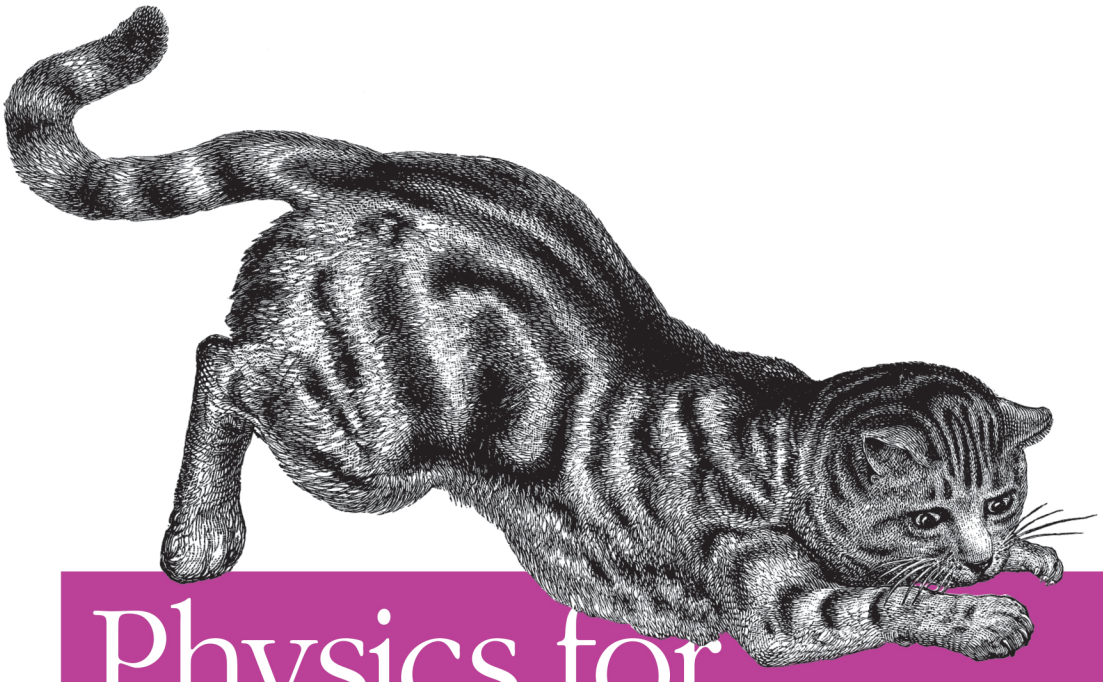
Science, Math, and Code for Realistic Effects

2nd Edition



Physics for Game Developers

David M. Bourg
& Bryan Bywalec

O'REILLY®

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.

"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

US $44.99 CAN $47.99
ISBN: 978-1-449-39251-2

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

Physics for Game Developers, Second Edition

by David M. Bourg and Bryan Bywalec

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
Production Editor: Christopher Hearse
Copyeditor: Rachel Monaghan
Proofreader: Amanda Kersey
Indexer: Lucie Haskins
Cover Designer: Randy Comer
Interior Designer: David Futato
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.

ISBN: 978-1-449-39251-2

[LSI]

Table of Contents

Part III. Physical Modeling

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, *kinematics*, 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 body[3] 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:

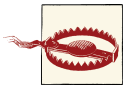
`Constant width`
: Used to indicate command-line computer output, code examples, Registry keys, and keyboard accelerators (see "Keyboard Accelerators" later in this book).

`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, **F**. When referring to the magnitude only of a vector quantity, we use standard type. For example, the magnitude of the vector force, **F**, is F with components along the coordinate axes, Fx, Fy, and Fz. 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:

$$F = ma$$

where F is the resultant force acting on the body, m is the mass of the body, and a is the linear acceleration of the body's center of gravity. We'll discuss this second law in greater detail later in this chapter, but before that there are some more fundamental issues that we must address.

Units and Measures

Over years of teaching various engineering courses, we've observed that one of the most common mistakes students make when performing calculations is using the wrong units for a quantity, thus failing to maintain consistent units and producing some pretty wacky answers. For example, in the field of ship performance, the most commonly misused unit is that for speed: people forget to convert speed in knots to speed in meters per second (m/s) or feet per second (ft/s). One knot is equal to 0.514 m/s, and considering that many quantities of interest in this field are proportional to speed squared, this mistake could result in answers that are as much as 185% off target! So, if some of your results look suspicious later on, the first thing you need to do is go back to your formulas and check their dimensional consistency.

To check dimensional consistency, you must take a closer look at your units of measure and consider their component dimensions. We are not talking about 2D or 3D type dimensions here, but rather the basic measurable dimensions that will make up various *derived* units for the physical quantities that we will be using. These basic dimensions are *mass*, *length*, and *time*.

It is important for you to be aware of these dimensions, as well as the combinations of these dimensions that make up the other derived units, so that you can ensure dimensional consistency in your calculations. For example, you know that the weight of an object is measured in units of force, which can be broken down into component dimensions like so:

$$F = (M)\ (L/T^2)$$

where M is mass, L is length, and T is time. Does this look familiar? Well, if you consider that the component units for acceleration are (L/T^2) and let a be the symbol for acceleration and m be the symbol for the mass of an object, you get:

$$F = ma$$

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:

$$R_f = 1/2\ \rho\ V^2\ S\ C_f$$

In this formula, R_f represents resistance (a force) due to friction, ρ is the density of water, V is the speed of the moving body, S is the submerged surface area of the body, and C_f is an empirical (experimentally determined) drag coefficient for the body. Now rewriting this formula in terms of basic dimensions instead of variables will show that the dimensions on the left side of the formula match exactly the dimensions on the right side. Since R_f is a force, its basic dimensions are of the form:

$$(M)\ (L/T^2)$$

as discussed earlier, which implies that the dimensions of all the terms on the right side of the equation, when combined, must yield an equivalent form. Considering the basic units for density, speed, and surface area:

- Density: $(M)/(L^3)$
- Speed: $(L)/(T)$
- Area: (L^2)

and combining these dimensions for the terms, $\rho\ V^2\ S$, as follows:

$$[(M)/(L^3)]\ [(L)/(T)]^2\ [L^2]$$

and collecting the dimensions in the numerator and denominator yields the following form:

$$(M\,L^2\,L^2) / (L^3\,T^2)$$

Canceling dimensions that appear in both the numerator and denominator yields:

$$M\,(L/T^2)$$

which is consistent with the form shown earlier for resistance, R_f. This exercise also reveals that the empirical term, C_f, for the coefficient of friction must be nondimensional—that is, it is a constant number with no units.

With that, let's take a look at some more common physical quantities that you will be using along with their corresponding symbols, component dimensions, and units in both the SI and English systems. This information is summarized in Table 1-1.

Table 1-1. Common physical quantities and units

Quantity	Symbol	Dimensions	Units, SI	Units, English
Acceleration, linear	A	L/T^2	m/s^2	ft/s^2
Acceleration, angular	α	radian/T^2	radian/s^2	radian/s^2
Density	ρ	M/L^3	kg/m^3	$slug/ft^3$
Force	F	$M\,(L/T^2)$	newton, N	pound, lbs
Kinematic viscosity	v	L^2/T	m^2/s	ft^2/s
Length	L (or x, y, z)	L	meters, m	feet, ft
Mass	m	M	kilogram, kg	slug
Moment (torque)	M[a]	$M\,(L^2/T^2)$	N-m	ft-lbs
Mass Moment of Inertia	I	$M\,L^2$	kg-m^2	lbs-ft-s^2
Pressure	P	$M/(L\,T^2)$	N/m^2	lbs/ft^2
Time	T	T	seconds, s	seconds, s
Velocity, linear	V	L/T	m/s	ft/s
Velocity, angular	ω	radian/T	radian/s	radian/s
Viscosity	μ	M/(L T)	N s/m^2	lbs • s/ft^2

[a] In general, we will use a capital *M* to represent a moment (torque) acting on a body and a lowercase *m* to represent the mass of a body. If we're referring to the basic dimension of mass in a general sense—that is, referring to the dimensional components of derived units of measure—we'll use a capital *M*. Usually, the meanings of these symbols will be obvious based on the context in which they are used; however, we will specify their meanings in cases where ambiguity may exist.

Coordinate System

Throughout this book we will refer to a standard, *right-handed* Cartesian coordinate system when specifying positions in 2D or 3D space. In two dimensions we will use the coordinate system shown in Figure 1-1(a), where rotations are measured positive counterclockwise.

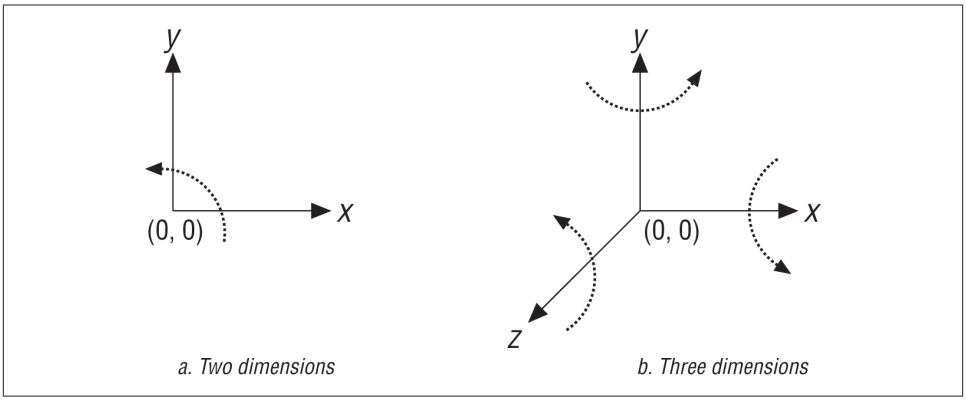


Figure 1-1. Right-handed coordinate system

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, **F**. When referring to the magnitude only of a vector quantity, we'll use standard type. For example, the magnitude of the vector force, **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:

$$|v| = ds/dt$$

where v is the speed, ds is a small distance (a *differential* distance), and dt is a small, differential, period of time. In reality, for our simulations, we'll never deal with infinitely small numbers; we'll use small numbers, such as time intervals of 1 millisecond, but not infinitely small numbers.

For our purposes, you can think of integrals as the reverse, or the inverse, of derivatives; integration is the inverse of differentiation. The symbol $\int$ represents integration. You can think of integration as a process of adding up a bunch of infinitely small chunks of some variable. Here again, we are not going to deal with infinitely small pieces of anything, but instead will consider small, discrete parcels of some variable—for example, a small, discrete amount of time, area, or mass. In these cases, we'll use the Σ symbol instead of the integration symbol. Consider a loaf of bread that's sliced into uniformly thick slices along its whole length. If you wanted to compute the volume of that loaf of bread, you can approximate it by starting at one end and computing the volume of the first slice, approximating its volume as though it were a very short, square cylinder; then

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 angular[1] 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:

$$m = \int \rho \, dV = \rho \int dV$$

In practice, you rarely need to take the volume integral to find the mass of a body, especially considering that many of the bodies we will consider—for example, cars and planes—are not of uniform density. Thus, you will simplify these complicated bodies by breaking them down into an ensemble of component bodies of known or easily calculable mass and simply sum the masses of all components to arrive at the total mass.

The calculation of the center of gravity of a body is a little more involved. First, divide the body into a finite number of elemental masses with the center of each mass specified relative to the reference coordinate system axes. We'll refer to these elemental masses as m_i. Next, take the *first moment* of each mass about the reference axes and then add up all of these moments. The first moment is the product of the mass times the distance along a given coordinate axis from the origin to the center of mass. Finally, divide this sum of moments by the total mass of the body, yielding the coordinates to the center of mass of the body relative to the reference axes. You must perform this calculation once for each dimension—that is, twice when working in 2D and three times when working in 3D. Here are the equations for the 3D coordinates of the center of mass of a body:

$$x_c = \{\int x_o \, dm\} / m$$
$$y_c = \{\int y_o \, dm\} / m$$
$$z_c = \{\int z_o \, dm\} / m$$

where $(x, y, z)_c$ are the coordinates of the center of mass for the body and $(x, y, z)_o$ are the coordinates of the center of mass of each elemental mass. The quantities $x_o \, dm$, $y_o \, dm$, and $z_o \, dm$ represent the first moments of the elemental mass, dm, about each of the coordinate axes.

Here again, don't worry too much about the integrals in these equations. In practice, you will be summing finite numbers of masses and the formulas will take on the friendlier forms shown here:

$$x_c = \{\Sigma \, x_o m_i\} / \{\Sigma \, m_i\}$$
$$y_c = \{\Sigma \, y_o m_i\} / \{\Sigma \, m_i\}$$
$$z_c = \{\Sigma \, z_o m_i\} / \{\Sigma \, m_i\}$$

Note that you can easily substitute weights for masses in these formulas since the constant acceleration due to gravity, g, would appear in both the numerators and denomi-

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:

$$m_t = \Sigma\, m_i$$
$$\mathbf{CG} = [\Sigma\, (\mathbf{cg}_i)\, (m_i)] \,/\, m_t$$

where m_t is the total mass, m_i is the mass of each point mass in the system, **CG** is the combined center of gravity, and $\mathbf{cg}_i$ is the location of the center of gravity of each point mass in design, or reference, coordinates. Notice that **CG** and $\mathbf{cg}_i$ are shown as vectors since they denote position in Cartesian coordinates. This is a matter of convenience since it allows you to take care of the *x*, *y*, and *z* components (or just *x* and *y* in two dimensions) in one shot.

In the code samples that follow, let's assume that the point masses making up the body are represented by an array of structures where each structure contains the point mass's design coordinates and mass. The structure will also contain an element to hold the coordinates of the point mass relative to the combined center of gravity of the rigid body, which will be calculated later.

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

// Assume that _NUMELEMENTS has been defined
PointMassElements[_NUMELEMENTS];
```

Here's some code that illustrates how to calculate the total mass and combined center of gravity of the elements:

```
int  i;
float TotalMass;
Vector CombinedCG;
Vector FirstMoment;

TotalMass = 0;
for(i=0; i<_NUMELEMENTS; i++)
   TotalMass FAC+= Elements[i].mass;

FirstMoment = Vector(0, 0, 0);
for(i=0; i<_NUMELEMENTS; i++)
{
   FirstMoment += Element[i].mass * Element[i].designPosition;
```

```
}
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}, $r_z^2 = x^2 + y^2$.

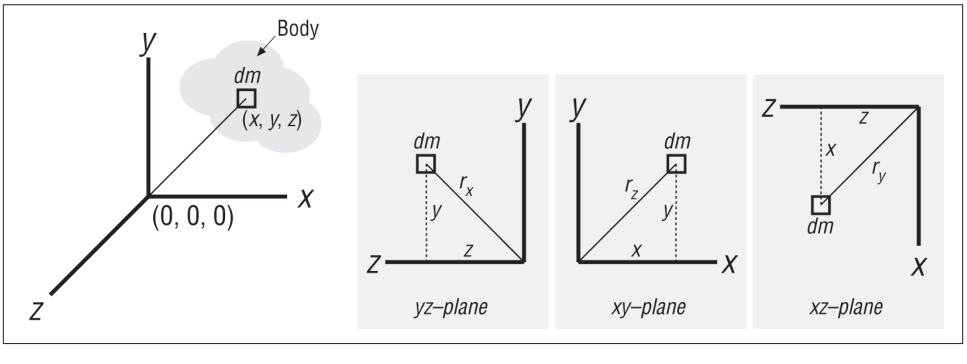


Figure 1-2. Arbitrary body in 3D

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

$$I_{xx} = \int r_x^2 \, dm = \int (y^2 + z^2) \, dm$$
$$I_{yy} = \int r_y^2 \, dm = \int (z^2 + x^2) \, dm$$
$$I_{zz} = \int r_z^2 \, dm = \int (x^2 + y^2) \, dm$$

Let's look for a moment at a common situation that arises in practice. Say you are given the moment of inertia, I_o, of a body about an axis, called the *neutral axis*, passing through

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:

$$I = I_o + md^2$$

where m is the mass of the body and d is the perpendicular distance between the parallel axes.

There is an important practical observation to make here: the new moment of inertia is a function of the distance separating the axes squared. This means that in cases where I_o is known to be relatively small and d relatively large, you can safely ignore I_o, since the md^2 term will dominate. You must use your best judgment here, of course. This formula for transfer of axes also indicates that the moment of inertia of a body will be at its minimum when calculated about an axis passing through the body's center of gravity. The body's moment of inertia about any parallel axis will always increase by an amount, md^2, when calculated about an axis not passing through the body's center of mass.

In practice, calculating mass moment of inertia for all but the simplest shapes of uniform density is a complicated endeavor, so we will often approximate the moment of inertia of a body about axes passing through its center of mass by using simple formulas for basic shapes that approximate the object. Further, we will break down complicated bodies into smaller components and take advantage of the fact that I_o may be negligible for certain components considering its md^2 contribution to the total body's moment of inertia.

Figure 1-3 through Figure 1-7 show some simple solid geometries for which you can easily calculate mass moments of inertia. The mass moment of inertia formulas for each of these simple geometries of homogenous density about the three coordinate axes are shown in the figure captions. You can readily find similar formulas for other basic geometries in college-level dynamics texts (see the Bibliography at the end of this book for a few sources).

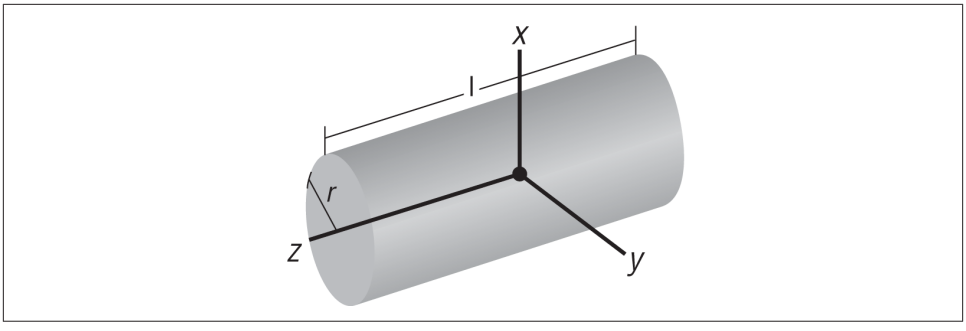


Figure 1-3. Circular cylinder: $I_{xx} = I_{yy} = (1/4)\ mr^2 + (1/12)\ ml^2$; $I_{zz} = (1/2)\ mr^2$

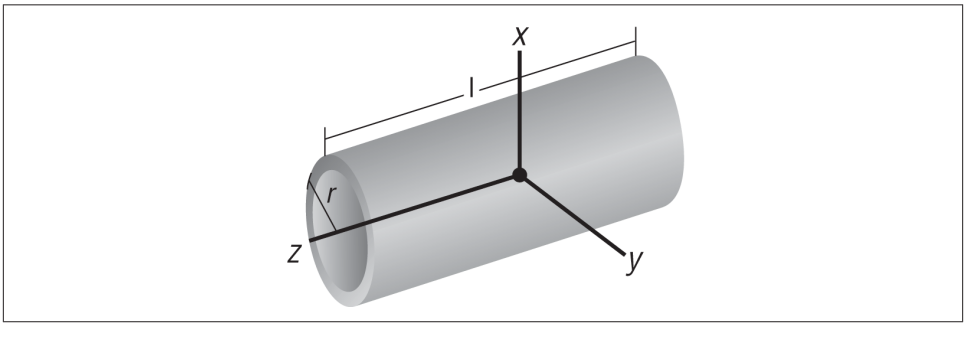


Figure 1-4. Circular cylindrical shell: $I_{xx} = I_{yy} = (1/2)\ mr^2 + (1/12)\ ml^2$; $I_{zz} = mr^2$

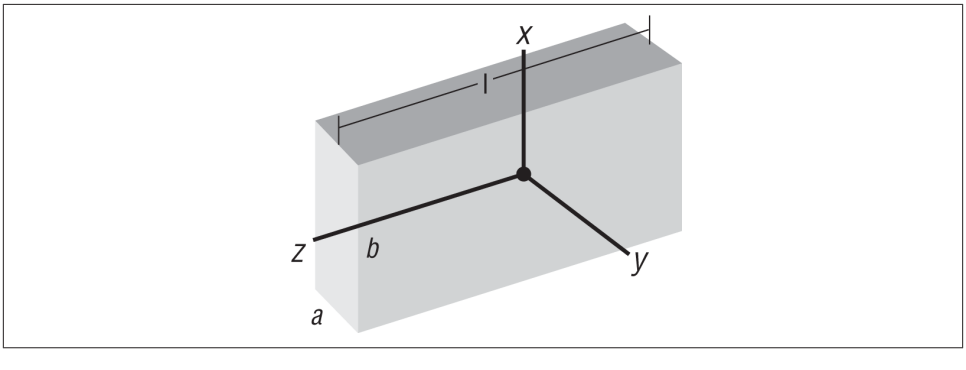


Figure 1-5. Rectangular cylinder: $I_{xx} = (1/12)\ m(a^2 + l^2)$; $I_{yy} = (1/12)\ m(b^2 + l^2)$; $I_{zz} = (1/12)\ m(a^2 + b^2)$

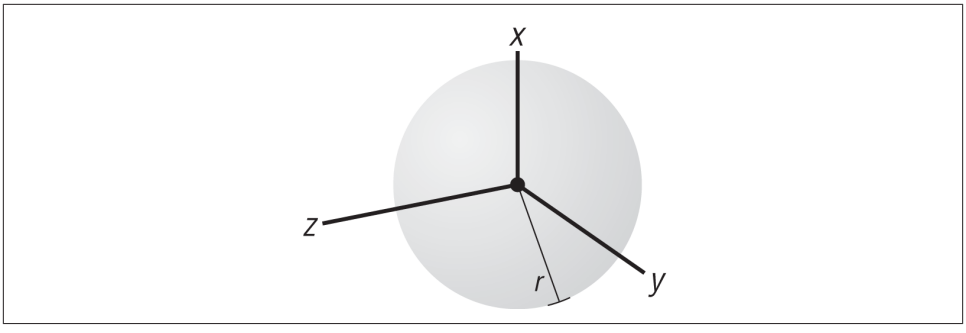


Figure 1-6. Sphere: $I_{xx} = I_{yy} = I_{zz} = (2/5)\ mr^2$

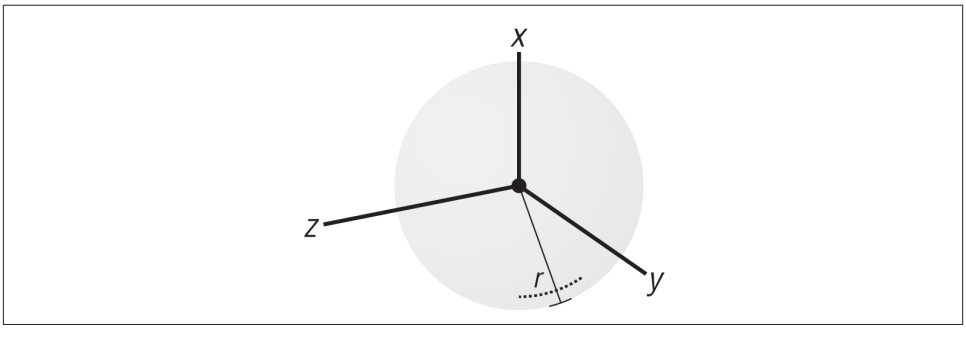


Figure 1-7. Spherical shell: $I_{xx} = I_{yy} = I_{zz} = (2/3)\ mr^2$

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.

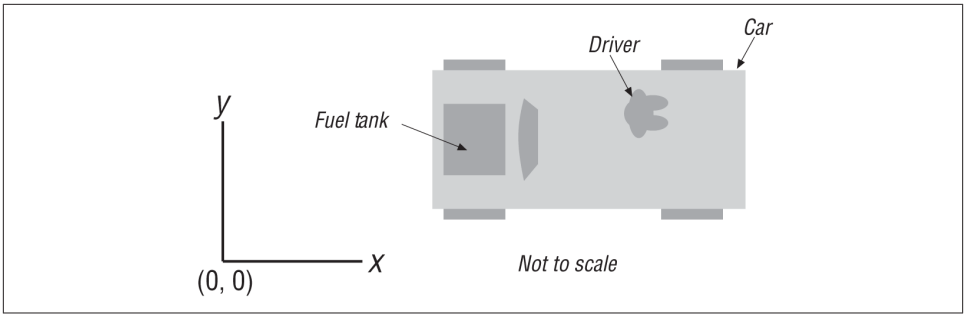


Figure 1-8. Example body consisting of car, driver, and fuel

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:

$$W_{fuel} = \rho v g = (750 \text{ kg/m}^3)\ (0.50 \text{ m})\ (0.90 \text{ m})\ (0.30 \text{ m})\ (9.81 \text{ m/s}^2)$$
$$= 993 \text{ N}$$

Acceleration due to gravity is the acceleration of a falling object as it falls toward the earth. The weight of an object is equal to its mass times the acceleration due to gravity. The symbol g is used to represent the acceleration due to gravity, and on Earth the value of g is approximately 9.8 m/s^2 at sea level. Units for weight in the metric system are Newtons, *N*.

Now, the total weight of the body is:

$$W_{total} = W_{car} + W_{driver} + W_{fuel}$$
$$W_{total} = 17{,}500\ N + 850\ N + 993\ N = 19{,}343\ N$$

To get the mass of the body, you simply divide the weight by the acceleration due to gravity.

$$M_{total} = W_{total/g} = 19{,}343\ N\ /(9.81\ m/s^2) = 1972\ kg$$

The next mass property we want is the location of the center of gravity of the body. In this example we will calculate the centroid relative to the global origin. We will also apply the first moment formula twice, once for the *x* coordinate and again for the *y* coordinate:

$$X_{cg\ body} = \{(x_{cg\ car})(W_{car}) + (x_{cg\ driver})(W_{driver}) + (x_{cg\ fuel})(W_{fuel})\} / W_{total}$$
$$X_{cg\ body} = \{(30.50\ m)(17{,}500\ N) + (31.50\ m)(850\ N) + (28.00\ m)(993\ N)\} / 19{,}343\ N$$
$$X_{cg\ body} = 30.42\ m$$
$$Y_{cg\ body} = \{(y_{cg\ car})(W_{car}) + (y_{cg\ driver})(W_{driver}) + (y_{cg\ fuel})(W_{fuel})\} / W_{total}$$
$$Y_{cg\ body} = \{(30.50\ m)(17{,}500\ N) + (31.00\ m)(850\ N) + (30.50\ m)(993\ N)\} / 19{,}343\ N$$
$$Y_{cg\ body} = 30.52\ m$$

Notice that we used weight in these equations instead of mass. Remember we can do this because the acceleration due to gravity built into the weight value is constant and appears in both the numerator and denominator, thus canceling out.

Now it's time to calculate the mass moment of inertia of the body. This is easy enough in this 2D example since we have only one rotational axis, coming out of the paper, and thus need only perform the calculation once. The first step is to calculate the local moment of inertia of each component about its own neutral axis. Given the limited information we have on the geometry and mass distribution of each component, we will make a simplifying approximation by assuming that each component can be represented by a rectangular cylinder, and will thus use the corresponding formula for moment of inertia from Figure 1-5. In the equations to follow, we'll use a lowercase *w* to represent width so as to not confuse it with weight, where we've been using a capital *W*.

$$I_{o\ car} = (m/12)\ (w^2 + L^2)$$

$$I_{o\ car} = ((17{,}500\ N / 9.81\ m/s^2) / 12\ )\ ((1.80\ m)^2 + (4.70\ m)^2) =$$
$$3765.5\ N - s^2 - m$$
$$I_{o\ driver} = (m/12)\ (w^2 + L^2)$$
$$I_{o\ driver} = ((850\ N / 9.81\ m/s^2) / 12)\ ((0.50\ m)^2 + (0.90\ m)^2) = 7.7$$
$$N - s^2 - m$$
$$I_{o\ fuel} = (m/12)\ (w^2 + L^2)$$
$$I_{o\ fuel} = ((993\ N / 9.81\ m/s^2) / 12)\ ((0.90\ m)^2 + (0.50\ m)^2) = 8.9\ N$$
$$- s^2 - m$$

Since these are the moments of inertia of each component about its own neutral axis, we now need to use the parallel axis theorem to transfer these moments to the neutral axis of the body, which is located at the body center of gravity that we recently calculated. To do this, we must find the distance from the body center of gravity to each component's center of gravity. The distances squared from each component to the body center of gravity are:

$$d^2_{car} = (x_{cg\ car} - X_{cg})^2 + (y_{cg\ car} - Y_{cg})^2$$
$$d^2_{car} = (30.50\ m - 30.42\ m)^2 + (30.50\ m - 30.53\ m)^2 = 0.01\ m^2$$
$$d^2_{driver} = (x_{cg\ driver} - X_{cg})^2 + (y_{cg\ driver} - Y_{cg})^2$$
$$d^2_{driver} = (31.50\ m - 30.42\ m)^2 + (31.25\ m - 30.53\ m)^2 = 1.68$$
$$m^2$$
$$d^2_{fuel} = (x_{cg\ fuel} - X_{cg})^2 + (y_{cg\ fuel} - Y_{cg})^2$$
$$d^2_{fuel} = (28.00\ m - 30.42\ m)^2 + (30.50\ m - 30.53\ m)^2 = 5.86\ m^2$$

Now we can apply the parallel axis theorem as follows:

$$I_{cg\ car} = I_o + md^2$$
$$I_{cg\ car} = 3765.5\ N - s^2 - m + (17{,}500\ N / 9.81\ m/s^2)\ (0.01\ m^2) =$$
$$3783.34\ N - s^2 - m$$
$$I_{cg\ driver} = I_o + md^2$$
$$I_{cg\ driver} = 7.7\ N - s^2 - m + (850\ N / 9.81\ m/s^2)\ (1.68\ m^2) =$$
$$153.27\ N - s^2 - m$$
$$I_{cg\ fuel} = I_o + md^2$$
$$I_{cg\ fuel} = 8.9\ N - s^2 - m + (993\ N / 9.81\ m/s^2)\ (5.86\ m^2) =$$
$$602.07\ N - s^2 - m$$

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:

$$I_{cg\ total} = I_{cg\ car} + I_{cg\ driver} + I_{cg\ fuel}$$
$$I_{cg\ total} = 3783.34\ N - s^2 - m + 153.27\ N - s^2 - m + 602.07\ N - s^2 - m = 4538.68\ N - s^2 - m$$

The mass properties of the body—that is, the combination of the car, driver, and full tank of fuel—are shown in Table 1-3.

Table 1-3. Example summary of mass properties

Property	Computed value
Total mass (weight)	1972 kg (19,343 N)
Combined center of mass location	$(x,y) = (30.42\ m, 30.53\ m)$
Mass moment of inertia	$4538.68\ N - s^2 - m$

It is important that you understand the concepts illustrated in this example well because as we move on to more complicated systems and especially to general motion in 3D, these calculations are only going to get more complicated. Moreover, the motion of the bodies to be simulated are functions of these mass properties, where mass will determine how these bodies are affected by forces, center of mass will be used to track position, and mass moment of inertia will determine how these bodies rotate under the action of noncentroidal forces.

So far, we have looked at moments of inertia about the three coordinate axes in 3D space. However, in general 3D rigid-body dynamics, the body may rotate about any axis—not necessarily one of the coordinate axes, even if the local coordinate axes pass through the body center of mass. This complication implies that we must add a few more terms to our set of *I*'s for a body to handle this generalized rotation. We will address this topic further later in this chapter, but before we do that we need to go over Newton's second law of motion in detail.

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:

$$F = ma$$

where F is the resultant force acting on the body, m is the mass of the body, and a is the linear acceleration of the body center of gravity.

If you rearrange this equation as follows:

$$F/m = a$$

you can see how the mass of a body acts as a measure of resistance to motion. Observe here that as mass increases in the denominator for a constant applied force, then the resulting acceleration of the body will decrease. You could say that the body of greater mass offers greater resistance to motion. Similarly, as the mass decreases for a constant applied force, then the resulting acceleration of the body will increase, and you could say that the body of smaller mass offers lower resistance to motion.

Newton's second law also states that the resulting acceleration is in the same direction as the resultant force on the body; thus, force and acceleration must be treated as vector quantities. In general, there may be more than one force acting on the body at a given time, which means that the resultant force is the vector sum of all forces acting on the body. So, you can now write:

$$\sum \mathbf{F} = m\mathbf{a}$$

where **a** represents the acceleration vector.

In 3D, the force and acceleration vectors will have x, y, and z components in the Cartesian reference system. In this case, the component equations of motion are written as follows:

$$\sum F_x = ma_x$$
$$\sum F_y = ma_y$$
$$\sum F_z = ma_z$$

An alternative way to interpret Newton's second law is that the sum of all forces acting on a body is equal to the rate of change of the body's momentum over time, which is the derivative of momentum with respect to time. Momentum equals mass times velocity, and since velocity is a vector quantity, so is momentum. Thus:

$$\mathbf{G} = m\mathbf{v}$$

where **G** is linear momentum of the body, *m* is the body's mass, and **v** is velocity of the center of gravity of the body. The time rate of change of momentum is the derivative of momentum with respect to time:

$$d\mathbf{G}/dt = d/dt\ (m\mathbf{v})$$

Assuming that the body mass is constant (for now), you can write:

$$d\mathbf{G}/dt = m\ d\mathbf{v}/dt$$

Observing that the time rate of change of velocity, d**v**/dt, is acceleration, we arrive at:

$$d\mathbf{G}/dt = m\mathbf{a}$$

and:

$$\sum \mathbf{F} = d\mathbf{G}/dt = m\mathbf{a}$$

So far we have considered only translation of the body without rotation. In generalized 3D motion, you must account for the rotational motion of the body and will thus need some additional equations to fully describe the body's motion. Specifically, you will require analogous formulas relating the sum of all moments (torque) on a body to the rate of change in its angular momentum over time, or the derivative of angular momentum with respect to time. This gives us:

$$\sum \mathbf{M}_{cg} = d/dt\ (\mathbf{H}_{cg})$$

where $\sum \mathbf{M}_{cg}$ is the sum of all moments about the body center of gravity, and **H** is the angular momentum of the body. $\mathbf{M}_{cg}$ can be expressed as:

$$\mathbf{M}_{cg} = \mathbf{r} \times \mathbf{F}$$

where **F** is a force acting on the body, and **r** is the distance vector from **F**, perpendicular to the line of action of **F** (i.e., perpendicular to the vector **F**), to the center of gravity of the body, and × is the vector cross-product operator.

The angular momentum of the body is the sum of the moments of the momentum of all particles in the body about the axis of rotation, which in this case we assume passes through the center of gravity of the body. This can be expressed as:

$$\mathbf{H}_{cg} = \sum \mathbf{r}_i \times m_i\ (\boldsymbol{\omega} \times \mathbf{r}_i)$$

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:

$$H_{cg} = \int \omega \, \mathbf{r}^2 \, dm$$

Given that the angular velocity is the same for all particles making up the rigid body, we have:

$$H_{cg} = \omega \int \mathbf{r}^2 \, dm$$

and recalling that moment of inertia, *I*, equals $\int \mathbf{r}^2 \, dm$, we get:

$$H_{cg} = I\omega$$

Taking the derivative with respect to time, we obtain:

$$dH_{cg}/dt = d/dt \, (I\omega) = I \, d\omega/dt = I\alpha$$

where α is the angular acceleration of the body about a given axis.

Finally, we can write:

$$\sum \mathbf{M}_{cg} = \mathbf{I}\alpha$$

As we stated in our discussion on mass moment of inertia, we will have to further generalize our formulas for moment of inertia and angular moment to account for rotation about any body axis. Generally, **M** and α will be vector quantities, while **I** will be a *tensor*[2] since the magnitude of moment of inertia for a body may vary depending on the axis of rotation (see the sidebar "Tensors" on page 22).

Tensors

A tensor is a mathematical expression that has magnitude and direction, but its magnitude may not be unique depending on the direction. Tensors are typically used to represent properties of materials where these properties have different magnitudes in different directions. Materials with properties that vary depending on direction are called *anisotropic* (*isotropic* implies the same magnitude in all directions). For example, consider the elasticity (or strength) of two common materials, a sheet of plain paper

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:

$$(d\mathbf{V}/dt)_{fixed} = (d\mathbf{V}/dt)_{rot} + (\boldsymbol{\omega} \times \mathbf{V})$$

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 ($\boldsymbol{\omega} \times \mathbf{V}$) term represents the difference between $\mathbf{V}$'s time derivative as measured in the fixed coordinate system and $\mathbf{V}$'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 $\mathbf{H}_{cg}$. Recall that $\mathbf{H}_{cg} = \mathbf{I}\boldsymbol{\omega}$ 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 $\mathbf{H}_{cg}$ in place of $\mathbf{V}$ in the derivative transform relation as follows:

$$\sum \mathbf{M}_{cg} = d\mathbf{H}_{cg}/dt = \mathbf{I}\,(d\boldsymbol{\omega}/dt) + (\boldsymbol{\omega} \times (\mathbf{I}\,\boldsymbol{\omega}))$$

where the moments, inertia tensor, and angular velocity are all expressed in local (body) coordinates. Although this equation looks a bit more complicated than the one we showed you earlier, it is much more convenient to use since $\mathbf{I}$ will be constant throughout your simulation (unless your body's mass or geometry changes for some reason during your simulation), and the moments are relatively easy to calculate in local coordinates. You'll put this equation to use in Chapter 15 when we show you how to develop a simple 3D rigid-body simulator.

Inertia Tensor

Take another look at the angular equation of motion and notice that we set the inertia term, $\mathbf{I}$, in bold, implying that it is a vector. You've already seen that, for two-dimensional problems, this inertia term reduces to a scalar quantity representing the moment of inertia about the single axis of rotation. However, in three dimensions there are three coordinate axes about which the body can rotate. Moreover, in generalized three dimensions, the body can rotate about any arbitrary axis. Thus, for three-dimensional problems, $\mathbf{I}$ is actually a 3×3 matrix—a second-rank tensor.

To understand where this inertia matrix comes from, you must look again at the angular momentum equation:

$$\mathbf{H}_{cg} = \int (\mathbf{r} \times (\boldsymbol{\omega} \times \mathbf{r}))\, dm$$

where $\boldsymbol{\omega}$ is the angular velocity of the body; $\mathbf{r}$ is the distance from the body's center of gravity to each elemental mass, dm; and $(\mathbf{r} \times (\boldsymbol{\omega} \times \mathbf{r}))dm$ is the angular momentum of each elemental mass. The term in parentheses is called a *triple vector product* and can be expanded by taking the vector cross products. $\mathbf{r}$ and $\boldsymbol{\omega}$ are vectors that can be written as follows:

$$\mathbf{r} = x\,\mathbf{i} + y\,\mathbf{j} + z\,\mathbf{k}$$
$$\boldsymbol{\omega} = \omega_x\,\mathbf{i} + \omega_y\,\mathbf{j} + \omega_z\,\mathbf{k}$$

Expanding the triple vector product term yields:

$$\begin{aligned}\mathbf{H}_{cg} = \int \{&[(y^2 + z^2)\omega_x - xy\omega_y - xz\omega_z]\,\mathbf{i} + \\ &[-yx\omega_x + (z^2 + x^2)\omega_y - yz\omega_z]\,\mathbf{j} + \\ &[-zx\omega_x - zy\omega_y + (x^2 + y^2)\omega z]\,\mathbf{k}\}\,dm\end{aligned}$$

To simplify this equation, let's replace a few terms by letting:

$$\begin{aligned}I_{xx} &= \int (y^2 + z^2)\,dm \\ I_{yy} &= \int (z^2 + x^2)\,dm \\ I_{zz} &= \int (x^2 + y^2)\,dm \\ I_{xy} = I_{yx} &= \int (xy)\,dm \\ I_{xz} = I_{zx} &= \int (xz)\,dm \\ I_{yz} = I_{zy} &= \int (yz)\,dm\end{aligned}$$

Substituting these *I* variables, some of which should look familiar to you, back into the expanded equation yields:

$$\begin{aligned}\mathbf{H}_{cg} = &[I_{xx}\,\omega_x - I_{xy}\,\omega_y - I_{xz}\,\omega_z]\,\mathbf{i} + \\ &[-I_{yx}\,\omega_x + I_{yy}\,\omega_y - I_{yz}\,\omega_z]\,\mathbf{j} + \\ &[-I_{zx}\,\omega_x - I_{zy}\,\omega_y + I_{zz}\,\omega_z]\,\mathbf{k}\end{aligned}$$

Simplifying this a step further by letting **I** be a matrix:

$$\mathbf{I} = \begin{matrix} I_{xx} & -I_{xy} & -I_{xz} \\ -I_{yx} & I_{yy} & -I_{yz} \\ -I_{zx} & -I_{zy} & I_{zz} \end{matrix}$$

yields the following equation:

$$\mathbf{H}_{cg} = \mathbf{I}\,\boldsymbol{\omega}$$

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):

$$I_{xy} = I_{yx} = \int (xy)\, dm$$
$$I_{xz} = I_{zx} = \int (xz)\, dm$$
$$I_{yz} = I_{zy} = \int (yz)\, dm$$

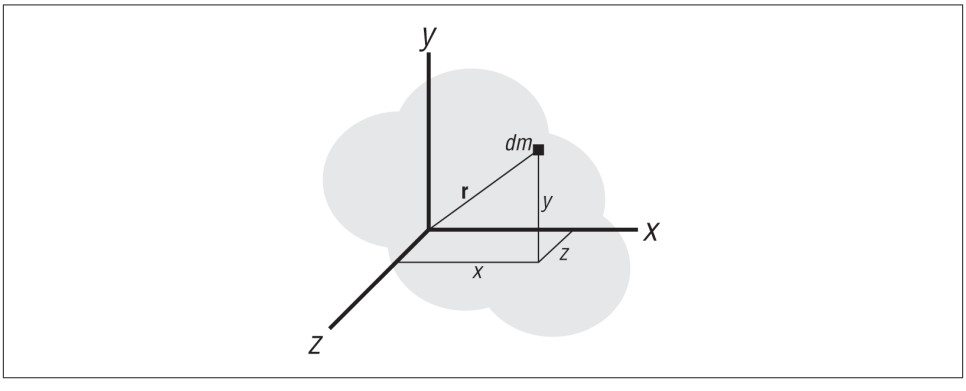


Figure 1-9. Products of inertia

Just like the parallel axis theorem, there's a similar transfer of axis formula that applies to products of inertia:

$$I_{xy} = I_{o(xy)} + m\, d_x\, d_y$$
$$I_{xz} = I_{o(xz)} + m\, d_x\, d_z$$
$$I_{yz} = I_{o(yz)} + m\, d_y\, d_z$$

where the I_o terms represent the local products of inertia (that is, the products of inertia of the object about axes that pass through its own center of gravity), m is the object's mass, and the d terms are the distances between the coordinate axes that pass through the object's center of gravity and a parallel set of axes some distance away (see Figure 1-10).

You'll notice that we did not give you any product of inertia formulas for the simple shapes shown earlier in Figure 1-3 through Figure 1-7. The reason is that the given moments of inertia were about the *principal axes* for these shapes. For any body, there exists a set of axes—called the principal axes—oriented such that the product of inertia terms in the inertia tensor are all zero.

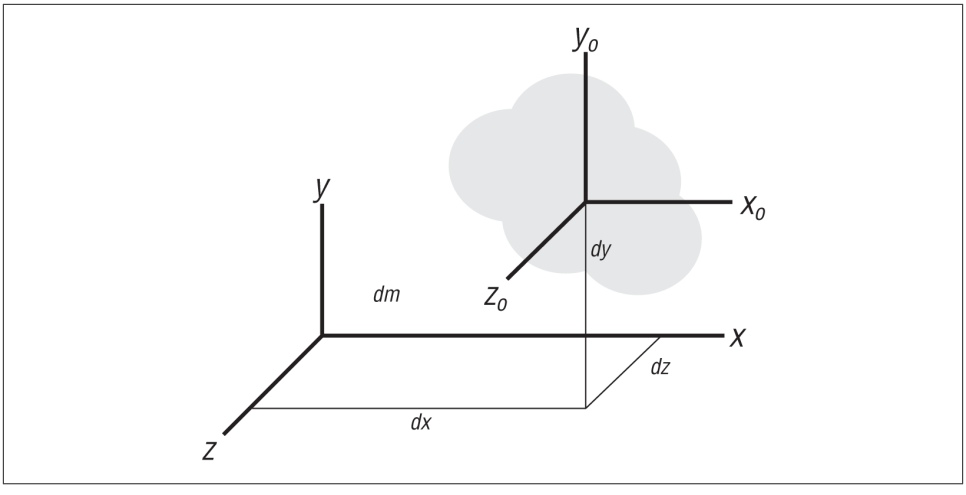


Figure 1-10. Transfer of axes

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.

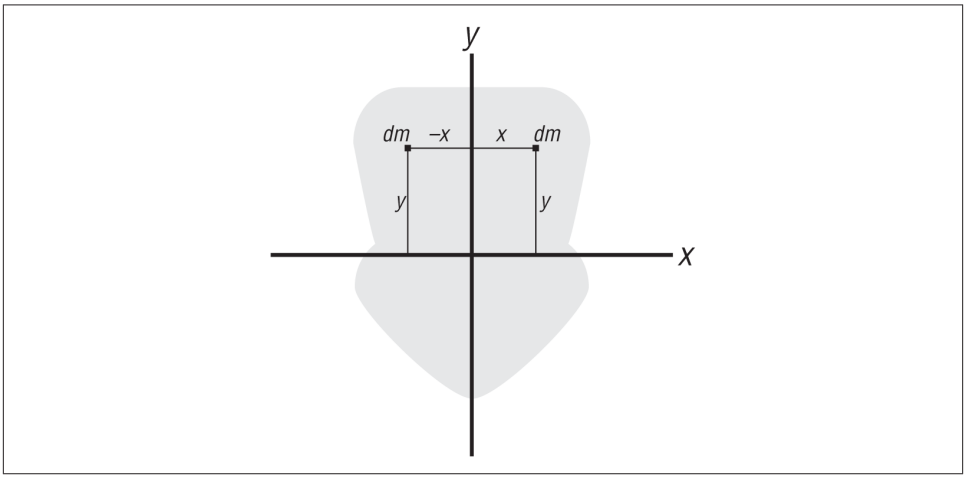


Figure 1-11. Symmetry

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);

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

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

        Iyz += 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 = -Ixy;
    InertiaTensor.e13 = -Ixz;

    InertiaTensor.e21 = -Ixy;
    InertiaTensor.e22 = Iyy;
    InertiaTensor.e23 = -Iyz;

    InertiaTensor.e31 = -Ixz;
    InertiaTensor.e32 = -Iyz;
    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 1.5*c*. 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:

$$\Delta t = 2L/c$$

Where *L* is the distance between the mirrors and *c* is the speed of light. Figure 1-12 shows what the clock would look like if you were above it traveling at its velocity.

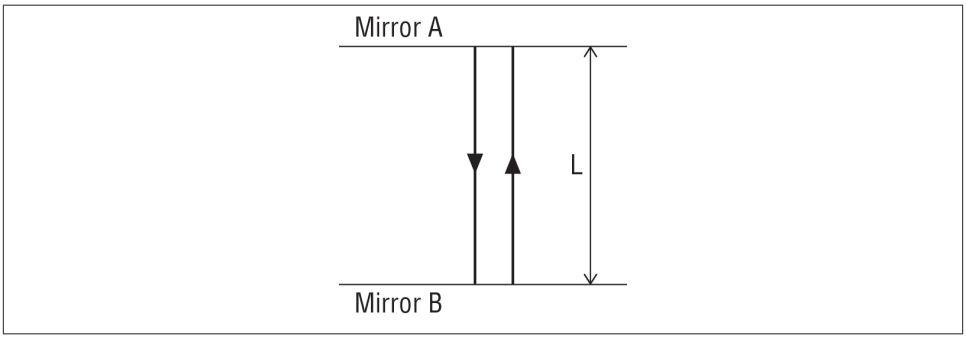


Figure 1-12. Traveling with the clock

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.

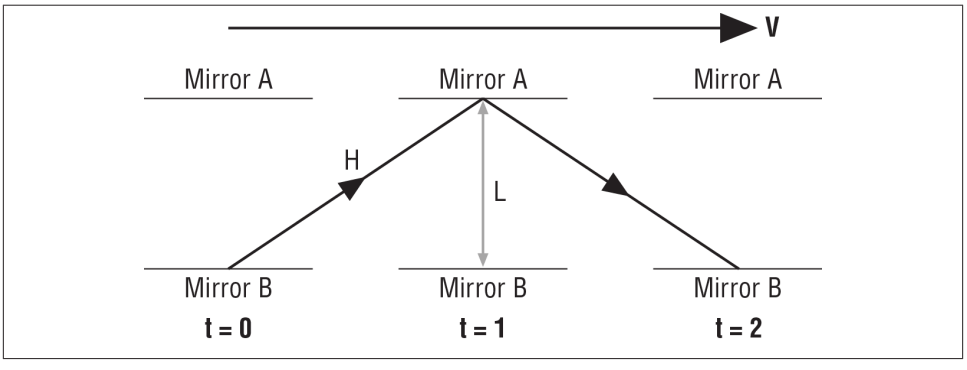


Figure 1-13. Stationary with respect to the clock

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:

$$c = 2L/\Delta t$$

but because the speed of light must be held constant in all frames of reference via locality, we also have:

$$c = 2H/\Delta t$$

For the two preceding equations to be equivalent, Δt must be different for each system.

This means that if I were in a rocket moving at high velocity past you as you read this book, you would look at the clock on my rocket wall and see it ticking more slowly than your clock. Now, it may seem as though I would look out of my rocket and see your clock running fast, but in fact the opposite is true. I would consider myself at rest and you speeding past me such that I would say *your* clock is running slowly. This may seem counterintuitive, but think of it in the same way as visual perspective. If you are at a great distance from me, then you appear to be small. That doesn't imply that I would appear to be huge to you!

Now the amount of dilation for a given velocity *v* is given by the *Lorentz transformation*:

$$\Delta t' = \gamma \Delta t$$

where:

$$\gamma = \frac{1}{\sqrt{1 - \frac{v^2}{c^2}}}$$

is called the *Lorentz factor*.

For velocities approaching the speed of light, the effect of time dilation is dramatic. Imagine if you had a twin sister. She boards a spaceship and is accelerated to three-fourths the speed of light relative to you on Earth. Upon her return, according to her clock, 20 years have elapsed since she left. However, due to time dilation you will have aged 30 years. While this seems paradoxical given that you both will have observed each other's time as running slowly compared to your own, the paradox is resolved by the fact that special relativity claims only that inertial frames of reference are identical. For the spaceship to return to Earth, it must accelerate by changing directions or, in other words, become a noninertial frame of reference. Once the ship is no longer an inertial frame of reference, there will appear to be a jump in the stay-at-home twin's age. Strange but true!

In addition to time dilation due to relative velocity, time also slows down in the presence of strong gravitational fields as a result of Einstein's theory of general relativity. This sort of time dilation is not relative in the sense that if I were closer to a black hole than you, we'd both agree that my clock is ticking less often than yours. However, given that all physical processes would be slower, there is no way to establish which clock is "faster" and which is "slower." It is all relative!

As some games require a way to speed up or slow down time, the applications of time dilation allow for a physical phenomenon to enable time manipulation. Imagine your character needing to be sent to the future to complete some task. That character could be put into a centrifuge and accelerated near the speed of light. Upon exiting the centrifuge, he would essentially have time-traveled into the future. However, if you plan to stay within the limits of physics, then it is a one-way trip: there is no way to reverse time in relativity! Additionally, if you would like to send a character to a nearby star, you can do so within the limits of physics as well. By accelerating a spaceship, a human being would be able to travel a great distance in a lifetime. In fact, with a constant acceleration of 9.8 m/s^2, which would make the astronauts feel like they were on Earth, you could travel the entire visible galaxy. However, you would have essentially time-traveled billions of years into the future. We are sure that you can imagine any number of scenarios where such a mechanism might make your game more interesting while always remaining in the realm of the physically possible!

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:

$$v = \Delta s/\Delta t$$

where v is speed, the magnitude of velocity $\mathbf{v}$, and Δs is distance traveled over the time interval Δt. Note that this relation reveals that the units for speed are composed of the basic dimension's length divided by time, L/T. Some common units for speed are meters per second, *m/s*; feet per second, *ft/sec*; and miles per hour, *mi/hr*.

Here's a simple example (illustrated in Figure 2-1): a car is driving down a straight road and passes marker one at time t_1 and marker two at time t_2, where t_1 equals 0 seconds and t_2 equals 1.136 seconds. The distance between these two markers, s, is 30 m. Calculate the speed of the car.

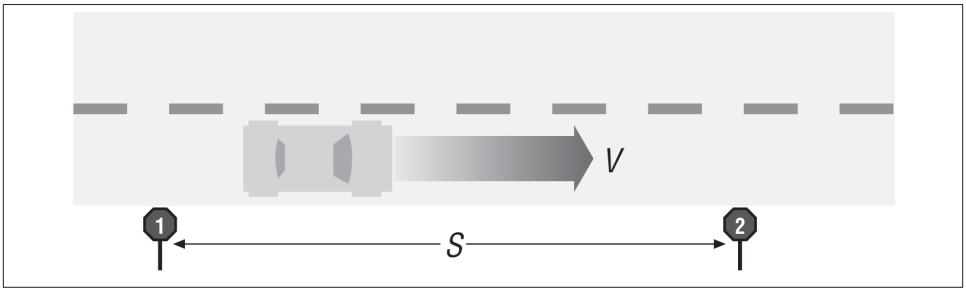


Figure 2-1. Example car speed

You are given that s equals 30 m; therefore, Δs equals 30 m and Δ_t equals $t_2 - t_1$ or 1.136 seconds. The speed of the car over this distance is:

$$v = \Delta s/\Delta t = 30 \text{ m}/1.136 \text{ sec} = 26.4 \text{ m/sec}$$

which is approximately 60 mi/hr. This is a simple one-dimensional example, but it brings up an important point, which is that the speed just calculated is the average speed of the car over that distance. You don't know anything at this point about the car's acceleration,

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:

$$v = \lim_{\Delta t \to 0} (\Delta s / \Delta t)$$

In differential terms, velocity is the derivative of displacement (change in position) with respect to time:

$$v = ds/dt$$

You can rearrange this relationship and integrate over the intervals from s_1 to s_2 and t_1 to t_2, as shown here:

$$\begin{gathered} v\ dt = ds \\ \int_{(s1 \text{ to } s2)} ds = \int_{(t1 \text{ to } t2)} v\ dt \\ s2 - s1 = \Delta s = \int_{(t1 \text{ to } t2)} v\ dt \end{gathered}$$

This relation shows that displacement is the integral of velocity over time. This gives you a way of working back and forth between displacement and velocity.

Kinematics makes an important distinction between displacement and distance traveled. In one dimension, displacement is the same as distance traveled; however, with vectors in space, displacement is actually the vector from the initial position to the final position without regard to the path traveled, while displacement is the difference between the starting position coordinates and the ending position coordinates. Thus, you need to be careful when calculating average velocity given displacement if the path from the starting position to the final position is not a straight line. When Δt is very small (as it approaches 0), displacement and distance traveled are the same.

Another important kinematic property is acceleration, which should also be familiar to you. Referring to your driving experience, you know that acceleration is the rate at which you can increase your speed. Your friend who boasts that his brand new XYZ 20II can go from 0 to 60 in 4.2 seconds is referring to acceleration. Specifically, he is referring to average acceleration.

Formally, average acceleration is the rate of change in velocity, or Δv over Δt:

$$a = \Delta v / \Delta t$$

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

$$a = \lim_{\Delta t \to 0} \Delta v / \Delta t$$
$$a = dv/dt$$

Thus, acceleration is the time rate of change in velocity, or, the derivative of velocity with respect to time.

Multiplying both sides by *dt* and integrating yields:

$$dv = a\ dt$$
$$\int_{(v1 \text{ to } v2)} dv = \int_{(t1 \text{ to } t2)} a\ dt$$
$$v2 - v1 = \Delta v = \int_{(t1 \text{ to } t2)} a\ dt$$

This relationship provides a means to work back and forth between velocity and acceleration.

Thus, the relationships between displacement, velocity, and acceleration are:

$$a = dv/dt = d^2s/dt^2$$

and:

$$v\ dv = a\ ds$$

This is the kinematic differential equation of motion (see the sidebar "Second Derivatives" on page 38 for some helpful background). In the next few sections you'll see some examples of the application of these equations for some common classes of problems in kinematics.

Second Derivatives

In Chapter 1, we explained that you need not worry about the use of derivatives and integrals in this book if you're unfamiliar with calculus since we'll show you how to implement the code to take care of them computationally. That still applies, but here we've introduced new notation:

$$d^2s/dt^2$$

which is an equation representing acceleration as the *second derivative* of distance traveled with respect to time. You can think of second derivatives as just two successive derivatives in the manner we explained in Chapter 1. In the case of distance traveled, velocity, and acceleration, the first derivative of distance traveled with respect to time is

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:

$$\int_{(v1 \text{ to } v2)} dv = \int_{(t1 \text{ to } t2)} a\, dt$$
$$\int_{(v1 \text{ to } v2)} dv = a \int_{(t1 \text{ to } t2)} dt$$
$$v_2 - v_1 = a \int_{(t1 \text{ to } t2)} dt$$
$$v_2 - v_1 = a\,(t_2 - t_1)$$
$$v_2 = a\,t_2 - a\,t_1 + v_1$$

When t_1 equals 0, you can rewrite this equation in the following form:

$$v_2 = a\,t_2 + v_1$$
$$v_2 = v_1 + a\,t_2$$

This simple equation allows you to calculate the instantaneous velocity at any given time by knowing the elapsed time, the initial velocity, and the constant acceleration.

You can also derive an equation for velocity as a function of displacement instead of time by considering the kinematic differential equation of motion:

$$v\,dv = a\,ds$$

Integrating both sides of this equation yields the following alternative function for instantaneous velocity:

$$\int_{(v1 \text{ to } v2)} v\,dv = a \int_{(s1 \text{ to } s2)} ds$$
$$(v_2^2 - v_1^2) / 2 = a\,(s_2 - s_1)$$
$$v_2^2 = 2a\,(s_2 - s_1) + v_1^2$$

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

$$v\ dt = ds$$

with the formula derived earlier for instantaneous velocity:

$$v_2 = v_1 + at$$

substituted for *v*. Doing so yields the formula:

$$s_2 = s_1 + v_1\ t + (a\ t^2) / 2$$

In summary, the three preceding kinematic equations derived are:

$$v_2 = v_1 + a\ t_2$$
$$v_2{}^2 = 2a\ (s_2 - s_1) + v_1{}^2$$
$$s_2 = s_1 + v_1\ t + (a\ t^2) / 2$$

Remember, these equations are valid only when acceleration is constant. Note that acceleration can be 0 or even negative in cases where the body is decelerating.

You can rearrange these equations by algebraically solving for different variables, and you can also derive other handy equations using the approach that we just demonstrated. For your convenience, we've provided some other useful kinematic equations for constant acceleration problems in Table 2-1.

Table 2-1. Constant acceleration kinematic formulas

To find:	Given these:	Use this:
a	$\Delta t, v_1, v_2$	$a = (v_2 - v_1) / \Delta t$
a	$\Delta t, v_1, \Delta s$	$a = (2\ \Delta s - 2\ v_1\ \Delta t) / (\Delta t)^2$
a	$v_1, v_2, \Delta s$	$a = (v_2{}^2 - v_1{}^2) / (2\ \Delta s)$
Δs	a, v_1, v_2	$\Delta s = (v_2{}^2 - v_1{}^2) / (2a)$
Δs	$\Delta t, v_1, v_2$	$\Delta s = (\Delta t / 2)\ (v_1 + v_2)$
Δt	a, v_1, v_2	$\Delta t = (v_2 - v_1) / a$
Δt	$a, v_1, \Delta s$	$\Delta t = \left(\sqrt{v_1^2 + 2a\Delta s}\ -\ v_1\right) / a$
Δt	$v_1, v_2, \Delta s$	$\Delta t = (2\Delta s) / (v_1 + v_2)$
v_1	$\Delta t, a, v_2$	$v_1 = v_2 - a\Delta t$
v_1	$\Delta t, a, \Delta s$	$v_1 = \Delta s/\Delta t - (a\Delta t)/2$

To find:	Given these:	Use this:
v_1	$a, v_2, \Delta s$	$v_1 = \sqrt{v_2^2 - 2a\Delta s}$

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:

$$a = -kv^2$$

where k is a constant and the negative sign indicates that this acceleration acts in the direction opposing the body's velocity. Now substituting this formula for acceleration into the previous equation and then rearranging yields:

$$\begin{aligned} a &= dv/dt \\ -kv^2 &= dv/dt \\ -k\,dt &= dv/v^2 \end{aligned}$$

If you integrate the right side of this equation from v_1 to v_2 and the left side from 0 to t, and then solve for v_2, you'll get this formula for the instantaneous velocity as a function of the initial velocity and time:

$$\begin{aligned} -k \int_{(0\ \text{to}\ t)} dt &= \int_{(v1\ \text{to}\ v2)} (1/v^2)\,dv \\ -k\,t &= 1/v_1 - 1/v_2 \\ v_2 &= v_1 / (1 + v_1 k\,t) \end{aligned}$$

If you substitute this equation for v in the relation $v = ds/dt$ and integrate again, you'll end up with a new equation for displacement as a function of initial velocity and time; see the following procedure:

$$v\,dt = ds;\ \textit{where}\ v = v_1 / (1 + v_1 k\,t)$$

$$\int_{(0 \text{ to } t)} v\, dt = \int_{(s1 \text{ to } s2)} ds$$
$$\int_{(0 \text{ to } t)} [v_1 / (1 + v_1 k\, t)]\, dt = \int_{(s1 \text{ to } s2)} ds$$
$$\ln(1 + v_1\, k\, t) / k = s_2 - s_1$$

If s_1 equals 0, then:

$$s = \ln(1 + v_1\, k\, t) / k$$

Note that in this equation, *ln* is the natural logarithm operator.

This example demonstrates the relative complexity of nonconstant acceleration problems versus constant acceleration problems. It's a fairly simple example where you are able to derive closed-form equations for velocity and displacement. In practice, however, there may be several different types of forces acting on a given body in motion, which could make the expression for induced acceleration quite complicated. This complexity would render a closed-form solution like the preceding one impossible to obtain unless you impose some simplifying restrictions on the problem, forcing you to rely on other solution techniques like numerical integration. We'll talk about this sort of problem in greater depth in Chapter 11.

2D Particle Kinematics

When we are considering motion in one dimension—that is, when the motion is restricted to a straight line—it is easy enough to directly apply the formulas derived earlier to determine instantaneous velocity, acceleration, and displacement. However, in two dimensions, with motion possible in any direction on a given plane, you must consider the kinematic properties of velocity, acceleration, and displacement as vectors.

Using rectangular coordinates in the standard Cartesian coordinate system, you must account for the *x* and *y* components of displacement, velocity, and acceleration. Essentially, you can treat the *x* and *y* components separately and then superimpose these components to define the corresponding vector quantities.

To help keep track of these *x* and *y* components, let **i** and **j** be unit vectors in the x- and y-directions, respectively. Now you can write the kinematic property vectors in terms of their components as follows:

$$\mathbf{v} = v_x\, \mathbf{i} + v_y\, \mathbf{j}$$
$$\mathbf{a} = a_x\, \mathbf{i} + a_y\, \mathbf{j}$$

If *x* is the displacement in the x-direction and *y* is the displacement in the y-direction, then the displacement vector is:

$$\mathbf{s} = x\, \mathbf{i} + y\, \mathbf{j}$$

It follows that:

$$\mathbf{v} = d\mathbf{s}/dt = dx/dt\ \mathbf{i} + dy/dt\ \mathbf{j}$$
$$\mathbf{a} = d\mathbf{v}/dt = d^2\mathbf{s}/dt^2 = d^2x/dt^2\ \mathbf{i} + d^2y/dt^2\ \mathbf{j}$$

Consider a simple example where you're writing a shooting game and you need to figure out the vertical drop in a fired bullet from its aim point to the point at which it actually hits the target. In this example, assume that there is no wind and no drag on the bullet as it flies through the air (we'll deal with wind and drag on projectiles in Chapter 6). These assumptions reduce the problem to one of constant acceleration, which in this case is that due to gravity. It is this gravitational acceleration that is responsible for the drop in the bullet as it travels from the rifle to the target. Figure 2-2 illustrates the problem.

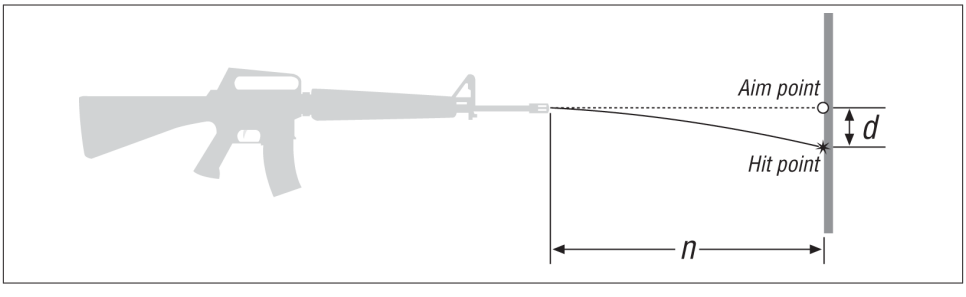


Figure 2-2. A 2D kinematics example problem

While we're talking about guns and shooting here, we should point out that these techniques can be applied just as easily to simulating the flight of angry birds in *Angry Birds* being shot from oversized slingshots, as in the very popular iPhone app. Heck, you can use these techniques to simulate flying monkeys, ballistic shoes, or coconuts being hurled at Navy combatants! This particle kinematic stuff is perfect for diversionary smartphone apps.

Let the origin of the 2D coordinate system be at the end of the rifle with the x-axis pointing toward the target and the y-axis pointing up. Positive displacements along the x-axis are toward the target, and positive displacements along the y-axis are upward. This implies that the gravitational acceleration will act in the negative y-direction.

Treating the *x* and *y* components separately allows you to break up the problem into small, easy-to-manage pieces. Looking at the *x* component first, you know that the bullet will leave the rifle with an initial muzzle velocity v_m in the x-direction, and since we are neglecting drag, this speed will be constant. Thus:

$$a_x = 0$$
$$v_x = v_m$$
$$x = v_x\, t = v_m\, t$$

Now looking at the *y* component, you know that the initial speed in the y-direction, as the bullet leaves the rifle, is 0, but the y-acceleration is $-g$ (due to gravity). Thus:

$$a_y = -g = dv_y/dt$$
$$v_y = a_y\, t = -g\, t$$
$$y = (1/2)\, a_y\, t^2 = -(1/2)\, g\, t^2$$

The displacement, velocity, and acceleration vectors can now be written as:

$$\mathbf{s} = (v_m\, t)\, \mathbf{i} - (1/2\, g\, t^2)\, \mathbf{j}$$
$$\mathbf{v} = (v_m)\, \mathbf{i} - (g\, t)\, \mathbf{j}$$
$$\mathbf{a} = -(g)\, \mathbf{j}$$

These equations give the instantaneous displacement, velocity, and acceleration for any given instant between the time the bullet leaves the rifle and the time it hits the target. The magnitudes of these vectors give the total displacement, velocity, and acceleration at a given time. For example:

$$s = \sqrt{(v_m t)^2 + (1/2 g t^2)^2}$$

$$v = \sqrt{(v_m)^2 + (gt)^2}$$

$$a = \sqrt{g^2} = g$$

To calculate the bullet's vertical drop at the instant the bullet hits the target, you must first calculate the time required to reach the target; then, you can use that time to calculate the *y* component of displacement, which is the vertical drop. Here are the formulas to use:

$$t_{hit} = x_{hit}/v_m = n/v_m$$
$$d = y_{hit} = -(1/2)\, g\, (t_{hit})^2$$

where *n* is the distance from the rifle to the target and *d* is the vertical drop of the bullet at the target.

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:

$$t_{hit} = 0.625 \text{ sec}$$
$$d = 1.9 \text{ m}$$

These results tell you that in order to hit the intended target at that range, you'll need to aim for a point about 2 m above it.

3D Particle Kinematics

Extending the kinematic property vectors to three dimensions is not very difficult. It simply involves the addition of one more component to the vector representations shown in the previous section on 2D kinematics. Introducing **k** as the unit vector in the z-direction, you can now write:

$$\mathbf{s} = x\,\mathbf{i} + y\,\mathbf{j} + z\,\mathbf{k}$$
$$\mathbf{v} = d\mathbf{s}/dt = dx/dt\,\mathbf{i} + dy/dt\,\mathbf{j} + dz/dt\,\mathbf{k}$$
$$\mathbf{a} = d^2\mathbf{s}/dt^2 = d^2x/dt^2\,\mathbf{i} + d^2y/dt^2\,\mathbf{j} + d^2z/dt^2\,\mathbf{k}$$

Instead of treating two components separately and then superimposing them, you now treat three components separately and superimpose these. This is best illustrated by an example.

Suppose that instead of a hunting game, you're now writing a game that involves the firing of a cannon from, say, a battleship to a target some distance away—for example, another ship or an inland target like a building. To add complexity to this activity for your user, you'll want to give her control of several factors that affect the shell's trajectory—namely, the firing angle of the cannon, both horizontal and vertical angles, and the muzzle velocity of the shell, which is controlled by the amount of powder packed behind the shell when it's loaded into the cannon. The situation is set up in Figure 2-3.

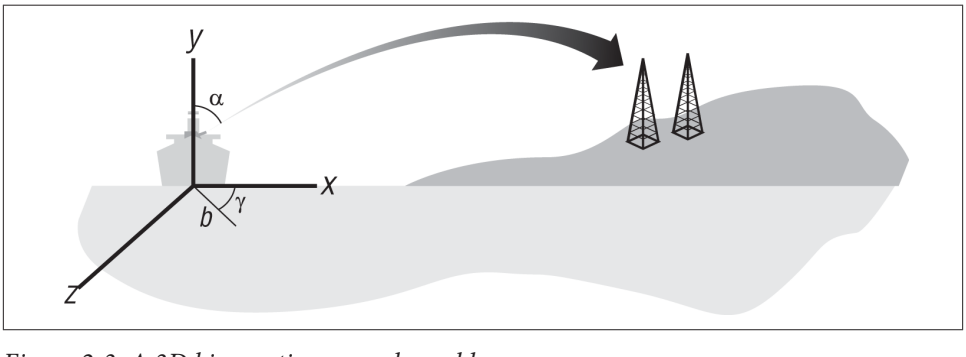


Figure 2-3. A 3D kinematics example problem

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:

$$\mathbf{v}_m = v_{mx}\,\mathbf{i} + v_{my}\,\mathbf{j} + v_{mz}\,\mathbf{k}$$

and you are given only the direction of $\mathbf{v}_m$ as determined by the direction in which the user points the cannon, and its magnitude as determined by the amount of powder the user packs into the cannon. To calculate the components of the muzzle velocity, you need to develop some equations for these components in terms of the direction angles of the cannon and the magnitude of the muzzle velocity.

You can use the direction cosines of a vector to determine the velocity components as follows:

$$\cos\theta_x = v_{mx}/v_m$$
$$\cos\theta_y = v_{my}/v_m$$
$$\text{cox}\,\theta_z = v_{mz}/v_m$$

Refer to Appendix A for a description and illustration of vector direction cosines.

Since the initial muzzle velocity vector direction is the same as the direction in which the cannon is aimed, you can treat the cannon as a vector with a magnitude of *L*, its length, and pointing in a direction defined by the angles given in this problem. Using the cannon length, *L*, and its components instead of muzzle velocity in the equations for direction cosines gives:

$$\cos\theta_x = L_x/L$$
$$\cos\theta_y = L_y/L$$
$$\cos\theta_z = L_z/L$$

In this example, you are given the angles α and γ (see Figure 2-4) that define the cannon orientation.

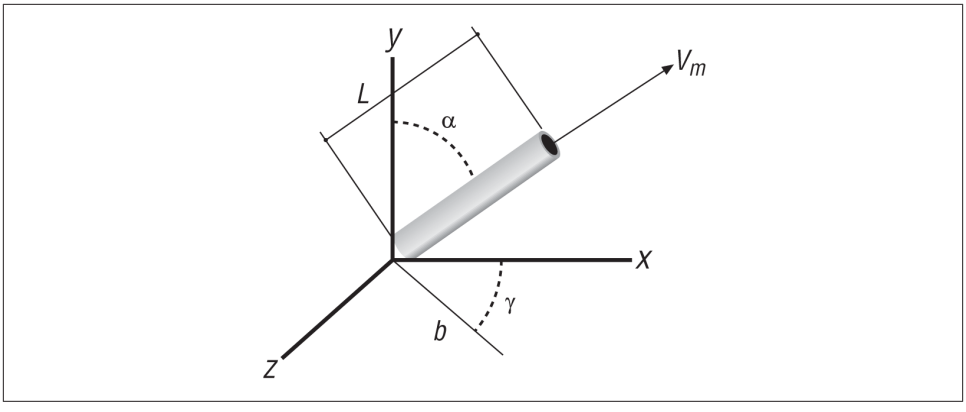


Figure 2-4. Cannon orientation

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

$$b = L \cos(90° - \alpha)$$

and the components of the cannon length, *L*, on each coordinate axis are:

$$Lx = b \cos \gamma$$
$$Ly = L \cos \alpha$$
$$Lz = b \sin \gamma$$

Now that you have the information required to compute direction cosines, you can write equations for the initial muzzle velocity components as follows:

$$v_{mx} = v_m \cos \theta_x$$
$$v_{my} = v_m \cos \theta_y$$
$$v_{mz} = v_m \cos \theta_z$$

Finally, you can write the *x* components of displacement, velocity, and acceleration as follows:

$$a_x = 0$$
$$v_x = v_{mx} = v_m \cos \theta_x$$
$$x = v_x t = (v_m \cos \theta_x) t$$

Y Components

The *y* components are just like the previous rifle example, again with the exception here of the initial velocity in the y-direction:

$$v_{my} = v_m \cos \theta_y$$

Thus:

$$a_y = -g$$
$$v_y = v_{my} + a\ t = (v_m \cos \theta_y) - g\ t$$

Before writing the equation for the *y* component of displacement, you need to consider the elevation of the base of the cannon, plus the height of the end of the cannon barrel, in order to calculate the initial *y* component of displacement when the shell leaves the cannon. Let y_b be the elevation of the base of the cannon, and let *L* be the length of the cannon barrel; then the initial *y* component of displacement, y_o, is:

$$y_o = y_b + L \cos \alpha$$

Now you can write the equation for *y* as:

$$y = y_o + v_{my}\ t + (1/2)\ a\ t^2$$
$$y = (y_b + L \cos \alpha) + (v_m \cos \theta_y)\ t - (1/2)\ g\ t^2$$

Z Components

The *z* components are largely analogous to the *x* components and can be written as follows:

$$a_z = 0$$
$$v_z = v_{mz} = v_m \cos \theta_z$$
$$z = v_z\ t = (v_m \cos \theta_z)\ t$$

The Vectors

With the components all worked out, you can now combine them to form the vector for each kinematic property. Doing so for this example gives the displacement, velocity, and acceleration vectors shown here:

$$\mathbf{s} = [(v_m \cos \theta_x)\ t]\ \mathbf{i} + [(y_b + L \cos \alpha) + (v_m \cos \theta_y)\ t - (1/2)\ g\ t^2]$$
$$\mathbf{j} + [(v_m \cos \theta_z)\ t\ ]\ \mathbf{k}$$
$$\mathbf{v} = [v_m \cos \theta_x\ ]\ \mathbf{i} + [(v_m \cos \theta_y) - g\ t\ ]\ \mathbf{j} + [v_m \cos \theta_z\ ]\ \mathbf{k}$$
$$\mathbf{a} = -g\ \mathbf{j}$$

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.

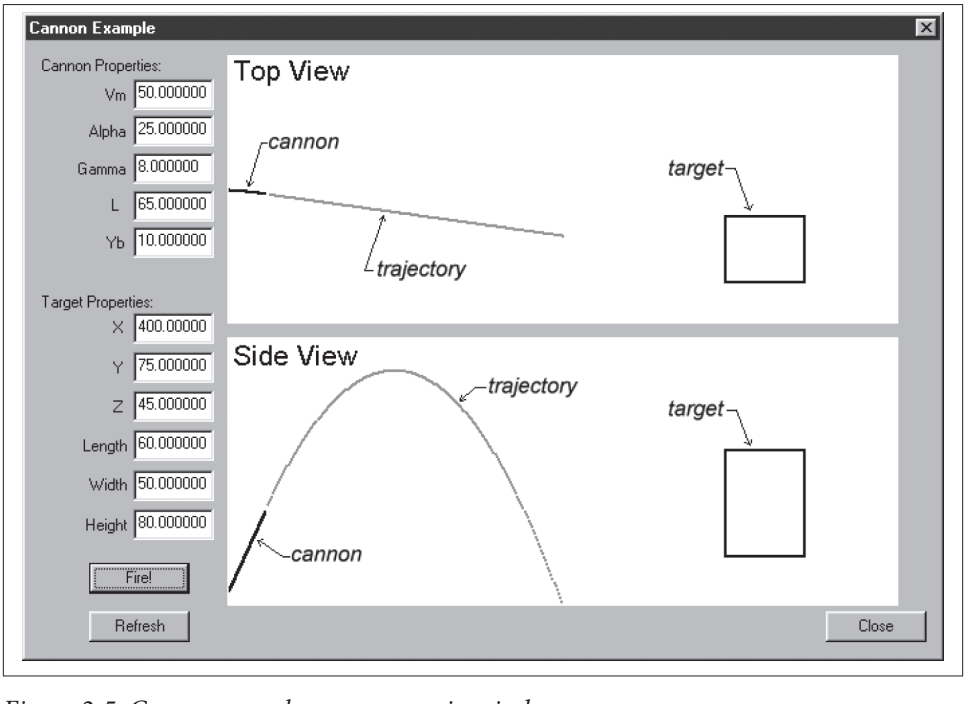


Figure 2-5. Cannon sample program main window

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.

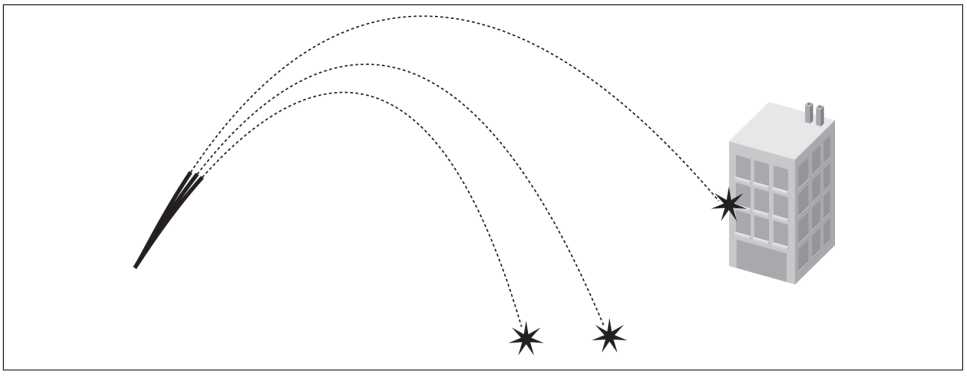


Figure 2-6. Trial shots (profile view)

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 happing 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 measure 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.

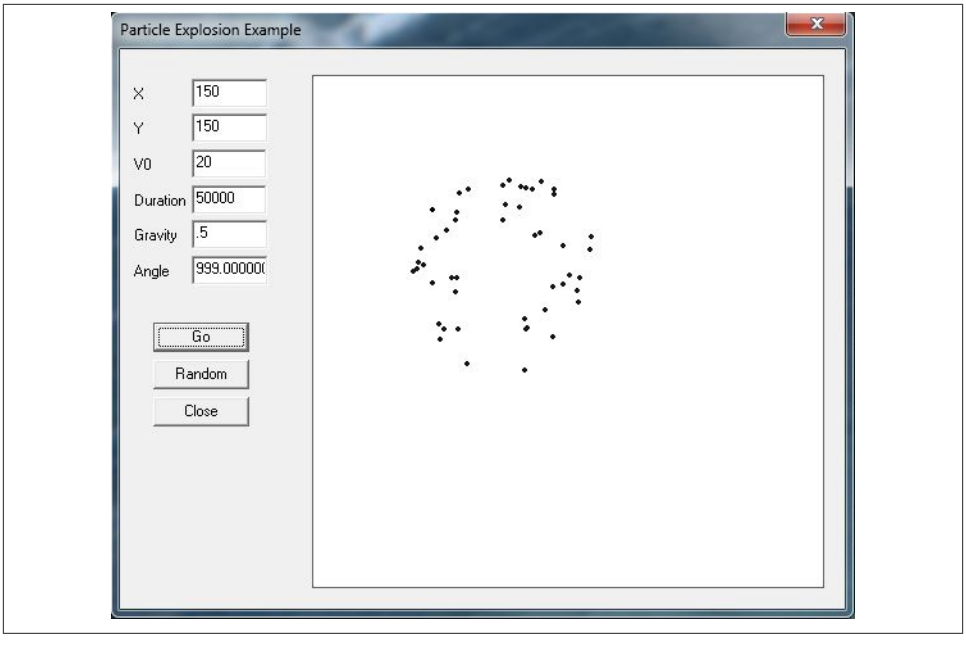


Figure 2-7. Particle explosion example program

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.

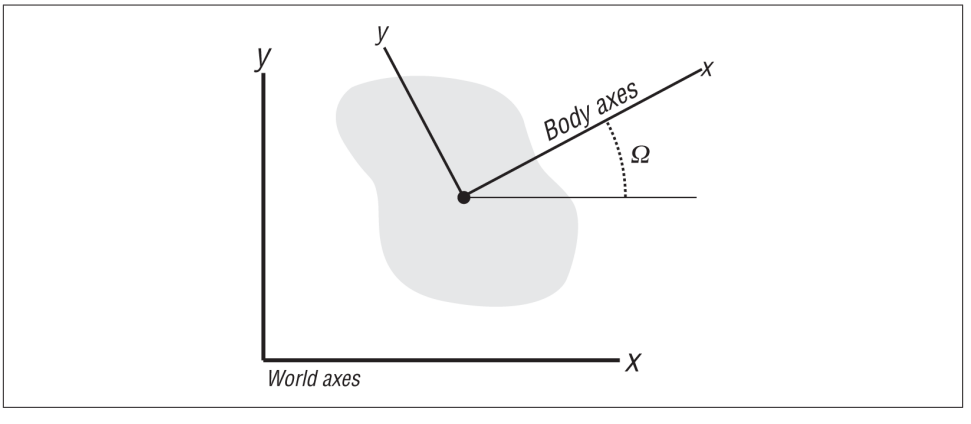


Figure 2-8. Local coordinate axes

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/s^2), respectively.

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

$$\omega = d\Omega/dt$$
$$\alpha = d\omega/dt = d^2\Omega/dt^2$$
$$\omega = \int \alpha \, dt$$
$$\Omega = \int \omega \, dt$$
$$\omega \, d\omega = \alpha \, d\Omega$$

In fact, you can substitute the angular properties Ω, ω, and α for the linear properties *s*, *v*, and *a* in the equations derived earlier for particle kinematics to obtain similar kinematic equations for rotation. For constant angular acceleration, you'll end up with the following equations:

$$\omega_2 = \omega_1 + \alpha \, t$$
$$\omega_2{}^2 = \omega_1{}^2 + 2 \, \alpha \, (\Omega_2 - \Omega_1)$$
$$\Omega_2 = \Omega_1 + \omega_1 t + (1/2) \, \alpha \, t^2$$

When a rigid body rotates about a given axis, every point on the rigid body sweeps out a circular path around the axis of rotation. You can think of the body's rotation as causing additional linear motion of each particle making up the body—that is, this linear motion is in addition to the linear motion of the body's center of mass. To get the total linear motion of any particle or point on the rigid body, you must be able to relate the angular motion of the body to the linear motion of the particle or point as it sweeps its circular path about the axis of rotation.

Before we show you how to do this, we'll explain why you would even want to perform such a calculation. Basically, in dynamics, knowing that two objects have collided is not always enough, and you'll often want to know how hard, so to speak, these two objects have collided. When you're dealing with interacting rigid bodies that may at some point make contact with one another or with other fixed objects, you need to determine not only the location of the points of contact, but also the relative velocity or acceleration between the contact points. This information will allow you to calculate the interaction forces between the colliding bodies.

The arc length of the path swept by a particle on the rigid body is a function of the distance from the axis of rotation to the particle and the angular displacement, Ω. We'll use *c* to denote arc length and *r* to denote the distance from the axis of rotation to the particle, as shown in Figure 2-9.

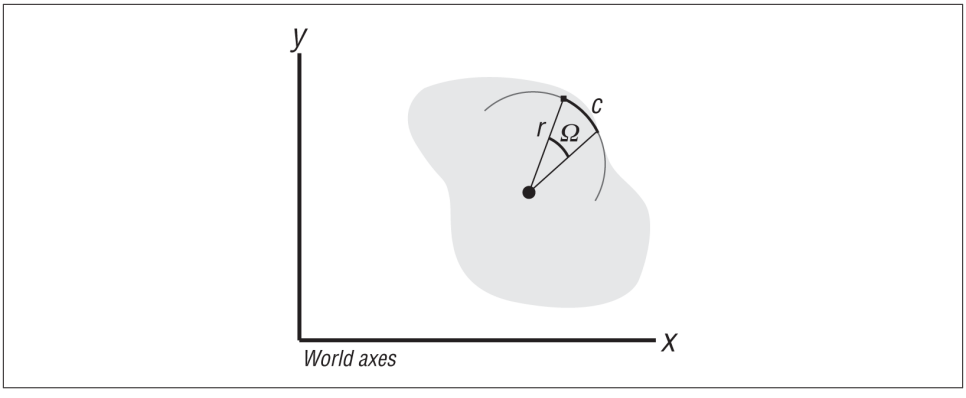


Figure 2-9. Circular path of particles making up a rigid body

The formula relating arc length to angular displacement is:

$$c = r\,\Omega$$

where Ω must be in radians, not degrees. If you differentiate this formula with respect to time:

$$dc/dt = r\,d\Omega/dt$$

you get an equation relating the linear velocity of the particle as it moves along its circular path to the angular velocity of the rigid body. This equation is written as follows:

$$v = r\,\omega$$

This velocity as a vector is tangent to the circular path swept by the particle. Imagine this particle as a ball at the end of a rod where the other end of the rod is fixed to a rotating axis. If the ball is released from the end of the rod as it rotates, the ball will fly off in a direction tangent to the circular path it was taking when attached to the rod. This is in the same direction as the tangential linear velocity given by the preceding equation. Figure 2-10 illustrates the tangential velocity.

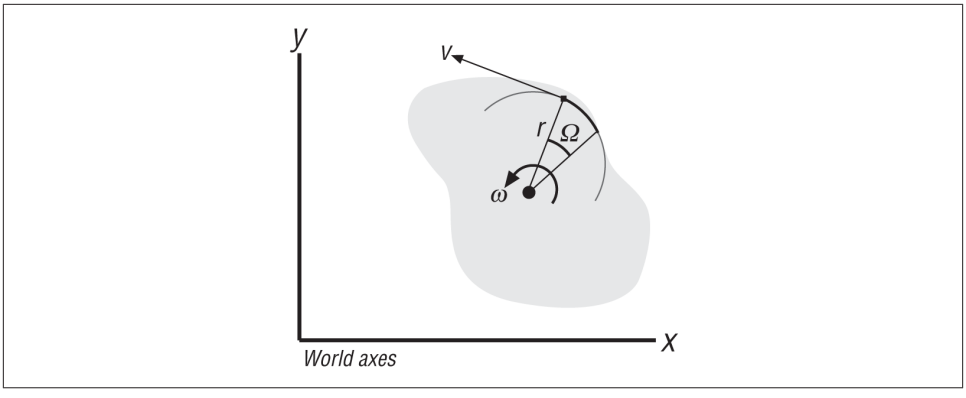


Figure 2-10. Linear velocity due to angular velocity

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

$$dv/dt = r\,d\omega/dt$$

yields this formula for the tangential linear acceleration as a function of angular acceleration:

$$a_t = r\,\alpha$$

Note that there is another component of acceleration for the particle that results from the rotation of the rigid body. This component—the *centripetal* acceleration—is normal, or perpendicular, to the circular path of the particle and is always directed toward the axis of rotation. Remember that velocity is a vector and since acceleration is the rate of change in the velocity vector, there are two ways that acceleration can be produced. One way is by a change in the magnitude of the velocity vector—that is, a change in speed—and the other way is a change in the direction of the velocity vector. The change in speed gives rise to the tangential acceleration component, while the direction change gives rise to the centripetal acceleration component. The resultant acceleration vector is the vector sum of the tangential and centripetal accelerations (see Figure 2-11). Centripetal acceleration is what you feel when you take your car around a tight curve even though your speed is constant.

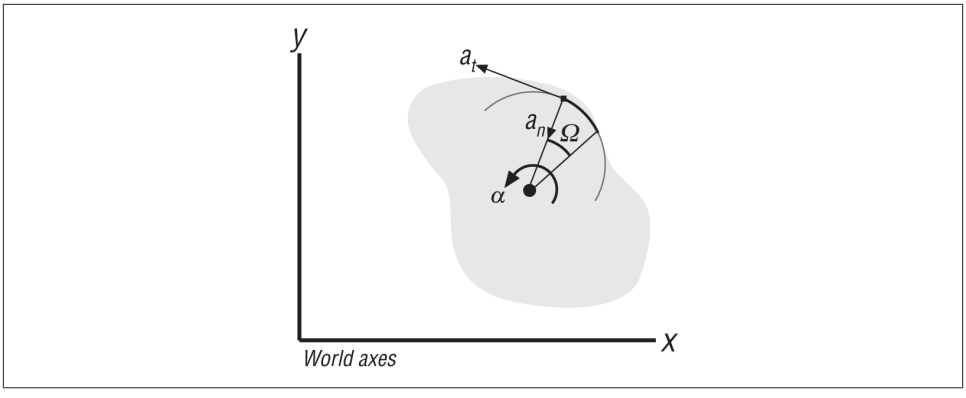


Figure 2-11. Tangential and centripetal acceleration

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

$$a_n = v^2/r$$

where v is the tangential velocity. Substituting the equation for tangential velocity into this equation for centripetal acceleration gives the following alternative form:

$$a_n = r\,\omega^2$$

In two dimensions you can easily get away with using these scalar equations; however, in three dimensions you'll have to use the vector forms of these equations. Angular velocity as a vector is parallel with the axis of rotation. In Figure 2-10 the angular velocity would be pointing out of the page directly at you. Its sense, or direction of rotation, is determined by the *right hand rule*. If you curl the fingers of your right hand in an arc around the axis of rotation with your fingers pointing toward the direction in which the body is rotating, then your thumb will stick up in the direction of the angular velocity vector.

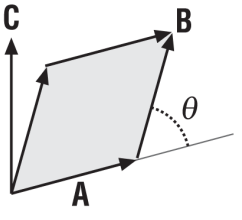
If you take the vector cross product (refer to the sidebar "Vector Cross Product" on page 67 for background and to Appendix A for a review of vector math) of the angular velocity vector and the vector from the axis of rotation to the particle under consideration, you'll end up with the linear, tangential velocity vector. This is written as follows:

$$\mathbf{v} = \boldsymbol{\omega} \times \mathbf{r}$$

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 **r**, and not the other way around, which would give the wrong direction for **v**.

Vector Cross Product

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



$$\mathbf{C} = \mathbf{A} \times \mathbf{B}$$
$$\mathbf{C} = \mathbf{AB} \sin \theta$$

The direction of **C** is determined by the right hand rule. As noted previously, the right hand rule is a simple trick to help you keep track of vector directions. Assume that **A** and **B** lie in a plane and let an axis of rotation extend perpendicular to this plane through a point located at the tail of **A**. Pretend to curl the fingers of your right hand around the axis of rotation from vector **A** toward **B**. 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 vector **C**.

In the preceding figure, a parallelogram is formed by A and B (the shaded region). The area of this parallelogram is the magnitude of C, which is AB sin θ.

There are two equations that you'll need in order to determine the vectors for tangential and centripetal acceleration. They are:

$$\mathbf{a}_n = \boldsymbol{\omega} \times (\boldsymbol{\omega} \times \mathbf{r})$$
$$\mathbf{a}_t = \boldsymbol{\alpha} \times \mathbf{r}$$

Another way to look at the quantities **v**, $\mathbf{a}_n$, and $\mathbf{a}_t$ is that they are the velocity and acceleration of the particle under consideration, on the rigid body, relative to the point about which the rigid body is rotating—for example, the body's center-of-mass location. This is very convenient because, as we said earlier, you'll want to track the motion of the rigid body as a particle when viewing the big picture without having to worry about

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 $\mathbf{v}_{cg}$, where $\mathbf{v}_{cg}$ 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 $\boldsymbol{\omega}$ about an axis that passes through the body's center of mass. The vector $\mathbf{r}$ is the vector from the rigid body's center of mass to the particular point of interest, *P*, located on the rigid body.

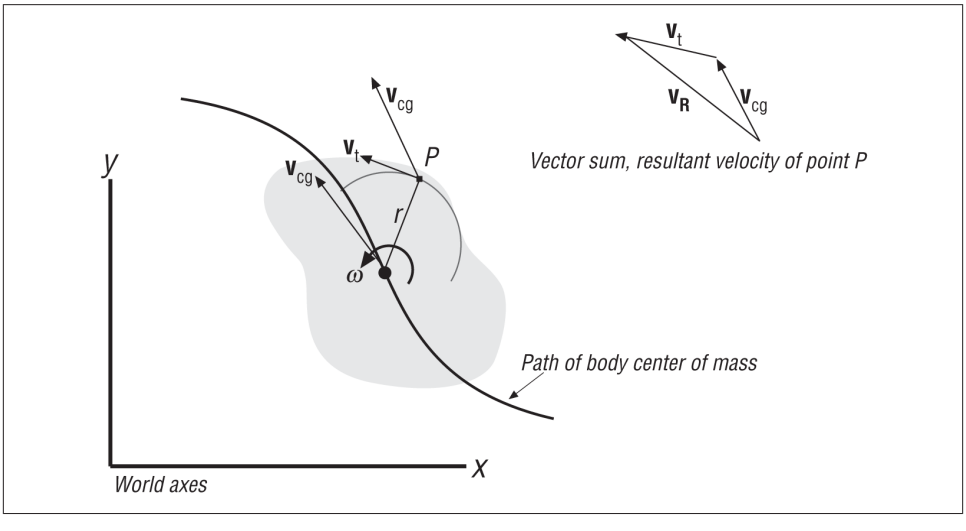


Figure 2-12. Relative velocity

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 $\boldsymbol{\omega}$. Here's what the vector equation looks like:

$$\mathbf{v}_R = \mathbf{v}_{cg} + \mathbf{v}_t$$

or:

$$\mathbf{v}_R = \mathbf{v}_{cg} + (\boldsymbol{\omega} \times \mathbf{r})$$

You can do the same thing with acceleration to determine point *P*'s resultant acceleration. Here you'll take the vector sum of the acceleration of the rigid body's center of

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:

$$\mathbf{a}_R = \mathbf{a}_{cg} + \mathbf{a}_n + \mathbf{a}_t$$

Figure 2-13 illustrates what's happening here.

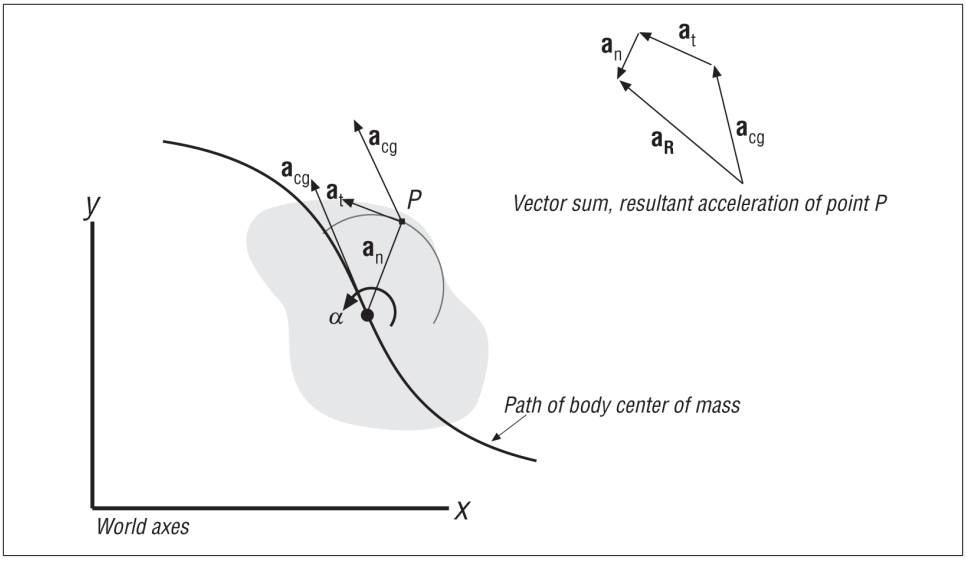


Figure 2-13. Relative acceleration

You can rewrite the equation for the resultant acceleration in the following form:

$$\mathbf{a}_R = \mathbf{a}_{cg} + (\boldsymbol{\omega} \times (\boldsymbol{\omega} \times \mathbf{r})) + (\boldsymbol{\alpha} \times \mathbf{r})$$

As you can see, using these principles of relative velocity and acceleration allows you to calculate the resultant kinematic properties of any point on a rigid body at any given time by determining what the body's center of mass is doing along with how the body is rotating.

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:

$$F_a = (G\ m_1\ m_2) / r^2$$

where *G* is the gravitational constant, Newton's so-called *universal constant*. *G* was first measured experimentally by Sir Henry Cavendish in 1798 and equals 6.673×10^{-11} $(N–m^2)/kg^2$ in metric units or 3.436×10^{-8} $ft^4/(lb\text{-}s^4)$ in English units.

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:

$$m\ a = (G\ M_e\ m) / (R_e + h)^2$$

where *m* is the mass of the body, *a* is the acceleration of the body due to the gravitational attraction between it and the earth, M_e is the earth's mass, R_e is the radius of the earth, and *h* is the altitude of the body. If you solve this equation for *a*, you'll have a formula for the acceleration due to gravity as a function of altitude:

$$a = g' = (G\ M_e) / (R_e + h)^2$$

The radius of the earth is approximately 6.38×10^6 m, and its mass is about 5.98×10^{24} kgs. Substituting these values in the preceding equation and assuming 0 altitude (sea level) yields the constant *g* that we've been using so far—that is, *g* at sea level equals 9.8 m/s^2.

Friction

Frictional forces (friction) always resist motion and are due to the interaction between contacting surfaces. Thus, friction is a contact force. Friction is always parallel to the contacting surfaces at the point of contact—that is, friction is tangential to the contacting surfaces. The magnitude of the frictional force is a function of the normal force between the contacting surfaces and the surface roughness.

This is easiest to visualize by looking at a simple block on a horizontal surface, as shown in Figure 3-1.

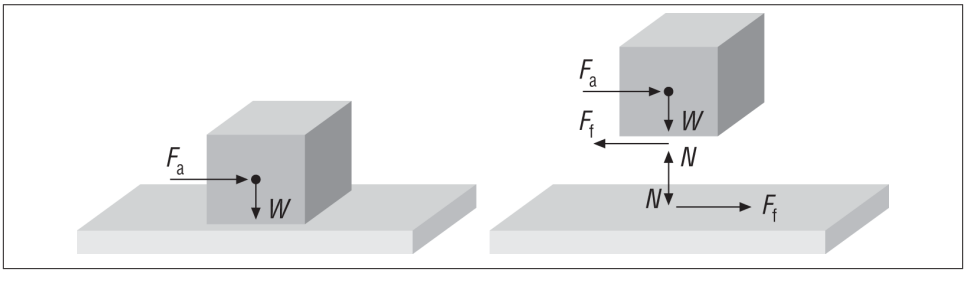


Figure 3-1. Friction: block in contact with horizontal surface

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:

$$F_{fmax} = \mu_s N$$

where μ_s is the experimentally determined coefficient of static[1] friction and N is the normal (perpendicular) force between the block and the surface, which equals the weight of the block in this case. As the applied force increases but is still less than F_{fmax}, the block will remain static and F_f will be equal in magnitude to the applied force. The block is in static equilibrium. When the applied force becomes greater than F_{fmax}, the frictional force can no longer impede the block's motion and the block will accelerate under the influence of the applied force. Immediately after the block starts its motion, the frictional force will decrease from F_{fmax} to F_{fk}, where F_{fk} is:

$$F_{fk} = \mu_k N$$

Here k means kinetic since the block is in motion, and μ_k, the coefficient of kinetic friction,[2] is less than μ_s. Like the static coefficient of friction, the kinetic coefficient of friction is determined experimentally. Table 3-1 shows typical coefficients of friction for several surfaces in contact.

Table 3-1. Coefficients of friction of common surfaces

Surface condition	M_s	M_u	% difference
Dry glass on glass	0.94	0.4	54%
Dry iron on iron	1.1	0.15	86%
Dry rubber on pavement	0.55	0.4	27%
Dry steel on steel	0.78	0.42	46%
Dry Teflon on Teflon	0.04	0.04	—
Dry wood on wood	0.38	0.2	47%
Ice on ice	0.1	0.03	70%
Oiled steel on steel	0.10	0.08	20%

The data in Table 3-1 is provided here to show you the magnitude of some typical friction coefficients and the relative difference between the static and kinetic coefficients for certain surface conditions. Other data is available for these and other surface conditions

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:

$$F_v = -C_f\, v$$

where C_f is the drag coefficient, v is the body's speed, and the minus sign means that the force opposes motion. This formula is valid for slow-moving objects in a viscous fluid. "Slow moving" implies that the flow around the body is *laminar*, which means that the flow streamlines are undisturbed and parallel.

For fast-moving objects, you'll use the formula for F_v written as a function of speed squared as follows:

$$F_v = -C_f\, v^2$$

"Fast moving" implies that the flow around the object is *turbulent*, which means that the flow streamlines are no longer parallel and there is a sort of mixing effect in the flow around the object. Note that the values of C_f are generally not the same for these two equations. In addition to the factors mentioned earlier, C_f depends significantly on whether the flow is laminar or turbulent.

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:

$$F = PA$$

This formula tells you that for constant pressure, the greater the area acted upon, the greater the resultant force. If you rearrange this equation solving for pressure, you'll see that pressure is inversely proportional to area—that is, the greater the area for a given applied force, the smaller the resulting pressure and vice versa.

$$P = F/A$$

An important characteristic of pressure is that it always acts normally (perpendicularly) to the surface of the body or object it is acting on. This fact gives you a clue as to the direction of the resultant force vector.

We wanted to mention pressure here because you'll be working with it to calculate forces when you get to the chapters in this book that cover the mechanics of ships, boats, and hovercraft. There, the pressures that you'll consider are hydrostatic pressure (buoyancy) and aerostatic lift. We'll take a brief look at buoyancy next.

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.

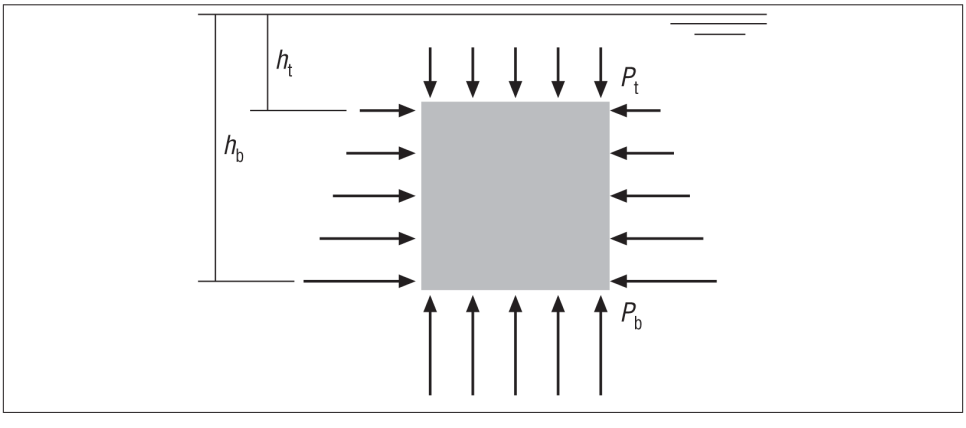


Figure 3-2. Immersed cube

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:

$$F_t = P_t\, A_t$$

$$F_t = (\rho\ g\ h_t)\ (s^2)$$

Similarly, the force acting upward on the bottom of the cube is equal to the pressure at the bottom times the surface area of the bottom.

$$F_b = P_b\ A_b$$
$$F_b = (\rho\ g\ h_b)\ (s^2)$$

The net vertical force (buoyancy) equals the difference between the top and bottom forces:

$$F_B = F_b - F_t$$
$$F_B = (\rho\ g\ h_b)\ (s^2) - (\rho\ g\ h_t)\ (s^2)$$
$$F_B = (\rho\ g)\ (s^2)\ (h_b - h_t)$$

This formula gives the magnitude of the buoyancy force. Its direction is straight up, counteracting the weight of the object.

There is an important observation we need to make here. Notice that $(h_b - h_t)$ is simply the height of the cube, which is s in this case. Substituting s in place of $(h_b - h_t)$ reveals that the buoyancy force is a function of the volume of the cube.

$$F_B = (\rho\ g)\ (s^3)$$

This is great since it means that all you need to do in order to calculate buoyancy is to first calculate the volume of the object and then multiply that volume by the specific weight[3] $(\rho\ g)$ of the fluid. In truth, that's a little easier said than done for all but the simplest geometries. If you're dealing with spheres, cubes, cylinders, and the like, then calculating volume is easy. However, if you're dealing with any arbitrary geometry, then the volume calculation becomes more difficult. There are two ways to handle this difficulty. The first way is to simply divide the object into a number of smaller objects of simpler geometry, calculate their volumes, and then add them all up. The second way is to use numerical integration techniques to calculate volume by integrating over the surface of the object.

You should also note that buoyancy is a function of fluid density, and you don't have to be in a fluid as dense as water to experience the force of buoyancy. In fact, there are buoyant forces acting on you right now, although they are very small, due to the fact that you are immersed in air. Water is many times more dense than air, which is why you notice the force of buoyancy when in water and not in air. Keep in mind, though,

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:

$$F_s = k_s (L - r)$$

Here, F_s is the spring force, k_s is the spring constant, L is the stretched or compressed length of the spring, and r is the rest length of the spring. In the metric system of units, F_s would be measured in newtons (1 N = 1 kg-m/s^2), L and r in meters, and k_s in kg/s^2. If the spring is connected between two objects, it exerts a force of F_s on one object and $-F_s$ on the other; these are *equal and opposite* forces.

Dampers are usually used in conjunction with springs in numerical simulations. They act like viscous drag in that they act against velocity. In this case, if the damper is connected between two objects that are moving toward or away from each other, the damper acts to slow the relative velocity between the two objects. The force developed by a damper is proportional to the relative velocity of the connected objects and a damping constant, k_d, that relates relative velocity to damping force.

$$F_d = k_d (v_1 - v_2)$$

This equation shows the damping force, F_d, as a function of the damping constant and the relative velocity of the connected points on the two connected bodies. In metric units, where the damping force is measured in newtons and velocity in m/s, k_d has units of kg/s.

Typically, springs and dampers are combined into a single spring-damper element, where a single formula represents the combined force. Using vector notation, we can write the formula for a spring-damper element connecting two bodies as follows:

$$\mathbf{F}_1 = -\{k_s (L - r) + k_d ((\mathbf{v}_1 - \mathbf{v}_2) \bullet \mathbf{L})/L\} \mathbf{L}/L$$

Here, $\mathbf{F}_1$ is the force exerted on body 1, while the force, $\mathbf{F}_2$, exerted on body 2 is:

$$\mathbf{F}_2 = -\mathbf{F}_1$$

L is the length of the spring-damper (*L*, not in bold print, is the magnitude of the vector **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 **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*.

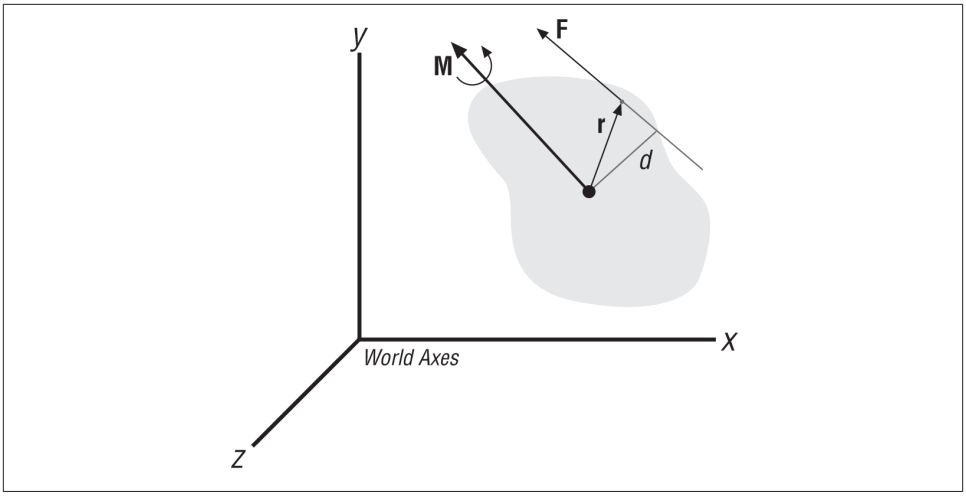


Figure 3-3. Force and torque

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:

$$\mathbf{M} = \mathbf{r} \times \mathbf{F}$$

The torque, **M**, is the vector cross product of the position vector, **r**, and the force vector, **F**.

In rectangular coordinates you can write the distance, force, and torque vectors as follows:

$$\mathbf{r} = x\,\mathbf{i} + y\,\mathbf{j} + z\,\mathbf{k}$$
$$\mathbf{F} = F_x\,\mathbf{i} + F_y\,\mathbf{j} + F_z\,\mathbf{k}$$
$$\mathbf{M} = M_x\,\mathbf{i} + M_y\,\mathbf{j} + M_z\,\mathbf{k}$$

The scalar components of **r** (*x*, *y*, and *z*) are the coordinate distances from the axis of rotation to the point of application of the force, **F**. The scalar components of the torque vector, **M**, are defined by the following:

$$M_x = y\,F_z - z\,F_y$$
$$M_y = z\,F_x - x\,F_z$$

$$M_z = x\,F_y - y\,F_x$$

Consider the rigid body shown in Figure 3-4 acted upon by the force **F** at a point away from the body's center of mass.

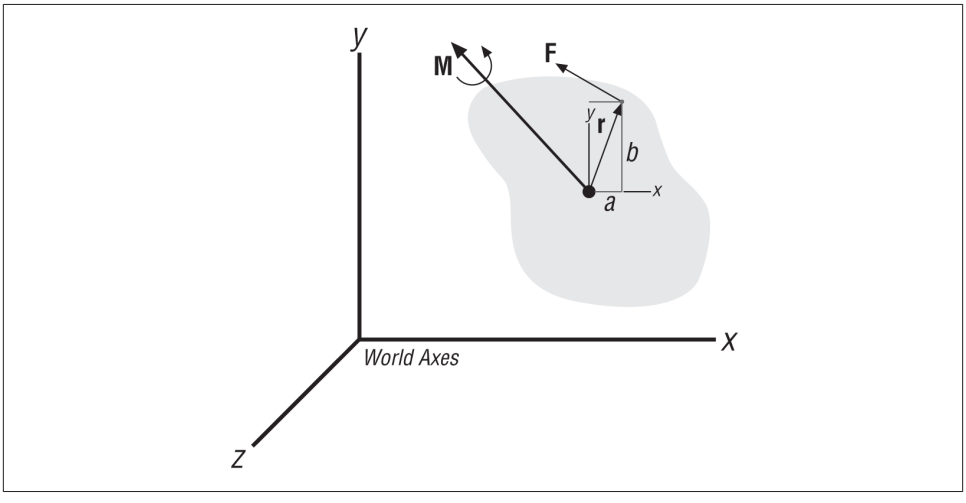


Figure 3-4. Torque example

In this example **F**, *a*, and *b* are given and are as follows:

$$\mathbf{F} = (-90 \text{ lbs})\,\mathbf{i} + (156 \text{ lbs})\,\mathbf{j} + (0)\,\mathbf{k}$$
$$a = 0.66 \text{ ft}$$
$$b = 0.525 \text{ ft}$$

Calculate the torque about the body's center of mass due to the force **F**.

The first step is to put together the distance vector from the point of application of **F** to the body's center of mass. Since the local coordinates *a* and *b* are given, **r** is simply:

$$\mathbf{r} = (0.66 \text{ ft})\,\mathbf{i} + (0.525 \text{ ft})\,\mathbf{j} + (0)\,\mathbf{k}$$

Now using the formula $\mathbf{M} = \mathbf{r} \times \mathbf{F}$ (or the formulas for the components of the torque vector shown earlier), you can write:

$$\mathbf{M} = [(0.66 \text{ ft})\,\mathbf{i} + (0.525 \text{ ft})\,\mathbf{j} + (0)\,\mathbf{k}] \times [(-90 \text{ lbs})\,\mathbf{i} + (156 \text{ lbs})\,\mathbf{j} + (0)\,\mathbf{k}]$$
$$\mathbf{M} = [(0.66 \text{ ft})\,(156 \text{ lbs}) - (0.525 \text{ ft})\,(-90 \text{ lbs})]\,\mathbf{k}$$
$$\mathbf{M} = (150.2 \text{ ft-lbs})\,\mathbf{k}$$

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 **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:

$$\mathbf{F} = m\mathbf{a}$$

When rigid bodies are involved, you must also consider that the forces acting on the body will tend to cause rotation of the body in addition to translation. The basic relationship here is:

$$\mathbf{M}_{cg} = \mathbf{I}\,\boldsymbol{\alpha}$$

where $\mathbf{M}_{cg}$ is the vector sum of all moments (torques) acting on the body, $\mathbf{I}$ is the body moment of inertia tensor, and $\boldsymbol{\alpha}$ is the angular acceleration.

Collectively, these two equations are referred to as the *equations of motion*.

There are two types of problems that you will encounter in kinetics. One type is where you know the force(s) acting on the body, or you can estimate them, and you must solve for the resulting acceleration of the body (and subsequently its velocity and displacement). Another type is where you know the body's acceleration, or can readily determine it using kinematics, and you must solve for the force(s) acting on the body.

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:

$$\Sigma F_x = m\, a_x$$
$$\Sigma F_y = m\, a_y$$

where ΣF_x means the sum of all forces in the x-direction, ΣF_y means the sum of all forces in the y-direction, a_x is the acceleration in the x-direction, and a_y is the acceleration in the y-direction.

The resultant force and acceleration vectors are:

$$\mathbf{a} = a_x\,\mathbf{i} + a_y\,\mathbf{j}$$

$$a = \sqrt{a_x^2 + a_y^2}$$

$$\Sigma\mathbf{F} = \Sigma F_x\,\mathbf{i} + \Sigma F_y\,\mathbf{j}$$
$$\Sigma F = \sqrt{(\Sigma F_x)^2 + (\Sigma F_y)^2}$$

Let's look at an example that appears simple but demonstrates the complexity of finding closed-form solutions. A ship floating in water, initially at rest, starts up its propeller generating a thrust, T, which starts the ship moving forward. Assume that the ship's forward speed is slow and the resistance to its motion can be approximated by:

$$R = -C\,v$$

where R is the total resistance, C is a drag coefficient, v is the ship speed, and the minus sign indicates that this resistive force opposes the forward motion of the ship. Find formulas for the ship's speed, acceleration, and distance traveled as functions of time, assuming that the propeller thrust and resistance force vectors act on a line of action passing through the ship's center of gravity. This assumption lets you treat the ship as a particle instead of a rigid body.

The first step in solving this problem is to identify all of the forces acting on the ship. Figure 4-1 shows a *free-body diagram* of the ship with all of the forces acting on it—namely, the propeller thrust, T; resistance, R; the ship's weight, W; and buoyancy, B.

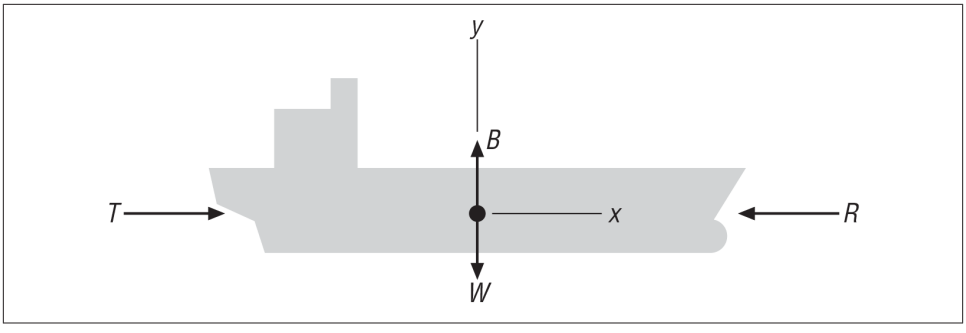


Figure 4-1. Free-body diagram of ship

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:

$$\Sigma F = m\,a$$
$$T - R = m\,a$$
$$T - (C\,v) = m\,a$$

Where *a* is the acceleration in the x-direction, and *v* is the speed in the x-direction.

The next step is to integrate this equation of motion in order to derive a formula for the speed of the ship as a function of time. To do this, you must make the substitution $a = dv/dt$, rearrange, integrate, and then solve for speed as follows:

$$T - (C\,v) = m\,(dv/dt)$$
$$dt = (m / (T - Cv))\,dv$$
$$\int_{(0 \text{ to } t)} dt = \int_{(v1 \text{ to } v2)} (m / (T - Cv))\,dv$$
$$t - 0 = -(m/C)\,\ln(T - Cv)\,|_{(v1 \text{ to } v2)}$$
$$t = -(m/C)\,\ln(T - Cv_2) + (m/C)\,\ln(T - Cv_1)$$
$$t = (m/C)\,[\ln(T - Cv_1) - \ln(T - Cv_2)]$$
$$(C/m)\,t = \ln\,[(T - Cv_1) / (T - Cv_2)]$$
$$e^{(C/m)\,t} = e^{\ln\,[(T - Cv1) / (T - Cv2)]}$$
$$e^{(C/m)\,t} = (T - Cv_1) / (T - Cv_2)$$
$$(T - Cv_2) = (T - Cv_1)\,e^{-(C/m)t}$$

$$v_2 = (T/C) - e^{-(C/m)\,t}\,(T/C - v_1)$$

where v_1 is the initial ship speed (which is constant) and v_2 is the ship speed at time t. v_2 is what you're after here, since it tells you how fast the ship is traveling at any instant of time.

Now that you have an equation for speed as a function of time, you can derive an equation for displacement (distance traveled, in this case) as a function of time. Here, you'll have to recall the formula $v\,dt = ds$, substitute the previous formula for speed, integrate, rearrange, and solve for distance traveled. These steps are shown here:

$$\begin{gathered}
v\,dt = ds \\
v_2\,dt = ds \\
((T/C) - e^{-(C/m)\,t}\,(T/C - v_1))\,dt = ds \\
\int_{(0\ \text{to}\ t)} (T/C) - e^{-(C/m)\,t}\,(T/C - v_1)\,dt = \int_{(s1\ \text{to}\ s2)} ds \\
(T/C) \int_{(0\ \text{to}\ t)} dt - (T/C - v_1) \int_{(0\ \text{to}\ t)} e^{-(C/m)\,t}\,dt = s_2 - s_1 \\
[(T/C)\,t + ((T/C) - v_1)(m/C)\,e^{-(C/m)\,t}]_{(0\ \text{to}\ t)} = s_2 - s_1 \\
[(T/C)\,t + ((T/C) - v_1)(m/C)\,e^{-(C/m)\,t}] - [0 + ((T/C) - v_1)(m/C)] = \\
s_2 - s_1 \\
(T/C)\,t + (T/C - v_1)\,(m/C)\,e^{-(C/m)\,t} - (T/C - v_1)\,(m/C) = s_2 - s_1 \\
s_2 = s_1 + (T/C)\,t + (T/C - v_1)\,(m/C)\,e^{-(C/m)\,t} - (T/C - v_1)\,(m/C)
\end{gathered}$$

Finally you can write an equation for acceleration by going back to the original equation of motion and solving for acceleration:

$$\begin{gathered}
T - (C\,v) = m\,a \\
a = (T - (C\,v))\,/\,m
\end{gathered}$$

where:

$$v = v_2 = (T/C) - e^{-(C/m)\,t}\,(T/C - v_1)$$

In summary, the equations for velocity, distance traveled, and acceleration are as follows:

$$\begin{gathered}
v_2 = (T/C) - e^{-(C/m)\,t}\,(T/C - v_1) \\
s_2 = s_1 + (T/C)\,t + (T/C - v_1)\,(m/C)\,e^{-(C/m)\,t} - (T/C - v_1)\,(m/C) \\
a = (T - (C\,v))\,/\,m
\end{gathered}$$

To illustrate the motion of the ship further, we've plotted the ship's speed, distance traveled, and acceleration versus time, as shown in Figure 4-2, Figure 4-3, and Figure 4-4. To facilitate these illustrations, we've assumed the following:

- 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.

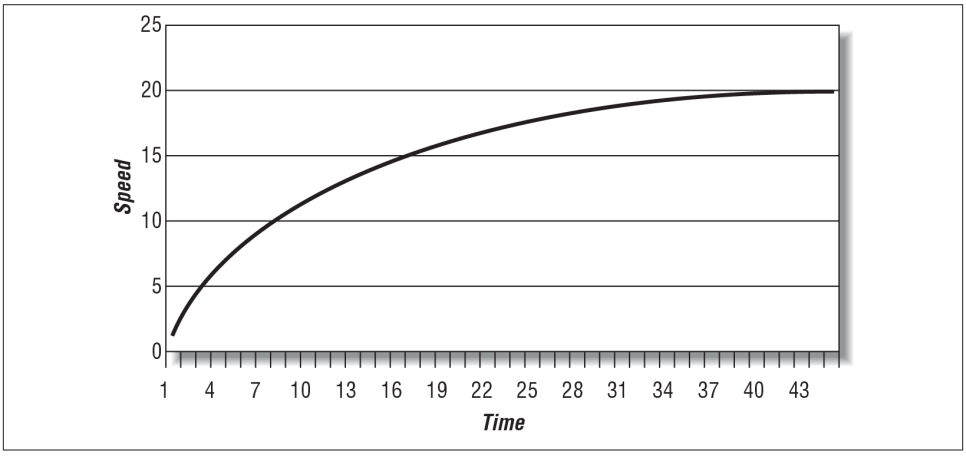


Figure 4-2. Speed versus time

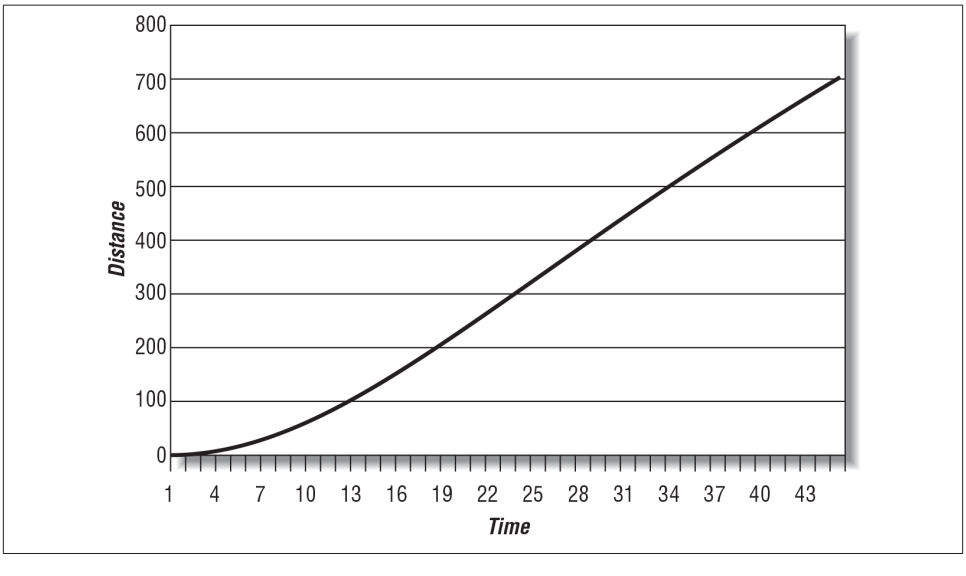


Figure 4-3. Distance versus time

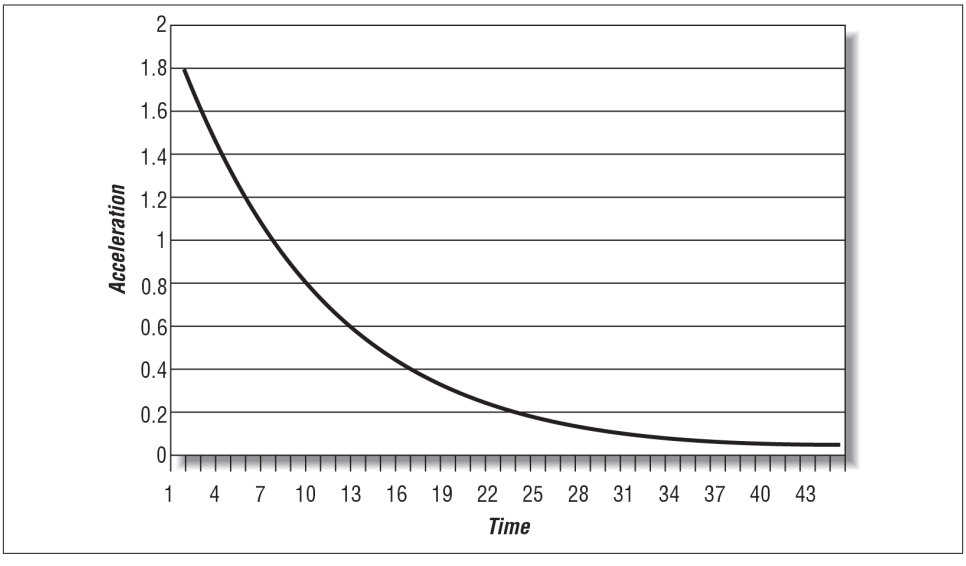


Figure 4-4. Acceleration versus time

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:

$$\Sigma F_x = m\, a_x$$
$$\Sigma F_y = m\, a_y$$

$$\Sigma F_Z = m\ a_Z$$

The resultant force and acceleration vectors are now:

$$\mathbf{a} = a_X\ \mathbf{i} + a_Y\ \mathbf{j} + a_Z\ \mathbf{k}$$
$$a = \sqrt{a_x^2 + a_y^2 + a_z^2}$$
$$\Sigma\mathbf{F} = \Sigma F_X\ \mathbf{i} + \Sigma F_Y\ \mathbf{j} + \Sigma F_Z\ \mathbf{k}$$
$$\Sigma F = \sqrt{(\Sigma F_x)^2 + (\Sigma F_y)^2 + (\Sigma F_z)^2}$$

To hammer these concepts home, we want to present another example.

Let's go back to the cannon example program discussed in Chapter 2. In that example, we made some simplifying assumptions so we could focus on the kinematics of the problem without complicating it too much. One of the more significant assumptions we made was that there was no drag acting on the projectile as it flew through the air. Physically, this would be valid only if the projectile were moving through a vacuum, which, of course, is unlikely here on Earth. Another significant assumption we made was that there was no wind to act on the projectile and affect its course. These two considerations, drag and wind, are important in real-life projectile problems, so to make this example a little more interesting—and more challenging to the user if this were an actual game—we'll add these two considerations now.

First, assume that the projectile is a sphere and the drag force acting on it as it flies through the air is a function of some drag coefficient and the speed of the projectile. This drag force can be written as follows:

$$\mathbf{F}_d = -C_d\ \mathbf{v}$$
$$\mathbf{F}_d = -C_d\ v_x\ \mathbf{i} - C_d\ v_y\ \mathbf{j} - C_d\ v_z\ \mathbf{k}$$

where C_d is the drag coefficient, v is the velocity of the projectile (v_x, v_y, and v_z are its components), and the minus sign means that this drag force opposes the projectile's motion. Actually, we're cheating a bit here since in reality the fluid dynamic drag would be more a function of speed squared. We're doing this here to facilitate a closed-form solution. Also, the drag coefficient here would be determined experimentally for each shape. Later, we'll discuss how experimental data is used on basic shapes to give data for similar ships.

Second, assume that the projectile is subjected to a blowing wind, the force of which is a function of some drag coefficient and the wind speed. This force can be written as follows:

$$\mathbf{F}_W = -C_W\ \mathbf{v}_W$$
$$\mathbf{F}_W = -C_W\ \mathbf{v}_{WX}\ \mathbf{i} - C_W\ \mathbf{v}_{WZ}\ \mathbf{k}$$

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:

$$F_{wx} = F_w \cos \gamma = -(C_w v_w) \cos \gamma$$
$$F_{wz} = F_w \sin \gamma = -(C_w v_w) \sin \gamma$$

We ignored the y-direction as we assume the wind is flowing parallel to the ground. Finally, let's apply a gravitational force to the projectile instead of specifying the effect of gravity as a constant acceleration, as we did in Chapter 2. This allows you to include the force due to gravity in the equations of motion. Assuming that the projectile is relatively close to sea level, the gravitational force can be written as:

$$\mathbf{F}_g = -m\, g\, \mathbf{j}$$

where the minus sign indicates that it acts in the negative y-direction (pulling the projectile toward the earth), and *g* on the righthand side of this equation is the acceleration due to gravity at sea level.

Now that all of the forces have been identified, you can write the equations of motion in each coordinate direction:

$$\Sigma F_x = F_{wx} + F_{dx} = m\,(dv_x/dt)$$
$$\Sigma F_y = F_{dy} + F_{gy} = m\,(dv_y/dt)$$
$$\Sigma F_z = F_{wz} + F_{dz} = m\,(dv_z/dt)$$

Note here that we already made the substitution *dv*/*dt* for acceleration in each equation. Following the same procedure shown in the previous section, you now need to integrate each equation of motion twice—once to find an equation for velocity as a function of time, and another to find an equation for displacement as a function of time. As before, we'll show you how this is done component by component.

You might be asking yourself, where's the thrust force from the cannon that propels the projectile in the first place? In this example, we're looking specifically at the motion of the projectile after it has left the muzzle of the cannon, where there is no longer a thrust force acting on the projectile; it isn't self-propelled. To account for the effect of the cannon thrust force, which acts over a very short period of time while the projectile is within the cannon, you have to consider the muzzle velocity of the projectile when it initially leaves the cannon. The components of the muzzle velocity in the coordinate

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.

$$-F_{wx} - F_{dx} = m(dv_x/dt)$$
$$-(C_w\, v_w \cos\gamma) - C_d\, v_x = m\, dv_x/dt$$
$$dt = m\, dv_x / [-(C_w\, v_w \cos\gamma) - C_d\, v_x]$$
$$\int_{(0 \text{ to } t)} dt = \int_{(vx1 \text{ to } vx2)} -m / [(C_w\, v_w \cos\gamma) + C_d\, v_x]\, dv_x$$
$$t = -(m/C_d) \ln((C_w\, v_w \cos\gamma) + C_d\, v_x)|_{(vx1 \text{ to } vx2)}$$
$$t = -(m/C_d) \ln((C_w\, v_w \cos\gamma) + C_d\, v_{x2}) + (m/C_d) \ln((C_w\, v_w \cos\gamma) + C_d\, v_{x1})$$
$$(C_d/m)\, t = \ln[((C_w\, v_w \cos\gamma) + C_d\, v_{x1}) / ((C_w\, v_w \cos\gamma) + C_d\, v_{x2})]$$
$$e^{(C_d/m)\, t} = e^{\ln[((C_w\, v_w \cos\gamma) + C_d\, v_{x1}) / ((C_w\, v_w \cos\gamma) + C_d\, v_{x2})]}$$
$$e^{(Cd/m)\, t} = ((C_w\, v_w \cos\gamma) + C_d\, v_{x1}) / ((C_w\, v_w \cos\gamma) + C_d\, v_{x2})$$
$$((C_w\, v_w \cos\gamma) + C_d\, v_{x2}) = ((C_w\, v_w \cos\gamma) + C_d\, v_{x1})\, e^{-(C_d/m)\, t}$$
$$v_{x2} = (1/C_d)\, [\, e^{(-C_d/m)\, t}\, (c_w\, v_w \cos\gamma + C_d\, v_{x1}) - (C_w\, v_w \cos\gamma)]$$

To get an equation for displacement as a function of time, you need to recall the equation $v\, dt = ds$, make the substitution for v (using the preceding equation) and then integrate one more time.

$$v_{x2}\, dt = ds_x$$
$$(1/C_d)\, [e^{(-C_d/m)\, t}\, (c_w\, v_w \cos\gamma + C_d\, v_{x1}) - (c_w\, v_w \cos\gamma)]\, dt = ds_x$$
$$\int_{(0 \text{ to } t)} (1/C_d)\, [e^{(-C_d/m)\, t}\, (c_w\, v_w \cos\gamma + C_d\, v_{x1}) - (c_w\, v_w \cos\gamma)]\, dt$$
$$=$$
$$= \int_{(sx1 \text{ to } sx2)} ds_x$$
$$s_{x2} = [(m/C_d)\, e^{(-C_d/m)\, t}\, (-(C_w\, v_w \cos\gamma) / C_d - v_{x1}) - ((C_w\, v_w \cos\gamma) / C_d)\, t] - [(m/C_d)\, (-(C_w\, v_w \cos\gamma) / C_d - v_{x1})] + s_{x1}$$

Yes, these equations are ugly. Just imagine if we hadn't made the simplifying assumption that drag is proportional to speed and not speed squared! You would have ended up with some really nice equations with an *arctan* term or two thrown in.

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:

$$-F_{dy} - F_{gy} = m\,(dv_y/dt)$$
$$-C_d\,v_y - m\,g = m\,(dv_y/dt)$$
$$\int_{(0\text{ to }t)} dt = -m \int_{(vy1\text{ to }vy2)} 1/(C_d\,v_y + m\,g)\,dv_y$$
$$v_{y2} = (1/C_d)\,e^{(-C_d/m)t}\,(C_d\,v_{y1} + m\,g) - (m\,g)/C_d$$

Now that you have an equation for velocity, you can proceed to get an equation for displacement as before:

$$v_{y2}\,dt = ds_y$$
$$[(1/C_d)\,e^{(-C_d/m)t}\,(C_d\,v_{y1} + m\,g) - (m\,g)/C_d]\,dt = ds_y$$
$$\int_{(0\text{ to }t)} [(1/C_d)\,e^{(-C_d/m)t}\,(C_d\,v_{y1} + m\,g) - (m\,g)/C_d]\,dt = \int_{(sy1\text{ to }sy2)} ds_y$$
$$s_{y2} = s_{y1} + [-(v_{y1} + (m\,g)/C_d)\,(m/C_d)\,e^{(-C_d/m)t} - t\,(m\,g)/C_d] + [(m/C_d)(v_{y1} + (m\,g)/C_d)]$$

OK, that's two down and only one more to go.

Z Components

With the *z* component, you get a break. You'll notice that the equations of motion for the *x* and *z* components look almost the same with the exception of the *x* and *z* subscripts and the sine versus cosine terms. Taking advantage of this fact, you can simply copy the *x* component equations and replace the *x* subscript with a *z* and the cosine terms with sines and be done with it:

$$v_{z2} = (1/C_d)\,[e^{(-C_d/m)t}\,(c_w\,v_w \sin\gamma + C_d\,v_{z1}) - (c_w\,v_w \sin\gamma)]$$
$$s_{z2} = [(m/C_d)\,e^{(-C_d/m)t}\,(-(C_w\,v_w \sin\gamma)\,/\,C_d - v_{z1}) - ((C_w\,v_w \sin\gamma)/C_d)\,t] - [(m/C_d)\,(-(C_w\,v_w \sin\gamma)/C_d - v_{z1})] + s_{z1}$$

Cannon Revised

Now that you have some new equations for the projectile's displacement in each coordinate direction, you can go to the cannon example source code and replace the old displacement calculation formulas with the new ones. Make the changes in the `DoSimulation` function as follows:

```
//--------------------------------------------------------------------------//
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