24-3](#).

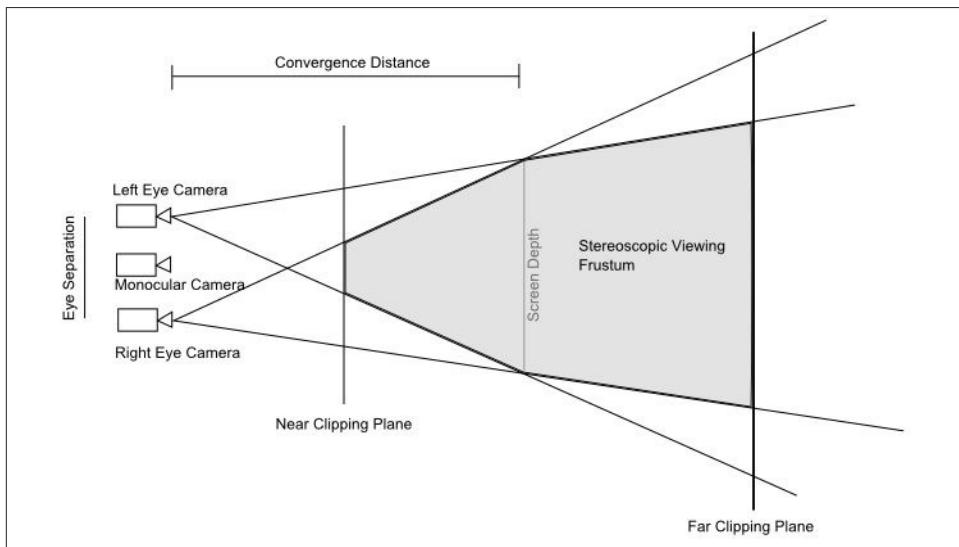


Figure 24-3. Viewing geometry of a stereoscopic display

These two cameras, offset from the monocular camera, generate a left and right frustums. As you can see, there is a location where these two frustums intersect; this is called the *convergence distance*. Objects that are placed at the convergence distance will have the same appearance to both cameras. Note that the cameras are all pointed in the same direction; this is called the *off-axis method*. This requires the frustums to be asymmetric, which most modern graphics libraries support. Now, at first glance, it might be tempting to toe-in the two frustums so that each camera's frustums are symmetrical, as shown in [Figure 24-4](#).

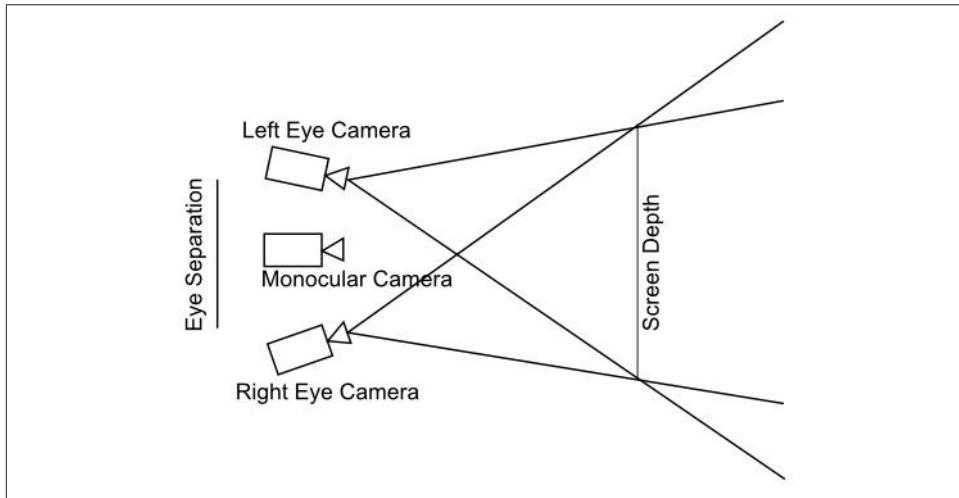


Figure 24-4. Toe-in method (incorrect)

This will create workable stereopairs, but along with the horizontal parallax will introduce some vertical parallax. This can cause eyestrain to the viewer and should be avoided. Instead, the off-axis technique should be used; it is illustrated in [Figure 24-5](#). One of the objects is beyond the screen in the background, and one is in front of the screen.

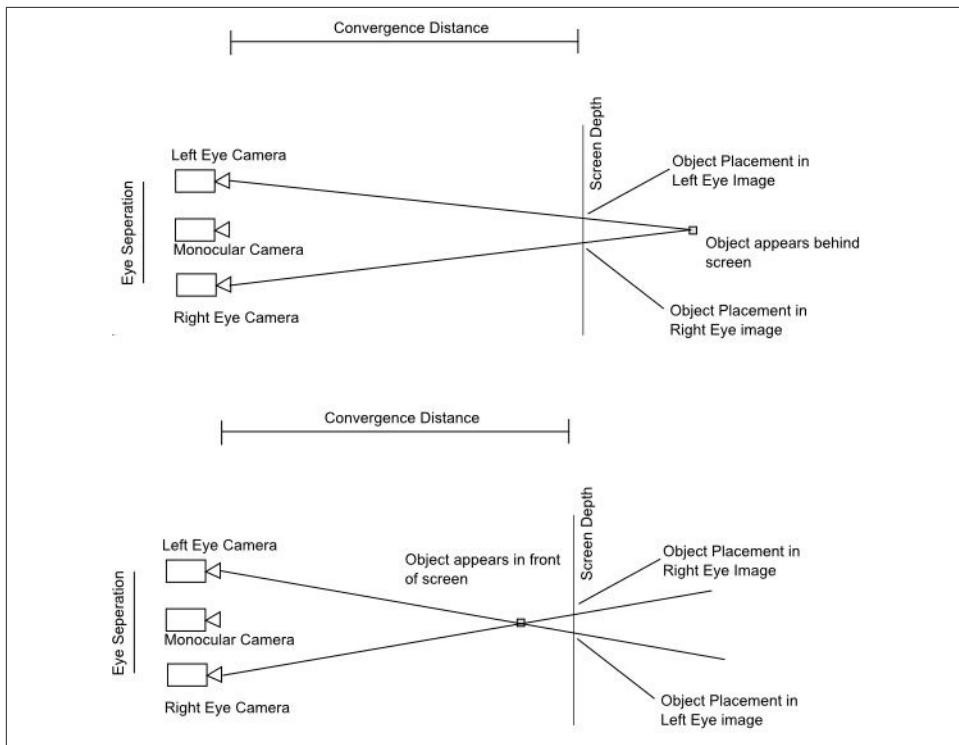


Figure 24-5. Off-axis method (correct)

You can see that if you wish to move something to greater than screen depth, the object must be shown farther to the left than if it were at screen depth for the left eye image. For the right eye image, the object must be shown farther to the right. If you want to show something coming out of the screen, the opposite is true. The left eye will see the object as farther to the right than if it were at screen depth. Also note that each object will have a slightly different angle as well. Again, the distance between the right eye image placement and the left eye image placement is referred to as parallax. The amount and relative orientation of parallax is the chief way your brain creates 3D images. In fact, the most important aspect of the physics of stereoscopic display for programmers to understand is that there is a parallax budget that they must use wisely in developing programs that take advantage of 3D display. This budget defines the ranges of parallax that your viewer's brain will be able to accept comfortably. We'll discuss this in detail at the end of the chapter.

For now, we'll consider that if we were to just show the right and left images on a screen without further work, you'd end up seeing two images with both eyes and no 3D effect would be produced. It is paramount that the image intended for the left eye is seen only by the left eye and vice versa. These two channels, the left and right eye, must be kept

as separate as possible. Let's see what the current options are for achieving such separation.

Types of Display

As just explained, 3D display technology depends on providing two distinct images, one to each eye. In the next sections we'll discuss the common techniques used today.

Complementary-Color Anaglyphs

Anyone who saw a 3D movie in the 80s remembers the cheap red and blue glasses one had to wear to get the effect. These were *complementary-color anaglyphs*. An anaglyph is the technique of encoding the separate images in a single photograph or video frame using color filters. The method calls for two horizontally shifted images to be viewed simultaneously. The images will contain the two images tinted in opposite colors of the scheme. While there are many color combinations that can be used, the most common today are red and cyan. These colors are chosen because the cyan and red filters are the most exclusive. Red and green filters were used earlier, but the green filter allows too much red light to leak through. This can cause what is called *binocular rivalry*, where your brain has a hard time figuring out which depth map to use. One way to illustrate this is via the simple drawing of a transparent cube, as shown in [Figure 24-6](#).

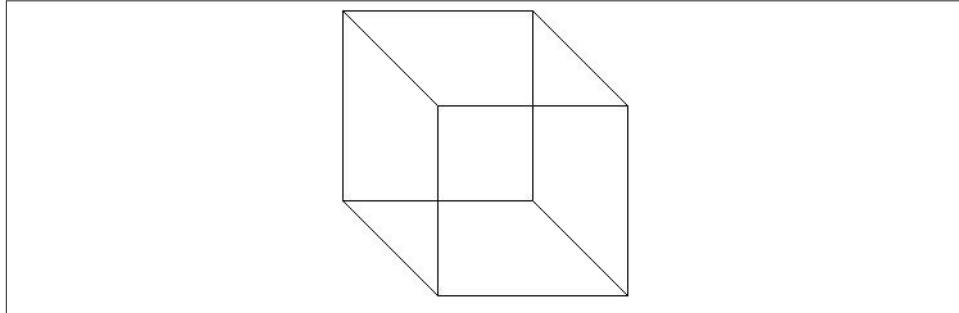


Figure 24-6. Cube demonstrating binocular rivalry

If you focus on the cube in [Figure 24-6](#), your brain may start to flip between interpreting the upper face as forward of the lower face, and the lower face being forward. While this is caused by incomplete depth cues, the same uneasy feeling can be caused when your brain receives leaks across the two channels in a stereoscopic display. As you can imagine, this would be pretty annoying during a video game.

As the glasses don't require any electronics to do this, it is an example of passive 3D technology. The major drawback of this method is that the red component of the images is muted to the viewer. There are many improvements that can be made to the system

to correct the color and account for some fuzziness. One example is the patented ColorCode 3D, which uses amber and blue filters. The advantage of this system is a nearly full color space and a fairly good image when not viewed with the glasses.

Anaglyphs fell out of favor with movie and game producers when polarization techniques came into maturity. These produce better color reproduction and reduce eyestrain. However, given the relatively inexpensive glasses required and that nothing special is required of the display other than that it be capable of displaying colors, anaglyphs have had a resurgence in printed material and online.

Linear and Circular Polarization

As polarized light plays a very important part in the largest 3D displays, movie screens, we'll review what polarization of light means and how to accomplish it. Light can be thought of as an electromagnetic wave traveling through space. Let's begin our discussion by considering a common light bulb. It emits electromagnetic waves in all directions and is nominally "white." An electromagnetic wave oscillates perpendicularly to its line of travel. This is called a *transverse wave*. In comparison, sound waves oscillate in the same direction they travel, creating regions of higher density. These are called *longitudinal waves*. Only transverse waves can be polarized because only transverse waves have oscillation in multiple orientations. Going back to our light bulb, it is emitting "dirty" light in that the electromagnetic waves are all at random orientations.

Most sources of electromagnetic radiation (i.e., light) are composed of many molecules that all have different orientations when they emit light, so the light is unpolarized. If that light passes through a polarization filter, it leaves the filter with the oscillations in only one direction. In fact, there are two types of filters. *Linear filters* produce oscillations in one direction. *Circular filters* (a special case of elliptical filters) create circularly polarized light that rotates in an either righthand or lefthand direction. As circular filters depend on linear filters first, we'll discuss those now.

The simplest and most common linear filter is the wire-grid polarizer. Imagine many very fine wires running parallel to one another with small gaps between them, as shown in [Figure 24-7](#). When unpolarized light hits the wires, the oscillations that are parallel to the waves excite the electrons in the wire and move them along the length of the wire. This phenomenon causes that component of the oscillations to be reflected. However, the electrons cannot easily move perpendicular to the length of the wires, so the reflection phenomenon doesn't occur. What we are left with on the far side of the filter is a beam of light with the oscillations all in the same direction.

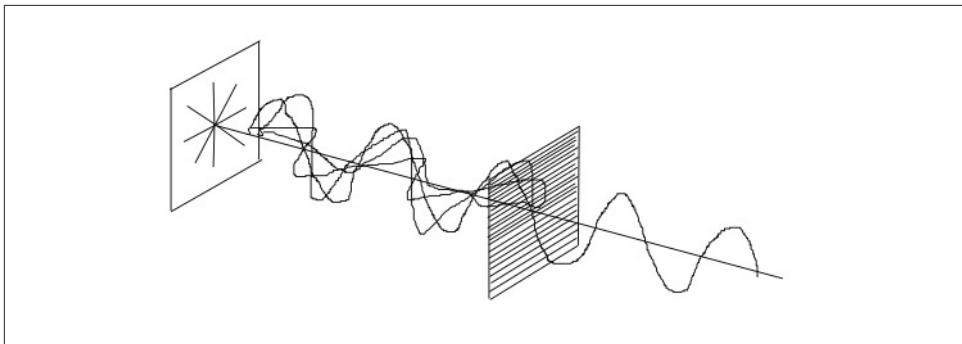


Figure 24-7. Wire-grid polarization

Early 3D display systems used linearly polarized light to separate the right eye channel from the left eye channel. However, there is one problem with using linear polarizers. It follows that if you place another wire-grid polarizer after the first, with its wires rotated 90 degrees, no light will pass through! In fact, if you have an old pair of sunglasses or 3D glasses and you hold the right eye lens against the left eye lens, you won't be able to see anything. That is because each filter is blocking out one direction of oscillations, preventing any light from coming through. If you rotated one of the lenses, then the combined lens will lighten as you align the polarization directions. The problem with these types of lenses is that if you were watching a movie and tilted your head to one side, the same effect would occur and the image would be greatly dimmed. This means your date could no longer rest his or her head on your shoulder while watching the movie. Something had to be done.

Circular polarization is another form of filtering out certain orientations so that you can control which light beams pass through which lens. However, in this case the direction of oscillation is not a single orientation but more accurately a pattern of oscillations parameterized by time. The first step to achieve circular polarization is to send the light through a linear polarizer as just discussed. That light is then sent through what is known as a *quarter-wave filter*. A typical arrangement is shown in [Figure 24-8](#).

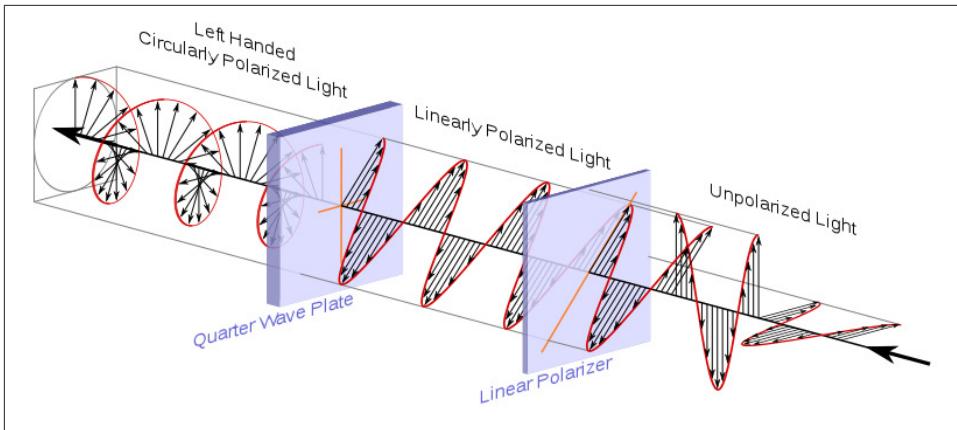


Figure 24-8. Circular polarization filter (public domain image by Dave3457; <http://commons.wikimedia.org>)

First a linear polarizer rotated to 45 degrees accepts incoming light and polarizes as we discussed earlier. The circular polarization effect is accomplished when a light wave polarized at 45 degrees hits the filter that accepts both 0- and 90-degree oscillations. As previously noted, this is called a quarter-wave filter. The resulting combination of 0- and 90-degree components of the intermediate 45-degree beam results in oscillations that turn right or left in a regular pattern. Patterns that turn counterclockwise are called left-handed. Patterns that turn clockwise are called right-handed.

The main benefit is that the lenses create the same pattern regardless of their rotation about the center of the lens. In other words, if you rotated the assembly shown in **Figure 24-8**, meaning both lenses about the center axis, there will be no change in the polarization. This reduces the effect of head position on the viewer's ability to fuse the right and left eye channels, reducing eyestrain and increasing comfort. As a side note, it is also required for use in digital cameras, as linear polarization would affect the autofocus and light-metering features of SLRs.

Like anaglyphs, polarized 3D systems also use glasses to separate two channels that are projected at the same time. The first systems used two projectors, each with a different linear polarization filter projecting on to the same screen with precise timing. As the glasses would allow only the correctly polarized light to be seen by either eye, the viewer perceived binocular disparity. However, the precise timing between the projectors would be subject to errors that cause eyestrain and binocular rivalry. Newer systems, including RealD, use an active polarization filter fitted to the projector. However, this is still classified as a passive system, because the glasses the user has are just normal passive filters. In this system, there is a single filter that can change its polarization up to 200 times a second. Every other frame is separately polarized, and binocular disparity is experienced without the complexity of an additional projector. Although this system

uses an active filter, the glasses don't have to actively change to separate the two channels, so this is another example of passive technology.

The main benefit of polarized systems over anaglyphs is that they provide full-color viewing and avoid binocular rivalry. This can increase the viewers' comfort when they are watching feature-length films. The disadvantages are cost and dimness. The glasses cost much more, and the complexity of projection is increased. It is impossible to create the effect in static media like print or web pages using standard displays. Also, because the lens on the projector is blocking out the portions of the light that don't have the correct polarization, the images appear dimmer to the viewer. This can cause up to 30% reduction in brightness and is the main point of contention for many directors.

Liquid-Crystal Plasma

The other display technologies discussed were passive technologies. The projection carries the two channels and the glasses separate the channels, one for each eye, without active participation from the glasses. Active technologies require that the glasses do the work of separating the channels while the display is less important. As the gaming industry is more sensitive to adapting 3D display technologies to work with existing computer monitors or TVs, it has generally focused on active technologies. The most common active technologies are based on *liquid-crystal shutter glasses*, or LC glasses. The LC glasses work by exploiting a property of liquid crystals that causes them to turn black when a voltage is applied to them. This is the same technology that creates the eight-segment digits on a simple calculator.

Basically, every other frame being displayed is shown only to one eye, as the LC glasses cause the lens to darken when the opposing eye's frame is being displayed. To make sure the glasses are preventing the correct image from being seen by the corresponding eye, the computer broadcasts a timing signal to the glasses either over a wire or wirelessly. At the appropriate time, the right eye lens has a voltage applied to it and the entire lens quickly turns black. As light can enter only the left eye, that eye sees the image on the screen. As the video being displayed moves to the next frame, this time for the right eye, the glasses simultaneously are triggered to remove the voltage from the right lens and apply it to the left lens. With the left lens now darkened, only the right eye sees the right eye image. As long as this is happening very quickly,—on the order of 60 times per second per eye, or a total refresh rate of 120 Hz—your brain can't detect that only one eye is seeing the information on the display at a time. Instead, it interprets it as each eye seeing distinct images continuously, and as long as the image follows the rules we discussed earlier, it interprets it as having depth.

As you can imagine, the main disadvantage of this technology for gaming would be that you have to ensure the frame rate stays relatively high. You are now rendering twice as many images as you normally would require. We'll discuss more about the rendering pipeline later. Also related to frame rate is the refresh rate of the display. As each eye is really seeing only half the frames, the overall frame rate is half whatever the refresh rate is on the screen. Older displays have refresh rates at 60 Hz and effectively halving that can create issues with eyestrain; however, new displays support 120 Hz refresh rates so that halving it still allows for smooth display. Also, the display will seem much dimmer with the glasses on, as your eyes are seeing, on average, only half the light they normally would. As you can see, dimness is a common problem among 3D display technologies.

The main advantage is that you don't need a special display. As long as your display is capable of the required refresh rate, then you can retrofit it with a pair of LC shutter glasses and get 3D display out of it. Nvidia released such a kit in 2008, called 3D Vision, that is relatively popular. Graphics cards are capable of automatically converting the depth of the object in the model world into a parallax so that older games can also be rendered in stereoscopy. This is something to consider as you design your next game and something we'll touch on again in a moment.

Autostereoscopy

While the newer 3D technologies are a far cry from the 3D of several decades ago, they still rely on people wearing glasses to view the image. This means the displays can never be used in a casual setting such as an arcade or street advertisement. Also, the younger segment of the gaming population might break or lose the glasses. Having to put on glasses to view the 3D images also provides a signal to your brain that what you are about to view is optical illusion. Put on the glasses, and your brain is already thinking that this isn't really in 3D.

Autostereoscopy endeavors to create the illusion of depth without the aid of any glasses or other wearable device. The first and most common way it does this is by introducing a *parallax barrier* between the display and the user. As discussed earlier, parallax—and by extension, binocular disparity—is what gives our brain a main source of information on depth. A parallax barrier uses the fact that each of your eyes sees things from a slightly different angle to separate the two channels required for stereoscopy. Physically, the barrier is a layer placed in or on the screen with a series of very precisely cut slits. Because your eyes are not in the same spot, the slits reveal different pixels on the screen to each eye. A basic illustration of this method is shown in [Figure 24-9](#).

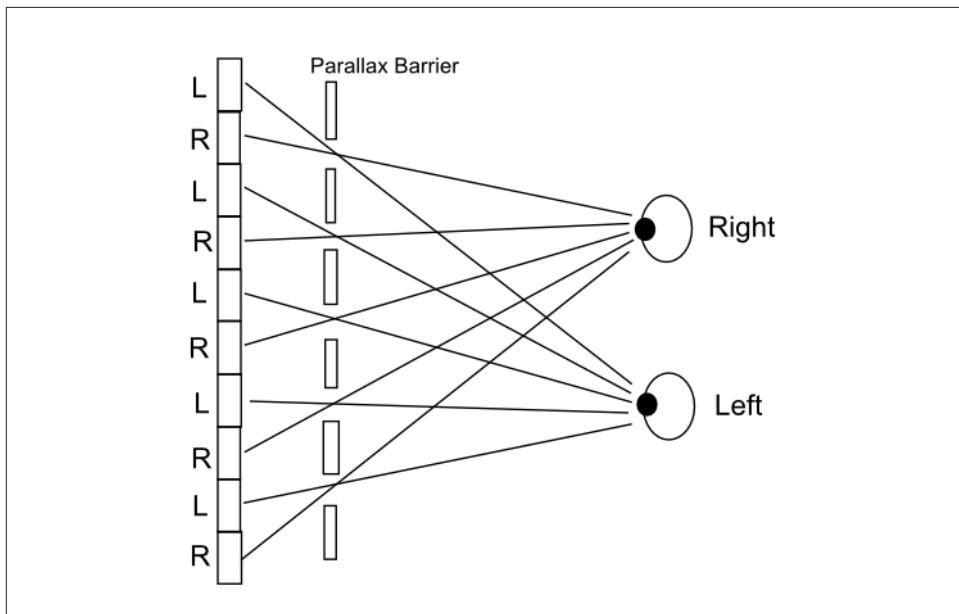


Figure 24-9. Parallax barrier

As shown in this figure, older screens used the slits to bar certain pixels from being seen by placing them above the screen. Newer screens like the one on the Nintendo DS place the barrier lower than the pixels, but before the backlight. This prevents your eyes from receiving the light from those pixels that are being shaded by the solid spaces between the slits. This results in a clearer image and a wider viewing angle.

Speaking of viewing angle, if the method works because your eyes aren't in the same spot, it is obvious that if you move your whole head then you'll also be seeing a different set of pixels. This is the drawback: that there is a relatively small area called the "sweet spot" that the user must position his head relative to the screen to perceive the 3D effect. It makes it inappropriate for movies, as only a portion of the seats would be in the sweet spot, but it is in use for handhelds where only one user will be viewing the screen at any given time. Another drawback is that because the slits are eliminating half the pixels from each eye, the system reduces the effective pixel count by one-half. This causes a reduction in resolution that can be countered by even higher pixel density.

Another method very similar to the parallax method is replacing the layer of slits with a series of lenses that direct light from certain pixels to a certain eye. These are called *lenticular lenses* and are illustrated in [Figure 24-10](#).

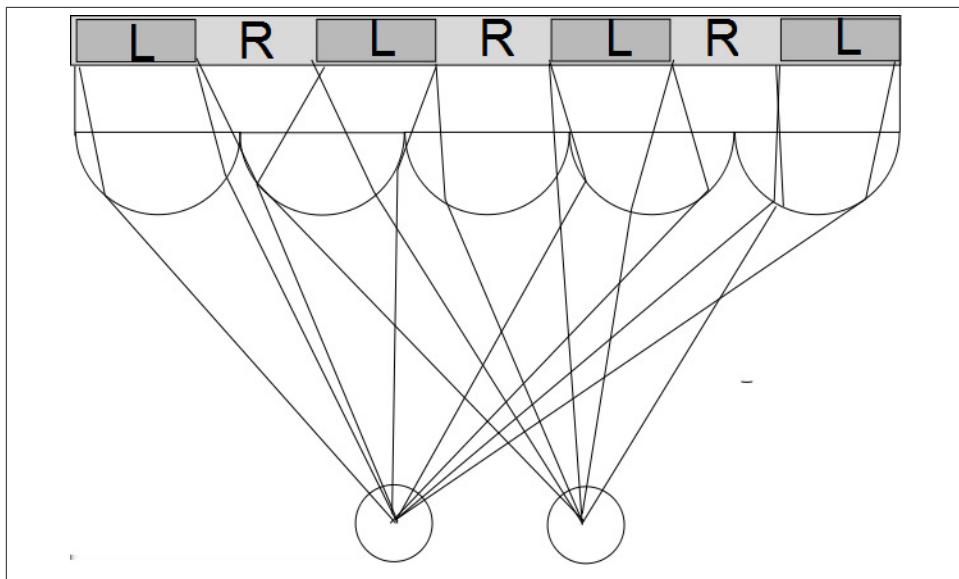


Figure 24-10. Lenticular lens screen

Here microscopic domed lenses are placed between the viewer and the screen. The lens focuses the light such that only certain pixels are seen by each eye, due to the slightly different angles by which the eyes view the lens. Benefits over the parallax barrier are that the position of the user is less restricted and the image is brighter. With both parallax barriers and lenticular lens arrays, it is possible to retrofit current screens with removable slide-in-place filters that allow for viewing of 3D content designed for use with those filters. At the time of writing, several large TV manufacturers are doing active research into widening the field of view of these technologies for use in a home entertainment environment.

Advanced Technologies

The displays we have discussed so far have all lacked some level of realism. For one, the eye doesn't have to refocus to observe objects at different depths, so your brain isn't totally fooled. Additionally, when you move around the object being projected and it doesn't change view, you still see the object at whatever angle the object was recorded. If you were viewing the world through a window, you could walk to the right and see more of the left-handed view. However, try as you might, you can't see around the corner of a building in a video game by moving your head at an angle of the screen. While it might be possible to recreate that effect with some sort of head tracking, there are some beyond-state-of-the-art technologies that could take this steps further.

One technology that is commonly thought of being able to produce 3D images is the *holograph*, a staple of science fiction. It seems like we should be able to just whip up

some dynamic holographs and be done with screens all together. Who wouldn't like to play a sports game as if it were a table-top miniature? However, due to the way they are recorded, holographs as we know them are static images. Once recorded, holographic images cannot be changed. Due to their ability to encode multiple angles of viewing, they would make wonderful display technology, and research is under way to find a material that can hold holographic data and be rewritten fast enough to induce the illusion of motion to the viewer. From time to time, you may see in the news some event incorporating computer-generated imagery displayed in what is called holographic display. These are not true holographs, but usually just a projection on a semitransparent screen. The images are still completely flat, and the illusion of 3D comes solely from the brain not registering the presence of the screen.

Another technology being actively researched is called *integral photography*. This is a lot like the use of a lenticular lens; however, instead of linear cylinders in an array, the lens field is more like a fly's eye. Each lens in the array captures a complete picture from a slightly different angle. Now, when projected through a similar integral lens, the light forms a 4D field that the viewer sees as a 3D scene appropriate for his or her viewing angle. If the view moves to the side, then he or she will see a new portion of the object that wasn't visible previously. This type of movement parallax creates very realistic 3D experiences. The advanced displays so accurately recreate the light that recorded the images that the eye can focus on different parts of the object (this is called the *wave front*) and therefore experience accommodation of the eyeball. Recall that this is the eighth item in our list from earlier in the chapter, and it is something the other displays are lacking. Some crude demonstrations of this technology have been presented, and it will be exciting to see how the research progresses.

Beyond any other method, the last one we'll talk about takes the bull by the horns. If you want a 3D image, just make the image three-dimensional. The other displays we discussed all attempt to recreate 3D screens using projection from a 2D surface. There is a group of display technologies known as *volumetric* that dispense with any 2D elements and attempt to create a light field with well-defined x , y , and z coordinates. These displays are far enough away from consumers that the definition of a volumetric display is still being argued. One of the biggest problems with the technology will be *occlusion* —that is, when an object passes in front of another object, you can't see the object behind the object that is closer to you. Pretty basic depth information, right? Well, if you are attempting to create a 3D light field, it is difficult to get the light to be blocked out when another rendered object passes in front of the original object. Simply not creating light there won't work, as each viewer would expect the farther object to be blocked at different angles. There are some existing demonstrators that use lasers to excite electrons in the air. When the lasers are focused on the same three-dimensional point, the combined energy creates a small pocket of plasma that gives off light. These small volumes of light are often referred to as *voxels* and correspond to pixels in 2D display technology. The current resolution and refresh rate is not going to be wowing any gamers in the near

future, but I for one can't wait for the day I can watch the New Orleans Saints play as a three-dimensional table-top game.

Programming Considerations

Now that you have a background in how the current 3D display technologies work, there are some aspects of each that you as the programmer should consider when writing games. There are two ways to add 3D stereoscopic content to games: active stereoization and passive stereoization. These are not to be confused with the passive and active technologies for viewing the 3D images. The stereoization process is the method by which the 3D images get created in the first place.

Active stereoization is the process by which the programmer creates two cameras, rendering separate images for each eye. *Passive stereoization* removes the requirements for two cameras, and adds the stereoization at the GPU level. Either method is going to cost something in performance. The worst-case cost is twice a monocular scene; however, some elements of the scene, like the shadow map, will not require recalculation for each eye.

Active Stereoization

Active stereoization is conceptually simpler and offers greater control over the process of stereoization. The most naive implementation is to simply have two cameras that render complete scenes and then pass the buffers labeled one for each eye. The buffers are then swapped in and out with the traditional definition of frame rate being half the actual frame rate. However, this simple implementation duplicates some elements of the scenes that are not eye dependent.

The advantage of this technique is precise control of what each eye is seeing. This allows the programmer to determine eye separation for each frame and could be used to actually disorient the user as a game element. Consider a flash-bang grenade going off: the programmer could alter the position of the cameras such that it would disorient the user in 3D for a short period after detonation. However, this technique would cause very real discomfort to the user, so it should be not used frequently!

The disadvantages are that the programmer is now responsible for managing an extra camera that must be rendered for each frame. For commercial titles, this method can be difficult considering that most games already have to be careful about how many times they invoke the render pipeline in order to maintain playable frame rates. Having to manage two cameras creates additional runtime burden on the program and makes the use of existing game engines a little more difficult.

Also due to the fact that not everyone's eyes have the same *separation* (intraocular distance) and not everyone's brain is willing to accept fabricated binocular disparity, the program must also provide options for the user to adjust the depth and complexity of

the stereo effect. If you do not provide a way for the user to tune the experience, a vocal minority will claim your game gives them a splitting headache. However, as stated before, if you are a curious amateur, the process of moving the camera and rendering stereoscopic images gives you a lot of insight into how the process works.

As discussed previously, we must take into account the intraocular distance of the viewer. This is the amount of parallax we want to give objects at infinity. This distance usually ranges between 3 cm and 6.5 cm. The large differences can arise when considering that you must consider both adults and children when creating your 3D scene. Now it is useful to develop a normalized measure of intraocular distance. Nvidia calls this *real eye separation* and gives the following formula for the value:

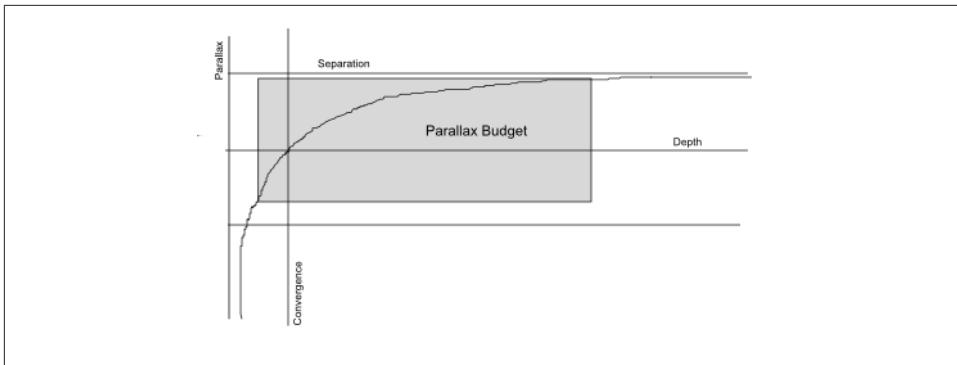


Figure 24-11. Parallax budget

The budget scales with convergence distance and separation. You should make sure as much of the important 3D action occurs between convergence and 10 times convergence. Your entire scene should be contained with negative convergence/2 to positive 100 times convergence.

In general, you must be most careful when trying to execute an out-of-screen effect. These effects are very impressive to the viewer but cause the most eyestrain due to the rapid change in parallax. Having the object first appear farther away than the screen and then moving it closer to the user provides a context for the brain and encourages fusion. If an object is going to be closer to the user than the screen, it is also important to prevent that image from being clipped by the edge of the screen. That would make a portion of the 3D object disappear, and the clipping always occurs at the convergence point. This will give conflicting cues to the viewer and cause the 3D effect to be diminished. Given the amount of control you need in order to prevent out-of-screen effects from causing viewer discomfort, it is often best to use in non-player-controlled scenes.

Another difficult part of the game to render is the 2D elements. The user interface or other menu items that have no depth are normally rendered at convergence depth. However, there are some elements that are 2D but should be rendered at some nonzero depth. The most important of these are mouse pointers and crosshairs. These should be rendered at the depth of the object below them. This change in depth of the user-controlled pointer helps maintain the idea that the objects are at different depths.

Passive Stereoization

Passive stereoization takes the responsibility for managing the stereoization process out of the programmer's hands. The programmer sends the render pipeline a single render command as usual, and the GPU handles generating the stereo images. Most systems rely on *heuristic subroutines* in the driver to take the monocular scene and generate binocular images. A heuristic subroutine is one that attempts to give a computer “com-

mon sense” about what it is trying to do to avoid having to do an exhaustive search for solutions to an existing problem. They are algorithms not based on rigid mathematical formulas but more like neural networks; they must be trained to do what you want them to. These algorithms decide which elements of the scene are eye dependent and which are not in a process that occurs entirely in the render pipeline.

It is possible for the programmer to defeat the “common sense” rules the computer is using in the pipeline so the method is not entirely fire-and-forget but does reduce a lot of the workload for development. One of the biggest benefits for larger game studios is that it avoids anyone having to reprogram existing game engines. By the same token, it allows existing games to be easily played in stereoscopic 3D. All of the preceding recommendations apply to passive stereoization except that there should be no user-adjustable settings in the game. Passive stereoization relies on third-party profiles that help the GPU do the work. The user will have set up a profile with whatever stereoization software he or she is using, such as NVIDIA’s 3D Vision. Other recommendations may be specific to the stereoizer, and manufacturers usually publish a best practices guide. The NVIDIA one is very helpful, and we recommend you read it if you are interested in using stereoization in your games.

CHAPTER 25

Optical Tracking

In previous chapters we discussed how accelerometers have changed the way that people interact with video games. The same sort of innovation is occurring with optical sensors. Cameras, both in visual and infrared spectrums, are being used to generate input for games. This chapter will focus on the Microsoft Kinect for Windows SDK and give an overview of how to make a simple game that combines optical tracking with physics. First we'll give a short introduction on the technologies these systems use to turn a camera into a tracking device.

Without getting too detailed, we should start by discussing a few things about digital cameras. First, most of us are familiar with the “megapixel” metric used to describe digital cameras. This number is a measure of how many pixels of information the camera records in a single frame. It is equal to the height of the frame in pixels multiplied by the width of the frame in pixels. A pixel, or picture element, contains information on intensity, color, and the location of the pixel relative to some origin. The amount of information depends on the bits per pixel and corresponds to the amount of color variation a particular pixel can display. Perhaps you've seen your graphics set to 16-bit or 24-bit modes. This describes how many colors a particular pixel can display. A 24-bit pixel can be one of 16.8 million different colors at any instant. It is commonly held that the human eye can differentiate among about 10 million colors; 24-bit color is called “true color,” as it can display more colors than your eye can recognize. You might also see 32-bit color modes; these include an extra 8 bits for a transparency channel that tells the computer what to do if this image were put on top of another image. This is sometimes referred to as *opacity* or *alpha*.

Optical tracking and computer vision, in general, work by detecting patterns in this wealth of pixel data. Pattern recognition is a mature field of computer science research. The human brain is an excellent pattern recognizer. For instance, look at [Figure 25-1](#). Most of us can't help but see a face in what is in reality a collection of three random

shapes. Our brains are so primed to recognize the basic pattern of a human face that we can do it even when we don't want to!

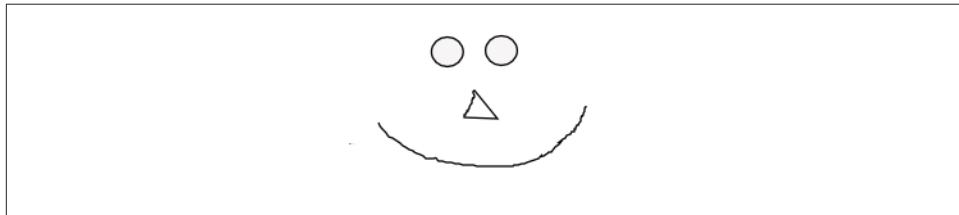


Figure 25-1. Four unrelated geometric entities

Computers, on the other hand, have a harder time looking at two circles and a few lines and saying, “Hey, this is a smiling face.”

Sensors and SDKs

The modern interest in computer vision as a consumer input for computer games has led to the development of several SDKs for performing computer-vision pattern recognition. One such system is Kinect for Windows. Although Microsoft provides a very high-level API with the Kinect, the downside is that you are locked into its hardware. The popular open source alternative is OpenCV, a library of computer-vision algorithms. Its advantage is that it can use a wide variety of camera hardware and not just the Kinect sensor.

Kinect

The Kinect was originally developed for the Xbox 360 but has recently been rebranded to include Kinect for Windows. As console game design has high entrance requirements, the Kinect for Windows allows more casual developers to try their hand at creating games with optical input. The system has a hardware component, called the Kinect sensor, and the previously mentioned Kinect SDK that does a lot of the heavy lifting for us in terms of pattern recognition and depth sensing. The hardware component consists of an infrared projector, infrared camera, visible light camera, and an array of microphones. The two cameras and the projector form the basis of the optical tracking system. The projector sends out infrared light that is invisible to humans. This light bounces off objects and is reflected back to the Kinect. The infrared camera records the reflected light pattern, and based on how it has been distorted, calculates how far the object is from the sensor. This exact method is carried out in the hardware of the sensor and is proprietary. However, the patent applications reveal that a special lens projects circles that, when reflected, become ellipses of varying shapes. The shape of the ellipse depends on the depth of the object reflecting it. In general, this is a much-improved version of

depth from focus, in which the computer assumes that blurry objects are farther away than objects in focus.

As for object detection, the Kinect comes with a great set of algorithms for skeleton direction. It can also be trained to detect other objects, but skeleton detection is really its forte. The skeleton detection is good because of the massive amount of training Microsoft used for the algorithms when creating the SDK. If you were to use an average computer to run the Kinect skeleton training program, it would take about three years. Luckily, Microsoft had 1,000 computers lying around, so it takes them only a day to run the training simulation. This gives you an idea of the amount of training you need to provide for consumer-level tracking in your own algorithms. The Kinect can track up to six people with two of them being in “active” mode. For these 2 people, 20 individual joints are tracked. The sensor can also track people while standing or sitting.

OpenCV

The OpenCV method for 3D reconstruction is, well, more open! The library is designed to work with any common webcam or other camera you can get connected to your computer. OpenCV works well with stereoscopic cameras and is also capable of attempting to map depth with a single camera. However, those results would not be accurate enough to control a game, so we suggest you stick with two cameras if you’re trying to use regular webcams.

Indeed, finding depth is relatively straightforward using OpenCV. The built-in function `ReprojectImageTo3D` calculates a vector for each pixel (x,y) based on a *disparity map*. A disparity map is a data set that describes how pixels have changed from one image to the next; if you have stereoscopic cameras, this essentially is the reverse of the technique we use in [Chapter 24](#) when dealing with 3D displays. To create a disparity map, OpenCV provides the handy function `FindStereoCorrespondenceGC()`. This function takes a set of images, assumes them to be from a stereoscopic source, and generates a disparity map by systematically comparing them. The documentation is very complete, and there are several books on the subject of OpenCV, including [*Learning OpenCV*](#) by Gary Bradski and Adrian Kaehler (O’Reilly), so we again will save the details for independent study.

Object detection is also possible with OpenCV. The common example in the OpenCV project uses *Harr-like* features to recognize objects. These features are rectangles whose mathematical structure allows for very fast computation. By developing patterns of these rectangles for a given object, a program can detect objects out of the background. For example, one such pattern could be if a selection rectangle includes an edge. The program would detect an edge in the pixel data by finding a sharp gradient between color and/or other attributes. If you detect the right number of edges in the right position, you have detected your object.

Hardcoding a pattern for the computer to look for would result in a very narrow set of recognition criteria. Therefore, computer-vision algorithms rely on a system of training rather than hard programming. Specifically, they use *cascade classifier training*.

The training process works well but requires a large image set. Typical examples require 6,000 negative images and 1,500 positive images. The negative images are commonly called background images. When training the algorithm, you take 1,200 of your positive images and draw selection rectangles around the object you are trying to detect. The computer learns that the pattern in the selection rectangles you've given it is one to be identified. This will take the average computer a long, long time. The remaining images are used for testing to ensure that your algorithm has satisfactory accuracy in detecting the patterns you've shown it. The larger the sample set, including different lighting, the more accurate the system will be. Once the algorithm is trained to detect a particular object, all you need is the training file—usually an *.xml* file—to share that training with another computer.

Numerical Differentiation

As previously noted, there are many ways to collect optical tracking data, but since we are focusing on the physics aspects, we'll now talk about how to process the data to get meaningfully physical simulation. By combining object detection with depth sensing, we can detect and then track an object as it moves in the camera's field of vision. Let's assume that you have used the frame rate or internal clock to generate data of the following format:

$$\{((x[i], y[i], z[i], t[i]), (x[i+1], y[i+1], z[i+1], t[i+1])) , \\ ((x[i+2], y[i+2], z[i+2], t[i+2])), \dots\}$$

Now, a single data point consisting of three coordinates and a timestamp doesn't allow us to determine what is going on with an object's velocity or acceleration. However, as the camera is supplying new position data at around 20–30 Hz, we will generate a time history of position or displacement. Using techniques similar to the numerical integration we used to take acceleration and turn it into velocities and then turn those velocities into position in earlier chapters, we can apply numerical differentiation to accomplish the reverse. Specifically, we can use the finite difference method.

For velocity, we need a first-order finite difference numerical differentiation scheme. Because we know the current data point and the previous data point, we are looking backward in time to get the current position. This is known as the *backward difference scheme*. In general, the backward difference is given by:

two data points and has a nonzero, fixed value. Therefore, the equation can be rewritten as:

might be required to provide a stable differential. Of particular note with central difference forms is that periodic functions that are in sync with your time step may result in zero slope. If the motion you are tracking is periodic, you should take care to avoid a time step near the period of oscillation. This is called *aliasing* and is a problem with all signal analysis, including computer graphics displays. Also, note that this cannot be computed until at least three time steps have been stored. In our notation, $t[i-1]$ is the center data point, $t[i-2]$ the backward value, and $t[i]$ the forward value. The acceleration function would therefore be as follows:

```
Vector findAcceleration (x[i-2], y[i-2], z[i-2], t[i-2], x[i-1], y[i-1], z[i-1],
                        t[i-1], x[i], y[i], z[i], t[i] ){
    float ax, ay, az, h;
    vector acceleration;

    h = t[i]-t[i-1];

    ax = (x[i] - 2*x[i-1] + x[i-2]) / h;
    ay = (y[i] - 2*y[i-1] + y[i-2]) / h;
    az = (z[i] - 2*z[i-1] + z[i-2]) / h;

    return acceleration = {ax, ay, az};
}
```

Now, let's say that you are tracking a ball in someone's hand. Until he lets it go, the velocity and acceleration we are calculating could change at any moment in any number of ways. It is not until the user lets go of the ball that the physics we have discussed takes over. Hence, you have to optically track it until he completes the throw. Once the ball is released, the physics from the rest of this book applies! You can then use the position at time of release, the velocity vector, and the acceleration vector to plot its trajectory in the game.

CHAPTER 26

Sound

In this chapter we'll explore some basic physics of sound and how you can capture 3D sound effects in games. We'll refer to the *OpenAL* audio API for some code examples, but the physics we discuss is independent of any particular API. If you're new to OpenAL, it's basically OpenGL for audio. OpenAL uses some very easy-to-understand abstractions for creating sound effects and handles all the mixing, filters, and 3D synthesis for you. You basically create sound *sources*, associate those sources with *buffers* that store the sound data, and then manipulate those sources by positioning them and setting their velocity (among other properties). You can have multiple sources, of course, but there's only one *listener*. You do have to set properties of the listener, such as the listener's position and velocity, in order to properly simulate 3D sound. We'll talk more about these things throughout the chapter.

What Is Sound?

If you look up the definition of sound online, you'll get answers like sound is a vibration; a sensation perceived by our brains through stimulation of organs in our inner ear; and a density or pressure fluctuation, or wave, traveling through a medium. So which is it? Well, it's all of them, and the interpretation you use depends on the context in which you're examining sound. For example, noise control engineers aiming to minimize noise on ships focus on vibrations propagating through the ship's structure, while medical doctors worry more about the biomechanics of our inner ear and how our brains interpret the sensations picked up by our ears, and physicists take a fundamental look at density and pressure fluctuations through compressible materials and how these waves interact with each other and the environment. We don't mean to suggest that each of these disciplines views sound only in a single way or context, but what we're saying is that each discipline often has its own perspective, priorities, and standard language for the subject. To us, in the context of games, sound is what the player hears through his speakers or headphones that helps to create an immersive gaming environment. How-

ever, in order to create realistic sounds that lend themselves to creating an immersive environment, complimenting immersive visuals and in-game behaviors, we need to understand the physics of sound, how we perceive it, and what sound tells us about its source and the environment.

Given that this is a book on game physics, we're going to take a fundamental view of sound, which is that sound is what our brain perceives as our ears sense density and pressure fluctuations in the air surrounding us. These density and pressure fluctuations are waves, and as such we'll refer to *sound waves*. Let's take a closer look.

If a compressible medium experiences a pressure change—say, due to a driven piston—it's volume will change and thus its density will change. In the case of a driven piston, the region directly in front of the piston will experience the compression first, resulting in a region of increased density and increased pressure. This is called *condensation*. For sound, you can think of that piston as the cone of a loudspeaker. That region of increased density and pressure will propagate through the medium, traveling at the speed of sound in that given medium. [Figure 26-1](#) illustrates this concept.

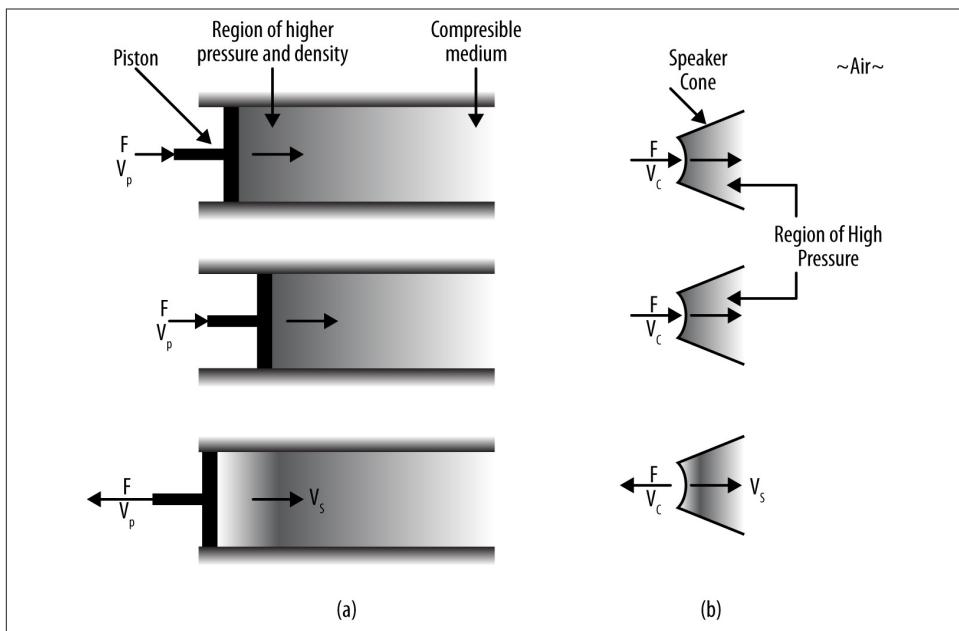


Figure 26-1. Driven piston and loudspeaker analogue

[Figure 26-1\(a\)](#) illustrates the driven piston concept, while [Figure 26-1\(b\)](#) illustrates the loudspeaker analogue. As the piston or cone displaces fluid (say, air), causing compression, and then withdraws, a single high-pressure region followed by a low-pressure region will be created. The low-pressure region resulting from withdrawal of the piston

is called *rarefaction*. That resulting solitary wave of pressure will head off through the air to the right in [Figure 26-1](#) at the speed corresponding to the speed of sound in air. If the piston, or speaker cone, pulses back and forth, as illustrated in [Figure 26-2](#), a series of these high-/low-pressure regions will be created, resulting in a continuous series of waves—a sound wave—propagating to the right.

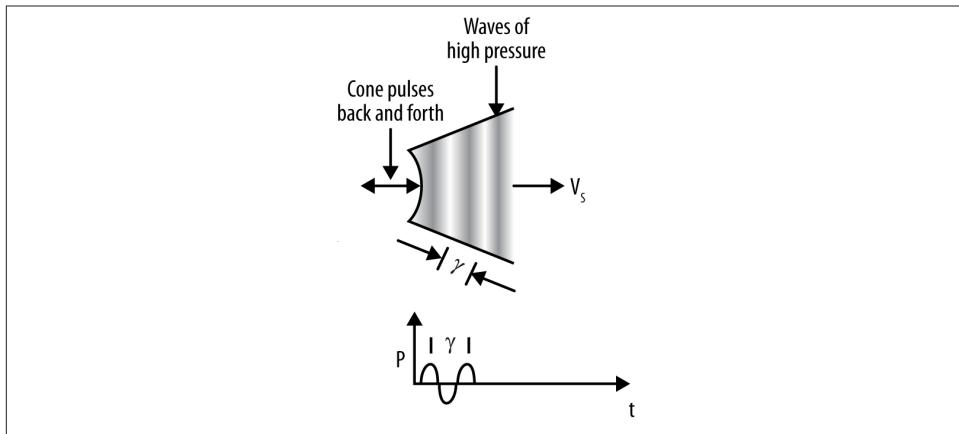


Figure 26-2. Sound wave

The *wavelength* of this sound wave (i.e., the distance measured from pressure peak to pressure peak) is a function of the frequency of pulsation, or vibration, of the cone. The resulting sound wave's frequency is related to the inverse of its wavelength—that is, $f = 1/\lambda$. The pressure amplitude versus time waveform for this scenario is illustrated in [Figure 26-2](#). We've illustrated the pressure wave as a harmonic sine wave, which need not be the case in reality since the sound coming from a speaker could be composed of an aggregate of many different wave components. We'll say more on this later.

One thing we do want to point out is that a sound wave is a longitudinal wave and not a transverse wave like an ocean wave, for example. In a transverse wave, the displacement of the medium due to the wave is perpendicular to the direction of travel of the wave. In a longitudinal wave, the displacement is along the direction of travel of the wave. The higher density and pressure regions of a sound wave are due to compression of the medium along the direction of travel of the wave. Thus, sound waves are longitudinal waves.

So, sound waves are variations in density and pressure moving through a medium. But how do we hear them? In essence the pressure wave, created by some mechanical vibration like that of a speaker cone, gets converted back to a vibration in our inner ear. And that vibration gets interpreted by our brains as sound—the sound we hear. [Figure 26-3](#) illustrates this concept.

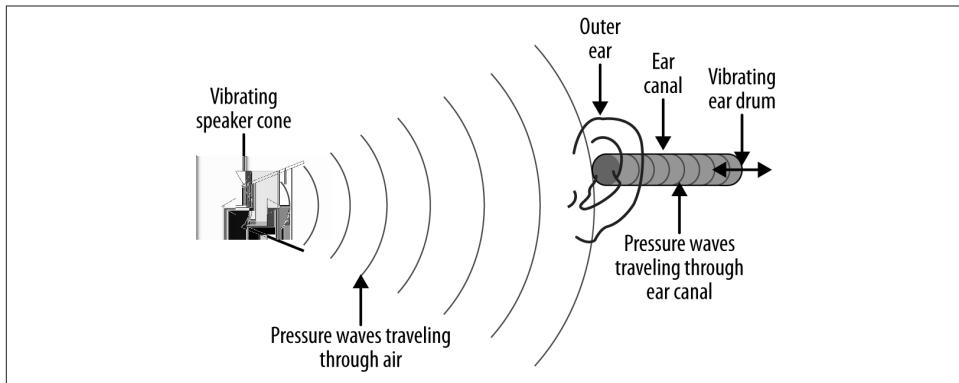


Figure 26-3. Illustration of how we hear

Our outer ears help capture and direct pressure waves into our ear canals. These pressure waves travel down the ear canal, impinging on the eardrum, which causes the eardrum to vibrate. This is where the pressure variations get converted back to mechanical vibration. Beyond the eardrum, biology and chemistry work their magic to convert those vibrations into electrical impulses that our brains interpret as sound.

Our ears are sensitive enough to detect pressure waves in the 20 to 20,000 hertz (Hz) frequency range. (A hertz is one cycle per second.) We interpret frequency as pitch. High-pitch sounds (think tweeters) correspond to high frequencies, and low-pitch sounds (think bass) correspond to low frequencies.

Aside from pitch, an obvious characteristic of sound that we perceive is its loudness. Loudness is related to the amplitude of the pressure wave, among other factors such as duration. We often think of loudness in terms of volume, or power, or intensity. All these characteristics are related, and we can write various formulas relating these characteristics to other features of the sound wave. Sound waves have kinetic energy, which is related to the mass of the medium disturbed by the pressure wave and the speed at which that mass is disturbed. Power is the time rate of change of energy transference. And intensity is related to how much power flows through a given area. The bottom line is that the more power a sound has, or the more intense it is, the louder it seems to you, the listener. At some point, a sound can be so intense as to cause discomfort or pain.

Customarily, intensity is measured in units of decibels. A decibel represents the intensity of a sound relative to some standard reference, which is usually taken as the sound intensity corresponding to the threshold of hearing. Zero decibels, or 0 dB, corresponds to the threshold of hearing. The intensity is so low you can't hear it. When sounds reach about 120–130 dB, they start to cause pain. **Table 26-1** lists some typical intensity values for common sounds.

Table 26-1. Typical sound intensities

Sound	Typical intensity
Jet airplane, fairly close	150 dB
Gun shot	160 to 180 dB depending on the gun
Crying baby	130 dB (painful!)
Loud scream	Up to 128 dB (world record set in 1988)
Typical conversation	50 to 60 dB
Whisper	About 10 dB

The intensity values shown in Table 26-1 are typical and surely there's wide variation in those levels depending on, for example, the type of aircraft, or the person you're talking to, or the softness of the person's voice whispering to you. It's common sense, but if you're writing a game you'll want to reflect some level of realism in the intensity of various sound effects in your game.

Now, intensity is actually a logarithmic scale. It is generally accepted that a sound measured 10 dB higher than another is considered twice as loud. This is perceived loudness. Thus, a crying baby is way more than twice as loud as a normal conversation. Parents already know that.

Characteristics of and Behavior of Sound Waves

Now that we've established what sound is, we're going to generally refer to *sound waves* throughout the remainder of this chapter. Remember, sound waves are pressure waves that get interpreted by our brains as sound. The bottom line is that we're dealing with waves, longitudinal waves. Therefore, we can use principles of wave mechanics to describe sound waves. Furthermore, since sound waves (think pressure waves) displace real mass, they can interact with the environment. We already know that pressure waves interact with our eardrums to trigger some biochemical action, causing our brains to interpret sound. Conversely, the environment can interact with the pressure wave to alter its characteristics.

Harmonic Wave

Let's consider a one-dimensional harmonic pressure wave—one that could be created by the driven piston shown in Figure 26-1(a). Let the x-direction correspond to distance, positive from left to right. Thus, the wave of Figure 26-1(a) travels in the positive x-direction. Let ΔP represent the change in pressure from the ambient pressure at any given time. Let A_p represent the amplitude of the pressure wave. Remember, the pressure will vary by some amount greater than the ambient pressure when condensation occurs to some amount lower than ambient pressure when rarefaction occurs. The range in

peak pressures relative to ambient is $-A_p$ to $+A_p$. Assuming a harmonic wave, we can write:

Superposition

In general, sound waves don't look like the pure harmonic wave shown in [Figure 26-4](#) unless the sound is a pure tone. A non-pure tone will have other wiggles in its plot resulting from components at other frequencies and phases. Similarly, if you record the sound pressure at a single point in a room, for example, where multiple sound sources exist, the pressure recording at the point in question will not correspond to the sound of any particular sound source. Instead, the recorded pressure time history will be some combination of all the sources present, and what you hear is some combination of all the sound sources.

A good approximation for how these various sound components combine is simply to sum the results of each component at the particular point in question. This is the principle of superposition.

[Figure 26-5](#) shows 10 different waveforms, each with different amplitudes, frequencies, and phases. The principle of superposition says that we can add all these waveforms to determine the combined result.

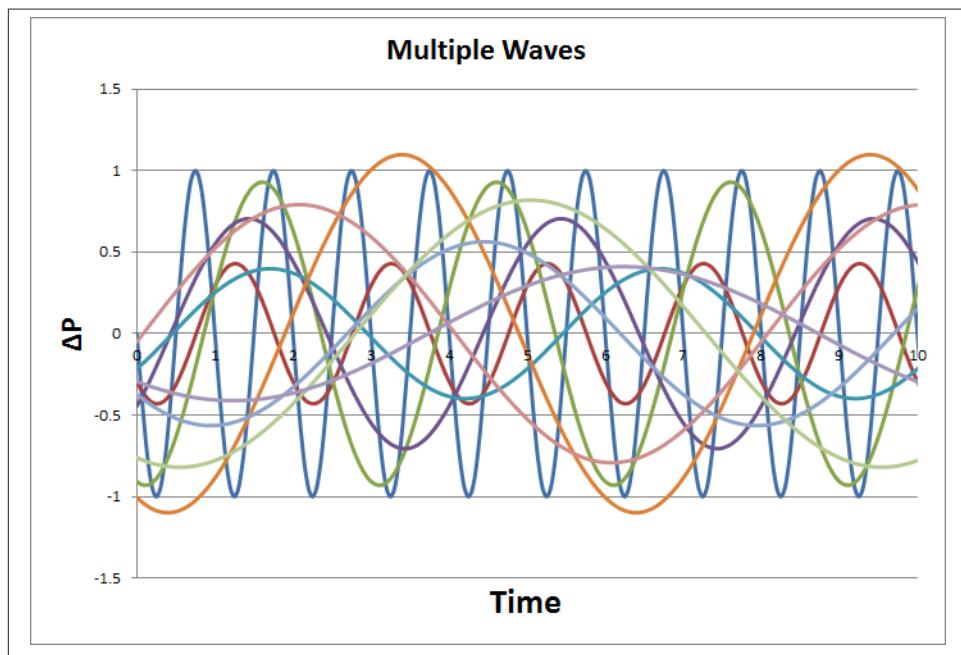


Figure 26-5. Ten different waves

Figure 26-6 shows the resulting wave. Note that the individual waves are added algebraically. At any given instant in time, some waves produce positive pressure changes, while others produce negative pressure changes. This means that some waves add together to make bigger pressure changes, but it also means that some can add together to make smaller pressure changes. In other words, waves can be either constructive or destructive. Some waves can cancel each other out completely, which is the basis for noise cancellation technologies.

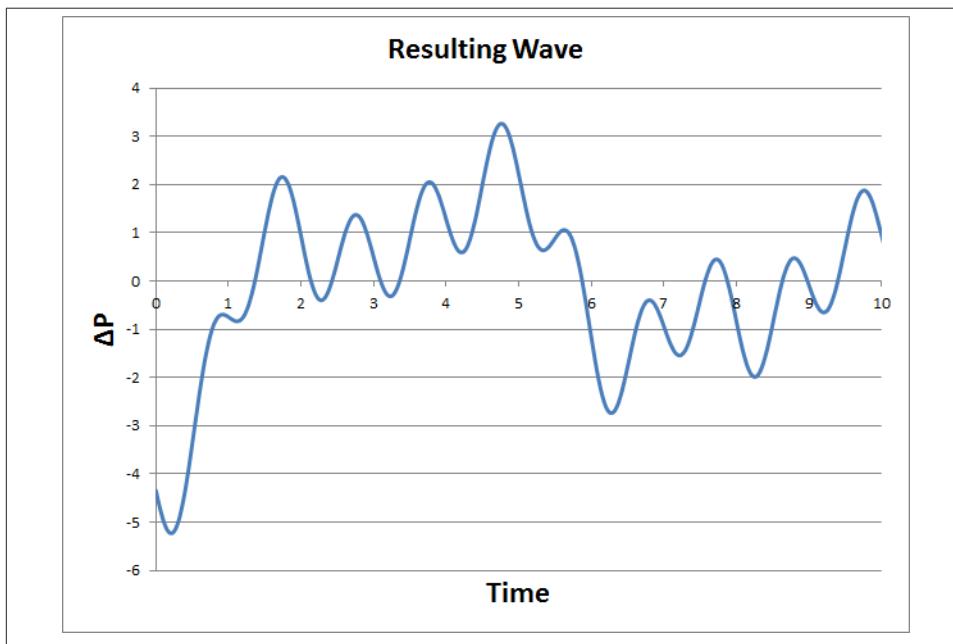


Figure 26-6. Resulting wave

Speed of Sound

Sound waves travel through a medium at some finite speed, which is a function of that medium's elastic and inertial properties. In general, sound travels faster in stiffer, less compressible mediums than it does in softer or more compressible mediums. For example, the speed of sound in air is about 340 m/s, depending on temperature, moisture content, and other factors, but it's about 1500 m/s in seawater. Water is a lot less compressible than air. Taking this a step further, the speed of sound in a solid such as iron is about 5,100 m/s.

You might say, “So what; why do I have to worry about the speed of sound in my game?” Well, the speed at which sound waves travel tells us something about the sound source, and you can leverage those cues in your game to enhance its immersive feel. Let’s say

an enemy unit fires a gun in your 3D shooter, and that enemy is some distance from your player. The player should see the muzzle flash before she hears the sound of the gun firing. This delay is due to the fact that light travels far faster than sound. The delay between seeing the muzzle flash and hearing the shot gives the player some sense of the distance from which the enemy is firing.

With respect to 3D sound effects, our ears hear sounds coming from an oblique direction at slightly different times because of the separation distance between our ears. That time lag, albeit very short, gives us some cues as to the direction from which the sound is coming. We'll say more on this later on this chapter.

Additionally, the Doppler effect, which we'll discuss later, is also a function of the speed of sound.

In OpenAL you set the desired sound speed using the `alSpeedOfSound` function, passing a single floating-point argument representing the sound speed. The specified value is saved in the `AL_SPEED_OF_SOUND` property. The default value is 343.3, which is the speed of sound in air at 20°C expressed in m/s.

Attenuation

Attenuation is the falloff in intensity of a sound over distance. Earlier we explained that sound intensity is related to how much power flows through a given area. Imagine a *point sound source*, which creates spherical pressure waves that propagate radially from the source. [Figure 26-7](#) illustrates this concept.

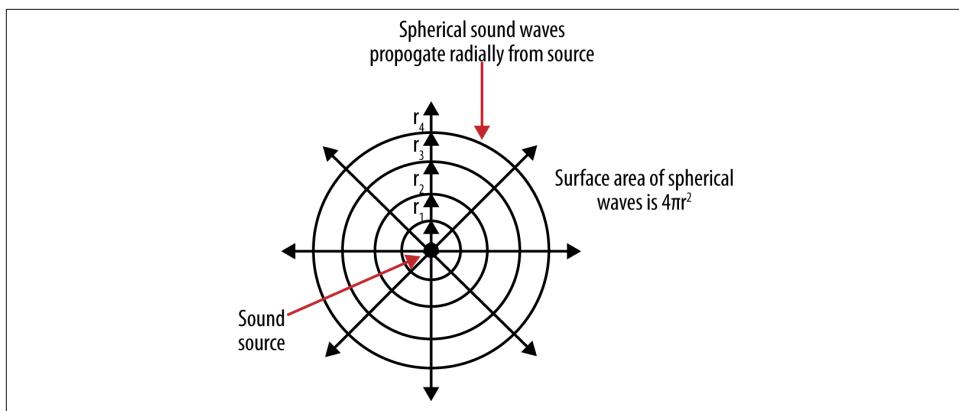


Figure 26-7. Spherical sound waves

Assuming the sound is being generated with a constant power, you can see that the area through which that power flows grows with increasing distance, r , from the source. Intensity is equal to power divided by area, thus the intensity at radius r_4 is less than that at, say, r_1 because the surface area at r_4 is larger. The surface area of a sphere is $4\pi r^2$. Without going into all the details, we can state that the amplitude of the spherical sound wave is inversely proportional to r^2 .

This is an ideal treatment so far. In reality attenuation is also a function of other factors, including the scattering and absorption of the sound wave as it interacts with the medium and the environment. You can model attenuation in many ways, taking into account various levels of detail at increasing computational expense. However, for games, relatively simple distance-based models are sufficient.

Attenuation provides another cue that tells us something about the sound source. In your game, you wouldn't want the intensity, or volume, of a sound generated far from the player to be the same as that from a source very close to the player. Attenuation tells the player something about the distance between him and the sound source.

OpenAL includes several different distance-based models from which you can choose. The OpenAL documentation describes the particulars of each, but the default model is an inverse distance-based model where the *gain* of the source sound is adjusted in inverse proportion to the distance from the sound source. Gain is an amplification factor applied to the recorded amplitude of the sound effect you're using.

You can change distance models in OpenAL using the `alDistanceModel` function (see the OpenAL programmers manual for valid parameters).

Reflection

When sound waves passing through one medium reach another medium or object, such as a wall, part of the original sound wave is reflected off the object, while part of it is absorbed by (and transmitted through) the object. Depending on the dispositions of the sound source and the listener, some sound waves will reach the listener via some direct path. Reflected waves may also reach the listener, although their energy may have been reduced after their interaction with whatever they bounced off. [Figure 26-8](#) illustrates this concept, where some sound waves reach the listener directly and others reach the listener after having been reflected from walls.

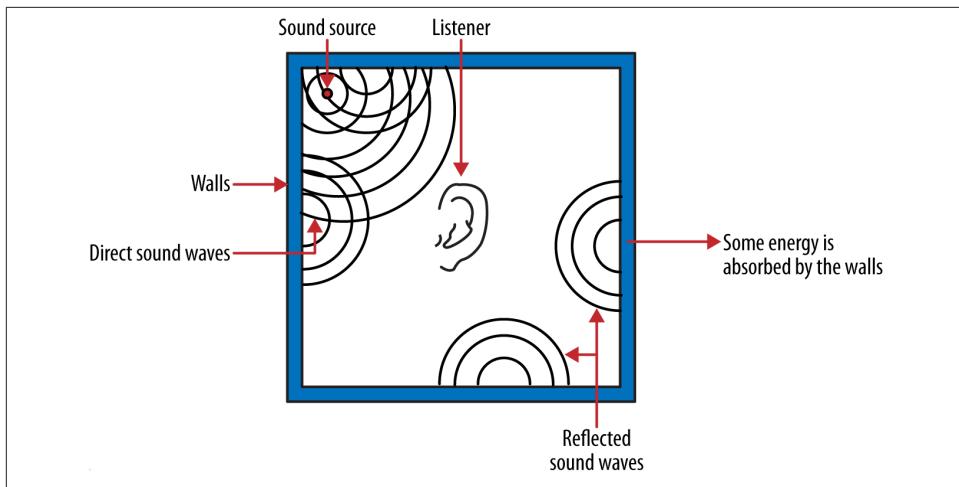


Figure 26-8. Reflected waves

The degree to which sound waves are reflected has to do with the characteristics of the material from which they bounce. Smooth, hard surfaces will tend to reflect more of the sound energy, while softer, irregular surfaces will tend to absorb more energy and scatter the waves that are reflected. These characteristics lend a certain quality to the ultimate sound the listener hears. The same sound played in a tiled bathroom will have a distinctly different quality than if it were played in a room with carpet, drapes, and tapestries. In the bathroom the sound may sound echoic, while in the carpeted room it may sound muted. Somewhere in between these two types of rooms, the sound may reverberate. Reverberation is a perceived prolonging of the original sound due to reflections of the sound within the space.

In your games, it would be prohibitively expensive (computationally speaking) to try to model various sound sources interacting with all the walls and objects in any given space within the game in real time. Such computations are possible and are often used in acoustic engineering and noise control applications, but again, it's too costly for a game. What you can do, however, is mimic the reflective or reverberant qualities of any given space in your game environment by adjusting the reverberation of your sound sources. One approach is to record sound effects with the quality you're looking for to represent the space in which that sound effect would apply. For example, you could record the echoic sound of dripping water in a stone room to enhance the atmosphere of a dungeon.

Alternatively, if you're using a system such as OpenAL and if the reverberation special effect is available on your sound card, you can assign certain reverberation characteristics to individual sound sources to mimic specific environments. This sort of approach falls within the realm of environmental modeling, and the OpenAL Effects Extension

Guide (part of the OpenAL documentation) gives some pretty good tips on how to use its special effects extensions for environmental modeling.

Doppler Effect

The Doppler effect results when there is a relative motion between a sound source and the listener. It manifests itself as an increase in frequency when the source and listener are approaching each other, and a decrease in frequency when the source and listener are moving away from each other. For example, the horn of an approaching train seems to increase in pitch as it gets closer but seems to decrease in pitch as the train passes and moves away. The Doppler effect is a very obvious clue as to the relative motion of a sound source that you can capture in your games. For example, you could model the sound of a speeding car with a Doppler effect complimenting visual cues of a car approaching and passing by a player.

What's happening physically is that the encounter frequency of the sound waves relative to the listener is augmented, owing to the relative velocity. An approaching velocity means there are more waves encountered by the listener per unit of time, which is heard as a higher frequency than the source frequency. Conversely, a departing velocity means there are fewer waves encountered per unit of time, which is heard as a lower frequency. Assuming still air, the increased frequency heard when the sound source and listener are approaching each other is given by the relation:

3D Sound

At one time not so long ago, “3D sound” was hyped as the next big thing. There’s no doubt that for a long time, game sound lagged far behind graphics capabilities. It’s also true that good 3D sound can compliment good visuals, helping to create a more immersive gaming environment. Unfortunately, a lot of early 3D sound just wasn’t that good. Things are getting better, though, and with the use of headphones and a good sound card, some amazing 3D sounds can be generated.

If used properly, 3D sound can give your player the sensation that sounds are coming from different distinct directions. For example, a shot fired from behind the player would be accompanied by the player hearing a gunshot sound as though it really were coming from behind. Such directional sound really adds to the immersive experience of a game.

How We Hear in 3D

3D sound—or more specifically, our ability to localize a sound—is the result of a complex interaction between the sound source and our bodies, not to mention the room or environment we happen to be in. Ignoring environmental interactions, [Figure 26-9](#) illustrates how a sound wave interacts with one’s body.

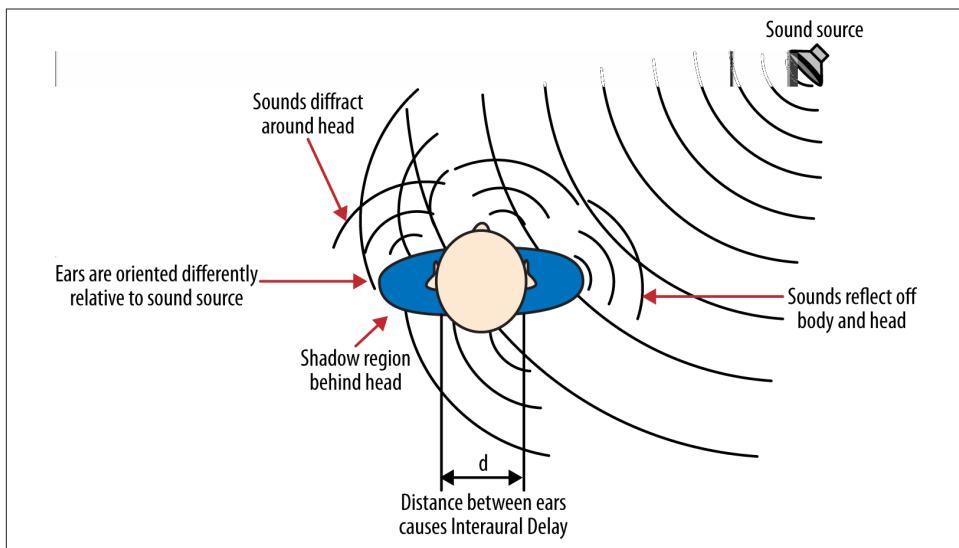


Figure 26-9. 3D sound

One of the first things you may notice is that our ears are separated by some finite distance. This means that the sound coming from the source on the right will reach the

right ear before it reaches the left ear. The time delay between the sound reaching each ear is called the *interaural delay*. We can approximately calculate the delay from a sound coming from the side by taking the distance separating the ears and dividing it by the speed of sound. In air and for a typical head size, that delay is around half a millisecond. The delay will be shorter depending on the orientation of your head with respect to the sound source. Whatever the delay is, our brains use that information to help determine the location from which the sound is coming.

Additionally, as the sound coming from the right in [Figure 26-9](#) reaches the head, some of the energy is reflected off the head. Reflections also occur off the shoulders and torso. Further, as the sound waves pass the head they tend to bend around it. Higher-frequency waves tend to get blocked by the head, and lower-frequency waves tend to pass by with little interruption. The resulting sound in the shadow region behind the head is somewhat different than the source due to the effective filtering that has occurred via interaction with the head. Also, notice that the orientation of the ears with respect to the sound source is different, and sound waves will interact with the ear and ear canal differently due to this differing orientation.

If the sound is coming from above or below the person in addition to being offset laterally, the sound will reflect off and diffract around different parts of the body in different ways.

Considering all these interactions, it would seem that the sound we end up hearing is quite different from the pure source sound. Well, the differences may not be that dramatic, but they are sufficient to allow our brains to pick up on all these cues, allowing us to locate the sound source. Given that we are all different shapes and sizes, our brains are tuned to our specific bodies when processing these localization cues.

It would seem that including believable 3D sound is virtually impossible to achieve in games given the complexity of sounds interacting with the listener. Certainly you can't model every potential game player along with your game sounds to compute how they interact with each other. That said, one approach to capturing the important localization cues is to use what are called *head-related transfer functions* (HRTFs).

If you were to place a small microphone in each ear and then record the sound reaching each ear from some known source, you'd have what is called a *binaural recording*. In other words, the two recordings—one for each ear—capture the sound received by each, which, given all the factors we described earlier, are different from each other. These two recordings contain information that our brains use to help us localize the source sound.

Now, if you compare these binaural recordings by taking the ratio of each to the source sound, you'd end up with what's called a *transfer function* for each ear. (The math is more complicated than we imply here.) These are the HRTFs. And you can derive an HRTF for a sound located at any position relative to a listener. So, the binaural recordings

for a source located at a specific location yield a pair of HRTFs. That's not too bad, but that's only for one single source location. You need HRTFs for every location if you are to emulate a 3D sound from any location. Obviously, generating HRTFs for every possible relative location isn't practical, so HRTFs are typically derived from binaural recordings taken at many discrete locations to create a library, so to speak, of transfer functions.

The HRTFs are then used to derive filters for a given sound you want to play back with 3D emulation. Two filters are required—one for each ear. And the HRTFs used to derive those filters are those that correspond closest to the location of the 3D sound source you're trying to emulate.

It is a lot of work to make all these recordings and derive the corresponding HRTFs. Sometimes the recordings are made using a dummy, and sometimes real humans are used. In either case, it is unlikely that you or your player resemble exactly the dummy or human subject used to make the recordings and HRTFs. This means the synthesized 3D sound may only approximate the cues for any particular person.

A Simple Example

OpenAL allows you to simulate 3D sound via easy-to-use source and listener objects with associated properties of each, such as position, velocity, and orientation, among others. You need only associate the sound data to a source and set its properties, listener position, velocity, and orientation. OpenAL will handle the rest for you. How good the results sound depends on the OpenAL implementation you're using and the sound hardware in use. OpenAL leaves implementation of things such as HRTFs to the hardware.

For demonstration purposes, we took the PlayStatic example provided in the Creative Labs OpenAL SDK and modified it slightly to have the sound source move around the listener. We've also included the Doppler effect to give the impression of the source moving toward or away from the listener. The relevant code is as follows:

```
int main()
{
    ALuint      uiBuffer;
    ALuint      uiSource;
    ALint       iState;

    // Initialize Framework
    ALFWInit();

    if (!ALFWInitOpenAL())
    {
        ALFWprintf("Failed to initialize OpenAL\n");
        ALFWShutdown();
        return 0;
    }
```

```

// Generate an AL Buffer
alGenBuffers( 1, &uiBuffer );

// Load Wave file into OpenAL Buffer
if (!ALFWLoadWaveToBuffer((char*)ALFWaddMediaPath(TEST_WAVE_FILE), uiBuffer))
{
    ALFWprintf("Failed to load %s\n", ALFWaddMediaPath(TEST_WAVE_FILE));
}

// Specify the location of the Listener
allListener3f(AL_POSITION, 0, 0, 0);

// Generate a Source to playback the Buffer
alGenSources( 1, &uiSource );

// Attach Source to Buffer
alSourcei( uiSource, AL_BUFFER, uiBuffer );

// Set the Doppler effect factor
alDopplerFactor(10);

// Initialize variables used to reposition the source
float x = 75;
float y = 0;
float z = -10;
float dx = -1;
float dy = 0.1;
float dz = 0.25;

// Set Initial Source properties
alSourcei(uiSource, AL_LOOPING, AL_TRUE);
alSource3f(uiSource, AL_POSITION, x, y, z);
alSource3f(uiSource, AL_VELOCITY, dx, dy, dz);

// Play Source
alSourcePlay( uiSource );

do
{
    Sleep(100);

    if(fabs(x) > 75) dx = -dx;
    if(fabs(y) > 5) dy = -dy;
    if(fabs(z) > 10) dz = -dz;
    alSource3f(uiSource, AL_VELOCITY, dx, dy, dz);

    x += dx;
    y += dy;
    z += dz;
    alSource3f(uiSource, AL_POSITION, x, y, z);
}

```

```

    // Get Source State
    alGetSourcei( uiSource, AL_SOURCE_STATE, &iState);
} while (iState == AL_PLAYING);

// Clean up by deleting Source(s) and Buffer(s)
alSourceStop(uiSource);
alDeleteSources(1, &uiSource);
alDeleteBuffers(1, &uiBuffer);

ALFWShutdownOpenAL();

ALFWShutdown();

return 0;
}

```

There's nothing fancy about this demonstration, and in fact there are no graphics. The program runs in a console window. That's OK, however, since you really need your ears and not your eyes to appreciate this demonstration. Be sure to use headphones if you test it yourself. The 3D effect is much better with headphones.

The lines of code within `main()` all the way up to the comment `Specify the location of the Listener` are just OpenAL initialization calls required to set up the framework and associate a sound file with a sound buffer that will hold the sound data for later playback.

The next line of code after the aforementioned comment sets the location of the listener. We specify the listener's location at the origin. In a game, you would set the listener location to the player's location as the player moves about your game world. In this example, the listener stays put.

A source is then created and associated with the previously created sound buffer. Since we want to include the Doppler effect, we set the Doppler factor to 10. The default is 1, but we amped it up to enhance the effect.

Next we create six new local variables to store the source's *x*, *y*, and *z* coordinates and the increments in position by which we'll move the source around. After initializing those variables, we set a few properties of the source—namely, we specify that we want the source sound to loop and then we set its initial position and velocity. The velocity properties are important for the Doppler effect. If you forget to set the velocity properties, you'll get no Doppler effect even if you move the source around by changing its position coordinates.

Next, the source is set to play, and a loop is entered to continuously update the source's position every 100 milliseconds. The code within the loop simply adds the coordinate increments to the current coordinates for the source and checks to be sure the source remains within certain bounds. If the source gets too far away, attenuation will be such that you won't hear it any longer, which just gets boring.

The remainder of the code takes care of housecleaning upon exit.

That's really all there is to creating 3D sound effects using OpenAL. Of course, managing multiple sounds with environmental effects in a real game is certainly more involved, but the fundamentals are the same.

APPENDIX A

Vector Operations

This appendix implements a class called `Vector` that encapsulates all of the vector operations that you need when writing 2D or 3D rigid-body simulations. Although `Vector` represents 3D vectors, you can easily reduce it to handle 2D vectors by eliminating all of the `z` terms or simply constraining the `z` terms to 0 where appropriate in your implementation.

Vector Class

The `Vector` class is defined with three components—`x`, `y`, and `z`—along with several methods and operators that implement basic vector operations. The class has two constructors, one of which initializes the vector components to 0, and the other of which initializes the vector components to those passed to the constructor:

```
-----  
// Vector Class and vector functions  
-----  
class Vector {  
public:  
    float x;  
    float y;  
    float z;  
  
    Vector(void);  
    Vector(float xi, float yi, float zi);  
  
    float Magnitude(void);  
    void Normalize(void);  
    void Reverse(void);  
  
    Vector& operator+=(Vector u);  
    Vector& operator-=(Vector u);  
    Vector& operator*=(float s);
```

```
Vector& operator/=(float s);

Vector operator-(void);

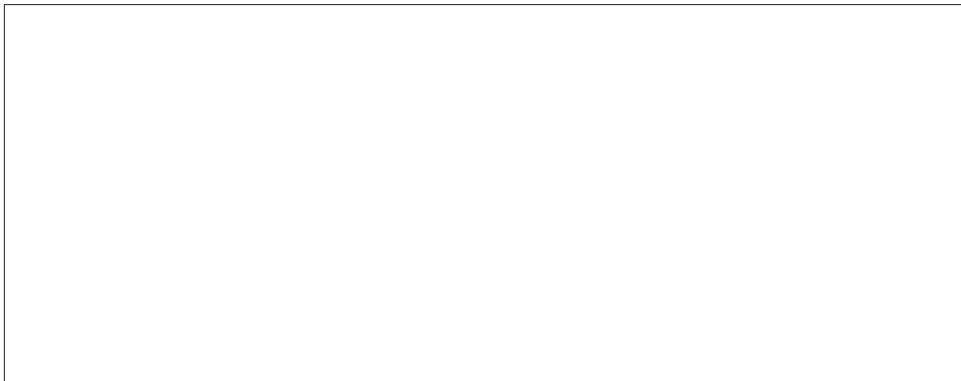
};

// Constructor
inline Vector::Vector(void)
{
    x = 0;
    y = 0;
    z = 0;
}

// Constructor
inline Vector::Vector(float xi, float yi, float zi)
{
    x = xi;
    y = yi;
    z = zi;
}
```

Magnitude

The **Magnitude** method simply calculates the scalar magnitude of the vector according to the formula:



Here's the code that calculates the vector magnitude for our `Vector` class:

```
inline      float Vector::Magnitude(void)
{
    return (float) sqrt(x*x + y*y + z*z);
}
```

Note, you can calculate the components of a vector if you know its length and *direction angles*. Direction angles are the angles between each coordinate axis and the vector, as shown in [Figure A-2](#).

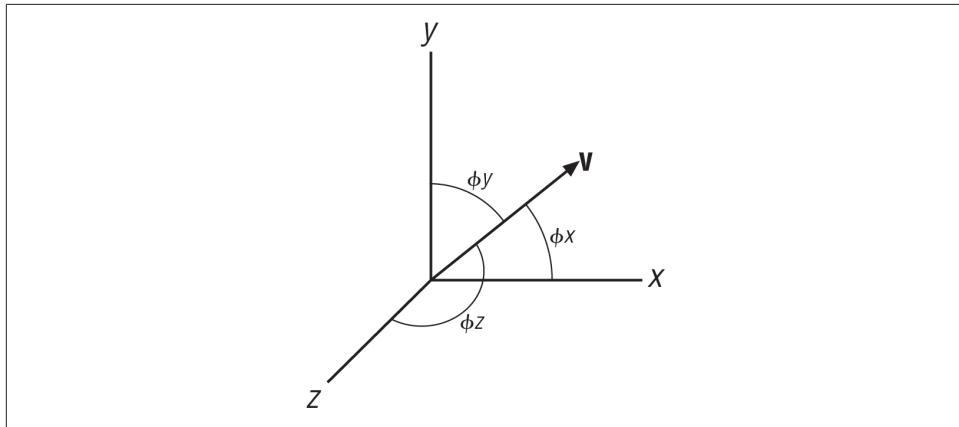


Figure A-2. Direction angles

The components of the vector shown in this figure are:

In other words, the length of the normalized vector is 1 unit. If \mathbf{v} is a non-unit vector with components x , y , and z , then we can calculate the unit vector \mathbf{u} from \mathbf{v} as follows:

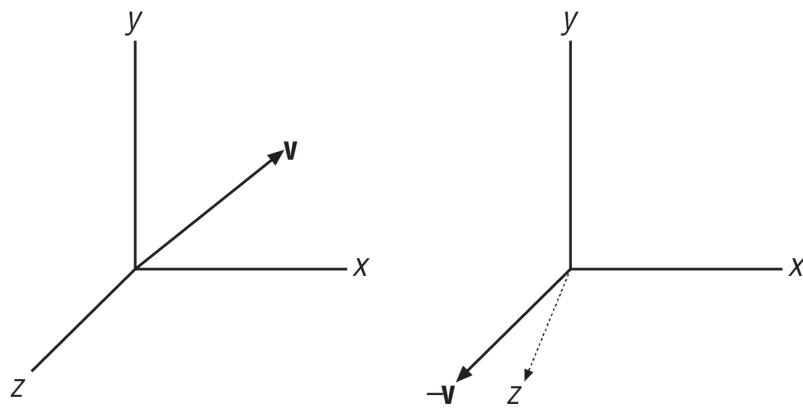


Figure A-3. Vector reversal

Vector Addition: The $+=$ Operator

This summation operator is used for vector addition, whereby the passed vector is added to the current vector, component by component. Graphically, vectors are added in tip-to-tail fashion, as illustrated in [Figure A-4](#).

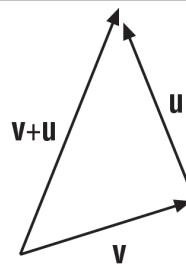


Figure A-4. Vector addition

Here's the code that adds the vector **u** to our **Vector** class vector:

```
inline Vector& Vector::operator+=(Vector u)
{
    x += u.x;
    y += u.y;
    z += u.z;
    return *this;
}
```

Vector Subtraction: The `-=` Operator

This subtraction operator is used to subtract the passed vector from the current one, which is performed on a component-by-component basis. Vector subtraction is very similar to vector addition except that you take the reverse of the second vector and add it to the first, as illustrated in [Figure A-5](#).

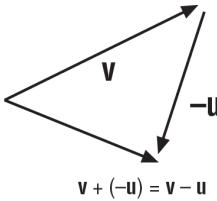


Figure A-5. Vector subtraction

Here's the code that subtracts vector **u** from our `Vector` class vector:

```
inline     Vector& Vector::operator-=(Vector u)
{
    x -= u.x;
    y -= u.y;
    z -= u.z;
    return *this;
}
```

Scalar Multiplication: The `*=` Operator

This is the scalar multiplication operator that's used to multiply a vector by a scalar, effectively scaling the vector's length. When you multiply a vector by a scalar, you simply multiply each vector component by the scalar quantity to obtain the new vector. The new vector points in the same direction as the unscaled one, but its length will be different (unless the scale factor is 1). This is illustrated in [Figure A-6](#).

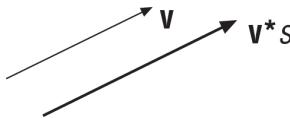


Figure A-6. Scalar multiplication

Here's the code that scales our `Vector` class vector:

```
inline     Vector& Vector::operator*=(float s)
{
    x *= s;
```

```
    y *= s;
    z *= s;
    return *this;
}
```

Scalar Division: The /= Operator

This scalar division operator is similar to the scalar multiplication operator except each vector component is divided by the passed scalar quantity:

```
inline     Vector& Vector::operator/=(float s)
{
    x /= s;
    y /= s;
    z /= s;
    return *this;
}
```

Conjugate: The – Operator

The conjugate operator simply takes the negative of each vector component and can be used when you are subtracting one vector from another or for reversing the direction of the vector. Applying the conjugate operator is the same as reversing a vector, as discussed earlier:

```
inline     Vector Vector::operator-(void)
{
    return Vector(-x, -y, -z);
}
```

Vector Functions and Operators

The following functions and overloaded operators are useful when you are performing operations with two vectors, or with a vector and a scalar, where the vector is based on the `Vector` class.

Vector Addition: The + Operator

This addition operator adds vector **v** to vector **u** according to the formula:

Vector Subtraction: The $-$ Operator

This subtraction operator subtracts vector v from vector u according to the formula:

If two vectors are parallel, then their cross product will be 0. This is useful when you need to determine whether or not two vectors are indeed parallel.

The cross-product operation is distributive; however, it is not commutative:

Triple Scalar Product

This function takes the triple scalar product of the vectors **u**, **v**, and **w** according to the formula:

APPENDIX B

Matrix Operations

This appendix implements a class called `Matrix3x3` that encapsulates all of the operations you need to handle 3×3 (nine-element) matrices when writing 3D rigid-body simulations.

Matrix3x3 Class

The `Matrix3x3` class is defined with nine elements, e_{ij} , where i represents the i th row and j the j th column. For example, e_{21} refers to the element on the second row in the first column. Here's how all of the elements are arranged:

$$\mathbf{M} = \begin{vmatrix} e_{11} & e_{12} & e_{13} \\ e_{21} & e_{22} & e_{23} \\ e_{31} & e_{32} & e_{33} \end{vmatrix}$$

The class has two constructors, one of which initializes the matrix elements to zero, and the other of which initializes the elements to those passed to the constructor:

```
class Matrix3x3 {
public:
    // elements eij: i -> row, j -> column
    float     e11, e12, e13, e21, e22, e23, e31, e32, e33;

    Matrix3x3(void);
    Matrix3x3(float r1c1, float r1c2, float r1c3,
              float r2c1, float r2c2, float r2c3,
              float r3c1, float r3c2, float r3c3 );

    float      det(void);
    Matrix3x3  Transpose(void);
    Matrix3x3  Inverse(void);
```

```

Matrix3x3& operator+=(Matrix3x3 m);
Matrix3x3& operator-=(Matrix3x3 m);
Matrix3x3& operator*=(float s);
Matrix3x3& operator/=(float s);
};

// Constructor
inline Matrix3x3::Matrix3x3(void)
{
    e11 = 0;
    e12 = 0;
    e13 = 0;
    e21 = 0;
    e22 = 0;
    e23 = 0;
    e31 = 0;
    e32 = 0;
    e33 = 0;
}

// Constructor
inline Matrix3x3::Matrix3x3(float r1c1, float r1c2, float r1c3,
                             float r2c1, float r2c2, float r2c3,
                             float r3c1, float r3c2, float r3c3 )
{
    e11 = r1c1;
    e12 = r1c2;
    e13 = r1c3;
    e21 = r2c1;
    e22 = r2c2;
    e23 = r2c3;
    e31 = r3c1;
    e32 = r3c2;
    e33 = r3c3;
}

```

Determinant

The method, `det`, returns the determinant of the matrix. The determinant of a 2×2 matrix:

$$\mathbf{m} = \begin{vmatrix} e_{11} & e_{12} \\ e_{21} & e_{22} \end{vmatrix}$$

is as follows:

We find the determinant of a 3×3 matrix by first expanding the matrix by minors, and then resolving the determinants of the 2×2 minors. Here's how you expand a 3×3 matrix by minors:

$$\mathbf{M} = e_{11} \begin{vmatrix} e_{22} & e_{23} \\ e_{32} & e_{33} \end{vmatrix} - e_{12} \begin{vmatrix} e_{21} & e_{23} \\ e_{31} & e_{33} \end{vmatrix} + e_{13} \begin{vmatrix} e_{21} & e_{22} \\ e_{31} & e_{32} \end{vmatrix}$$

Here's how this all looks in code:

```
inline      float      Matrix3x3::det(void)
{
    return  e11*e22*e33 -
            e11*e32*e23 +
            e21*e32*e13 -
            e21*e12*e33 +
            e31*e12*e23 -
            e31*e22*e13;
}
```

Transpose

The method `Transpose` transposes the matrix by swapping rows with columns—that is, the elements in the first row become the elements in the first column and so on for the second and third rows and columns. The following relations are true for transpose operations:

Here \mathbf{M}^{-1} is the inverse of matrix \mathbf{M} , and \mathbf{I} is the identity matrix. For a 3×3 matrix, we find the inverse as follows:

$$\mathbf{M}^{-1} = \frac{1}{\det[\mathbf{M}]} \begin{vmatrix} E_{11} & E_{12} & E_{13} \\ E_{21} & E_{22} & E_{23} \\ E_{31} & E_{32} & E_{33} \end{vmatrix}$$

Here E_{ij} represents the cofactor of element e_{ij} , which we can find by taking the determinant of the minor of each element. Only square matrices, those with the same number of rows as columns, can be inverted. Note, however, that not all square matrices can be inverted. A matrix can be inverted only if its determinant is nonzero.

The following relation applies to matrix inversion:

```
    e12 += m.e12;
    e13 += m.e13;
    e21 += m.e21;
    e22 += m.e22;
    e23 += m.e23;
    e31 += m.e31;
    e32 += m.e32;
    e33 += m.e33;
    return *this;
}
```

Matrix addition (and subtraction) is commutative, associative, and distributive; thus:

```
    e32 *= s;
    e33 *= s;
    return *this;
}
```

The following relation applies for scalar multiplication (and division):



Matrix Subtraction: The `-` Operator

This operator subtracts matrix `m2` from `m1` on an element-by-element basis:

```
inline     Matrix3x3 operator-(Matrix3x3 m1, Matrix3x3 m2)
{
    return     Matrix3x3(
                m1.e11-m2.e11,
                m1.e12-m2.e12,
                m1.e13-m2.e13,
                m1.e21-m2.e21,
                m1.e22-m2.e22,
                m1.e23-m2.e23,
                m1.e31-m2.e31,
                m1.e32-m2.e32,
                m1.e33-m2.e33);
}
```

Scalar Divide: The `/` Operator

This operator divides every element in the matrix `m` by the scalar `s`:

```
inline     Matrix3x3 operator/(Matrix3x3 m, float s)
{
    return     Matrix3x3(
                m.e11/s,
                m.e12/s,
                m.e13/s,
                m.e21/s,
                m.e22/s,
                m.e23/s,
                m.e31/s,
                m.e32/s,
                m.e33/s);
}
```

Matrix Multiplication: The `*` Operator

This operator, when applied between two matrices, performs a matrix multiplication. In matrix multiplication, each element, e_{ij} , is determined by the product of the i th row in the first matrix and the j th column of the second matrix:

```
inline     Matrix3x3 operator*(Matrix3x3 m1, Matrix3x3 m2)
{
    return Matrix3x3(m1.e11*m2.e11 + m1.e12*m2.e21 + m1.e13*m2.e31,
                     m1.e11*m2.e12 + m1.e12*m2.e22 + m1.e13*m2.e32,
                     m1.e11*m2.e13 + m1.e12*m2.e23 + m1.e13*m2.e33,
                     m1.e21*m2.e11 + m1.e22*m2.e21 + m1.e23*m2.e31,
                     m1.e21*m2.e12 + m1.e22*m2.e22 + m1.e23*m2.e32,
                     m1.e21*m2.e13 + m1.e22*m2.e23 + m1.e23*m2.e33,
                     m1.e31*m2.e11 + m1.e32*m2.e21 + m1.e33*m2.e31,
                     m1.e31*m2.e12 + m1.e32*m2.e22 + m1.e33*m2.e32,
                     m1.e31*m2.e13 + m1.e32*m2.e23 + m1.e33*m2.e33 );
}
```

Two matrices can be multiplied only if one has the same number of columns as the other has rows. Matrix multiplication is not commutative, but it is associative; thus:

```
}

inline     Vector operator*(Vector u, Matrix3x3 m)
{
    return Vector(      u.x*m.e11 + u.y*m.e21 + u.z*m.e31,
                      u.x*m.e12 + u.y*m.e22 + u.z*m.e32,
                      u.x*m.e13 + u.y*m.e23 + u.z*m.e33);
}
```


APPENDIX C

Quaternion Operations

This appendix implements a class called `Quaternion` that encapsulates all of the operations you need to handle quaternions when writing 3D rigid-body simulations.

Quaternion Class

The `Quaternion` class is defined with a scalar component, n , and vector component, \mathbf{v} , where \mathbf{v} is the vector, $x\mathbf{i} + y\mathbf{j} + z\mathbf{k}$. The class has two constructors, one of which initializes the quaternion to 0, and the other of which initializes the elements to those passed to the constructor:

```
class Quaternion {
public:
    float      n;      // number (scalar) part
    Vector     v;      // vector part: v.x, v.y, v.z

    Quaternion(void);
    Quaternion(float e0, float e1, float e2, float e3);

    float      Magnitude(void);
    Vector     GetVector(void);
    float      GetScalar(void);
    Quaternion operator+=(Quaternion q);
    Quaternion operator-=(Quaternion q);
    Quaternion operator*=(float s);
    Quaternion operator/=(float s);
    Quaternion operator~(void) const { return Quaternion( n,
                                                       -v.x,
                                                       -v.y,
                                                       -v.z); }

};

// Constructor
inline    Quaternion::Quaternion(void)
```

```
{  
    n    = 0;  
    v.x = 0;  
    v.y = 0;  
    v.z = 0;  
}  
  
// Constructor  
inline Quaternion::Quaternion(float e0, float e1, float e2, float e3)  
{  
    n    = e0;  
    v.x = e1;  
    v.y = e2;  
    v.z = e3;  
}
```

Magnitude

The method `Magnitude` returns the magnitude of the quaternion according to the following formula:

```
        return n;  
    }
```

Quaternion Addition: The $+=$ Operator

This operator performs quaternion addition by simply adding the quaternion, \mathbf{q} , to the current quaternion on a component-by-component basis.

If \mathbf{q} and \mathbf{p} are two quaternions, then:

```

    v.x -= q.v.x;
    v.y -= q.v.y;
    v.z -= q.v.z;
    return *this;
}

```

Scalar Multiplication: The *= Operator

This operator simply multiplies each component in the quaternion by the scalar s . This operation is similar to scaling a vector, as described in [Appendix A](#):

```

inline     Quaternion Quaternion::operator*=(float s)
{
    n *= s;
    v.x *= s;
    v.y *= s;
    v.z *= s;
    return *this;
}

```

Scalar Division: The /= Operator

This operator simply divides each component in the quaternion by the scalar s :

```

inline     Quaternion Quaternion::operator/=(float s)
{
    n /= s;
    v.x /= s;
    v.y /= s;
    v.z /= s;
    return *this;
}

```

Conjugate: The ~ Operator

This operator takes the conjugate of the quaternion, $\sim\mathbf{q}$, which is simply the negative of the vector part. If $\mathbf{q} = [n, x \mathbf{i} + y \mathbf{j} + z \mathbf{k}]$, then $\sim\mathbf{q} = [n, (-x) \mathbf{i} + (-y) \mathbf{j} + (-z) \mathbf{k}]$.

The conjugate of the product of quaternions is equal to the product of the quaternion conjugates, but in reverse order:

Quaternion Functions and Operators

The functions and overloaded operators that follow are useful when you are performing operations with two quaternions, or with a quaternion and a scalar, or a quaternion and a vector. Here, the quaternions are assumed to be of the type `Quaternion`, and vectors of the type `Vector`, as discussed in [Appendix A](#).

Quaternion Addition: The `+` Operator

This operator performs quaternion addition by simply adding the quaternion `q1` to quaternion `q2` on a component-by-component basis:

```
inline     Quaternion operator+(Quaternion q1, Quaternion q2)
{
    return     Quaternion(     q1.n + q2.n,
                                q1.v.x + q2.v.x,
                                q1.v.y + q2.v.y,
                                q1.v.z + q2.v.z);
}
```

Quaternion Subtraction: The `-` Operator

This operator performs quaternion subtraction by simply subtracting the quaternion `q2` from quaternion `q1` on a component-by-component basis:

```
inline     Quaternion operator-(Quaternion q1, Quaternion q2)
{
    return     Quaternion(     q1.n - q2.n,
                                q1.v.x - q2.v.x,
                                q1.v.y - q2.v.y,
                                q1.v.z - q2.v.z);
}
```

Quaternion Multiplication: The `*` Operator

This operator performs quaternion multiplication according to the following formula:

Here's the code that multiplies two Quaternions, q1 and q2:

```
inline     Quaternion operator*(Quaternion q1, Quaternion q2)
{
    return     Quaternion(q1.n*q2.n - q1.v.x*q2.v.x
                        - q1.v.y*q2.v.y - q1.v.z*q2.v.z,
                        q1.n*q2.v.x + q1.v.x*q2.n
                        + q1.v.y*q2.v.z - q1.v.z*q2.v.y,
                        q1.n*q2.v.y + q1.v.y*q2.n
                        + q1.v.z*q2.v.x - q1.v.x*q2.v.z,
                        q1.n*q2.v.z + q1.v.z*q2.n
                        + q1.v.x*q2.v.y - q1.v.y*q2.v.x);
}
```

Scalar Multiplication: The * Operator

This operator simply multiplies each component in the quaternion by the scalar s . There are two forms of this operator, depending on the order in which the quaternion and scalar are encountered:

```
inline     Quaternion operator*(Quaternion q, float s)
{
    return     Quaternion(q.n*s, q.v.x*s, q.v.y*s, q.v.z*s);
}

inline     Quaternion operator*(float s, Quaternion q)
{
    return     Quaternion(q.n*s, q.v.x*s, q.v.y*s, q.v.z*s);
}
```

Vector Multiplication: The * Operator

This operator multiplies the quaternion q by the vector v as though the vector v were a quaternion with its scalar component equal to 0. There are two forms of this operator, depending on the order in which the quaternion and vector are encountered. Since v is assumed to be a quaternion with its scalar part equal to 0, the rules of multiplication follow those outlined earlier for quaternion multiplication:

```
inline     Quaternion operator*(Quaternion q, Vector v)
{
    return     Quaternion(      -(q.v.x*v.x + q.v.y*v.y + q.v.z*v.z),
                                q.n*v.x + q.v.y*v.z - q.v.z*v.y,
                                q.n*v.y + q.v.z*v.x - q.v.x*v.z,
                                q.n*v.z + q.v.x*v.y - q.v.y*v.x);
}

inline     Quaternion operator*(Vector v, Quaternion q)
{
    return     Quaternion(      -(q.v.x*v.x + q.v.y*v.y + q.v.z*v.z),
                                q.n*v.x + q.v.z*v.y - q.v.y*v.z,
                                q.n*v.y + q.v.x*v.z - q.v.z*v.x,
```

```

        q.n*v.z + q.v.y*v.x - q.v.x*v.y);
}

```

Scalar Division: The / Operator

This operator simply divides each component in the quaternion by the scalar s :

```

inline     Quaternion operator/(Quaternion q, float s)
{
    return     Quaternion(q.n/s, q.v.x/s, q.v.y/s, q.v.z/s);
}

```

QGetAngle

This function¹ extracts the angle of rotation about the axis represented by the vector part of the quaternion:

```

inline     float QGetAngle(Quaternion q)
{
    return     (float) (2*acos(q.n));
}

```

QGetAxis

This function returns a unit vector along the axis of rotation represented by the vector part of the quaternion q :

```

inline     Vector QGetAxis(Quaternion q)
{
    Vector v;
    float m;

    v = q.GetVector();
    m = v.Magnitude();

    if (m <= tol)
        return Vector();
    else
        return v/m;
}

```

QRotate

This function rotates the quaternion p by q according to the formula:

1. For a description of how quaternions are used to represent rotation, refer to the section “[Quaternions](#)” on [page 232](#) in [Chapter 11](#).

Here, $\sim\mathbf{q}$ is the conjugate of the unit quaternion \mathbf{q} . Here's the code:

```
inline     Quaternion QRotate(Quaternion q1, Quaternion q2)
{
    return     q1*q2*(~q1);
}
```

QVRotate

This function rotates the vector \mathbf{v} by the unit quaternion \mathbf{q} according to the formula:

-
2. You can verify this by recalling the trigonometric relation $\cos^2\theta + \sin^2\theta = 1$.

Performing this multiplication yields:

$$\mathbf{R} = \begin{vmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{vmatrix}$$

and let \mathbf{q} be a quaternion:

```

r32 = 2 * (q.v.y*q.v.z + q.n*q.v.x);
r33 = q00 - q11 - q22 + q33;

tmp = fabs(r31);
if(tmp > 0.99999)
{
    r12 = 2 * (q.v.x*q.v.y - q.n*q.v.z);
    r13 = 2 * (q.v.x*q.v.z + q.n*q.v.y);

    u.x = RadiansToDegrees(0.0f); //roll
    u.y = RadiansToDegrees((float) (-(pi/2) * r31/tmp)); // pitch
    u.z = RadiansToDegrees((float) atan2(-r12, -r31*r13)); // yaw
    return u;
}

u.x = RadiansToDegrees((float) atan2(r32, r33)); // roll
u.y = RadiansToDegrees((float) asin(-r31)); // pitch
u.z = RadiansToDegrees((float) atan2(r21, r11)); // yaw
return u;
}

```

Conversion Functions

These two functions are used to convert angles from degrees to radians and radians to degrees. They are not specific to quaternions but are used in some of the code samples shown earlier:

```

inline float DegreesToRadians(float deg)
{
    return deg * pi / 180.0f;
}

inline float RadiansToDegrees(float rad)
{
    return rad * 180.0f / pi;
}

```

Bibliography

A wise old professor once told us that it is not important to know the answers to everything as long as you know where to find the answers when you need them. In that spirit, we've compiled a list of references to books, articles, and Internet resources that you might find useful when looking for additional information on the various topics discussed throughout this book. We've tried to categorize them as best we could, however, keep in mind that several references cover more than just the subject matter referred to in the category headings we've assigned.

General Physics and Dynamics

Anand, D. K., and, Cunniff, P. F, *Engineering Mechanics - Dynamics*, Houghton Mifflin Company, Boston, 1973.

Beer, Ferdinand P., and, Johnston, E. Russell Jr., *Vector Mechanics for Engineers*, McGraw-Hill Book Company, New York, 1988.

Dugas, Rene, *A History of Mechanics*, Dover Publications, Inc., New York, 1988.

Ginsberg, Jerry H., *Advanced Engineering Dynamics*, Cambridge University Press, New York, 1995.

Lindeburg, Michael R., *Engineer-in-Training Reference Manual*, Professional Publications, Inc., Belmont, CA, 1990.

Meriam, J. L., and, Kraige, L. G., *Engineering Mechanics, Volume 2, Dynamics*, John Wiley & Sons, New York, 1987.

Rothbart, Harold A., Editor, *Mechanical Design Handbook*, McGraw-Hill, New York, 1996.

Serway, Raymond A., *Physics for Scientists & Engineers*, Saunders College Publishing, New York, 1986.

Mathematics and Numerical Methods

Boyce, William E., and, DiPrima, Richard C., *Elementary Differential Equations*, John Wiley & Sons, New York, 1986.

Kreyszig, Erwin, *Advanced Engineering Mathematics*, John Wiley & Sons, New York, 1988.

Larson, Roland E., and, Hostetler, Robert P., *Calculus with Analytic Geometry*, D. C. Heath and Company, Lexington, Massachusetts, 1986.

Press, Flannery, Teukolsky, and Vetterling, *Numerical Recipes in Pascal*, Cambridge University Press, New York, 1989.

Computational Geometry

Arvo, James, Editor, *Graphics Gems II*, Academic Press, 1991.

Bobic, Nick, “Advanced Collision Detection Techniques,” Gamasutra, March 2000.

DaLoura, Mark, Editor *Game Programming Gems*, Chapter 4.5, Charles River Media, Inc., Massachusetts, 2000.

Foley, van Dam, Feiner, and Hughes, *Computer Graphics: Principles and Practice*, Addison-Wesley Publishing Company, New York, 1996.

Glassner, Andrew, Editor, *Graphics Gems*, Academic Press, 1990.

Goodman, J. E., and, O’Rourke, J., Editors, *Handbook of Discrete and Computational Geometry*, CRC Press LLC, 1997.

Heckbert, Paul, Editor, *Graphics Gems IV*, Academic Press, 1994.

Kirk, David, Editor, *Graphics Gems III*, Academic Press, 1992.

Mirtich, Brian, “Fast and Accurate Computation of Polyhedral Mass Properties,” Volume 1, number 2, 1996.

Mirtich, Brian, “Rigid Body Contact: Collision Detection to Force Computation,” MERL Technical Report 98-01, Proc. of Workshop on Contact Analysis and Simulation, IEEE International Conference on Robotics and Automation, May 1998.

Mirtich, Brian, “Efficient Algorithms for Two-Phase Collision Detection,” MERL Technical Report 97-23, Practical Motion Planning in Robotics: Current Approaches and Future Directions, K. Gupta and A.P. del Pobil, editors, 1998.

Mirtich, Brian, “V-Clip: Fast and Robust Polyhedral Collision Detection,” MERL Technical Report 97-05, ACM Trans. on Graphics 17 (3), July 1998.

O'Rourke, Joseph, "comp.graphics.algorithms Frequently Asked Questions," Copyright 2000 by Joseph O'Rourke.

O'Rourke, Joseph, *Computational Geometry in C*, Cambridge University Press, New York, 1998.

Paeth, Alan, Editor, *Graphics Gems V*, Academic Press, 1995.

Projectiles

Power, H. L. and, Iversen, J. D., "Magnus Effect on Spinning Bodies of Revolution," AIAA Journal Vol. 11, No. 4, April 1973.

McCoy RL. *A Brief History of Exterior Ballistics. Modern Exterior Ballistics*. Pennsylvania, Schiffer Publishing Ltd; 1999.

Sports Ball Physics

Adair, Robert K., *The Physics of Baseball*, Harper Perennial, New York, 1994.

Davies, John M., "The Aerodynamics of Golf Balls," Journal of Applied Physics, Volume 20, No. 9, September 1949.

Jorgensen, Theodore P., *The Physics of Golf*, Springer, New York, 1999.

MacDonald, William M., "The Physics of the drive in golf," Am. J. Phys. 59 (3), March 1991.

McPhee, John J., and, Andrews, Gordon C., "Effect of sidespin and wind on projectile trajectory, with particular application to golf," Am. J. Phys. 56 (10), October 1988.

Mehta, Rabindra D., "Aerodynamics of Sports Balls," Ann. Rev. Fluid Mech., 1985. 17: 151-89.

Shepard, Ron, "Amateur Physics for the Amateur Pool Player," Ron Shepard, 1997.

Watts, Robert G., and, Baroni, Steven, "Baseball-bat collisions and the resulting trajectories of spinning balls," Am. J. Phys. 57 (1), January 1989.

Watts, Robert G., and, Sawyer, Eric, "Aerodynamics of a knuckleball," Am. J. Phys. 43 (11), November 1975.

Aerodynamics

Abbot, Ira H., and, Von Doenhoff, Albert E., *Theory of Wing Sections*, Dover Publications, Inc., New York, 1959.

Hoerner, Sighard F. and, Borst, Henry V., *Fluid Dynamic Lift*, Hoerner Fluid Dynamics, Bakersfield, CA, 1985.

Hoerner, Sighard F., *Fluid Dynamic Drag*, Hoerner Fluid Dynamics, Bakersfield, CA, 1992.

Thwaites, Bryan, Editor, *Incompressible Aerodynamics*, Dover Publications, Inc., New York, 1960.

Hydrostatics and Hydrodynamics

Clayton, B. R., and, Bishop, R. E. D., *Mechanics of Marine Vehicles*, Gulf Publishing Company, Houston, TX, 1982.

Daugherty, Franzini, and, Finnemore, *Fluid Mechanics with Engineering Applications*, McGraw-Hill Book Company, New York, 1985.

Gillmer, Thomas C., and, Johnson, Bruce, *Introduction to Naval Architecture*, Naval Institute Press, Annapolis, Maryland, 1982.

Lewis, Edward V., Editor, *Principles of Naval Architecture Second Revision, Volume II, Resistance, Propulsion and Vibration*, The Society of Naval Architects and Marine Engineers, Jersey City, New Jersey, 1988.

Lewis, Edward V., Editor, *Principles of Naval Architecture Second Revision, Volume I, Stability and Strength*, The Society of Naval Architects and Marine Engineers, Jersey City, New Jersey, 1988.

Newman, *Marine Hydrodynamics*, The MIT Press, Cambridge, Massachusetts, 1989.

Zubaly, Robert B., *Applied Naval Architecture*, The Society of Naval Architects and Marine Engineers, Jersey City, New Jersey, 1996.

Automobile Physics

Beckman, Brian, “Physics of Racing Series,” Copyright 1991 by Brian Beckman, Stuttgart-West, 1998.

Gillespie, Thomas, *Fundamentals of Vehicle Dynamics*. Society of Automotive Engineers Inc, 1992.

Real-time Physics Simulations

DaLoura, Mark, Editor *Game Programming Gems*, Section 2, Charles River Media, Inc., Massachusetts, 2000.

Hecker, Chris, “Physics, The Next Frontier,” Game Developer, October/November 1996.

Hecker, Chris, “Physics, Part 2: Angular Effects,” Game Developer, December 1996/January 1997.

Hecker, Chris, “Physics, Part 3: Collision Response,” Game Developer, March 1997.

Hecker, Chris, “Physics, Part 4: The Third Dimension,” Game Developer, June 1997.

Katz, Amnon, *Computational Rigid Vehicle Dynamics*, Krieger Publishing Company, Malabar, Florida, 1997.

Lander, Jeff, “Collision Response: Bouncy, Trouncy, Fun,” Gamasutra, February 08, 2000.

Lander, Jeff, “Crashing into the New Year,” Gamasutra, February 10, 2000.

Lander, Jeff, “Lone Game Developer Battles Physics Simulator,” Gamasutra, February 15, 2000.

Lander, Jeff, “Trials and Tribulations of Tribology,” Gamasutra, May 10, 2000.

Lander, Jeff, “Physics on the Back of a Cocktail Napkin,” Gamasutra, May 16, 2000.

Mirtich, Brian, “Impulse-based Dynamic Simulation of Rigid Body Systems,” Ph.D. thesis, University of California, Berkeley, December 1996.

Mirtich, Brian, and Canny, John, “Impulse-based Simulation of Rigid Bodies,” Proc. of 1995 Symposium on Interactive 3D Graphics, April 1995.

Mirtich, Brian, and Canny, John, “Impulse-based Dynamic Simulation,” Proc. of Workshop on Algorithmic Foundations of Robotics, February 1994.

Witkin, Andrew, and, Baraff, David, “An Introduction to Physically Based Modeling,” 1997. (see also SIGGRAPH ’95 course entitled “An Introduction to Physically Based Modeling”)

Digital Physics

Parkinson, Bradford. *Global Positioning System: Theory & Applications*. American Institute of Aeronautics and Astronautics, 1996.

Bradski, Gary, and, Kaehler, Adrian. *Learning OpenCV*. O’Reilly Media, 2008.

Chipley, Michael, et al. *FEMA 426 Reference Manual to Mitigate Potential Terrorist Attacks Against Buildings*. Federal Emergency Management Agency, 2003.

Witkin, Andrew, and, Baraff, David, “An Introduction to Physically Based Modeling,” 1997. (see also SIGGRAPH ’95 course entitled “An Introduction to Physically Based Modeling”)

Allan, Alasdair. *Basic Sensors in IOS*. Sebastopol, CA: O’Reilly, 2011. Print.

Index

Symbols

- * (multiplication) operator
 - in matrix operations, 513, 514, 514
 - in quaternion operations, 235, 236, 521, 522, 522
 - in vector operations, 503, 504
- *= (multiplication) operator
 - in matrix operations, 511
 - in quaternion operations, 520
 - in vector operations, 500
- + (addition) operator
 - in matrix operations, 512
 - in quaternion operations, 521
 - in vector operations, 501
- += (addition) operator
 - in matrix operations, 510
 - in quaternion operations, 519
 - in vector operations, 499
- / (division) operator
 - in matrix operations, 513
 - in vector operations, 504
- /= (division) operator
 - in matrix operations, 512
 - in quaternion operations, 520, 523
 - in vector operations, 501
- ^ (cross-product) operator, in vector operations, 8, 502
- ~ (conjugate) operator in quaternion operations, 235, 520

- (conjugate) operator in vector operations, 501
- (subtraction) operator
 - in matrix operations, 513
 - in quaternion operations, 521
 - in vector operations, 502
- = (subtraction) operator
 - in matrix operations, 511
 - in quaternion operations, 519
 - in vector operations, 500

A

- Abbott, Ira H., 300
- acceleration
 - about, 37
 - accelerometer theory on, 415
 - angular, 6, 62–69, 100
 - average, 37
 - centrifugal, 342
 - centripetal, 65, 342
 - constant, 39–41
 - equations for, 89
 - equations of motion and, 91
 - instantaneous, 38
 - linear, 4, 4, 6, 20, 80, 391
 - nonconstant, 41
 - relative, 69
 - rotational, 80
 - second derivatives of, 38
 - tangential, 65

We'd like to hear your suggestions for improving our indexes. Send email to index@oreilly.com.

velocity and, 36–38
acceleration vector, computing, 20, 475
accelerometers
about, 413–415
common specifications, 417
controlling sprites example, 420–426
data clipping, 417
digital, 86
MEMS, 414–416
sensing orientation with, 167, 418
sensing tilt, 420–426
active stereoization, 467–469
ACVs (air cushion vehicles) (see hovercraft)
adaptive step size method, 151
addition (+) operator
in matrix operations, 512
in quaternion operations, 521
in vector operations, 501
addition (+=) operator
in matrix operations, 510
in quaternion operations, 519
in vector operations, 499
aerodynamic drag
in billiards example, 391
in cars, 339
on hovercraft, 288, 347–350
aerostatic lift, 345
ailerons, in aircraft, 251, 295, 303
aim (guns)
breathing and body position, 360–361
defined, 353
elevation adjustments, 359
shooting positions, 361
taking aim, 355
zeroing the sights, 357–360
air cushion vehicles (ACVs) (see hovercraft)
air drag
particle simulation, 172
for ships and boats, 330
shooting guns example, 130
aircraft
control surfaces in, 295
controlling, 303
flight controls in, 227, 250–254
forces acting on, 293, 297–303
geometry of, 294
modeling, 305–319
parts of, 295
quaternion operations in, 239–241
weather helm and, 348
airfoil
about, 295
lift and drag forces, 297–298
stalled, 301
alpha (opacity), 471
altitude
barometers and, 449
defined, 427
gravity and, 73
trilateration technique and, 433
anaglyphs, complementary-color, 458
Anderson, Byron, 334
angle of attack
critical, 301
defined, 295
lift and drag forces, 297
in lift and drag, 298
stalls and, 301
angular acceleration
angular velocity and, 62–69
rigid-body kinetics and, 100
units and symbols for, 6
angular displacement, 62, 64
angular effects in collision response, 213–225
angular impulse
in collisions, 112–115
golf example, 117
in impulse-momentum principle, 104
angular kinetic energy, 106
angular momentum
equation for, 24
laws of motion and, 21
angular motion
defined, 9
particle explosions and, 363
rigid-body kinetics and, 99–102
angular velocity
angular acceleration and, 62–69
Euler integration and, 249
laws of motion and, 22
rotation in 3D rigid-body simulation and, 230, 233
units and symbols for, 6
anisotropic materials, 22
antipodal locations, 435
Archimedes' principle of buoyancy, 323
arm rod (golf swings), 371
aspect ratio, aircraft wing area, 295

atan function, 420
atan2 function, 435
atmospheric pressure, 440, 449
atomic clock drift, 433
attenuation in sound, 485
automobiles (see cars)
autostereoscopy, 463
average acceleration, 37

B

backward difference scheme, 474
ballistic coefficient (BC), 354
ballistics and firearms, 353
(see also shooting guns example)
about, 353
particle explosions, 363–366
projectile motion, 353–354
recoil and impact, 361
taking aim, 355–361
Bancroft, Stephen, 433
barometers, 439, 448–450
baseball examples
collisions in, 109–111
Magnus effect in, 135
BC (ballistic coefficient), 354
Bernouilli, David, 125
Bernoulli's equation, 125, 297
billiards
about, 378–380
calculating forces, 388–392
collision example, 107–109
handling collisions, 393–400
implementing example, 380–382
initializing example, 383–386
stepping simulation, 386
binaural recording, 490
binocular disparity, 451
binocular rivalry, 458
binocular vision, 451–454
biomechanics, 369, 369
(see also sports)
boats (see ships and boats)
Borst, Henry V., 301
boundary layer in fluid dynamic drag, 126, 332
bounding boxes, 286
bounding circle check, 208, 218
bounding spheres, 286
Bourg, David M., 161
Bradski, Gary, 473

breadth-to-draft ratio, 329
breadth-to-length ratio, 333
buoyancy
about, 77–79
calculating, 78
pressure and, 77
in ship stability, 323
buttons, pressure-sensitive, 442–444

C

cannon ball game example (see shooting guns example)
capacitive touch screens, 404, 408–410
capsizing ships and boats, 323, 325
cars
about, 339
power and, 340
resistance and, 339
steering, 342–345
stopping distance in, 341
Cartesian coordinate system
about, 6
2D particle kinematics and, 42
Cavendish, Henry, 72
cavitation, 335
center of gravity (see center of mass)
center of mass
calculating, 10–12
defined, 9
impact and, 110
load cells and, 445–448
local coordinate system and, 62
quaternion rotation and, 232
rigid-body kinematics and, 61
in ship stability, 323
torque and, 83
2D example, 17
center of percussion, 110
center of pressure, 288
central difference, second-order, 475
central impact, 107, 208
centrifugal acceleration, 342
centripetal acceleration, 65, 342
chord length, aircraft wings, 295
chord line, 295
circular cylinder, mass moment of inertia formula, 13
circular cylindrical shell, mass moment of inertia formula, 13

circular polarization of light, 459–462
cloth simulation, 255
coefficient of restitution
 applying formula for, 111
 collision response and, 287
 defined, 107
 physics models and, 284
coefficient of rolling resistance, 340
coefficients of friction, 74, 284
collision detection
 about, 103, 206, 285
 billiards example, 393–400
 bounding circle check, 208, 218
 collision response and, 287
 continuous, 286
 particle-to-ground, 175–181
 particle-to-obstacle, 181–186
 physics engines and, 283, 285
 vertex-edge collisions, 221–225
 vertex-vertex collisions, 219–221
collision response
 about, 103
 angular effects in, 213–225
 billiards example, 393–400
 collision detection and, 287
 conservation of momentum principle, 105, 111, 362
 implementing, 205–225
 impulse-momentum principle, 104, 286
 linear, 206–213
 particle-to-ground, 175–181
 particle-to-obstacle, 181–186
 physics engines and, 283, 286
collisions
 about, 103
 angular impulse in, 112–115, 117
 baseball and bat example, 109–111
 billiard ball example, 107–109, 378
 friction and, 115–118
 golf example, 116–118
 handling in billiards example, 393–400
 impact force and, 105–111
 implementing in particle simulation, 175–186
 impulse-momentum principle, 104, 286
 inelastic, 106
 line of action of, 107
 linear impulse in, 112–115, 207
 penetration in, 207, 211, 287
 plastic, 106
compartments in ships and boats, 325
complementary-color anaglyphs, 458
condensation, 478
condition table, motion-identifying, 447
conjugate operator
 in quaternion operations, 235, 520
 in vector operations, 501
connecting objects
 about, 255
 connecting particles, 258–265
 connecting rigid bodies, 265–279
 hanging rope or vine example, 258–265
 linked-chain example, 265–279
conservation of momentum principle, 105, 111, 362
constant acceleration, 39–41
contact forces
 about, 71
 in billiards example, 391
 center of mass and, 83
 collision response and, 287
 friction as, 73–75
contact manifold, 285, 393
continuous collision detection, 286
convergence distance, 456, 468
conversion functions, in quaternion operations, 527
coordinate systems
 accelerometer example, 419
 Cartesian, 6, 42
 in particle simulation, 168
 geographic, 427, 433
 local, 62
 rotating, 229
 3D rigid-body simulation, 243
coordinate transformation, 2D rigid-body simulation, 197
coupled motions, 326, 328
critical attack angle, 301
cross track error, 436
cross-product (^) operator, in vector operations, 8, 502

D

dampers
 about, 79
 connecting objects, 257
 equation for, 79, 257

uses for, 80
data clipping in accelerometers, 417
degrees of freedom
controlling sprites example, 421–426
rigid bodies and, 227, 418
ships and boats, 326, 326–328
DegreesToRadians function, 527
density
sound and, 478
uniform, 100
units and symbols for, 5, 6
depth from focus (cameras), 472
derivatives
defined, 8
second, 38
determinant, in matrix operations, 508
differentiation schemes (optical tracking), 474–476
digital accelerometers, 86
digital signal processing, 413
dihedral angle, 311
direct central impact, 107
direct force effectors, 288
direct impact, 107
direction
acceleration and, 20
in vector operations, 7
of rotation, 66, 80
tensors and, 22
velocity and, 36, 46–47
wind, 93
direction angles, 497
direction cosines, vector, 46, 497
disparity maps, 473
dispersive signal technology, 404
displacement
about, 37, 321
accelerometer theory on, 415
angular, 62, 64
distance traveled versus, 37
formula for, 40
volume of ships and, 323
weight of ship and, 323
distance
calculating between latitude and longitude
coordinates, 433–438
convergence, 456, 468
equations of motion and, 91
intraocular, 467
skidding, 342
stopping, 341
distance tolerance, 209, 287
distance traveled
displacement versus, 37
equations for, 89
second derivatives of, 38
time and, 36
division (/) operator
in matrix operations, 513
in vector operations, 504
division (/=) operator
in matrix operations, 512
in quaternion operations, 520, 523
in vector operations, 501
Doppler effect, 485, 488
dot-product (*) operator, 8, 503
drag coefficient
aircraft and, 302
calculating, 129
cars and, 340
fluid dynamic drag and, 75
hovercraft and, 348
physics models and, 284
terminal velocity and, 132
drag forces
aircraft in flight and, 293, 297–302, 302
on cars, 339
fluid dynamic, 75
on hovercraft, 347–350
nonconstant acceleration and, 41
particle kinetics in 3D, 92
resistance in ships and, 328–331
speed and, 125
viscous, 75

E

Einstein, Albert, 30, 415
electromagnetic waves, 459
elevators, in aircraft, 252, 304
engineering strain, 444
equal and opposite forces, 79, 289
equations of motion
defined, 85
linked-chain example, 274
numerical integrators and, 288
particle kinetics in 2D, 88–91
particle kinetics in 3D, 91
in particle simulation, 169

- real-time simulations and, 144–146
in 2D rigid-body simulation, 197
- Euler angles
constructing quaternions from, 236, 524
extracting from quaternions, 238, 250, 525
steering cars and, 342
in 3D motion, 62, 227
- Euler’s method
for billiards example, 386
improved, 153–158
modeling golf swings, 375
in particle simulation, 169
in real-time simulations, 146–152
in 3D rigid-body simulation, 247
in 2D rigid-body simulation, 199
- exotic touch screens technologies, 404
- explosions
about, 362
kinematic particle explosion, 54–60
kinematic particle explosions, 363–366
particle, 363–366
polygon, 366–368
external ballistics, 353
- F**
- fade effect, 60
- far zero, 358
- field forces
about, 71, 72
center of mass and, 83
- filters
accelerometers and, 413
polarized light and, 459
in 3D sound, 491
- firearms (see ballistics and firearms)
- fireworks exploding, 60
- flag model, 255
- flaps, in aircraft, 251, 295, 302, 303
- flight controls, 3D rigid-body simulation, 227, 250–254, 307–319
- flight simulation (see aircraft)
- fluid dynamic drag
about, 75
around a sphere, 125–128
boundary layer in, 126, 332
drag coefficient, 75, 129
laminar flow, 75
in projectiles, 124–130
Reynolds number, 127
- separation point in, 127
of spinning sphere, 132
turbulent flow, 75
turbulent wake, 127
- football simulation game, 54
- force
about, 71
aggregating, 289
on aircraft in flight, 293, 297–303
buoyancy, 77–79
calculating in billiards example, 388–392
equal and opposite, 79, 289
fluid dynamic drag, 75
impact, 105–111
impulse, 104
linear acceleration and, 80
Newton’s second law of motion and, 4, 20
pressure versus, 76
springs and dampers, 79
static, 74
torque versus, 80–83
units and symbols for, 4, 6
- force effectors, 287
- forces at a distance, 71, 72
- forward azimuth, 435
- FourSquare app, 428
- frequency
measuring for accelerometers, 413
measuring for sound, 480
- friction
about, 73–75
in billiards example, 391
calculating, 74
coefficients of, 74
collisions and, 115–118
recoil and, 362
skidding distance and, 341
- frictional drag
aircraft and, 303
Bernoulli’s equation and, 125
moving through fluid and, 5, 329
- frustum, viewing, 454–458
- fuselage, in aircraft, 295
- fusion in binocular vision, 451
- G**
- game engines, 283, 293
- geocaching, 428
- geographic coordinate system, 427, 433

Gillespie, Thomas, 345
Global Positioning System (GPS)
about, 427, 429, 449
calculating between latitude and longitude, 433–438
location-based gaming and, 427
trilateration technique, 429–433
GM (stability index), 324
golf examples
in collisions, 116–118
Magnus effect in, 135
modeling golf swings, 370–378
physics engine considerations, 282
two-rod model, 371
Google Maps, 449
GPS (Global Positioning System)
about, 427, 429, 449
calculating between latitude and longitude, 433–438
location-based gaming and, 427
trilateration technique, 429–433
gravitational force
accelerometer theory on, 415
aircraft in flight and, 293
as force effector, 288
Newton's law of gravitation, 72
particle simulation and, 166–169
projectiles and, 120
zeroing the sights and, 357–359
great-circle heading, 433, 435
Greenwich Observatory (UK), 427, 429
grid partitioning, game space, 286
ground plane, particle-to-ground collisions, 175–181
guns, shooting (see ballistics and firearms)
gyroscopes, 419

H

Hamilton, William, 232
haptic feedback in touch screens, 411
harmonic wave, 479, 481
Harr-like features (optical tracking), 473
haversine formula for distance, 433
head-related transfer functions (HRTFs), 490
heave motion in ships and boats, 327, 327
hectoPascals (hPa), 449
heuristic subroutines, 469
high lift devices, 302
high-pass filters (accelerometers), 413

higher-order terms, 146
Hoerner, Sighard F., 301, 303
holographs, 465
Hooke's law, 79, 257
horsepower in cars, 340
hovercraft
about, 345–347
aerodynamic drag on, 288, 347–350
implementing collision response, 205–225
resistance in, 347–350
steering, 347, 350
2D rigid-body simulation, 189–204
hPa (hectoPascals), 449
HRTFs (head-related transfer functions), 490
hull in ships and boats, 322
hull speed, 333
human action modeling (see sports)
hydrodynamic lift, 321
hydrostatic pressure, 76, 440

I

impact
billiard characteristics for, 378
center of mass and, 110
central, 107, 208
collisions and, 105–111
conservation of momentum principle, 105, 111, 362
direct, 107
direct central, 107
oblique, 107
improved Euler method, 153–158
impulse
angular, 104, 112–115, 117
defined, 104
linear, 104, 112–115, 207
projectiles and, 120
impulse torque, 104
impulse-momentum principle, 104, 286
indirect force effectors, 288
induced drag, 347
inelastic collisions, 106
inertia tensors
about, 24
angular momentum equation, 24
calculating, 28, 245
products of inertia, 26
symmetry and, 27
transfer of axis formula, 26

inertia, defined, 342
infrared touch screens, 404
initial value problem, 145
instantaneous acceleration, 38
instantaneous velocity
 calculating, 39
 defined, 37
integral photography, 466
integrals, defined, 8
integrators
 3D rigid-body simulation, 247–250
 about, 162
 numerical, 288
 particle simulation, 169
 2D rigid-body simulation, 198–200
intensity of sound, 480
interaural delay, 490
internal ballistics, 353
International System of Units (SI), 5
International Towing Tank Conference (ITTC),
 329
intraocular distance, 467
inverse, in matrix operations, 509
isotropic materials, 22
ITTC (International Towing Tank Conference),
 329

J

jittering problem in objects, 287
Jorgensen, Theodore P., 371–378

K

Kaehler, Adrian, 473
Kinect system, 472
kinematic viscosity, 6
kinematics
 about, 35
 angular velocity and acceleration, 62–69
 constant acceleration, 39–41
 local coordinate axes, 62
 nonconstant acceleration, 41
 particle explosion, 54–60, 363–366
 rigid-body, 61
 3D particle, 45–54
 2D particle, 42–45
 velocity and acceleration, 36–38
kinetic energy
 about, 106

of bullets, 362, 364–366
converting, 105
of sound waves, 480
kinetic weapons, 362
kinetics
 about, 85–86
 problem-solving guidelines, 86
 rigid-body, 99–102
 3D particle, 91–99
 2D particle, 87–91
kneeling (shooting position), 361
Kutta condition, 297
Kutta-Joukouski theorem, 134

L

laminar flow, 75
latitude
 calculating distance between longitude and,
 433–438
 defined, 427
 trilateration technique and, 429–433
LC (liquid-crystal) shutter glasses, 462
LED technology, 404
length
 of aircraft wings, 295
 of ships and boats, 322
 units and symbols for, 4, 6
lenticular lenses, 464
lift force
 about, 132–134
 aerostatic lift, 345
 aircraft in flight and, 293, 297–302
 calculating, 134
lift-to-drag ratio, 300
light
 polarization of, 459–462
 speed of, 30–33, 432
linear acceleration
 in billiards example, 391
 force and, 80
 Newton's second law of motion and, 4, 20
 units and symbols for, 4, 6
linear collision response, 206–213
linear impulse
 in collisions, 112–115, 207
 in impulse-momentum principle, 104
linear kinetic energy, 106
linear momentum, 21
linear motion, 9

- linear polarization of light, 459–462
 linear velocity
 equation for, 64
 linked-chain example, 274
 units and symbols for, 6
 linked-chain example, 265–279
 liquid-crystal (LC) shutter glasses, 462
 liquid-crystal plasma displays, 462
 load cells
 center of gravity and, 445–448
 defined, 439
 gaming uses for, 439, 444
 strain gauges and, 444
 local coordinate system, 62, 296
 locality principle, 30
 location-based gaming
 about, 427
 geographic coordinate system and, 427
 longitude
 calculating distance between latitude and, 433–438
 defined, 427
 origin of determining, 429
 trilateration technique and, 429–433
 longitudinal waves, 459
 Lorentz factor, 32
 Lorentz transformation, 31–33
 loudness in sound, 480
 low-pass filters (accelerometers), 413
- ## M
- magnitude
 in polygon explosions, 367
 in quaternion operations, 234, 518
 scalars and, 7
 tensors and, 22
 of torque, 81
 in vector operations, 7, 496
 of velocity, 36
 Magnus effect, 132–137
 MakeEulerAnglesFromQ function, 238, 250, 525
 MakeQFromEulerAngles function, 236, 524
 maneuverability of ships and boats, 335
 mass, 415
 (see also center of mass)
 accelerometer theory on, 415
 calculating, 9
 defined, 9
- Newton's second law of motion and, 4, 20
 2D example, 16
 units and symbols for, 4, 4, 6
 variable, 138
 virtual, 332
 mass flow rate, 348
 mass moment of inertia
 calculating, 12–15
 defined, 9
 physics models and, 284
 2D example, 17–19
 units and symbols for, 6
 mass properties
 center of mass, 9, 10–12
 defined, 9
 mass, 9, 9
 mass moment of inertia, 9, 12–15
 2D example, 15–19
 matrix addition, 510, 512
 matrix functions and operators
 matrix addition, 510, 512
 matrix multiplication, 513
 matrix subtraction, 511, 513
 scalar division, 512, 513
 scalar multiplication, 511, 514
 vector multiplication, 514
 matrix multiplication, 513
 matrix operations
 matrix functions and operators, 512–514
 Matrix3x3 class, 507–512
 matrix subtraction, 511, 513
 Matrix3x3 class
 about, 507
 det method, 508
 Inverse method, 509
 matrix addition, 510
 matrix subtraction, 511
 scalar division, 512
 scalar multiplication, 511
 Transpose method, 509
 mean camber line, 295
 measures (see units and measures)
 MEMS (microelectromechanical systems), 413–416, 448
 Mercator projection, 437
 metacenter, in ship stability, 324
 microelectromechanical systems (MEMS), 413–416, 448
 mixed-reality games, 428

- Miyamoto, Shigeru, 444
- models
- 3D rigid-body simulation, 243–247
 - about, 162
 - aircraft flight, 305–319
 - connecting objects examples, 255–279
 - particle simulation, 166–169
 - physics, 283
 - 2D rigid-body simulation, 190
- modulo operator, 436
- moment (see torque)
- moment of inertia
- calculating, 12–15, 28, 245
 - defined, 9
 - laws of motion and, 22
 - rigid-body kinetics and, 100
 - 3D example, 23
 - 2D example, 17–19
- momentum
- angular, 21, 24
 - conservation of momentum principle, 105, 111, 362
 - linear, 21
- momentum drag, 348
- motion
- angular, 9, 99–102, 363
 - coupled, 326, 328
 - linear, 9
 - plane, 61
 - projectile, 353–354
 - ship, 326–328
- motion-identifying condition table, 447
- mouse-based input versus touch screens, 412
- movement parallax, 453
- multiplication (*) operator
- in matrix operations, 513, 514, 514
 - in quaternion operations, 235, 236, 521, 522, 522
 - in vector operations, 503, 504
- multiplication (*=) operator
- in matrix operations, 511
 - in quaternion operations, 520
 - in vector operations, 500
- muzzle velocity, 354
- ## N
- NACA foil sections, 300
- neutral axis, 12
- Newton, Isaac, 3
- Newton's laws
- conservation of momentum principle, 105, 111, 362
 - equal and opposite forces, 79, 289
 - equations of motion and, 85, 88
 - Impulse-Momentum Principle and, 105
 - of motion, 415
 - of gravitation, 72
 - of motion, 3, 5, 20–24, 72, 73
- nonconstant acceleration, 41
- normalize, in vector operations, 497
- numerical integrators, 288
- ## O
- object detection (optical tracking), 473
- Objective-C
- accelerometer code example, 421–426
 - calculating distances code example, 434, 437
- objects
- aggregating forces, 289
 - collision detection considerations, 286
 - connecting, 255–279
 - jittering problem, 287
 - simulated objects manager for, 284
 - weight of, 4, 323, 354
- oblique impact, 107
- occlusion in volumetric displays, 466
- off-axis method (cameras), 456
- opacity (alpha), 471
- OpenAL API
- about, 477
 - alDistanceModel function, 486
 - alSpeedOfSound function, 485
 - AL_SPEED_OF_SOUND property, 485
 - Doppler effect, 488
 - reverberation special effect, 487
 - 3D sound example, 491–494
- OpenCV method for 3D reconstruction, 473
- optical sensors and tracking
- about, 471, 472
 - Kinect system, 472
 - numerical differentiation, 474–476
 - OpenCV method for 3D reconstruction, 473
- optimal imaging techniques, 404
- orientation
- computing in 2D rigid-body simulation, 196, 199
 - defining in cannon ball game example, 46

expressing in 3D rigid-body simulations, 227, 230
importance in rigid bodies, 35
quaternions and, 232, 239–241, 244, 249
sensing with accelerometers, 167, 418
tracking during body rotation, 62
out-of-screen effects, 469
overshoot angle, 336
oversteering cars, 343

P

parallax, 452
parallax barrier in autostereoscopy, 463
parallel axis theorem, 13, 18
parameter tuning, 119
particle explosions, 54–60, 363–366
particle kinematics
 particle explosions, 54–60, 363–366
 3D, 45–54
 2D, 42–45
particle kinetics
 3D, 91–99
 2D, 87–91
particle simulation
 about, 161–166
 basic simulator, 170–172
 implementing collisions, 175–186
 implementing external forces, 172–174
 integrating particles, 169
 rendering particles, 170
 simple model, 166–169
 tuning, 186
particle-to-ground collisions, 175–181
particle-to-obstacle collisions, 181–186
particles
 about, 35
 cloth simulation, 255
 connecting, 258–265
 hanging rope or vine example, 258–265
 integrating in simulator, 169
 rendering in simulator, 170
 uses for, 161
passive stereoization, 467, 469
pattern recognition, 471
penalty methods, 104
penetration in collisions, 207, 211, 287
percussion, center of, 110
photodetectors, 404
photons, defined, 30

physics engines
 about, 281
 building, 281–283
 collision detection in, 283, 285
 collision response in, 283, 286
 force effectors and, 287
 general-purpose, 281
 numerical integrators and, 288
 physics models, 283
 purpose-built, 281
 simulated objects manager, 284
physics models, 283
piezoresistors, 416, 448
piezoresistive strain gauge, 445
pinned joint, 275
pitch
 flight control action for, 227, 250, 252, 304
 local coordinate axes and, 62, 296
 quarter-chord point and, 301, 304
 in ships and boats, 323, 327, 328
 in sound, 480
plane motion, 61
planing vessels, 321, 331
plastic collisions, 106
plenum chamber (hovercraft), 346
point sound source, 485
point-blank weapons, 355
polarization of light, 459–462
polling rate, 413
polygon explosions, 366–368
pounds per square foot (psf), 76
pounds per square inch (psi), 76
power
 of car engines, 340
 defined, 340
 hovercraft hover height and, 345
 of sound wave, 480
pressure
 atmospheric, 440, 449
 Bernoulli's equation and, 125, 297
 buoyancy and, 77
 center of, 288
 defined, 440
 force versus, 76
 hydrostatic, 76, 440
 sound and, 478
 units and symbols for, 6, 440
pressure drag, 329, 340

pressure sensors
about, 439–442
load cells and, 444–448
pressure-sensitive buttons and, 442–444
principal axes, 26
products of inertia, 26
projectiles
about, 119–120
bullets in motion, 353–354
drag and, 124–132
football simulation game, 54
golf ball flight, 370
hitting the target example, 49–54
impulse forces, 104
Magnus effect, 132–137
particle explosions as, 54–60, 363–366
recoil and impact, 361
simple trajectories, 120–124
taking aim, 355–361
terminal velocity and, 130
variable mass and, 138
prone (shooting position), 361
propeller walk, 337
propulsion of ships and boats, 334
pseudoranges, 431
psf (pounds per square foot), 76
psi (pounds per square inch), 76

Q

QGetAngle function, 523
QGetAxis function, 523
QRotate function, 523
quantum tunneling, 417
quarter-chord point, 301, 304
quaternion addition, 519, 521
Quaternion class
about, 234, 517
conjugate operator, 235, 520
GetScalar method, 518
GetVector method, 518
Magnitude method, 234, 518
quaternion addition, 519
quaternion subtraction, 519
scalar division, 520
scalar multiplication, 520
quaternion functions and operators
conjugate operator, 235, 520
conversion functions, 527
DegreesToRadians function, 527

MakeEulerAnglesFromQ function, 238, 250, 525
MakeQFromEulerAngles function, 236, 524
QGetAngle function, 523
QGetAxis function, 523
QRotate function, 523
quaternion addition, 519, 521
quaternion multiplication, 235, 521
quaternion subtraction, 519, 521
QVRotate function, 235, 240, 247, 524
RadiansToDegrees function, 527
scalar division, 520, 523
scalar multiplication, 520, 522
vector multiplication, 236, 522
quaternion multiplication, 235, 521
quaternion operations
Quaternion class, 234, 517–520
quaternion functions and operators, 521–527
for rigid-body rotation, 232–238
3D simulation and, 239–241
quaternion subtraction, 519, 521
QVRotate function, 235, 240, 247, 524

R

RadiansToDegrees function, 527
rarefaction, 479
real-time simulations
about, 143
equations of motion and, 144–146
Euler's method, 146–152
improved Euler method, 153–158
physics engine considerations, 282
Runge–Kutta method in, 155–158
Taylor's theorem in, 146, 153
recoil of firearms, 361
rectangular cylinder, mass moment of inertia formula, 13
reflection in sound waves, 486
relative acceleration, 69
relative normal velocity, 209
relative velocity
about, 68
between connected objects, 258
collision detection and, 207, 209, 286
particle-to-ground collisions and, 179
relativistic time, 29–33, 433
rendering
about, 162

- particle simulation, 170
simulated objects manager and, 285
2D rigid-body simulation, 200
residual resistance, 330
resistance
 in cars, 339
 equation for, 87
 in hovercraft, 347–350
 residual, 330
 rolling, 340, 392
 in ships and boats, 328–334
 zeroing the sights and, 357–359
resistive touch screens, 403, 404–408
restitution, coefficient of (see coefficient of restitution)
reverberation special effect, 487
reverse geocaching, 428
reverse, in vector operations, 498
Reynolds number, 127, 329
rhumb line, 433, 436
right hand rule, 80
rigid bodies
 about, 35
 billiards example, 381
 circular path of particles making up, 63
 connecting, 265–279
 conservation of momentum principle, 105, 111, 362
 linked-chain example, 265–279
 penalty methods for, 103
 rotation in 3D simulation, 227–241
 3D simulation for, 243–254, 381
 2D simulation for, 189–204
rigid-body kinematics, 61
rigid-body kinetics, 99–102
roadway bank, 344
Robbins effect, 132–137
roll
 flight control action for, 227, 250
 local coordinate axes and, 62, 296
 in ships and boats, 323, 327, 327
roll period, 327
rolling resistance, 340, 392
rope example, 258–265
rotation in 3D rigid-body simulation
 about, 227
 quaternions and, 232–241
 rotation matrices and, 228–232
 rotational restraint, 275–279
rotation matrices, 228–232
rotational acceleration, 80
rotational inertia (see mass moment of inertia)
rotational restraint, 275–279
rudders
 in aircraft, 295, 304
 in ships and boats, 336
Runge-Kutta method, 155–158, 373, 375
- ## S
- Saffer, Dan, 412
scalar division
 in matrix operations, 512, 513
 in quaternion operations, 520, 523
 in vector operations, 501, 504
scalar multiplication
 in matrix operations, 511, 514
 in quaternion operations, 520, 522
 in vector operations, 500, 504, 505
scalars
 defined, 7
 examples of, 7
 magnitude and, 7
 tensors and, 22
second derivatives, 38
second zero, 358
second-order central difference, 475
separation point in fluid dynamic drag, 127
ships and boats
 about, 321–322
 geometry of, 322
 maneuverability of, 335–337
 parts of, 322
 propulsion of, 334
 resistance in, 328–334
 ship motions, 326–328
 sinking, 325
 stability of, 323–325
 2D particle kinetics example, 87–91
 types of, 321
 typical speeds for, 333
shooting guns example
 air drag and, 130
 challenges of, 354
 particle kinematics suggestions, 55
 physics engine considerations, 282
 taking aim, 355
 3D particle kinematics, 45–54
 3D particle kinetics, 92–99

2D particle kinematics, 43–45
shooting positions, 361
SI (International System of Units), 5
simulated objects manager, 284
simulations
 cloth, 255
 football game, 54
 force effectors in, 287
 hitting the target example, 49–54
 implementing collision response, 205–225
 particle, 161–187
 physics engine considerations, 283
 real-time, 143–159, 282
 springs and dampers in, 80
 3D rigid-body, 227–241, 243–254
 2D rigid-body, 189–204
 updating, 262–265, 271–275
 sinking ships and boats, 325
 skidding distance, 342
 skin friction
 on aircraft, 303
 on cars, 340
 on hovercraft, 347
 planing vessels and, 331
 sliding friction force, 391
 sound and sound waves
 about, 477–481
 characteristics of, 481–488
 3D, 489–494
span, aircraft wings, 295
special effects
 fade effect, 60
 out-of-screen effects, 469
 particle explosions, 55
 reverberation, 487
specific weight, 78
speed
 acceleration and, 37
 calculating, 36
 defined, 36
 drag and, 125
 equations of motion and, 91
 hull, 333
 of light, 30–33, 432
 of sound, 484
 units and symbols for, 5, 36
 velocity and, 36
spheres
 bounding, 286
calculating distance along, 433
fluid dynamic drag around, 125–128
great-circle heading, 435
Magnus effect and, 132
mass moment of inertia formula, 15
spherical law of cosines, 433
spherical shell, mass moment of inertia formula,
 15
sports
 about, 369
 baseball examples, 109–111, 135
 billiards examples, 107–109, 378–400
 football game simulation, 54
 golf examples, 116–118, 135, 282, 370–378
spring-damper element formula, 79, 256, 257
springs
 about, 79
 accelerometer theory on, 415
 cloth simulation, 255
 connecting objects, 257
 equation for, 79, 257
 swinging rope example, 259–265
 uses for, 80
stability index (GM), 324
stability of ships and boats, 323–325
standing (shooting position), 361
static forces, 74
steering
 in cars, 342–345
 in hovercraft, 347, 350
 throttle, 337
Stein, Jonathan Y., 413
stereoization process
 about, 467
 active stereoization, 467–469
 passive stereoization, 467, 469
stereopsis, 453
stereoscopic displays, 451
 (see also 3D display)
 about, 451, 454
 viewing frustum, 454–458
stopping distance in cars, 341
strain energy, 105
strain gauges, 444
street games, 428
subtraction (–) operator
 in matrix operations, 513
 in quaternion operations, 521
 in vector operations, 502

- subtraction ($-$) operator
in matrix operations, 511
in quaternion operations, 519
in vector operations, 500
summation in binocular vision, 451
superelevation (roadway banking), 344
superposition principle, 483
suppression in binocular vision, 451
surface acoustic wave technology, 404
surface area, units and symbols for, 5
symmetry, plane of, 27
- T**
- tangential acceleration, 65
tangential velocity, 64, 111
Taylor's theorem, 146, 153
tensors
about, 22
inertia, 24–29, 245
scalars and, 22
vectors and, 22
10/10 maneuver, 336
terminal ballistics, 353
terminal velocity, 130
3D display
about, 451
binocular vision, 451–454
programming considerations, 467–470
stereoscopic basics, 454–458
types of, 458–467
3D particle kinematics
about, 45
hitting the target, 49–54
shooting guns example, 45–54
vectors, 48
x components, 46–47
y components, 47
z components, 48
3D particle kinetics
about, 91–94
shooting guns example, 92–99
x components, 94
y components, 95
z components, 95
3D rigid-body simulation
about, 243
billiard ball example, 381
flight controls, 250–254, 307–319
integrator for, 247–250
model for, 243–247
quaternions in, 239–241
rotation in, 227–241
3D sound, 489–494
throttle steering, 337
thrust
defined, 72, 302
flight control actions, 250, 293
thrust vectoring, 336
thrust-to-propeller RPM ratio, 334
thrust-to-throttle curve ratio, 334
thrust-to-weight ratio, 302
tilt
controlling sprite with, 420–426
sensing with accelerometers, 420
time
equations of motion and, 91
GPS background and, 429–433
laws of motion and, 21
relativistic, 29–33
relativistic, 433
speed and, 36
tracking button position over, 444
units and symbols for, 4, 6
TNT equivalency, 367
toe-in method (cameras), 456
tons per centimeter immersion (TPCM), 327
torque
about, 80
calculating, 80
force versus, 80–83
impulse, 104
laws of motion and, 21
magnitude of, 81
rotational acceleration and, 80
units and symbols for, 6
touch screens
about, 403
custom gestures and, 412
example program, 410–411
haptic feedback, 411
modeling in games, 411
mouse-based input versus, 412
step-by-step physics, 404–410
types of, 403
TPCM (tons per centimeter immersion), 327
trajectories
football simulation game, 54
hitting the target example, 49–54

simple, 120–124
terminal velocity and, 130
zeroing the sights, 357–360
transfer functions (sound), 490
transfer of axis formula, 26
transforming coordinates, 2D rigid-body simulation, 197
transitional ballistics, 353
transpose, in matrix operations, 509
transverse waves, 459
trilateration technique, 429–433
triple scalar product, 505
triple vector product, 24
truncation error, 146–149
tuning
 particle simulation, 186
 2D rigid-body simulation, 204
turbulent flow, 75
turbulent wake, 127
two-rod model, 371
2D particle kinematics, 42–45
2D particle kinetics, 87–91
2D rigid-body simulation
 about, 189
 basic simulator, 201–204
 implementing collision response, 205–225
 integrator for, 198–200
 model for, 190–197
 rendering, 200
 tuning, 204

U

understeering cars, 343
uniform density, 100
unit quaternion, 232, 241, 249
units and measures
 checking dimensional consistency, 4–6
 common mistakes when calculating, 4
 for pressure, 76, 440
 for sound, 480
universal constant, 72

V

variable mass, 138
vector addition, 499, 501
Vector class
 about, 495
 conjugate operator, 501

Magnitude method, 496
Normalize method, 497
in particle simulation, 167
Reverse method, 498
scalar division and, 501
scalar multiplication and, 500
vector addition and, 499
vector subtraction and, 500
vector cross product, 8, 67, 502
vector direction cosines, 46, 497
vector dot-product operator, 8, 503
vector functions and operators
 conjugate operator, 501
 scalar division, 501, 504
 scalar multiplication, 500, 504
 triple scalar product, 505
 vector addition, 499, 501
 vector cross product, 8, 67, 502
 vector dot product, 8, 503
 vector subtraction, 500, 502
vector multiplication
 in matrix operations, 514
 in quaternion operations, 236, 522
vector operations
 direction in, 7
 magnitude in, 7, 496
 Vector class, 495–501
 vector functions and operators, 501–505
vector subtraction, 500, 502

vectors
 acceleration, 20, 475
 defined, 7
 examples of, 7
 orthogonal, 231
 tensors and, 22
 3D particle kinematics and, 48
 time derivative of, 23
velocity
 about, 36–37
 acceleration and, 36–38
 angular, 6, 22, 62–69, 230, 233, 249
 Bernoulli's equation and, 125
 equations for, 88
 instantaneous, 37, 39
 linear, 6, 64, 274
 magnitude of, 36
 muzzle, 354
 relative, 68, 179, 207, 209, 258, 286
 Reynolds number and, 130

second derivatives of, 38
tangential, 64, 111
terminal, 130
vertex-edge collisions, 221–225
vertex-vertex collisions, 219–221
vertical force, 391
vibration energy, 404
viewing frustum, 454–458
Vincenty formula, 433
vines (swinging) example, 258–265
virtual mass, 332
viscosity
 kinematic, 6
 units and symbols for, 6
viscous drag, 347
volume of ships, displacement and, 323
volumetric displays, 451, 466
Von Doenhoff, Albert E., 300
voxels, 466

W

wave drag, 330, 348
weather helm, 348
weight of objects
 about, 4
 ammunition, 354
 buoyancy and, 323
weighted average, 447–448

wetted drag, 348–350
wind force
 as force effector, 288
 particle kinetics in 3D, 93
 in particle simulation, 172–174
 zeroing the sights and, 359
wings, in aircraft, 295
wire-grid polarizer, 459

X

x-axis, rotation around, 7, 230

Y

y-axis, rotation around, 7, 230
yaw
 flight control action for, 227, 250, 304
 local coordinate axes and, 62, 296
 in ships and boats, 327

Z

z-axis, rotation around, 7, 229
zero range, 358
zeroing the sights
 about, 357
 gravity and resistance, 357–359
 wind and, 359

About the Authors

David Bourg is a naval architect involved in various military and commercial proposal, design, and construction efforts. Since 1998, David has served as an independent consultant working for various regional clients engaged in both commercial and military shipbuilding for whom he provides design and analysis services including but not limited to concept design, proposal writing, detailed design and analysis, visualization, and software development. He coordinated and led the winning design and proposal effort for the US Coast Guard Point Class (patrol boat) Replacement Program. In 2006, David joined fellow naval architect Kenneth Humphreys to form MiNO Marine, LLC, a naval architecture and marine professional services firm.

In addition to *Physics for Game Developers*, David has published two other books. He earned a PhD in engineering and applied science in 2008 from the University of New Orleans. He has served as an adjunct professor at the University of New Orleans School of Naval Architecture and Marine Engineering, where he has taught various courses since 1993.

Ever since his father read Stephen Hawking's *A Brief History of Time* to him in middle school, **Bryan Bywalec** wanted to be an astrophysicist. While he will always have a passion for pure physics, he became more and more obsessed in high school with the application of those physical principles he was learning. Having been around sailboats his entire life, his decision to seek a degree in naval architecture at the University of New Orleans surprised few.

While working on his degree, Mr. Bywalec was employed as a network administrator for the College of Engineering. Having an office in an electronics lab, he explored the world of enterprise computing and became very interested in high performance clusters, remote administration of desktops, and robotics.

Upon graduating in 2007, he began his career at MiNO Marine, LLC, and under the guidance of David Bourg and Kenneth Humphreys, now focuses on finite element analysis of complex welded steel structures. His structural analysis work depends largely on the accurate approximations of non-linear physical systems. Bryan has completed several computational fluid dynamics simulations of exhaust gases from ship stacks and current flow around offshore structures.

In addition to his work as a naval architect, Bryan strives to create innovative ways to connect everyday objects to various control networks. From unlocking doors via text message to developing a real-time street car tracking program, he constantly searches for opportunities to integrate technology into his life.

Colophon

The animals on the cover of *Physics for Game Developers, 2nd Edition* are a cat and a mouse. The age-old rivalry between cat and mouse has been the topic of many children's

books and Saturday cartoons. From traditional folk tales, such as Aesop's fables and Grimm Brothers' fairy tales, to today's cartoons, such as *Tom & Jerry*, the cat has chased and bullied the mouse and the mouse has avoided becoming lunch. The cat may be bigger and stronger, but the mouse is small, fast, and can fit in tight spaces, so the end result is often a battle of wits.

The cover image is from a 19-century engraving from the Dover Pictorial Archive. The cover font is Adobe ITC Garamond. The text font is Adobe Minion Pro; the heading font is Adobe Myriad Condensed; and the code font is Dalton Maag's Ubuntu Mono.