A PLATFORM-AGNOSTIC APPROACH

Game Programming ALGORITHMS and TECHNIQUES

Sanjay **MADHAV**

Game Programming Algorithms and Techniques

This page intentionally left blank

Game Programming Algorithms and Techniques

A Platform-Agnostic Approach

Sanjay Madhav

Addison-Wesley

Upper Saddle River, NJ • Boston • Indianapolis • San Francisco
New York • Toronto • Montreal • London • Munich • Paris • Madrid
Capetown • Sydney • Tokyo • Singapore • Mexico City

Visit us on the Web: informit.com/aw

Library of Congress Control Number: 2013950628

ISBN-13: 978-0-321-94015-5
ISBN-10: 0-321-94015-6

Text printed in the United States on recycled paper at R.R. Donnelley in Crawfordsville, IN. First printing: December 2013

Editor-in-Chief
Mark Taub

Executive Editor
Laura Lewin

Development Editor
Chris Zahn

Managing Editor
Kristy Hart

Project Editor
Elaine Wiley

Copy Editor
Bart Reed

Indexer
WordWise Publishing Services

Proofreader
Jess DeGabriele

Technical Reviewers
Alexander Boczar
Dustin Darcy
Jeff Wofford

Editorial Assistant
Olivia Basegio

Cover Designer
Chuti Prasersith

Compositor
Nonie Ratcliff

Graphics
Laura Robbins

To my family for supporting me and to all my students
for not driving me crazy (yet).

This page intentionally left blank

Contents

ACKNOWLEDGMENTS

Even though my name might be the only one on the cover, this book simply would not have been possible without the help and support of many other individuals. I'd first like to thank my mom and dad for their support throughout the years as I pursued my education and then career. I'd also like to thank my sister Nita for going along with all the crazy and creative things we did when we were younger, and for providing great advice when we became older.

The team at Pearson also has been a pleasure to work with throughout the writing process. It starts with Laura Lewin, my executive editor, who has backed the premise of the book since day one. She and assistant editor Olivia Basegio have provided very useful guidance throughout the writing process. The editorial and production teams, especially Chris Zahn, have also been stellar in helping make the book ready for production, including art, copy editing, and typesetting.

I also want to acknowledge the technical reviewers who blocked time out of their busy schedules to help ensure the accuracy of the book—Alexander Boczar, Dustin Darcy, and Jeff Wofford. Their feedback has been invaluable, and I'm confident that the book is technically sound because of it.

My colleagues at USC, especially those in the Information Technology Program, have really shaped who I am as an instructor. I'd especially like to thank the current director of the department, Michael Crowley, as well as the director who first hired me as a part-time lecturer, Ashish Soni. I'd also like to thank Michael Zyda for establishing and leading the Computer Science (Games) program at USC. My lab assistants over the years, most notably Xin Liu and Paul Conner, have also been immensely helpful.

Finally, I'd like to thank Jason Gregory for his mentorship that dates back almost ten years. Without his guidance, I may have never ended up in the game industry, and almost certainly would never have ended up following in his footsteps and teaching at USC. He has taught me so much about games and teaching games over the years, and for that I'm grateful.

ABOUT THE AUTHOR

Sanjay Madhav is a lecturer at the University of Southern California, where he teaches several courses about and related to video game programming. Prior to joining USC full time, he worked as a programmer at several video game developers, including Electronic Arts, Neversoft, and Pandemic Studios. Although he has experience programming a wide range of systems, his primary interest is in gameplay mechanics. Some of his credited games include *Medal of Honor: Pacific Assault*, *Tony Hawk's Project 8*, *Lord of the Rings: Conquest*, and *The Saboteur*.

In 2008, Sanjay began teaching part-time at USC while still working full time in the game industry. After Pandemic Studios was shuttered at the end of 2009, he decided to refocus his efforts on teaching up-and-coming game programmers. His flagship course is an undergraduate-level game programming course that he has taught for more than ten consecutive semesters.

This page intentionally left blank

PREFACE

It wasn't long ago that the knowledge necessary to program commercial video games was only available to a small number of game industry veterans. In that bygone era, learning about the actual algorithms that were used in AAA games was akin to learning some dark and forbidden knowledge (hence titles such as Michael Abrash's seminal *Graphics Programming Black Book*). If one wanted to pursue a formal education in game programming, the choices were more or less limited to a handful of specialized trade schools. But over the past ten years, video game education has changed dramatically. Several top universities now offer courses and degrees in video game programming, and more join the ranks every single year.

A side effect of this explosion of video game curriculum is that the expectations for new hires in the industry have risen dramatically. In the early 2000s, all that was expected from junior programmers was a solid computer science background and a demonstrable passion for creating video games. The hope was that with this solid foundation, strong candidates would learn the more advanced game programming techniques on the job. Today, with so many excellent places to get a more games-focused education, an increasing number of companies now expect their junior programmers to have experience with the breadth of topics relevant to programming games.

Why Another Game Programming Book?

The expansion of video game curriculum has also increased the need for books designed with a university setting in mind. However, the majority of game programming books currently on the market target one of two types of readers: hobbyists who want to learn a little bit about making games or professionals who already have years of game industry experience. Both types of books can be frustrating in an academic setting. The hobbyist books may not be academically rigorous enough for a university, and the industry-targeted books will often sail well over the heads of inexperienced students.

One of the courses I teach at the University of Southern California is ITP 380: Video Game Programming. The students who enter the class are typically in their second or third year and already know the basics of programming. Some of the students also have experience in prototyping games using engines such as GameMaker and Unity, but ITP 380 is the students' first real exposure to programming video games. My hope is that this book will perfectly complement my course and any courses like it at other universities. Although the primary target audience is students in a university setting, anyone who is interested in learning about programming games should find this book extremely valuable.

A unique characteristic of this book is that the first 12 chapters are 100% platform and framework agnostic. This means that irrespective of programming language or target platform, the vast majority of the book will be entirely applicable to most 2D and 3D games. This separates this book from most other titles on the market, which use a specific version of a framework that usually becomes obsolete within a couple of years. By being agnostic, this book should maintain its relevance for a longer period of time. This is also helpful for a university setting because different schools may choose to use different languages and frameworks. That being said, the value of code samples is indisputable. Because of this, the last two chapters are overviews of two different games written in two different frameworks, and code samples are provided for these games.

Who Should Read This Book?

This book assumes that the reader already knows how to program in an object-oriented language (such as C++, C#, or Java) and is familiar with standard data structures such as linked lists, binary trees, and hash tables. These topics usually are covered in the first two semesters of a standard computer science curriculum, so anyone who has completed such coursework should be fine. Furthermore, having some exposure to topics in calculus will help readers grasp the concepts discussed in the linear algebra and physics chapters.

Although not a requirement, it is also helpful if the reader has prior experience in the basics of game design, or at the very least is familiar with games. At times, this book will discuss programming mechanics for specific genres, so knowing the references will be very helpful. Having some experience with prototyping games using tools such as GameMaker might also be helpful, but is by no means a requirement.

Though this book is intended for an academic setting, it should be useful for anyone who already knows general programming but is interested in professionally programming games. Topics are always presented through a very practical and application-focused lens, unlike some decidedly academic books.

Finally, because this book has a broad coverage of several different game programming topics, it may prove useful for a junior programmer who wants to brush up on different aspects of game programming. That being said, a seasoned game developer may not find that much in the way of new information in this book.

How This Book Is Organized

The first 12 chapters of this book are an overview of many of the algorithms and techniques used in game programming. These topics run the gamut from 2D and 3D graphics, physics,

artificial intelligence, cameras, and more. Although Chapters 1 through 12 are designed to be read sequentially, it is possible to go out of order to some degree. The graph in Figure P.1 demonstrates the dependencies between the chapters—so at the very least, no chapter should be read prior to its dependencies being read.

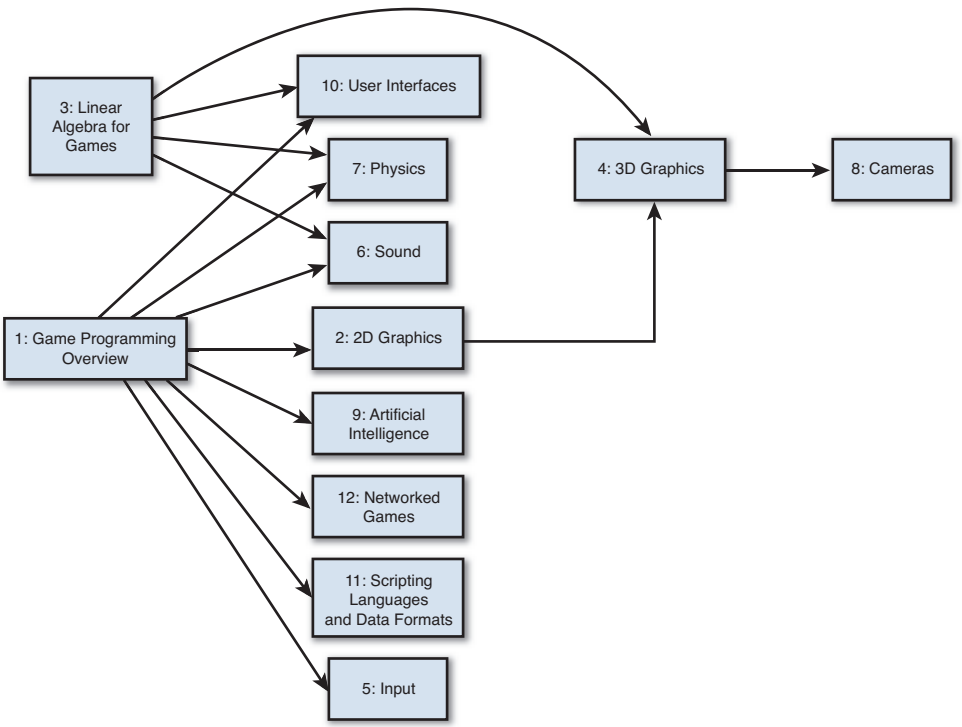


Figure P.1 Dependency graph for Chapters 1–12.

The last two chapters of the book present sample game case studies that were created to illustrate many of the algorithms and techniques covered in the first 12 chapters. The two sample games are a 2D side-scroller for iOS devices (written in Objective-C using cocos2d) and a 3D tower defense game for PC/Mac/Linux (written in C# using XNA/MonoGame). The full source code for these two case studies is available online on this book's website: http://gamealgorithms.net.

Book Features and Conventions

This section describes some of the features used in the book and the conventions followed for items such as code and equations.

Sidebars

At certain points in this book, you will find sidebars and notes. This section contains examples of both.

SIDEBAR

Sidebars are brief discussions of how a particular algorithm or technique was dealt with in actual shipped games. In some instances, these might be personal anecdotes based on projects I worked on in the industry. In other cases, they will be stories about other games. In any event, these asides are designed to give further insight into how the concept in question might be considered when you're actually creating a game.

Note

Notes are intended to be a bit more fun and tangential. Although they may not have as much pedagogical value, they still might provide further insight into the topic in question.

Pseudocode

In order to maintain language-neutrality, algorithms will often be presented in pseudocode form. The syntax used for the pseudocode has similarity with the scripting language Lua, though there are also clear influences from C/C++ as well as C#. Code snippets will be presented in fixed-font blocks, like so:

```
function Update(float deltaTime)
    foreach Object o in world
       // Comment
       o.Update(deltaTime)
    loop
end
```

This book utilizes syntax highlighting similar to many IDEs. Keywords appear in blue, comments are in green, class names are in teal, and variables are *italicized*. Member functions and variables are accessed with a dot in all instances.

In some cases, the pseudocode may be interspersed with prose explaining the code further. In these cases, it will be clear that it's not the end of the code snippet. Furthermore, in cases where this is done, the full code will be presented at the end of the section with a numbered code listing such as Listing P.1.

Listing P.1 Sample Numbered Code Listing

```
 function Update(float deltaTime)
    foreach Object o in world
       // Comment
       if o is alive
          o.Update(deltaTime)
       end
    loop
end
```

Note that the preceding pseudocode simply states to check whether the `Object` "is alive" rather than explicitly calling a particular function. This shortcut will be used in instances where the meaning is clear.

Finally, in some cases, parts of the code may be omitted to save space. This will be done sparingly, usually only when it's a repeat of previously shown code. The format for omitted code will be a comment followed by ellipses on the subsequent line:

```
function Update(float deltaTime)
    // Update code
    ...
end
```

Equations

Certain chapters (particularly the ones on linear algebra, 3D rendering, and physics) will present concepts directly in equation form. This is only done when a particular concept is more clear as an equation instead of in pseudocode. Equations will typically be centered and appear on their own line, like so:

$$f(x) = a + b$$

Companion Website

This book features a companion website at http://gamealgorithms.net. Of particular note, this site provides access to the full source code for the game case studies in Chapter 13, "Sample Game: Side-Scroller for iOS," and Chapter 14, "Sample Game: Tower Defense for PC/Mac," as well as any errata for the book. Finally, the site also has a forum where readers can post questions regarding both the book and game programming in general.

This page intentionally left blank

CHAPTER 1

GAME PROGRAMMING OVERVIEW

This chapter provides a brief history of the evolving roles of game programmers through the different eras of video game development. Once that bit of history is established, the chapter then covers three important concepts in programming any game: the game loop, management of time, and game objects.

Evolution of Video Game Programming

The first commercial video game, *Computer Space*, was released in 1971. Created by future Atari founders Nolan Bushnell and Ted Dabney, the game was not powered by a traditional computer. The hardware had no processor or RAM; it simply was a state machine created with several transistors. All of the logic of *Computer Space* had to be implemented entirely in hardware.

But when the Atari Video Computer System (Atari 2600) exploded onto the scene in 1977, developers were given a standardized platform for games. This is when video game creation became more about programming software as opposed to designing complex hardware. Though games have changed a great deal since the early Atari titles, some of the core programming techniques developed during that era are still used today. Unlike most of the book, no algorithms will be presented in this section. But before the programming begins, it's good to have a bit of context on how the video game industry arrived at its current state.

Although the focus of this section is on home console game development, the transitions described also occurred in computer game development. However, the transitions may have occurred a little bit earlier because computer game technology is usually a couple of years ahead of console game technology. This is due to the fact that when a console is released, its hardware is locked for the five-plus years the console is in the "current generation." On the other hand, computer hardware continuously improves at a dizzying pace. This is why when PC-focused titles such as *Crysis* are released, the graphical technologies can best many console games. That being said, the advantage of a console's locked hardware specification is that it allows programmers to become intimately familiar with the system over the course of several years. This leads to late-generation titles such as *The Last of Us* that present a graphical fidelity rivaling that of even the most impressive PC titles.

In any event, console gaming really did not take off until the original Atari was released in 1977. Prior to that, there were several home gaming systems, but these systems were very limited. They came with a couple of games preinstalled, and those were the only titles the system could play. The video game market really opened up once cartridge-based games became possible.

Atari Era (1977–1985)

Though the Atari 2600 was not the first generalized gaming system, it was the first extraordinarily successful one. Unlike games for modern consoles, most games for the Atari were created by a single individual who was responsible for all the art, design, and programming. Development cycles were also substantially shorter—even the most complicated games were finished in a matter of months.

Programmers in this era also needed to have a much greater understanding of the low-level operations of the hardware. The processor ran at 1.1 MHz and there was only 128 *bytes* of RAM. With these limitations, usage of a high-level programming language such as C was impractical due to performance reasons. This meant that games had to be written entirely in assembly.

To make matters worse, debugging was wholly up to the developer. There were no development tools or a **software development kit** (SDK).

But in spite of these technical challenges, the Atari was a resounding success. One of the more technically advanced titles, *Pitfall!*, sold over four million copies. Designed by David Crane and released in 1982, it was one of the first Atari games to feature an animated human running. In a fascinating GDC 2011 postmortem panel, listed in the references, Crane describes the development process and technical challenges that drove *Pitfall!*.

NES and SNES Era (1985–1995)

In 1983, the North American video game market suffered a dramatic crash. Though there were inarguably several contributing factors, the largest might have been the saturation of the market. There were dozens of gaming systems available and thousands of games, some of which were notoriously poor, such as the Atari port of *Pac-Man* or the infamous *E.T.* movie tie-in.

The release of the Nintendo Entertainment System in 1985 is largely credited for bringing the industry back on track. Since the NES was noticeably more powerful than the Atari, it required more man hours to create games. Many of the titles in the NES era required a handful of programmers; the original *Legend of Zelda*, for instance, had three credited programmers.

The SNES continued this trend of larger programming teams. One necessity that inevitably pops up as programming teams become larger is some degree of specialization. This helps ensure that programmers are not stepping on each other's toes by trying to write code for the same part of the game at the same time. For example, 1990's *Super Mario World* had six programmers in total. The specializations included one programmer who was solely responsible for Mario, and another solely for the map between the levels. *Chrono Trigger* (1995), a more complex title, had a total of nine programmers; most of them were also in specialized roles.

Games for the NES and SNES were still written entirely in assembly, because the hardware still had relatively small amounts of memory. However, Nintendo did actually provide development kits with some debugging functionality, so developers were not completely in the dark as they were with the Atari.

Playstation/Playstation 2 Era (1995–2005)

The release of the Playstation and N64 in the mid 1990s finally brought high-level programming languages to console development. Games for both platforms were primarily written in C, although assembly subroutines were still used for performance-critical parts of code.

The productivity gains of using a higher-level programming language may at least partially be responsible for the fact that team sizes did not grow during the initial years of this era. Most early games still only had eight to ten programmers in total. Even relatively complex games, such as 2001's *Grand Theft Auto III*, had engineering teams of roughly that size.

But while the earlier titles may have had roughly the same number of programmers as the latter SNES games, by the end of this era teams had become comparatively large. For example, 2004's *Full Spectrum Warrior*, an Xbox title, had roughly 15 programmers in total, many of which were in specialized roles. But this growth was minimal compared to what was to come.

Xbox 360, PS3, and Wii Era (2005–2013)

The first consoles to truly support high definition caused game development to diverge on two paths. AAA titles have become massive operations with equally massive teams and budgets, whereas independent titles have gone back to the much smaller teams of yesteryear.

For AAA titles, the growth has been staggering. For example, 2008's *Grand Theft Auto IV* had a core programming team of about 30, with an additional 15 programmers from Rockstar's technology team. But that team size would be considered tame compared to more recent titles—2011's *Assassin's Creed: Revelations* had a programming team with a headcount well over 75.

But to independent developers, digital distribution platforms have been a big boon. With storefronts such as XBLA, PSN, Steam, and the iOS App Store, it is possible to reach a wide audience of gamers without the backing of a traditional publisher. The scope of these independent titles is typically much smaller than AAA ones, and in several ways their development is more similar to earlier eras. Many indie games are made with teams of five or less. And some companies have one individual who's responsible for all the programming, art, and design, essentially completing the full circle back to the Atari era.

Another big trend in game programming has been toward **middleware**, or libraries that implement solutions to common game programming problems. Some middleware solutions are full game engines, such as Unreal and Unity. Other middleware may only implement a specific subsystem, such as Havok Physics. The advantage of middleware is that it can save time and money because not as many developers need to be allocated to work on that particular system. However, that advantage can become a disadvantage if a particular game design calls for something that is not the core competency of that particular middleware.

The Future

Any discussion of the future would be incomplete without acknowledging mobile and web-based platforms as increasingly important for games. Mobile device hardware has improved at a rapid pace, and new tablets have performance characteristics exceeding that of the Xbox 360 and PS3. The result of this is that more and more 3D games (the primary focus of this book) are being developed for mobile platforms.

But traditional gaming consoles aren't going anywhere any time soon. At the time of writing, Nintendo has already launched their new console (the Wii U), and by the time you read this, both Microsoft's Xbox One and Sony's Playstation 4 will also have been released. AAA games

for these platforms will undoubtedly have increasingly larger teams, and video game expertise will become increasingly fractured as more and more game programmers are required to focus on specializations. However, because both Xbox One and PS4 will allow self-publishing, it also means independent developers now have a full seat at the table. The future is both exciting and bright for the games industry.

What's interesting is that although much has changed in programming games over the years, many concepts from the earlier eras still carry over today. In the rest of this chapter we'll cover concepts that, on a basic level, have not changed in over 20 years: the game loop, management of time, and game object models.

The Game Loop

The **game loop** is the overall flow control for the entire game program. It's a loop because the game keeps doing a series of actions over and over again until the user quits. Each iteration of the game loop is known as a **frame**. Most real-time games update several times per second: 30 and 60 are the two most common intervals. If a game runs at 60 FPS (**frames per second**), this means that the game loop completes 60 iterations every second.

There can be many variations of the game loop, depending on a number of factors, most notably the target hardware. Let's first discuss the traditional game loop before exploring a more advanced formulation that's designed for more modern hardware.

Traditional Game Loop

A traditional game loop is broken up into three distinct phases: processing inputs, updating the game world, and generating outputs. At a high level, a basic game loop might look like this:

```
while game is running
    process inputs
    update game world
    generate outputs
loop
```

Each of these three phases has more depth than might be apparent at first glance. For instance, processing inputs clearly involves detecting any inputs from devices such as a keyboard, mouse, or controller. But those aren't the only inputs to be considered; any external input must be processed during this phase of the game loop.

As one example, consider a game that supports online multiplayer. An important input for such a game is any data received over the Internet, because the state of the game world will directly be affected by this information. Or take the case of a sports game that supports instant replay. When a previous play is being viewed in replay mode, one of the inputs is the saved replay

information. In certain types of mobile games, another input might be what's visible by the camera, or perhaps GPS information. So there are quite a few potential input options, depending on the particular game and hardware it's running on.

Updating the game world involves going through everything that is active in the game and updating it as appropriate. This could be hundreds or even thousands of objects. Later in this chapter, we will cover exactly how we might represent said game objects.

As for generating outputs, the most computationally expensive output is typically the graphics, which may be 2D or 3D. But there are other outputs as well—for example, audio, including sound effects, music, and dialogue, is just as important as visual outputs. Furthermore, most console games have "rumble" effects, where the controller begins to shake when something exciting happens in the game. The technical term for this is **force feedback**, and it, too, is another output that must be generated. And, of course, for an online multiplayer game, an additional output would be data sent to the other players over the Internet.

We'll fill in these main parts of the game loop further as this chapter continues. But first, let's look at how this style of game loop applies to the classic Namco arcade game *Pac-Man*.

The primary input device in the arcade version of *Pac-Man* is a quad-directional joystick, which enables the player to control Pac-Man's movement. However, there are other inputs to consider: the coin slot that accepts quarters and the Start button. When a *Pac-Man* arcade cabinet is not being played, it simply loops in a demo mode that tries to attract potential players. Once a quarter is inserted into the machine, it then asks the user to press Start to commence the actual game.

When in a maze level, there are only a handful of objects to update in *Pac-Man*—the main character and the four ghosts. Pac-Man's position gets updated based on the processed joystick input. The game then needs to check if Pac-Man has run into any ghosts, which could either kill him or the ghosts, depending on whether or not Pac-Man has eaten a power pellet. The other thing Pac-Man can do is eat any pellets or fruits he moves over, so the update portion of the loop also needs to check for this. Because the ghosts are fully AI controlled, they also must update their logic.

Finally, in classic *Pac-Man* the only outputs are the audio and video. There isn't any force feedback, networking, or anything else necessary to output. A high-level version of the *Pac-Man* game loop during the gameplay state would look something like what is shown in Listing 1.1.

Listing 1.1 Theoretical *Pac-Man* Game Loop

```
while player.lives > 0
   // Process Inputs
   JoystickData j = grab raw data from joystick
```

```
    // Update Game World
    update player.position based on j
    foreach Ghost g in world
       if player collides with g
          kill either player or g
       else
          update AI for g based on player.position
       end
    loop

    // Pac-Man eats any pellets
    ...

    // Generate Outputs
    draw graphics
    update audio
loop
```

Note that the actual code for *Pac-Man* does have several different states, including the aforementioned attract mode, so these states would have to be accounted for in the full game's code. However, for simplicity the preceding pseudo-code gives a representation of what the main game loop might look like if there were only one state.

Multithreaded Game Loops

Although many mobile and independent titles still use a variant of the traditional game loop, most AAA console and PC titles do not. That's because newer hardware features CPUs that have multiple cores. This means the CPU is physically capable of running multiple lines of execution, or **threads**, at the same time.

All of the major consoles, most new PCs, and even some mobile devices now feature multicore CPUs. In order to achieve maximum performance on such systems, the game loop should be designed to harness all available cores. Several different methods take advantage of multiple cores, but most are well beyond the scope of this book. However, multithreaded programming is something that has become prevalent in video games, so it bears mentioning at least one basic multithreaded game loop technique.

Rendering graphics is an extremely time-consuming operation for AAA games. There are numerous steps in the rendering pipeline, and the amount of data that needs to be processed is rather massive; some console games now render well over a million polygons per frame. Suppose it takes 30 milliseconds to render the entire scene for a particular game. It also takes an additional 20 milliseconds to perform the game world update. If this is all running on a single thread, it will take a total of 50 milliseconds to complete a frame, which will result in an unacceptably low 20 FPS. But if the rendering and world update could be completed in parallel, it

would only take 30 milliseconds to complete a frame, which means that a 30 FPS target can be achieved.

In order for this to work, the main game loop thread must be changed so it processes all inputs, updates the game world, and outputs anything other than the graphics. It then must hand off any relevant data to a secondary rendering thread, which can then draw all the graphics.

But there is a catch to this: What should the main thread do while the rendering thread is drawing? We don't want it to simply wait for the drawing to finish, because that would be no faster than performing all the actions on one thread. The way this problem is solved is by having the rendering thread always lag one frame behind the main thread. So every frame, the main thread is updating the world while the rendering thread draws the results of the last main thread update.

One big drawback of this delay is an increase of **input lag**, or how long it takes for a player's action to be visible onscreen. Suppose the "jump" button is pressed during frame 2. In the multithreaded loop, the input will not get processed until the beginning of frame 3, and the graphics will not be visible until the end of frame 4. This is illustrated in Figure 1.1.

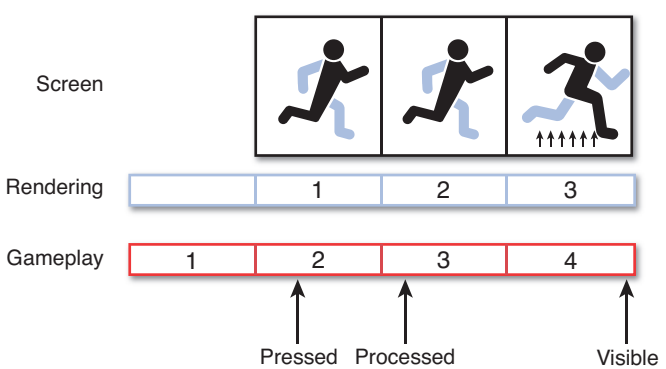


Figure 1.1 The jump is delayed a couple of frames due to input lag.

If a particular game relies on extremely quick response time, including fighting games such as *Street Fighter*, this increased input lag may be deemed unacceptable. But for most other genres, the increased lag may not be particularly noticeable. Several other factors increase input lag. The game loop can be one of these factors, and some, like the display lag most LCD panels

have, might be out of the programmer's control. For more information on this topic, check the references at the end of this chapter, which includes an interesting set of articles on the topic of measuring and solving input lag in games.

ADAPTING TO MULTICORE CONSOLES

The original Xbox and GameCube were both single-core systems, so games that were developed for these platforms essentially ran variants of the traditional game loop. But the traditional loop style became obsolete when the Xbox 360 and PS3 systems were released. Suddenly, game developers who had been accustomed to single-threaded development had to deal with the new world of multicore programming.

The initial attempt that most development studios made was the rendering/gameplay split, as described in this section. This solution ended up being shipped in most of the early titles for both consoles.

But the problem was that such an approach didn't take advantage of all the available cores. The Xbox 360 can run six threads at once, and the PS3 is able to run two threads on its general-purpose core and six on math-focused cores. So simply splitting the rendering from everything else would not use all the available threads. If a console has three available slots for threads but only two are in use, the game is only using two-thirds the total capacity.

Over time, developers improved their techniques and were able devise ways to fully utilize all the cores. This is part of the reason why the later Xbox 360 and PS3 titles look so much better than the earlier ones—they're actually using all the horsepower that's available.

Time and Games

The majority of video games have some concept of time progression. For real-time games, that progression of time is typically measured in fractions of a second. As one example, a 30 FPS title has roughly 33ms elapse from frame to frame. But even turn-based titles do feature a progression of time, except this progression is measured in turns instead of in seconds. In this section, we look at how time should be taken into account when programming a game.

Real Time and Game Time

It is very important to distinguish **real time**, the amount of time that has elapsed in the real world, from **game time**, which is how much time has elapsed in the game's world. Although there may often be a 1:1 correspondence between real time and game time, that certainly is not always the case. Take, for instance, a game in a paused state. Although a great deal of time might be elapsing in the real world, the game time is stopped entirely. It's not until the game is unpaused that the game time starts updating again.

There are several other instances where the real time and game time might diverge. For example, to allow for more nuanced gunfights, *Max Payne* uses a "bullet time" gameplay mechanic that reduces the speed of the game. In this case, the game time must update at a substantially slower rate than actual time. On the opposite end of the spectrum, many sports games feature sped-up time. In a football game, rather than requiring a player to sit through 15 full minutes per quarter, the game may update the clock twice as fast, so it actually only takes half the time. And some games may even have time progress in reverse. For example, *Prince of Persia: The Sands of Time* featured a unique mechanic where the player could rewind the game time to a certain point.

With all these different ways real time and game time might diverge, it's clear that our video game's loop should take elapsed game time into account. The following section discusses how our game loop might be updated to account for this requirement.

Logic as a Function of Delta Time

Early games were often programmed with a specific processor speed in mind. A game's code might be written explicitly for an 8 MHz processor, and as long as it worked properly it was considered acceptable. In such a setup, code that updates the position of an enemy might look something like this:

```
// Update x position by 5 pixels
enemy.position.x += 5
```

If this code moves the enemy at the desired speed on an 8 MHz processor, what happens on a 16 MHz processor? Well, assuming that the game loop now runs twice as fast, that means that the enemy will also now move twice as fast. This could be the difference between a game that is challenging and one that is impossible. Now, imagine running this 8 MHz–designed game on a modern processor that is hundreds of times faster. The game would be over before you even blinked!

In other words, if the preceding enemy movement pseudocode were run 30 times per second (30 FPS), the enemy would move a total of 150 pixels in one second. However, at 60 FPS, the enemy would move a total of 300 pixels during that same period of time. To solve this issue, we

need to introduce the concept of **delta time**: the amount of elapsed game time since the last frame.

In order to convert the preceding pseudocode to use delta time, we need to think of the movement not in terms of pixels per frame, but in terms of pixels per second. So if the ideal movement speed is 150 pixels per second, this pseudocode would be preferable:

```
// Update x position by 150 pixels/second
enemy.position.x += 150 * deltaTime
```

Now the code will work perfectly fine regardless of the frame rate. At 30 FPS, the enemy will move 5 pixels per frame, for a total of 150 pixels per second. At 60 FPS, the enemy will only move 2.5 pixels per frame, but that will still result in a total of 150 pixels per second. The movement certainly will be smoother in the 60 FPS case, but the overall per-second speed will be identical.

As a rule of thumb, whenever an object in the game world is having its properties modified in a way that should be done over the course of several frames, the modification should be written as a function of delta time. This applies to any number of scenarios, including movement, rotation, and scaling.

But how do you calculate what the delta time should be every frame? First, the amount of real time that has elapsed since the previous frame must be queried. This will depend greatly on the framework, and you can check the sample games to see how it's done in a couple of them. Once the elapsed real time is determined, it can then be converted to game time. Depending on the state of game, this may be identical in duration or it may have some factor applied to it.

This improved game loop would look something like what's shown in Listing 1.2.

Listing 1.2 Game Loop with Delta Time

```
while game is running
   realDeltaTime = time since last frame
   gameDeltaTime = realDeltaTime * gameTimeFactor

   // Process inputs
   ...
   update game world with gameDeltaTime

   // Render outputs
   ...
loop
```

Although it may seem like a great idea to allow the simulation to run at whatever frame rate the system allows, in practice there can be several issues with this. Most notably, any game that has

even basic physics (such as a platformer with jumping) will have wildly different behavior based on the frame rate. This is because of the way numeric integration works (which we'll discuss further in Chapter 7, "Physics"), and can lead to oddities such as characters jumping higher at lower frame rates. Furthermore, any game that supports online multiplayer likely will also not function properly with variable simulation frame rates.

Though there are more complex solutions to this problem, the simplest solution is to implement **frame limiting**, which forces the game loop to wait until a target delta time has elapsed. For example, if the target frame rate is 30 FPS and only 30ms has elapsed when all the logic for a frame has completed, the loop will wait an additional ~3.3ms before starting its next iteration. This type of game loop is demonstrated in Listing 1.3. Even with a frame-limiting approach, keep in mind that it still is imperative that all game logic remains a function of delta time.

Listing 1.3 Game Loop with Frame Limiting

```
// 33.3ms for 30 FPS
targetFrameTime = 33.3f
while game is running
   realDeltaTime = time since last frame
   gameDeltaTime = realDeltaTime * gameTimeFactor

   // Process inputs
   ...

   update game world with gameDeltaTime

   // Render outputs
   ...

   while (time spent this frame) < targetFrameTime
      // Do something to take up a small amount of time
      ...
   loop
loop
```

There is one further case that must be considered: What if the game is sufficiently complex that occasionally a frame actually takes *longer* than the target frame time? There are a couple of solutions to this problem, but a common one is to skip rendering on the subsequent frame in an attempt to catch back up to the desired frame rate. This is known as **dropping a frame**, and will cause a perceptible visual hitch. You may have noticed this from time to time when playing a game and performing things slightly outside the parameters of the expected gameplay (or perhaps the game was just poorly optimized).

Game Objects

In a broad sense, a **game object** is anything in the game world that needs to be updated, drawn, or both updated *and* drawn on every frame. Even though it's described as a "game object," this does not necessarily mean that it must be represented by a traditional object in the object-oriented sense. Some games employ traditional objects, but many employ composition or other, more complex methods. Regardless of the implementation, the game needs some way to track these objects and then incorporate them into the game loop. Before we worry about incorporating the objects into the loop, let's first take a look at the three categories of game objects a bit more closely.

Types of Game Objects

Of the three primary types of game objects, those that are both updated and drawn are the most apparent. Any character, creature, or otherwise movable object needs to be updated during the "update game world" phase of the game loop and needs to be drawn during the "generate outputs" phase. In *Super Mario Bros.*, Mario, any enemies, and all of the dynamic blocks would be this type of game object.

Objects that are only drawn but not updated are sometimes called **static objects**. These objects are those that are definitely visible to the player but never need to be updated. An example of this type of object would be a building in the background of a level. A building isn't going to get up and move or attack the player, but it certainly needs to be drawn.

The third type of game object, those that are updated but not drawn, is less apparent. One example is the camera. You technically can't see the camera (you can see *from* the camera), but many games feature moving cameras. Another example is what's known as a **trigger**. Many games are designed so that when the player moves to a certain location, something happens. For example, a horror game might want to have zombies appear when the player approaches a door. The trigger is what detects that the player is in position and triggers the appropriate action. So a trigger is an invisible box that must be updated to check for the player. It shouldn't be drawn (unless in debug mode) because it suspends disbelief for the gamer.

Game Objects in the Game Loop

To use game objects in the game loop, we first need to determine how to represent them. As mentioned, there are several ways to do this. One such approach, which uses the OOP concept of interfaces, is outlined in this section. Recall that an **interface** is much like a contract; if a class implements a particular interface, it is promising to implement all of the functions outlined in the interface.

First, we need to have a base game object class that all the three types of game objects can inherit from:

```
class GameObject
    // Member data/functions omitted
    ...
end
```

Any functionality that all game objects should share, regardless of type, could be placed in this base class. Then we could declare two interfaces, one for drawable objects and one for updatable objects:

```
interface Drawable
   function Draw()
end

interface Updateable
   function Update (float deltaTime)
end
```

Once we have these two interfaces, we can then declare our three types of game objects relative to both the base class and said interfaces:

```
// Update-only Game Object
class UGameObject inherits GameObject, implements Updateable
   // Overload Update function
   ...
end

// Draw-only Game Object
class DGameObject inherits GameObject, implements Drawable
   // Overload Draw function
   ...
end

// Update and Draw Game Object
class DUGameObject inherits UGameObject, implements Drawable
   // Inherit overloaded Update, overload Draw function
   ...
end
```

If this were implemented in a language that provides multiple inheritance, such as C++, it might be tempting to have `DUGameObject` just directly inherit from `UGameObject` and `DGame Object`. But this will make your code very complicated, because `DUGameObject` will inherit from two different parents (`UGameObject` and `DGameObject`) that in turn both inherit from the same grandparent (`GameObject`). This issue is known as the **diamond problem**, and although there are solutions to this problem, it's typically best to avoid the situation unless there's a very good reason for it.

Once these three types of classes are implemented, it's easy to incorporate them into the game loop. There could be a `GameWorld` class that has separate lists for all the updateable and drawable game objects in the world:

```
class GameWorld
    List updateableObjects
    List drawableObjects
end
```

When a game object is created, it must be added to the appropriate object list(s). Conversely, when an object is removed from the world, it must be removed from the list(s). Once we have storage for all our game objects, we can flesh out the "update game world" part of our loop, as shown in Listing 1.4.

Listing 1.4 Final Game Loop

```
while game is running
   realDeltaTime = time since last frame
   gameDeltaTime = realDeltaTime * gameTimeFactor

   // Process inputs
   ...

   // Update game world
   foreach Updateable o in GameWorld.updateableObjects
      o.Update(gameDeltaTime)
   loop

   // Generate outputs
   foreach Drawable o in GameWorld.drawableObjects
      o.Draw()
   loop

   // Frame limiting code
   ...
loop
```

This implementation is somewhat similar to what Microsoft uses in their XNA framework, though the version presented here has been distilled to its essential components.

Summary

This chapter covered three core concepts that are extremely important to any game. The game loop determines how all the objects in the world are updated every single frame. Our management of time is what drives the speed of our games and ensures that gameplay can

be consistent on a wide variety of machines. Finally, a well-designed game object model can simplify the update and rendering of all relevant objects in the world. Combined, these three concepts represent the core building blocks of any real-time game.

Review Questions

1. Why were early console games programmed in assembly language?
2. What is middleware?
3. Select a classic arcade game and theorize what it would need to do during each of the three phases of the traditional game loop.
4. In a traditional game loop, what are some examples of outputs beyond just graphics?
5. How does a basic multithreaded game loop improve the frame rate?
6. What is input lag, and how does a multithreaded game loop contribute to it?
7. What is the difference between real time and game time, and when would game time diverge from real time?
8. Change the following 30 FPS code so it is not frame rate dependent:

```
position.x += 3.0
position.y += 7.0
```

9. How could you force a traditional game loop to be locked at 30 FPS?
10. What are the three different categories of game objects? Give examples of each.

Additional References

Evolution of Video Game Programming

Crane, David. "GDC 2011 Classic Postmortem on Pitfall!" (http://tinyurl.com/6kwpfee**).** The creator of *Pitfall!*, David Crane, discusses the development of the Atari classic in this one-hour talk.

Game Loops

Gregory, Jason. *Game Engine Architecture*. Boca Raton: A K Peters, 2009. This book dedicates a section to several varieties of multithreaded game loops, including those you might use on the PS3's asymmetrical CPU architecture.

West, Mick. "Programming Responsiveness" and "Measuring Responsiveness" (http://tinyurl.com/594f6r **and** http://tinyurl.com/5qv5zt**).** These Gamasutra articles written by Mick West (co-founder of Neversoft) discuss factors that can cause increased input lag as well as how to measure input lag in games.

Game Objects

Dickheiser, Michael, Ed. *Game Programming Gems 6*. Boston: Charles River Media, 2006.
One of the articles in this volume, "Game Object Component System," describes an alternative to a more traditional object-oriented model. Although this implementation may be a bit complex, more and more commercial games are trending toward game object models that use composition ("has-a") instead of strict "is-a" relationships.

This page intentionally left blank

CHAPTER 2

2D GRAPHICS

With the explosion of web, smartphone, and indie games, 2D has had a renaissance of sorts. Developers are drawn to 2D because the typical budget and team can be much smaller. Gamers are drawn toward 2D because of the purity and simplicity of the games.

Though the primary focus of this book is on 3D games, it would be a mistake not to cover the core concepts behind 2D graphics. It also should be noted that many of the topics covered in latter chapters, whether physics, sound, or UI programming, are equally applicable in both 2D and 3D games.

2D Rendering Foundations

To fully understand 2D rendering, it is important to understand the limitations of display devices when these techniques were first developed. Even though we now almost exclusively use LCD or plasma displays, many of the rendering concepts that were originally developed with older monitors in mind are still in use today.

CRT Monitor Basics

For many years, **cathode ray tube** (CRT) displays were the predominant display technology. A CRT features an array of picture elements known as **pixels**. For a color display, each pixel contains a red, green, and blue sub-pixel, which can then be combined to create specific colors. The resolution of the display determines the total number of pixels. For instance, a 300×200 display would have 200 total rows, or **scan lines**, and each scan line would have 300 pixels, for a grand total of 60,000 pixels. The (0,0) pixel usually refers to the top-left corner, though not all displays follow this format.

In a CRT, all of the drawing is done by an electron gun that fires a narrow stream of electrons. This gun starts drawing in the top-left corner of the screen and shifts its aim across horizontally to draw the first scan line (see Figure 2.1). It then repositions its aim so it can start all the way at the start of the subsequent scan line, repeating this process until all scan lines have been drawn.

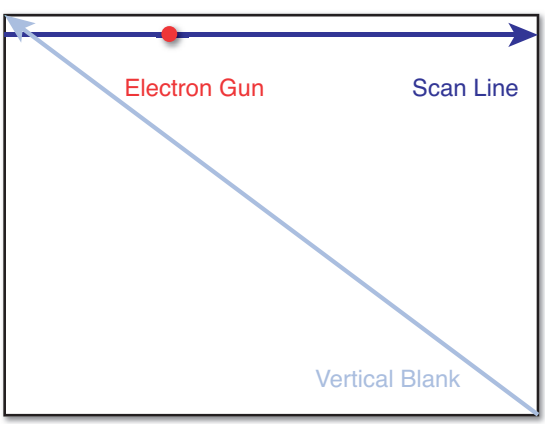


Figure 2.1 Basic CRT drawing.

When the electron gun has finished drawing one frame, its aim will be positioned at the bottom-right corner of the CRT. The amount of time it takes for the electron gun to shift its aim from the bottom-right corner all the way back to the top-left corner is known as the **vertical blank interval** (VBLANK). This interval is a fraction of a second, though the precise timing

depends on whether or not the CRT is for a television or computer, and additionally the region where the device is intended for use.

Early gaming hardware such as the Atari did not have enough memory to store the pixel data for the entire screen at once, which further complicated the rendering systems on that system. This topic is covered in greater detail in David Crane's GDC talk that was referenced in Chapter 1, "Game Programming Overview."

Color Buffers and Vertical Sync

Newer hardware featured more than enough memory to have a **color buffer** that could store the pixel data for the entire screen at once. But this did not mean that the game loop could ignore the CRT gun entirely. Suppose the electron gun is half way through drawing the screen. At this exact time, it just so happens that the game loop hits the "generate outputs" phase. So it starts writing the pixel information into the color buffer for the next frame, while the CRT is still drawing the last frame. The result of this is **screen tearing**, which is when the screen shows part of two different frames at once. An example of this is shown in Figure 2.2.

What's worse is that the game loop might not actually finish writing all its image information for the next frame before it gets drawn on the screen. So not only will the second half of the screen be the wrong frame, it may even be missing graphical elements!

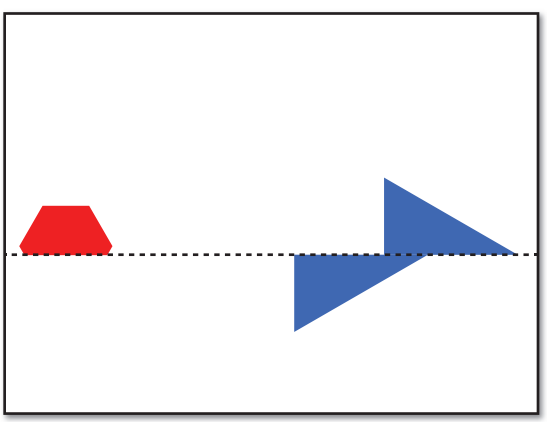


Figure 2.2 Screen tearing caused by updating video data while the CRT is drawing.

One solution to this problem is to synchronize it so the game loop renders only during the vertical blank interval. This will fix the split-image problem, but it limits the amount of time the game loop has to render to the color buffer to only that VBLANK period, which isn't going to be enough time for a modern game.

It is instead possible to solve screen tearing with a rendering technique called **double buffering**. In double buffering, there are *two* color buffers. The game alternates between drawing to these two buffers. On one frame, the game loop might write to buffer A while the CRT displays buffer B. Then on the next frame, the CRT will display buffer A while the game loop writes to buffer B. As long as both the CRT and game loop aren't accessing the same buffer at the same time, there is no risk of the CRT drawing an incomplete frame.

In order to fully prevent screen tearing, the buffer swap must happen during VBLANK. This is often listed as VSYNC in the graphics settings for games, though technically it's a misnomer because VSYNC is the signal the monitor sends the system to let it know VBLANK has commenced. In any event, because the buffer swap is a relatively fast operation, the game has a much longer period of time to render the entire frame (though ideally this should be less than the amount of time it takes the CRT to draw a frame). So long as the buffer swap occurs during VBLANK, screen tearing will be entirely avoided.

Here's what a game world render function might look like with double buffering:

```
function RenderWorld()
    // Draw all objects in the game world
    ...

    wait for VBLANK
    swap color buffers
end
```

Some games do allow buffer swaps to occur as soon as rendering finishes, which means there may be some screen tearing. This is typically allowed when a user wants to run the game at a frame rate much higher than the screen refresh rate. If a particular monitor has a 60 Hz refresh rate, synchronizing the buffer swaps to VBLANK would cap the frame rate at 60 FPS. But players who are very conscientious of reducing their input lag (and have fast enough computers) may be able to achieve much higher frame rates if that cap is removed.

Even though CRT monitors are rarely used today, double buffering still will prevent screen tearing if buffer swaps are timed correctly on an LCD. Some games even use **triple buffering**, in which three color buffers are used instead of two. Triple buffering can help smooth out the frame rate if there is high volatility, but at the cost of increased input lag.

Sprites

A **sprite** is a 2D visual object within the game world that can be drawn using a single image on any given frame. Typically sprites are used to represent characters and other dynamic objects. For simple games, sprites might also be used for backgrounds, though there are more efficient

approaches, especially for static backgrounds. Most 2D games will have dozens if not hundreds of sprites, and for mobile games the sprites often make up the majority of the overall download size of the game. Because of this, it's important to try to use sprites as efficiently as possible.

The first decision one has to make with sprites is the image format in which they will be stored. This will vary depending on the platform as well as memory constraints. A PNG file may take up less space, but hardware typically cannot natively draw PNGs, so they must be converted when loaded by the game. A TGA file can usually be drawn directly, but TGA file sizes are typically massive. On iOS devices, the preferred file format is PVR, because it is a compressed image format that can also be drawn natively by the hardware.

The process of loading an image file into memory can also vary greatly depending on the platform and framework. For frameworks such as SDL, XNA, and cocos2d, functionality for loading many file formats is built in. But if you are writing a 2D game from scratch, one relatively simple cross-platform library is stb_image.c (http://nothings.org/stb_image.c), which can load several file formats, including JPEG and PNG. Because it's written in portable C code, it will work in C, C++, and Objective-C programs.

Drawing Sprites

Suppose you have a basic 2D scene with a background image and a character in the center. The simplest approach to drawing this scene would be to first draw the background image and then draw the character. This is much like how a painter would paint the scene on a canvas, and because of this, the approach is known as the **painter's algorithm**. In the painter's algorithm, all the sprites in a scene are sorted from back to front (see Figure 2.3). When it's time to render the scene, the presorted scene can then be traversed in order so that it can be drawn appropriately.

This can be implemented without much issue in a 2D game. Each sprite object can have an integral draw order value, in addition to any other information such as image file data and the sprite's x and y positions:

```
class Sprite
    ImageFile image
    int drawOrder
    int x, y
    function Draw()
       // Draw the image at the correct (x,y)
       ...
    end
end
```

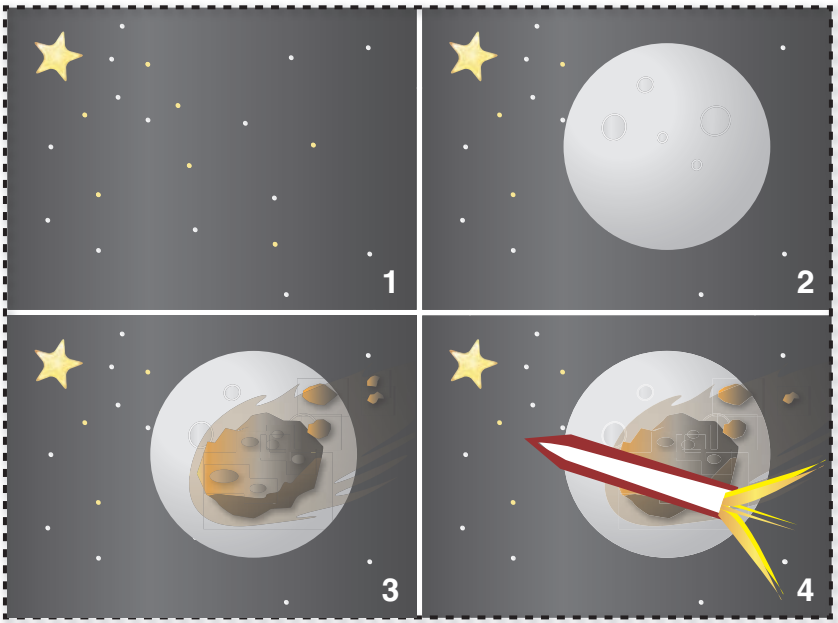


Figure 2.3 Painter's algorithm applied to a 2D space scene.

Then, the game world's list of drawable objects can be changed so it's a sorted container that's sorted based on the draw order. This way, during the rendering step, the sorted container can be linearly traversed to draw the sprites in the correct order.

```
SortedList spriteList

// When creating a new sprite...
Sprite newSprite = specify image and desired x/y
newSprite.drawOrder = set desired draw order value
// Add to sorted list based on draw order value
spriteList.Add(newSprite.drawOrder, newSprite)

// When it's time to draw...
foreach Sprite s in spriteList
   s.Draw()
loop
```

As we will discuss in Chapter 4, "3D Graphics", the painter's algorithm can also be utilized in a 3D environment, though there are some distinct drawbacks that come out of this. But for 2D scenes, the painter's algorithm generally works well.

Some 2D libraries, including cocos2d, allow a scene to be composed of any number of layers, each of which can have a specific draw order. Thus, a game might have a background layer, a

character layer, and a UI layer (in that order, from back to front). Each layer can then contain any number of sprites. In our example, all the sprites in the background layer will automatically be placed behind all the sprites in the character layer, and both the background and character layers will show up behind the UI one. Within each individual layer, you can still specify the draw order for a particular sprite, except the draw order is only relative to other sprites in that same layer.

As for the x- and y-coordinates of the sprite, the location it references depends on the library. In some cases, it might be the top-left corner of the sprite relative to the top left of the screen, whereas others might refer to the center of the sprite relative to the bottom left. The choice is arbitrary and comes down to how a particular developer felt like implementing the functionality.

Animating Sprites

For most 2D games, animation is based on the principles of traditional flipbook animation: a series of static 2D images are played in rapid succession to create an illusion of motion (see Figure 2.4). For smooth animation, typically a minimum of 24 FPS is required, which is the traditional frame rate that is used in film. This means that for every one second of animation, you need 24 individual images. Some genres, such as 2D fighting games, may use 60 FPS animations, which increases the required number of images dramatically.

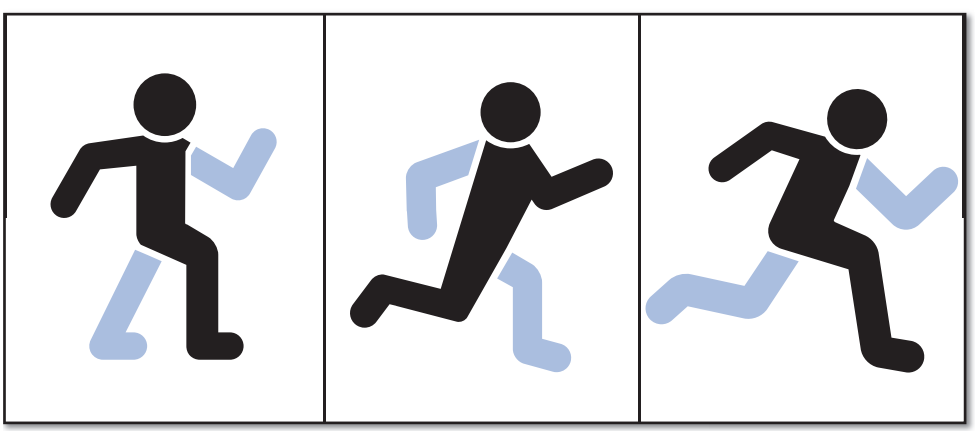


Figure 2.4 Animated run cycle.

A typical approach is to have an array of images that represents all the possible states of a particular character, regardless of the particular animation. For example, a character that has both a walk and run cycle, each ten frames in length, would have an array of 20 images in total. To keep things simple, these images would be stored sequentially, which would mean frames 0–9 would correspond to the walk cycle and frames 10–19 would correspond to the run cycle.

But this means we need some way to specify which frames correspond to which animations. A simple way to encapsulate this animation frame information is to create an `AnimFrameData` structure that specifies the start frame and number of frames for one particular animation:

```
struct AnimFrameData
    // The index of the first frame of an animation
    int startFrame
    // The total number of frames for said animation
    int numFrames
end
```

We would then have an `AnimData` struct that stores both the array of all the possible images, as well as the `FrameData` corresponding to each particular animation:

```
struct AnimData
    // Array of images for all the animations
    ImageFile images[]
    // The frame data for all the different animations
    AnimFrameData frameInfo[]
end
```

We then need an `AnimatedSprite` class that inherits from `Sprite`. Because it inherits from `Sprite`, it will already have position and draw order functionality, as well as the ability to draw a single `ImageFile`. However, because an `AnimatedSprite` is much more complicated than a single-image `Sprite`, several additional member variables are required.

`AnimatedSprite` must track the current animation number, the current frame in that animation, and the amount of game time the current frame has been displayed. You may notice that the animation FPS is also stored as a separate member variable. This allows the animation to dynamically speed up or slow down. For instance, as a character gains speed, it would be possible to have the run animation play more rapidly.

```
class AnimatedSprite inherits Sprite
    // All of the animation data (includes ImageFiles and FrameData)
    AnimData animData
    // The particular animation that is active
    int animNum
    // The frame number of the active animation that's being displayed
    int frameNum
    // Amount of time the current frame has been displayed
    float frameTime
    // The FPS the animation is running at (24FPS by default).
    float animFPS = 24.0f

    function Initialize(AnimData myData, int startingAnimNum)
    function UpdateAnim(float deltaTime)
    function ChangeAnim(int num)
end
```

The `Initialize` function takes in a reference to the corresponding `AnimData` for this particular `AnimatedSprite`. By passing in a reference, an unlimited number of animated sprites can use the same image data, which will save large quantities of memory. Furthermore, the function requires passing in the first animation that will be played so that the remaining member variables can be initialized by the `ChangeAnim` function.

```
function AnimatedSprite.Initialize(AnimData myData, int startingAnimNum)
    animData = myData
    ChangeAnim(startingAnimNum)
end
```

The `ChangeAnim` function gets called when the `AnimatedSprite` swaps to a particular animation. It sets both the frame number and the amount of time in the frame to zero, and sets the active image to be the first frame in the animation. In both this function and in `UpdateAnim`, we use the `image` variable to represent the active image. That's because the base class (`Sprite`) uses this variable to determine what it should draw on a particular frame.

```
function AnimatedSprite.ChangeAnim(int num)
    animNum = num
    // The active animation is now at frame 0 and 0.0f time
    frameNum = 0
    animTime = 0.0f
    // Set active image, which is just the starting frame.
    int imageNum = animData.frameInfo[animNum].startFrame
    image = animData.images[imageNum]
end
```

The `UpdateAnim` function is where most of the heavy lifting of `AnimatedSprite` occurs. Much of its complexity stems from the fact that we can't assume that the animation frame rate is slower than the game frame rate. For example, a game could be running at 30 FPS but we may want to play a 24 FPS animation at 2× speed (or 48 FPS). This means that `UpdateAnim` will quite often need to skip ahead multiple frames in the animation. To take this into account, once a particular frame has been displayed longer than the animation frame time, we need to calculate how many frames to jump ahead. This also means that when a looping animation wraps around, we can't just reset it to frame 0. It might be that the animation is playing so quickly that we need to reset to frame 1 or 2.

```
function AnimatedSprite.UpdateAnim(float deltaTime)
    // Update how long the current frame has been displayed
    frameTime += deltaTime

    // This check determines if it's time to change to the next frame.
    if frameTime > (1 / animFPS)
       // The number of frames to increment is the integral result of
       // frameTime / (1 / animFPS), which is frameTime * animFPS
       frameNum += frameTime * animFPS
```

```
        // Check if we've advanced past the last frame, and must wrap.
        if frameNum >= animData.frameInfo[animNum].numFrames
            // The modulus (%) makes sure we wrap correctly.
            // (Eg. If numFrames == 10 and frameNum == 11, frameNum would
            // wrap to 11 % 10 = 1).
            frameNum = frameNum % animData.frameInfo[animNum].numFrames
        end

        // Update the active image.
        // (startFrame is relative to all the images, while frameNum is
        // relative to the first frame of this particular animation).
        int imageNum = animData.frameInfo[animNum].startFrame + frameNum
        image = animData.images[imageNum]

        // We use fmod (floating point modulus) for the same reason
        // as the % above.
        frameTime = fmod(frameTime, 1 / animFPS)
    end
end
```

Although this implementation of `AnimatedSprite` will work for looping animations, it has no support for transitions between animations. If such functionality is required, it's advisable to create an animation state machine. Chapter 9, "Artificial Intelligence," discusses the state design pattern in the context of AI, but said state pattern could just as easily apply to an animation state machine.

Sprite Sheets

In order to line up animated sprites properly, it is preferable to have all the sprites for a particular character be the same size. In the past, many libraries also required that all image files had dimensions that were powers of two, in which case a 30×30 image would have to be padded to 32×32. Although most modern graphics libraries do allow non-power-of-two textures, power-of-two textures are still preferable due to a concept known as mipmapping (which is beyond the scope of this book).

You could certainly have each frame of an animated sprite be an individual image file (or **texture**). Although this is a simple system, it typically ends up wasting a great deal of memory. That's because sprites usually aren't rectangles, even though the image file must be. If the size of a particular frame's image file is 100KB, but 15% of this space unused, that means 15KB is being wasted. This can add up quickly if there are a lot of frames of an animation; even just 100 total frames (~5 total seconds of animation) would waste almost 1.5MB!

A solution to this problem is to use a single image file that contains all the sprites, called a **sprite sheet**. In a sprite sheet, it is possible to pack in the sprites closely and overlap the unused

space. This means that when the sprite sheet is opened, a bit of work is necessary to reconstruct the correct images in memory. But at least the file size can be significantly reduced, which will reduce the total installation size of the game. Figure 2.5 demonstrates saving space with a sprite sheet.

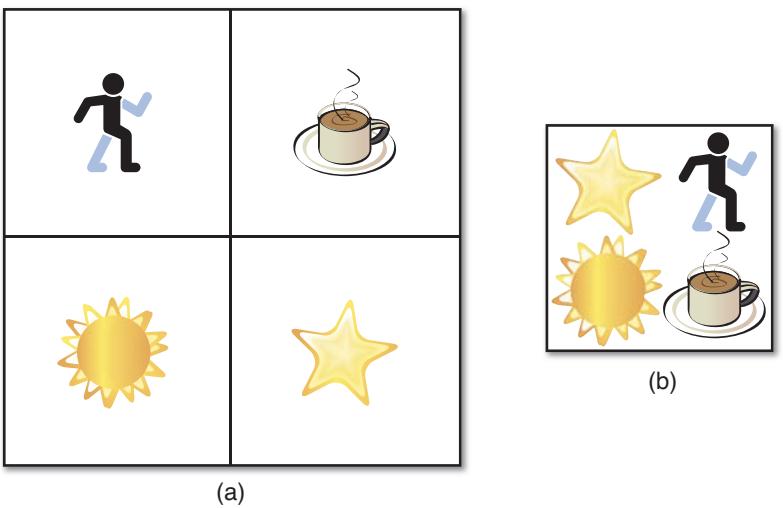


Figure 2.5 Individual sprites (a), and those sprites packed in a sprite sheet (b).

A popular tool for creating sprite sheets is TexturePacker (www.texturepacker.com), which is natively supported by many 2D libraries, including cocos2d. The Pro version of TexturePacker features some additional space-saving techniques, including rotating sprites to pack them together more tightly, and dithering, which can reduce the amount of color information a file needs to store.

Another advantage of using a sprite sheet is that most graphics processors must have a texture loaded into their own memory in order for it to be drawn. If you are switching between several individual texture files frequently when rendering a frame, this can result in a noticeable performance loss, especially if they are larger sprites. By having all of the sprites in one large texture, it is possible to eliminate this cost of switching.

Depending on how many sprites a game has, and the size of those sprites, it may not always be possible to fit all the sprites within one sprite sheet. Most hardware has a maximum texture size; for instance, current-generation iOS devices can't have textures larger than 2048×2048. So if an iOS game has more sprites than can be fit into a 2048×2048 sprite sheet, it will need to have multiple sprite sheets.

Scrolling

In a relatively simple 2D game such as *Pac-Man* or *Tetris*, everything fits on one single screen. More complex 2D games, however, often have worlds that are larger than what can be displayed on a single screen. For these games, a common approach is to have the level scroll as the player moves across it. In this section, we cover progressively more complex types of scrolling.

Single-Axis Scrolling

In **single-axis scrolling**, the game scrolls only in the x or y direction. Infinite scroller games such as *Jetpack Joyride,* shown in Figure 2.6, often feature this type of scrolling. How this is implemented depends on how large the levels are. For games where all the background images can be loaded into memory at once, as we will assume in this section, the algorithms are not too complex.

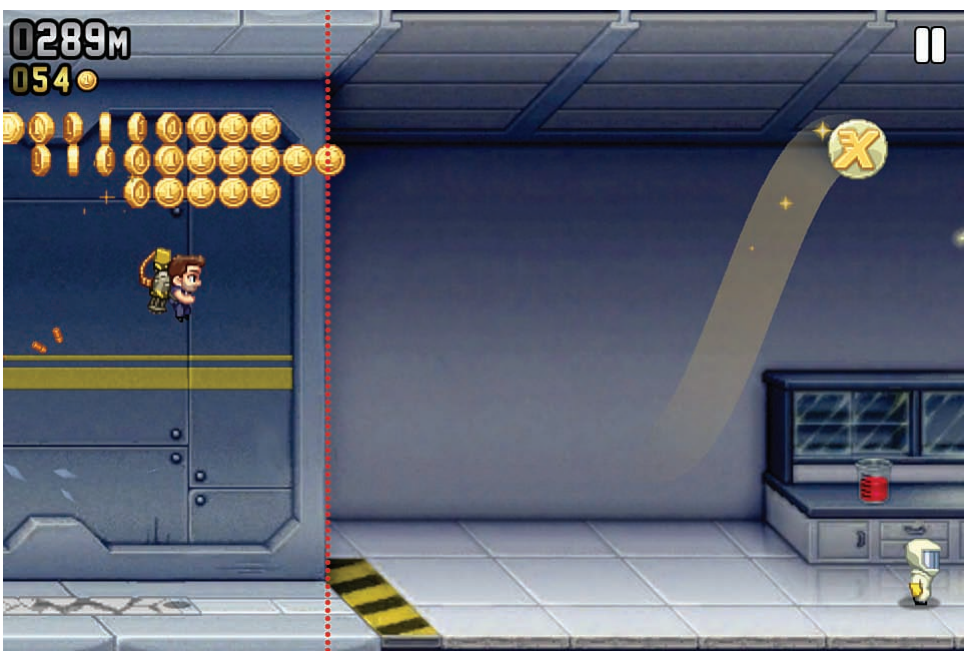


Figure 2.6 Single direction scrolling in *Jetpack Joyride*. The dotted line demonstrates a boundary.

The easiest approach is to split up the background of the level into screen-sized image segments. That's because a level may be 20 or 30 screens-sized segments, which definitely would be larger than what would fit in a single texture. Each segment can then be positioned at the appropriate x- and y-coordinates in the world when loaded. Luckily, loading the segments at their correct positions in the world is a quick calculation if the background sprites are drawn

relative to their top-left corner. As each background is loaded at the correct world position, it could be added to a linked list. For example, the setup code for horizontal scrolling would be as follows:

```
const int screenWidth = 960 // An iPhone 4/4S sideways is 960x640
// All the screen-sized image backgrounds
string backgrounds[] = { "bg1.png", "bg2.png", /*...*/}
// The total number of screen-sized images horizontally
int hCount = 0
foreach string s in backgrounds
   Sprite bgSprite
   bgSprite.image.Load(s)
   // 1st screen would be x=0, 2nd x=960, 3rd x=1920, ...
   bgSprite.x = hCount * screenWidth
   bgSprite.y = 0
   bgSpriteList.Add(bgSprite)
   screenCount++
loop
```

Once the `bgSpriteList` has been populated, we then need to determine which backgrounds should be drawn and in what position. If the background segments are the size of the screen, no more than two background segments can be onscreen at once. We need a way to track what's on screen and display the appropriate backgrounds.

One common approach is to have x- and y-coordinates for the camera also expressed as a position in the world. The camera starts out pointing at the center of the first background image. In horizontal scrolling, the camera's x-position is set to the player's x-position so long as the position does not scroll behind the first background or past the last background.

Although this could be done with `if` statements, as the complexity of the system increases these statements would become increasingly complex. A better solution is to use a **clamp** function, which takes a value and restricts it to be between a minimum and maximum value. In this case, we want the camera's x-position to be equal to the player's x-position, with a minimum valid value of half the screen width and a maximum valid value of the center of the final background.

Once we have the position of the camera, we can then determine which backgrounds are visible based on the camera. Finally, all the sprites, including the backgrounds, can be drawn with an offset based on this camera position.

```
// camera.x is player.x as long as its clamped within the valid range
camera.x = clamp(player.x, screenWidth / 2,
                           hCount * screenWidth - screenWidth / 2)

Iterator i = bgSpriteList.begin()
while i != bgSpriteList.end()
   Sprite s = i.value()
```

```
    // find the first bg image to draw
    if (camera.x - s.x) < screenWidth
        // Image 1: s.x = 0, camera.x = 480, screenWidth/2 = 480
        // 0 - 480 + 480 = 0
        draw s at (s.x - camera.x + screenWidth/2, 0)
        // draw the bg image after this, since it might also be visible
        i++
        s = i.value()
        draw s at (s.x - camera.x + screenWidth/2, 0)
        break
    end
    i++
loop
```

This provided code implements scrolling where the camera and player can move forward and backward through the level, as in games such as *Super Mario World*. However, if we want to limit the scrolling so that the camera never goes backward, all we have to do is change the code so that `camera.x` is only changed when `player.x` is a greater value.

It's also worth noting that in a framework that supports layers, such as cocos2d, it's possible to simplify this code by changing the position of the background layer when it's time to scroll. This is what's employed for the sample iOS game outlined in Chapter 13, "Sample Game: Side-Scroller for iOS." But for frameworks that don't support such a feature, you'll have to implement it in a manner similar to the preceding one.

Infinite Scrolling

Infinite scrolling is when the game continues to scroll until the player loses. Of course, there can never be an infinite number of background segments, so at some point the game will have to repeat images. In the case of Chapter 13's infinite scroller, there actually are only two nebula images, which repeat indefinitely. However, most infinite scrollers will have a greater number of possibilities as well as random generation thrown in to add some variety. Typically the game may have a series of four or five background segments that must appear in order and then a clean break after which a different series of segments can be selected.

Parallax Scrolling

In **parallax scrolling**, the background is broken up into multiple layers at different depths. Each layer then scrolls at a different speed, which gives an illusion of depth. One example could be a game where there's a cloud layer and a ground layer. If the cloud layer scrolls more slowly than the ground layer, it gives the impression that the clouds are further away than the ground.

This technique has been used in traditional animation for nearly a century, but the first game to popularize parallax scrolling was the 1982 arcade game *Moon Patrol*. On smartphone app

stores, the majority of high-profile 2D games with scrolling utilize some form of parallax scrolling, whether it's *Jetpack Joyride* or *Angry Birds Space*. Typically only three layers are necessary to create a believable parallax effect, as illustrated in Figure 2.7, but more layers add more depth to the effect.

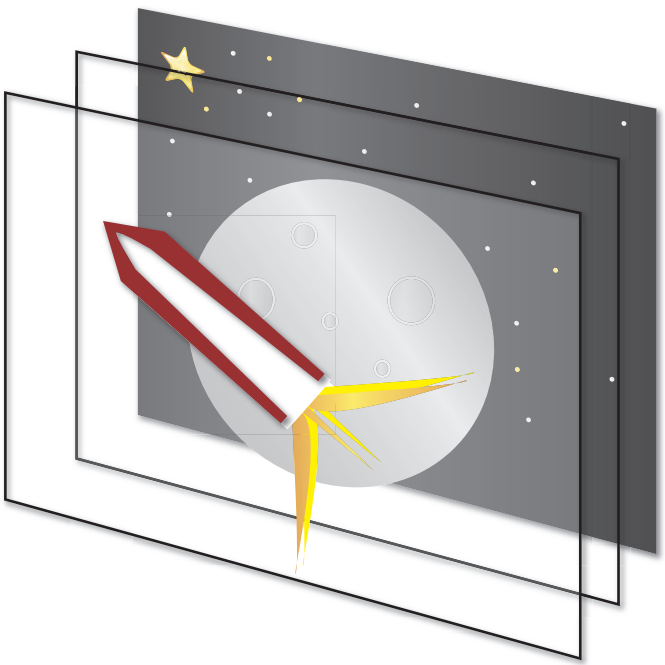


Figure 2.7 Space scene broken into three layers to facilitate parallax scrolling.

The single direction scrolling pseudocode could be updated to instead have multiple background sprite lists, one for each layer. To implement the parallax effect, when it's time to calculate the offset of the sprite based on the camera, an additional factor must be applied. If a particular background layer is intended to move at a fifth of the speed of the closest layer, the offset calculation code would be as follows:

```
float speedFactor = 0.2f
draw s at (s.x - (camera.x - screenWidth/2) * speedFactor, 0)
```

For infinite scrollers, certain background layers may have less variety than others. The cloud layer may only have ten total segments that are repeated in perpetuity, whereas more readily apparent layers might have the aforementioned randomization. And, of course, frameworks that already support layers work perfectly with parallax scrolling as well.

For further detail on parallax scrolling, you should look at the sample 2D game implemented in Chapter 13. It implements infinite horizontal scrolling with multiple background layers in order to create the parallax effect.

Four-Way Scrolling

In four-way scrolling, the game world scrolls both horizontally and vertically, as in most 2D Mario games since *Super Mario Bros. 2*. Because it's scrolling in both directions, up to four different background segments can be onscreen at any one time.

In order to update the horizontal scrolling code so it works in both directions, we will need to add code that checks whether or not the camera's y-position needs to be updated. But first, we need to declare variables that tell us what the screen height is and how many vertical segments tall the level is:

```
const int screenHeight = 640 // Height of iPhone 4/4S horizontally
int vCount = 2
```

One issue that comes up with four-way scrolling is placement of the origin. Because most 2D frameworks have a coordinate system where the top left of the screen is (0,0), it simplifies things if the top left of the game world is also (0,0). So if our game supports two vertical segments, (0,0) would be one segment above the starting segment, as in Figure 2.8.

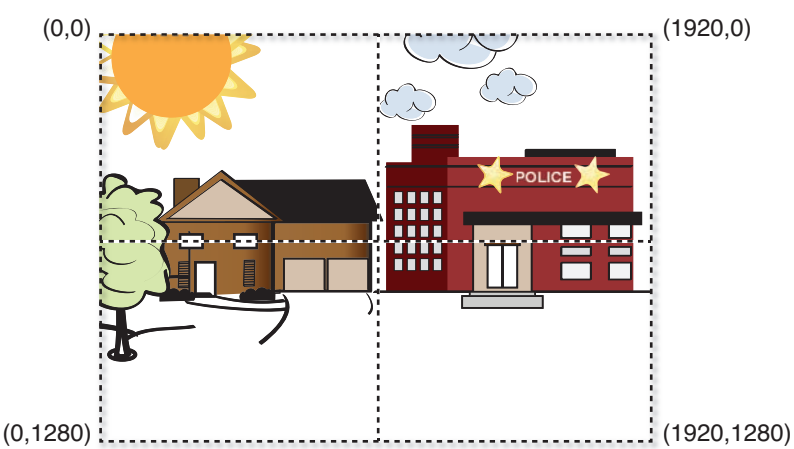


Figure 2.8 Scene broken up into four segments to support four-way scrolling.

Because we can also scroll vertically, we need to update the y-position of the camera, as well:

```
camera.y = clamp(player.y, screenHeight / 2,
                           vCount * screenHeight - screenHeight / 2)
```

Calculating the new location of the background sprite based on the camera also must take into account the y-position:

```
draw s at (s.x - camera.x + screenWidth/2,
           s.y - camera.y - screenHeight / 2)
```

To determine which background segments need to be drawn, we can't use the same approach as in horizontal scrolling. That's because there isn't just a single list of background sprites to iterate through. A basic approach might be to use a two-dimensional array to store the segments and determine both the row and column of the top-left segment on the screen. Once that segment is computed, it is trivial to get the other three visible segments.

```
// This assumes we have an 2D array [row][column] of all the segments
for int i = 0, i < vCount, i++
   // Is this the correct row?
   if (camera.y - segments[i][0].y) < screenHeight
      for int j = 0, j < hCount, j++
         // Is this the correct column?
         if (camera.x - segments[i][j].x) < screenWidth
            // This is the top left visible segment
         end
      loop
   end
loop
```

Tile Maps

Suppose you are creating a 2D top-down action roleplaying game in the vein of *The Legend of Zelda*. There are massive outdoor areas and dungeons, so you need some way to draw these levels. A naïve approach might be to simply create a unique texture to represent every single scene. But this might duplicate a great deal of content because the same tree or rock might appear several times in one scene. And to make matters worse, those trees or rocks may show up in several different scenes, as well. This means using a unique texture for every scene may end up wasting a great deal of memory.

Tile maps solve this problem by partitioning the world into squares (or other polygons) of equal size. Each square then references the sprite that is at that location in the grid. The reference can often just be a numeric lookup into the **tile set**, or set of all the possible tiles. So if the tree is at index 0 in the tile set, every square that has a tree can be represented by the number 0. Figure 2.9 demonstrates a sample scene that uses tile maps.

Even though squares are by far the most popular type of tile used in tile maps, this is not an absolute requirement. Some games choose to instead use hexagons, and others still might use parallelograms. It will solely depend on the type of view that is desired.

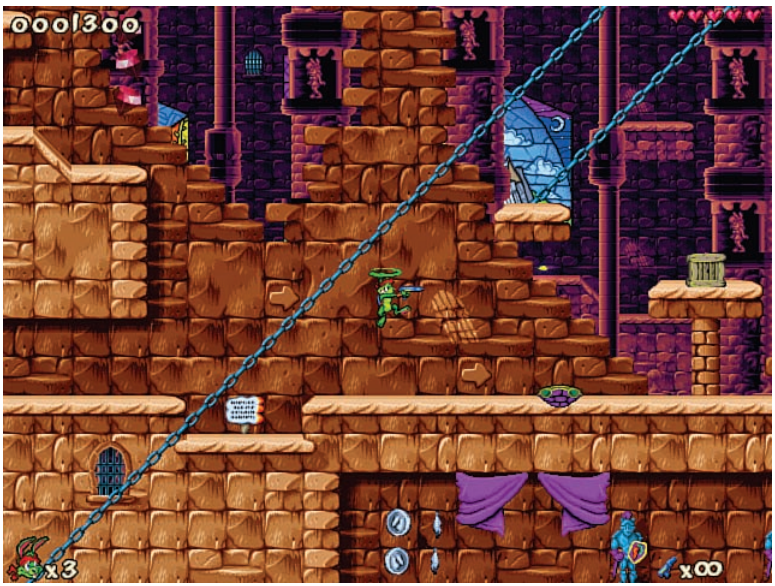


Figure 2.9 Scene from the classic platformer *Jazz Jackrabbit 2*, which uses tile maps for its levels.

In any event, tile maps provide a great way to save memory and make life much easier for the artists and designers. And 2D games with procedurally generated content, such as *Spelunky*, would be difficult to implement without tile maps.

Simple Tile Maps

Certain frameworks, such as cocos2d, have built-in support for tile maps generated by the free program Tiled (www.mapeditor.org). It's a great program, and I heartily recommend it if your framework of choice supports it. However, if Tiled is not supported by a particular framework, it still is possible to implement tile maps without major difficulty. This is especially true if each level fits on a single screen and scrolling isn't required.

The first order of business is to determine the size of the tiles. This will vary depending on the device, but a popular size for many smartphone games is 32×32 pixels. This means that on a Retina iPhone with a resolution of 960×640, the screen would be able to fit 30×20 tiles.

Once the size of the tiles has been settled, the next step is to create the tile set that the game will use. All of the tiles should be placed into a single sprite sheet, if possible, for the gains outlined earlier. The code to render the tile map then needs some way to convert from the unique tile ID into the actual tile image. A simple convention is to have the tile in the top-left corner of the sprite sheet be tile 0, and then continue numbering across the row. The numbering for subsequent rows then continues at the number where the previous row left off.

Once we have a tile set, we can then create level data using these tiles. Although the level data could be a hardcoded 2D array of integers, it will work better if the level data is stored in an external file. The most basic file format would be a text file that lists the tile ID of each tile at each location on the screen. So a level that's 5×5 tiles in size may look like this:

```
// Basic level file format
5,5
0,0,1,0,0
0,1,1,1,0
1,1,2,1,1
0,1,1,1,0
0,0,1,0,0
```

In this file format, the first line is ignored so the user can write a comment. The second line specifies the length and width of the level, and then the raw tile ID data is listed for each row and column. So the first row for this level has two tiles with ID 0, one with ID 1, and then two more with ID 1.

Levels would then be represented by the following class:

```
class Level
    const int tileSize = 32
    int width, height
    int tiles[][]
    function Draw()
end
```

The tiles array stores the tile IDs from the level file, and the width/height corresponds to those numbers in the level file. Loading the actual level file into the class is very language specific, but the file is so basic that parsing it should not prove to be difficult. Once the parsing is implemented, the next step is to implement the level's draw function:

```
function Draw()
    for int row = 0, row < height, row++
       for int col = 0, col < width, col++
          // Draw tiles[row][col] at (col*tileSize, row*tileSize)
       loop
    loop
end
```

Drawing the actual texture is also something that's platform specific, but most 2D frameworks support drawing only a subset of a texture at a location. As long as the tile ID can be converted to the correct sub-image in the tile set, drawing it is relatively straightforward.

One important thing to note is that it may be possible to create several different tile sets that still use the same tile IDs. This is one easy way to implement seasonal or holiday-themed "skins"

for a game. In the winter, it would be possible to change the tile set into something that's covered in snow instead. It would be the same exact level, but it would look more festive.

Although this text-based tile map file format may work for simple levels, in a commercial game a more robust format would be preferable. Chapter 11, "Scripting Languages and Data Formats," discusses other common file formats, some of which may be preferable formats for tile maps.

Isometric Tile Maps

The tile maps we've discussed to this point are displayed as a flat surface. This works well for a top-down or straight-on-the-side view, but if we want a view with a greater sense of depth, we need a different way to look at the scene. In an **isometric view**, the view is rotated slightly so that there is greater depth. Games such as *Diablo* and *Fallout* utilize isometric views, and a sample isometric view is shown in Figure 2.10.

Figure 2.10 Sample isometric scene.

Authoring tiles for an isometric view is a bit different from a standard view. Instead of squares, the tiles must either be diamonds or hexagons. It is also very common to have multiple layers of tiles when isometric tile maps are utilized, and higher layers may have structures constructed out of multiple adjacent tiles. To support multiple layers, we need to update our tile map format

so data can be expressed for multiple layers. Furthermore, the drawing code will need to be updated so that layers are drawn in the appropriate order.

Summary

2D graphics have been around since the very inception of the video game industry, and this chapter discussed many of these core techniques. Although these techniques were developed with the limitations of CRT monitors in mind, most of them have carried over to modern games. Even today, we use double buffering in order to prevent screen tearing. And even though the resolutions of sprites have increased dramatically over the years, they are still the primary visual building block of any 2D games, whether animated or not. Finally, our discussion of scrolling and tile maps should have given you insight into how these slightly more complex systems might be implemented in a modern 2D game.

Review Questions

1. How does an electron gun draw the scene on a CRT? What is VBLANK?
2. What is screen tearing, and how can it be avoided?
3. What are the advantages of using double buffering instead of a single-color buffer?
4. What is the painter's algorithm?
5. What advantage do sprite sheets have over individual sprites?
6. When should the camera's position update in single-direction scrolling?
7. If all background segments can't fit into memory at once in bidirectional scrolling, what data structure should you use to track the segments?
8. What is the difference between a tile map and tile set?
9. For animated sprites, why is it preferable to have the animation FPS as a member variable?
10. How does an isometric view differ from a top-down view?

Additional References

Cocos2d

The most popular 2D library used for iOS games is cocos2d, which is available at www.cocos2d-iphone.org.

Itterheim, Stephen. *Learn Cocos2d 2: Game Development for iOS*. New York: Apress, 2012. There are many books on cocos2d, but this is one of the stronger ones.

SDL

Simple DirectMedia Layer (SDL) is another library choice. This popular cross-platform 2D game library is written in C, but has been ported to many other languages. SDL is available at www.libsdl.org.

CHAPTER 3

LINEAR ALGEBRA FOR GAMES

Linear algebra is a broad category of mathematics with more than enough material for a separate book. But for 3D computer games, only a very specific subset of linear algebra is typically utilized. Rather than covering the overall theories and proofs behind linear algebra, this chapter will present the practical aspects that are most relevant to game programmers.

It cannot be overstated how important vectors and matrices are in the game programmer's toolbox; most programmers in the industry use them every single day.

Vectors

A **vector** represents both a magnitude and a direction in an *n*-dimensional space and has one real number component per dimension. In video games, vectors are usually either 2D or 3D, depending on the number of dimensions the game supports. However, as we will discuss in Chapter 4, "3D Graphics," there are instances where 4D vectors may be used, even though the game itself is only 3D.

A vector can be written out on paper in several different ways, but the approach used in this book is to draw an arrow above a letter to denote that it's a vector. The components of the vector are then referenced by subscript for each dimension. The 3D vector $\vec{v}$ could be represented as follows:

$$\vec{v} = \langle v_x, v_y, v_z \rangle$$

So the vector $\langle 1, 2, 3 \rangle$ would have an x-component of 1, a y-component of 2, and a z-component of 3.

In code, vectors are usually represented as a class with one `float` per dimension:

```
class Vector3
    float x
    float y
    float z
end
```

A vector has *no concept of position*. This means that two vectors are identical as long as they have the same magnitude (or length) and point in the same direction. Figure 3.1 illustrates a vector field that contains several vectors. All of the vectors in this field have the same magnitude and direction, and therefore they are equivalent.

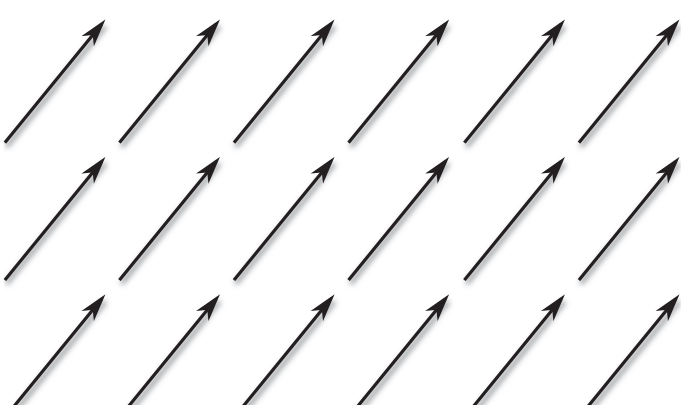


Figure 3.1 Vector field where all the vectors are equivalent.

This equality regardless of where the vectors are drawn is extremely valuable. When solving vector problems, you'll find that it often helps if a vector is drawn at a different location. Because changing where a vector is drawn does not change the vector itself, this is a useful trick to keep in mind.

Even though where we draw a vector doesn't change its value, it simplifies things if a vector is drawn such that the start, or **tail**, of the vector is at the origin (0, 0). The arrow part of the vector (the **head**) can then be thought as "pointing at" a specific position in space, which actually corresponds to the value of the vector. So if the 2D vector ⟨1, 2⟩ is drawn with its tail at the origin, the head will point at the position (1, 2). This is shown in Figure 3.2.

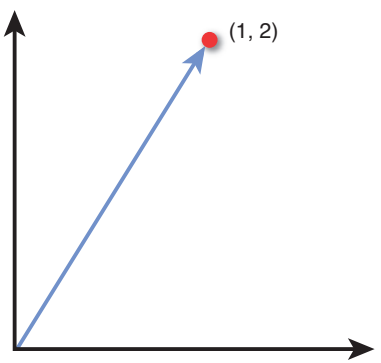


Figure 3.2 The 2D vector ⟨1, 2⟩ drawn with its tail at the origin and its head "pointing at" (1, 2).

Several basic operations can be performed on a vector. It is valuable to know the arithmetic of how to calculate these operations, but it is far more useful to understand the geometric implications of them. That's because this geometric understanding is what will enable vectors to solve problems when you're programming a game.

In the subsequent sections, we cover the most common vector operations. Note that although the diagrams are mostly in 2D, unless otherwise noted all operations work on both 2D and 3D vectors.

Addition

To add two vectors together, each component is individually added to the corresponding component of the other vector:

$$\vec{c} = \vec{a} + \vec{b} = \langle a_x + b_x, a_y + b_y, a_z + b_z \rangle$$

For the geometric representation of addition, draw the vectors such that the head of one vector touches the tail of the other vector. The result of the addition is the vector drawn from the tail of the first vector to the head of the second vector, as shown in Figure 3.3.

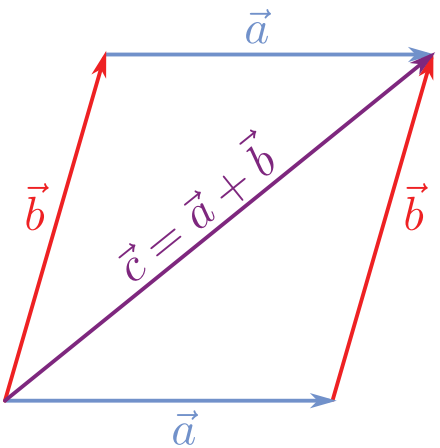


Figure 3.3 Vector addition and the parallelogram rule.

Notice how there are two permutations: one where $\vec{a}$ is the first vector and one where $\vec{b}$ is the first vector. But regardless of the configuration, the result is the same. This is known as the **parallelogram rule**, and it holds because vector addition is commutative, just like addition between two real numbers:

$$\vec{a} + \vec{b} = \vec{b} + \vec{a}$$

Subtraction

In vector subtraction, the components of two vectors are subtracted from each other.

$$\vec{c} = \vec{b} - \vec{a} = \langle b_x - a_x,\ b_y - a_y,\ b_z - a_z \rangle$$

To geometrically subtract two vectors, draw them such that both vectors' tails are at the same position, as in Figure 3.4(a). Then construct a vector from the head of one vector to the head of the other. Because subtraction isn't commutative, the order is significant. The best way to remember this is that if you want the resultant vector to go *from* $\vec{a}$ *to* $\vec{b}$, the calculation is $\vec{b} - \vec{a}$.

Subtraction is extremely important, because it enables you to construct a vector between two points. But if we always draw our vectors such that the tail is at the origin and the head aims at only one point, how can we make a vector *between two points*?

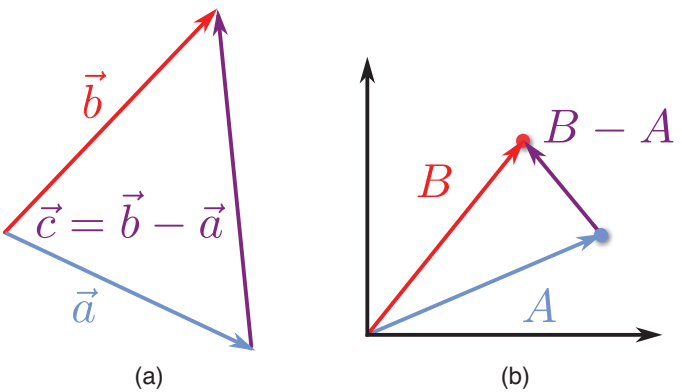


Figure 3.4 Vector subtraction (a), and representing points as vectors from the origin (b).

Well, suppose we have point A at the position (5, 2) and point B at (3, 5). What if we instead represented these points as vectors with their tails at the origin and their heads at the respective points? As we covered earlier, the values of x and y will be identical either way. But if we think of them as vectors, subtraction will allow us to construct a vector between the two, as demonstrated in Figure 3.4(b). The same principle applies as before: *from* A *to* B is $B - A$.

This aspect of vector subtraction is frequently used in games. For instance, imagine you are designing a UI arrow system for a game. The player is periodically given objectives he must travel to, and the game needs to display an arrow that points the player in the proper direction. In this case, the direction in which the arrow needs to face is the vector *from* the player to the objective, which we would calculate with:

```
arrowVector = objective.position - player.position
```

Length, Unit Vectors, and Normalization

As mentioned, vectors represent both a magnitude and a direction. The magnitude (**length**) of a vector is typically denoted by two lines on either side of the vector's symbol, such as $\|\vec{a}\|$. The formula to calculate the length is as follows:

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

This may seem very similar to the distance formula ($d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$), and that's because we are using it! We are calculating the distance from the origin to the position at which the vector is pointing.

Although it is not as much of an issue as in the past, taking a square root is still a relatively expensive operation. If you absolutely need the actual length of a vector for a numerical calculation, there is no way to get around the square root. However, suppose you only want to know whether object A or object B is closer to the player. In this case, you could make a vector from the player to object A as well as a vector from the player to object B. It would be possible to then take the lengths of the vectors and see which one is shorter. But because the length cannot be negative, it is also true that the comparison between the square of the length is logically equivalent:

$$\left\|\vec{a}\right\| < \left\|\vec{b}\right\| \equiv \left\|\vec{a}\right\|^2 < \left\|\vec{b}\right\|^2$$

Therefore, the expense of a square root can be avoided in instances where only a comparison is necessary. If we square both sides of the length equation, we're left with the **length squared**:

$$\left\|\vec{a}\right\|^2 = {a_x}^2 + {a_y}^2 + {a_z}^2$$

Hand in hand with length is the idea of a **unit vector**, or a vector whose length is one. A unit vector is typically represented with a "hat" above the vector's symbol, as in $\hat{u}$. Converting a non-unit vector into a unit vector is known as **normalization**. To normalize a vector, each component must be divided by the length of the vector:

$$\hat{a} = \left\langle \frac{a_x}{\left\|\vec{a}\right\|}, \frac{a_y}{\left\|\vec{a}\right\|}, \frac{a_z}{\left\|\vec{a}\right\|} \right\rangle$$

Certain vector operations discussed in subsequent sections require the vectors to be normalized. However, when a vector is normalized, it will lose any magnitude information. So you have to be careful not to normalize the wrong vectors. A good rule of thumb to follow is if you only care about the direction of the vector (as in the previous UI arrow example), you should normalize the vector. If you care about both the direction and magnitude, the vector should be stored in its un-normalized form.

THE LEGENDARY "JOHN CARMACK INVERSE SQUARE ROOT"

In the 1990s, personal computers did not have processors that could efficiently perform mathematical operations on floating point numbers. On the other hand, operations on integers were far more efficient.

This was a problem in 3D games, because almost all calculations are done with floats instead of integers. This forced developers to come up with clever ways to perform common operations. One such problem area was the square root, and specifically the inverse square root. Why the inverse? When normalizing a vector, you can either divide each

component by a square root or multiply each component by one over the square root. Multiplication was typically more efficient than division, so that's why the inverse was chosen.

When the *Quake III Arena* source code was released, programmers noticed it contained a rather impressive implementation of the inverse square root. Here is the C code, presented with the actual comments (certain colorful language has been censored):

```
float Q_rsqrt(float number)
{
   int i;
   float x2, y;
   const float threehalfs = 1.5F;

   x2 = number * 0.5F;
   y  = number;
   i  = * ( int * ) &y; // evil floating point bit level hacking
   i  = 0x5f3759df - ( i >> 1 ); // what the f***?
   y  = * ( float * ) &i;
   y  = y * ( threehalfs - ( x2 * y * y ) ); // 1st iteration

   return y;
}
```

On first glance, this code may not make much sense. But the way it works is by performing a form of Newtonian approximation. The line with the hexadecimal value is the initial guess, and then the first iteration results in an answer that is reasonably accurate.

What's also interesting is that even though this square root is often attributed to John Carmack, the founder and technical director of id Software, he actually didn't write it. Rys Sommefeldt tried to track down the actual author in this interesting article on Beyond3D: http://www.beyond3d.com/content/articles/8/.

Scalar Multiplication

Vectors can be multiplied by a **scalar**, or a single real number. To multiply a vector by a scalar, you simply multiply each component by the scalar:

$$s \cdot \vec{a} = \left\langle s \cdot a_x,\ s \cdot a_y,\ s \cdot a_z \right\rangle$$

If you multiply a vector by a positive scalar, it will only change the magnitude of the vector. If you multiply the vector by a negative scalar, it will also invert the direction of the vector. Figure 3.5 demonstrates a few different results of scalar multiplication.

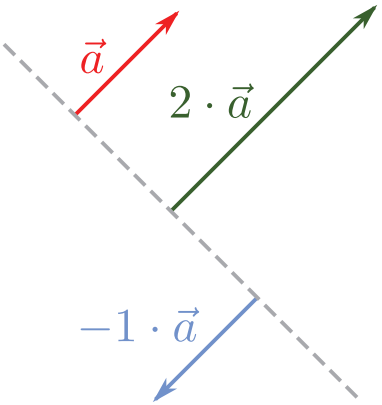


Figure 3.5 Scalar multiplication.

Dot Product

Vectors do not have a single traditional multiplication operator. Instead, they have two different products: the **dot product** and the **cross product**. The dot product results in a scalar, whereas the cross product results in a vector. One of the most common uses of the dot product in games is to find the angle between two vectors.

The dot product calculation is as follows:

$$\vec{a} \cdot \vec{b} = a_x \cdot b_x + a_y \cdot b_y + a_z \cdot b_z$$

But notice how that equation does not have an angle term anywhere in it. In order to find the angle between two vectors, you must use the trigonometric formulation of the dot product:

$$\vec{a} \cdot \vec{b} = \left\|\vec{a}\right\| \left\|\vec{b}\right\| \cos\theta$$

This formulation is based on the Law of Cosines, and a full derivation of it can be found in the Lengyel book in this chapter's references. Once we have this formulation, we can then solve for θ:

$$\theta = \arccos\left(\frac{\vec{a} \cdot \vec{b}}{\left\|\vec{a}\right\| \left\|\vec{b}\right\|}\right)$$

If the vectors $\vec{a}$ and $\vec{b}$ are unit vectors, the division can be omitted because the lengths of both vectors will just be one. This is demonstrated in Figure 3.6(a). The simplification of this

equation is one of the reasons why you should normalize vectors in advance if you only care about their direction.

Because the dot product can be used to calculate an angle between two vectors, a couple of special cases are important to remember. If the dot product between two unit vectors results in 0, it means they are perpendicular to each other because $\cos(90^\circ) = 0$. If the dot product results in 1, it means the vectors are parallel and facing in the same direction, and –1 means they are parallel and face in the opposite direction.

Another important aspect of the dot product is what's known as the **scalar projection**, which is illustrated in Figure 3.6(b). In this application, you have both a unit vector and a non-unit vector. The unit vector is extended such that a right triangle is formed with the other vector. In this case, the dot product will return the length of the extended unit vector.

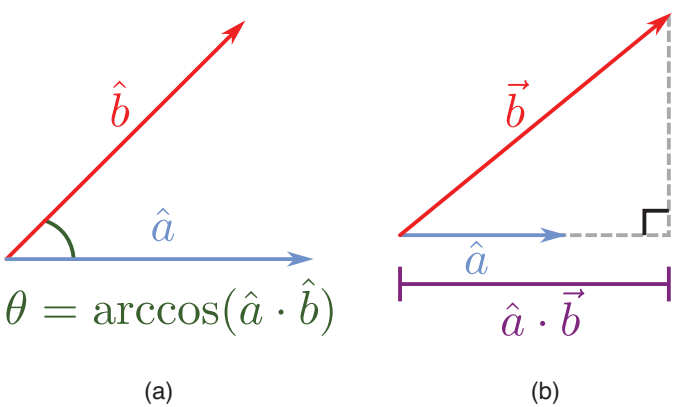


Figure 3.6 Angle between unit vectors (a), and scalar projection (b).

As with multiplication between two real numbers, the dot product is commutative, distributive over addition, and associative:

$$\vec{a} \cdot \vec{b} = \vec{b} \cdot \vec{a}$$
$$\vec{a} \cdot \left(\vec{b} + \vec{c}\right) = \vec{a} \cdot \vec{b} + \vec{a} \cdot \vec{c}$$
$$\vec{a} \cdot \left(\vec{b} \cdot \vec{c}\right) = \left(\vec{a} \cdot \vec{b}\right) \cdot \vec{c}$$

Another useful tip is to also think of the length squared calculation as being equivalent to the dot product of the vector with itself—or in other words:

$$\vec{v} \cdot \vec{v} = \left\|\vec{v}\right\|^2 = {v_x}^2 + {v_y}^2 + {v_z}^2$$

Sample Problem: Vector Reflection

Now that we've covered several different types of vector operations, let's apply them to an actual video game problem. Suppose you have a ball that is travelling toward a wall. When the ball hits the wall, you want it to reflect off of the wall. Now, if the wall is parallel to one of the coordinate axes, this is simple to solve. For example, if the ball bounces off a wall that parallels the x-axis, you can simply negate the velocity in the y-direction.

But negating the x or y velocity does not work if the wall is not parallel to an axis. The only solution to this generalized reflection problem is to use vector math. If we can compute the reflection with vectors, it will work for any arbitrary orientation of the wall.

Vector problems can be difficult to solve without visualizing them on a sheet of paper. Figure 3.7(a) shows the initial state of this problem. There are two known values: the vector $\vec{v}$, which is the velocity of the ball prior to reflection, and the normal $\hat{n}$, which is a unit vector that is perpendicular to the surface of reflection. We need to solve for the vector $\vec{v}'$, which represents the velocity after reflection.

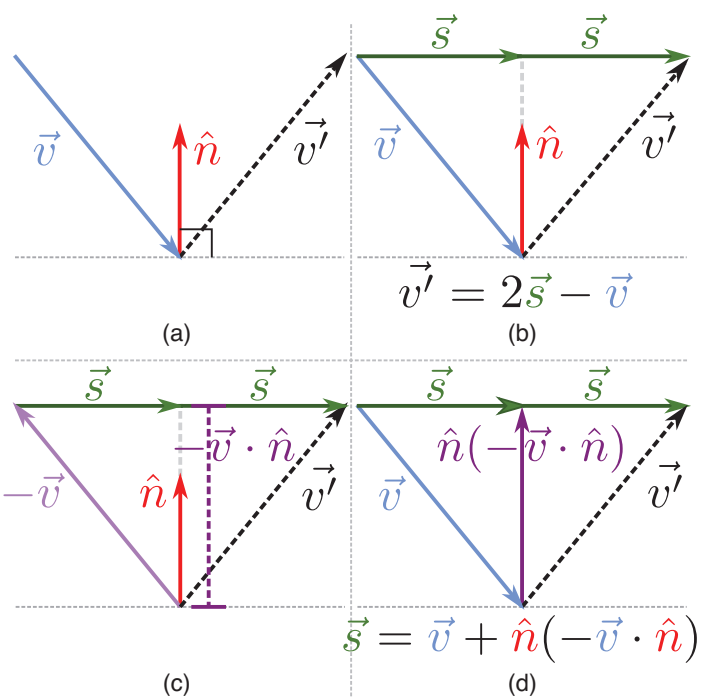


Figure 3.7 Steps to solve vector reflection.

If we had a vector *from* the tail of $\vec{v}$ *to* the head of $\vec{v}'$, it would be possible to calculate $\vec{v}'$ by using vector subtraction. Figure 3.7(b) draws this hypothetical vector as $2\vec{s}$. This means that if we can figure out the value of $\vec{s}$, it will be possible to solve the entire problem.

If there were some way to extend $\hat{n}$ so that its head were drawn at the same location as the head of $\vec{s}$, it would be possible to calculate $\vec{s}$ by vector addition. Figure 3.7(c) shows that if the direction of $\vec{v}$ is reversed, we can use scalar projection to determine the distance of the extended $\hat{n}$. Because $\hat{n}$ is a unit vector, if we scalar multiply it by this computed distance, we will get our extended $\hat{n}$. Figure 3.7(d) shows this extended vector, which can then be used to calculate the value of $\vec{s}$.

Now that we have the value of $\vec{s}$, we can substitute back into the equation from Figure 3.6(b) and perform some simple algebra to get the final solution for $\vec{v}'$:

$$\begin{aligned}
\vec{v}' &= 2\vec{s} - \vec{v} \\
\vec{s} &= \vec{v} + \hat{n}\left(-\vec{v} \cdot \hat{n}\right) \\
\vec{v}' &= 2\left(\vec{v} + \hat{n}\left(-\vec{v} \cdot \hat{n}\right)\right) - \vec{v} \\
\vec{v}' &= 2\vec{v} + 2\hat{n}\left(-\vec{v} \cdot \hat{n}\right) - \vec{v} \\
\vec{v}' &= \vec{v} - 2\hat{n}\left(\vec{v} \cdot \hat{n}\right)
\end{aligned}$$

When you're solving vector problems, it is useful to make sure you ultimately are performing valid vector operations. For example, if a solution to a vector problem suggested that a vector should be added to a scalar, it would have to mean the solution is wrong. This is much like checking to make sure the units line up in physics.

So let's look at our solution to the vector reflection problem. In order to do vector subtraction with $\vec{v}$, the result of $2\hat{n}\left(\vec{v} \cdot \hat{n}\right)$ must be a vector. If we look at this expression more closely, we first scalar multiply $\hat{n}$ by 2, which results in a vector. We then scalar multiply this vector by the scalar result of a dot product. So, ultimately, we are subtracting two vectors, which certainly is valid.

This vector reflection problem is a great example of how to use the vector operations we've covered thus far. And it also turns out that it's an extraordinarily common interview question posed to potential game programmers.

Cross Product

The **cross product** between two vectors results in a third vector. Given two vectors, there is only a single plane that contains both vectors. The cross product finds a vector that is perpendicular to this plane, as in Figure 3.8(a), which is known as a **normal** to the plane.

Because it returns a normal to a plane, the cross product only works with 3D vectors. This means that in order to take a cross product between two 2D vectors, they first must be converted to 3D vectors via the addition of a z-component with a value of 0.

The cross product is written as follows:

$$\vec{c} = \vec{a} \times \vec{b}$$

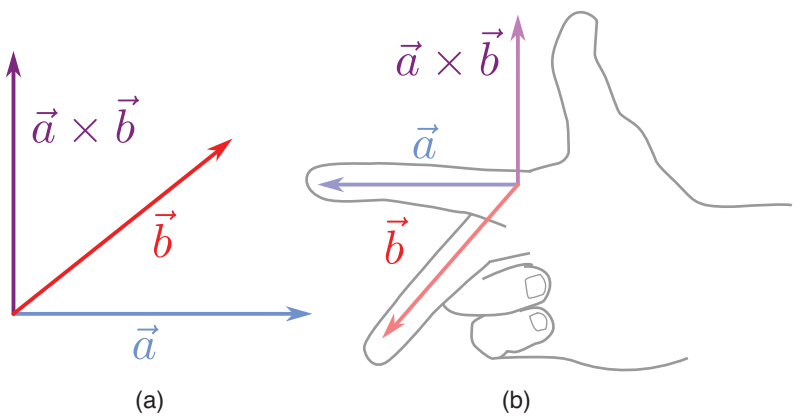


Figure 3.8 Cross product (a), and right-hand rule (b).

An important thing to note is that there is technically a second vector that is perpendicular to the plane: the vector that points in the opposite direction of $\vec{c}$. So how do you know which direction the cross product result should face? The answer is tied into what is known as the **handedness** of the coordinate system. In a right-handed system, you take your right hand and create a 90° angle between your thumb and index finger. You likewise make a 90° angle between your index finger and your middle finger. You then line up your index finger with $\vec{a}$ and your middle finger with $\vec{b}$. The direction your thumb points in is the direction the cross product will face. Figure 3.8(b) shows an application of the **right-hand rule**.

You may have noticed that if you switch it so your index finger lines up with $\vec{b}$ and your middle finger with $\vec{a}$, your thumb points in the opposite direction. This is because the cross product is not commutative, but in fact *anti*-commutative:

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

It is important to remember that *not all games use a right-handed coordinate system*. We will describe coordinate systems in greater detail later, but for now let's assume a right-handed coordinate system.

Now that the cross product has been covered conceptually, let's look at how it is calculated:

$$\vec{c} = \vec{a} \times \vec{b} = \left\langle a_y b_z - a_z b_y,\ a_z b_x - a_x b_z,\ a_x b_y - a_y b_x \right\rangle$$

Remembering how to calculate the cross product just from the preceding formula is difficult. Luckily, there is a mnemonic to help us remember: "xyzzy." The way the mnemonic works is it tells us the order of the subscripts for the first component of the cross product:

$$c_x = a_y b_z - a_z b_y$$

If you can remember the base equation is $c = a \cdot b - a \cdot b$, "xyzzy" tells you which subscript to use for each term. Once you have the first line, you can figure out c_y and c_z by rotating the subscripts in the following manner: $x \rightarrow y \rightarrow z \rightarrow x$. This will then give you the next two lines of the cross product:

$$c_y = a_z b_x - a_x b_z$$
$$c_z = a_x b_y - a_y b_x$$

As with the dot product, there is a special case to watch out for. If the cross product returns a vector where all three components are 0, this means that the two input vectors are **collinear**, or they lie on the same line. Two collinear vectors cannot form a plane, and because of that there is no normal for the cross product to return.

Because a triangle is guaranteed to always be on a single plane, we can use the cross product to determine the vector that's perpendicular to the triangle. This perpendicular vector is known as the **normal** of the triangle. Figure 3.9 shows triangle ABC. In order to calculate the normal, we first must construct vectors from A to B and from A to C. Then if we take the cross product between these two vectors, in a right-handed coordinate system we would get a vector going into the page, which corresponds to the direction of the normal.

Because we only care about the direction of the normal, and not the magnitude, we should make sure to always normalize this vector. This concept of a normal does not apply only to triangles; any polygon has a single normal as long as it lies on a single plane.

Note that we could also take the cross product in the opposite order, in which case we would still get a vector perpendicular to the triangle's plane, although it will be facing in the opposite direction. Chapter 4 discusses the concept of winding order, which determines the order in which we should perform the cross product.

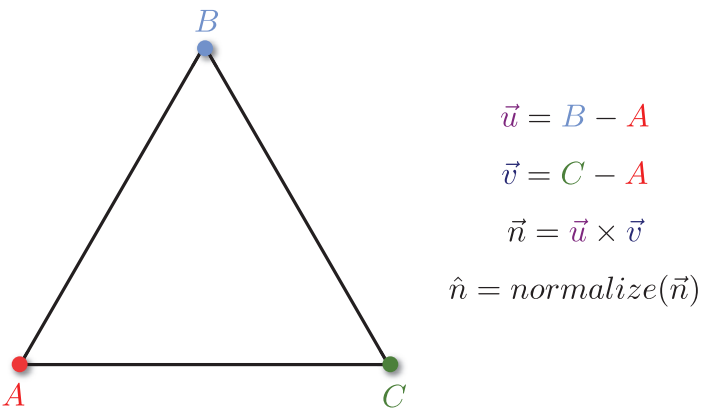


Figure 3.9 The normal going into the page for triangle ABC (in a right-handed coordinate system).

Sample Problem: Rotating a 2D Character

Let's take a look at another sample problem. Suppose we have a top-down action game where a player character moves through the world. We are scripting an in-game cutscene where an explosion occurs at a specific location, and we want the player to turn and face this explosion. However, the cutscene can trigger in a variety of scenarios, so the initial direction the player is facing is variable. We also do not want the player to suddenly snap to face the explosion; rather, it should be a smooth transition over a brief period of time.

We need to determine two different things in order to perform a smooth transition. First, we need the angle between the current facing vector and the new facing vector of the player. Once we have that, we then need to figure out whether the character should rotate clockwise or counterclockwise. Figure 3.10 presents a visualization of this problem.

The vector c_y represents the current direction the player is facing. P is the position of the player in the world, and E is the position of the explosion. The first step to solve this problem is to construct a vector that points in the direction of the explosion. We want the vector *from* P *to* E, so the calculation should be $E - P$. Because we only care about the direction in this problem, and not the distance, we should go ahead and normalize this vector. Therefore, we can define $\hat{n}$ as the unit vector that points in the direction of the explosion.

Now that we have the current and new directions, how do we find the angle between the two vectors? The answer is the dot product. Because both $\hat{c}$ and $\hat{n}$ are unit vectors, the angle θ can be calculated with $\arccos(\hat{c} \cdot \hat{n})$.

But the dot product can't tell us whether it's θ in the clockwise or counterclockwise direction. To determine this, we must use the cross product. Because the cross product only works on 3D vectors, $\hat{c}$ and $\hat{n}$ must be converted to 3D vectors by adding a z-component of 0:

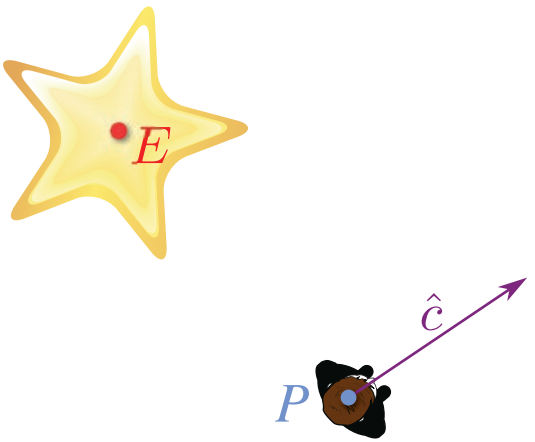


Figure 3.10 The player needs to rotate to face the explosion.

$$\hat{c} = \langle c_x,\ c_y,\ 0 \rangle$$
$$\hat{n} = \langle n_x,\ n_y,\ 0 \rangle$$

Once the vectors are 3D, we can calculate $\hat{c} \times \hat{n}$. If the z-component of the resultant vector is positive, it means that the rotation is counterclockwise. On the other hand, a negative z-component means the rotation is clockwise. The way to visualize this is by using the right-hand rule. If you go back to Figure 3.10 and position your index finger along $\hat{c}$ and your middle finger along $\hat{c}$, your thumb should point out of the page. In a right-handed coordinate system, this is a positive z value, which is a counterclockwise rotation.

Linear Interpolation

Linear interpolation, or **lerp**, calculates a value somewhere linearly between two other values. For example, if $a = 0$ and $b = 10$, a linear interpolation of 20% *from* a *to* b is 2. Lerp is not limited to only real numbers; it can be applied to values in any number of dimensions. It is possible with points, vectors, matrices, quaternions, and many other types of values. Regardless of the number of dimensions, lerp can be applied with the same generic formula:

$$Lerp(a, b, f) = (1 - f) \cdot a + f \cdot b$$

Here, a and b are the values being interpolated between and f is a fractional value in the range of [0,1] *from* a *to* b.

In games, a common application of lerp is to find a point between two other points. Suppose there is a character at point a and it needs to be smoothly moved to b over time. Lerp will allow you to slowly increase f until it hits 1, at which point the character will be at point b (see Figure 3.11).

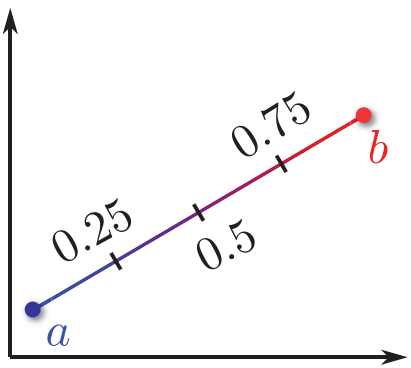


Figure 3.11 Lerp from *a* to *b* with different *f* values.

Coordinate Systems

Not all games use the same 3D coordinate system. Some coordinate systems may have the y-axis as up, whereas others may use the z-axis. Some might be right-handed coordinate systems while others might be left-handed. Selection of a coordinate system is arbitrary and often comes down to personal preference. Most games seem to prefer systems where the y-axis is up and the z-axis goes into or out of the screen, but there is no mathematical basis for this preference.

Because there is no one set way to express a 3D system, whenever you're working with a new framework it is important to check which coordinate system it uses. It ultimately doesn't matter which coordinate system a game uses, as long as it's consistent across the codebase (see the sidebar "A Coordinate System Mix-Up" for a humorous instance where this wasn't the case).

Sometimes it may be a requirement to convert from one coordinate system to another. A common occurrence of this is when a 3D modeling program uses a different coordinate system than the game itself. Converting from one coordinate system to another may require negating a component or swapping components. It just depends on what the two coordinate systems are. For example, Figure 3.12 shows two y-up coordinate systems: a right-handed one and a left-handed one. In this particular case, the conversion is relatively simple: To switch from one to the other, just negate the z-component.

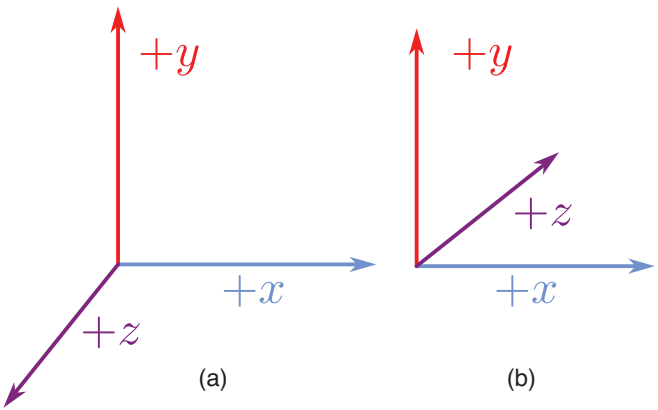


Figure 3.12 Sample right-handed (a) and left-handed (b) coordinate systems.

A COORDINATE SYSTEM MIX-UP

On *Lord of the Rings: Conquest*, one of my primary responsibilities was to provide programming support for the sound team. Early on in the project, the lead sound designer came to me with a peculiar issue. He noticed that when an explosion occurred on the left side of the screen, it came out of the right speaker, and vice versa. He had checked his speaker wires several times, and was sure that he had the correct speakers connected to the correct inputs on the receiver.

A bit of digging around uncovered the issue: Most of our game's code used a left-handed coordinate system, whereas our third-party sound library used a right-handed coordinate system. This resulted in the wrong coordinates being passed into the sound system. The solution was to convert the coordinates to right-handed when passing them into the sound library. Once this was changed, the sounds started coming out of the correct speakers for the first time on the project.

By default, DirectX uses a left-handed y-up coordinate system, and OpenGL uses a right-handed y-up coordinate system. However, this does not mean that *all* games using these libraries must use the default coordinate systems because it's relatively straightforward to configure both to use a different coordinate system.

Sometimes the x-, y-, and z-axes are instead referred to as the $\hat{i}$, $\hat{j}$, and $\hat{k}$ **basis vectors**. The basis vectors are what formally define the coordinate system. As we will discuss in Chapter 4, the generic way to change from any arbitrary coordinate system to another is by using a **change of basis matrix**. But before we can construct one of these, we first must cover matrices.

Matrices

A **matrix** is a grid of real numbers, with *m* rows and *n* columns, and is typically expressed by a capital letter. For games, 3×3 and 4×4 matrices are the most common. A 3×3 matrix could be written as such, where the letters *a* through *i* simply represent numbers within the matrix:

$$A = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix}$$

Another method for matrix declaration is to use subscript coordinates to refer to the specific elements. The preceding matrix could be rewritten as follows:

$$A = \begin{bmatrix} A_{1,1} & A_{1,2} & A_{1,3} \\ A_{2,1} & A_{2,2} & A_{2,3} \\ A_{3,1} & A_{3,2} & A_{3,3} \end{bmatrix}$$

In 3D games, matrices are used to **transform**, or change, vectors and positions. We will cover these types of transformations in Chapter 4. For now, let's cover the basic matrix operations.

Addition/Subtraction

Two matrices can be added or subtracted if they both have the same dimensions. To add them, simply add the corresponding components to each other.

$$\begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} + \begin{bmatrix} j & k & l \\ m & n & o \\ p & q & r \end{bmatrix} = \begin{bmatrix} a+j & b+k & c+l \\ d+m & e+n & f+o \\ g+p & h+q & i+r \end{bmatrix}$$

Subtraction is the same, except the components are subtracted. Both of these operations rarely see use in games.

Scalar Multiplication

As can be done with vectors, it is possible to multiply a matrix by a scalar.

$$s \cdot \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} = \begin{bmatrix} s \cdot a & s \cdot b & s \cdot c \\ s \cdot d & s \cdot e & s \cdot f \\ s \cdot g & s \cdot h & s \cdot i \end{bmatrix}$$

Multiplication

Matrix multiplication is an operation that sees widespread use in games. The easiest way to visualize matrix multiplication between two matrices with the same dimensions is to think in terms of vectors and dot products. Suppose you have matrices A and B, as shown here:

$$A = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \quad B = \begin{bmatrix} e & f \\ g & h \end{bmatrix}$$

To calculate $C = A \times B$, take the first row of A and dot product it with the first column of B. The result of this dot product is the top-left value in C. Then dot product the first row of A with the second column of B, which gives the top-right value in C. Repeat this process for the second row of A to complete the multiplication.

$$C = A \times B = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \times \begin{bmatrix} e & f \\ g & h \end{bmatrix} = \begin{bmatrix} a \cdot e + b \cdot g & a \cdot f + b \cdot h \\ c \cdot e + d \cdot g & c \cdot f + d \cdot h \end{bmatrix}$$

Matrix multiplication does not require matrices to have identical dimensions, but it does require that the number of columns in A be equal to the number of rows in B. For instance, the following multiplication is also valid:

$$\begin{bmatrix} a & b \end{bmatrix} \times \begin{bmatrix} c & d \\ e & f \end{bmatrix} = \begin{bmatrix} a \cdot c + b \cdot e & a \cdot d + b \cdot f \end{bmatrix}$$

Multiplying two 4×4 matrices follows the same principle as smaller ones; there are just more elements to calculate:

$$\begin{bmatrix} C_{1,1} & C_{1,2} & C_{1,3} & C_{1,4} \\ C_{2,1} & C_{2,2} & C_{2,3} & C_{2,4} \\ C_{3,1} & C_{3,2} & C_{3,3} & C_{3,4} \\ C_{4,1} & C_{4,2} & C_{4,3} & C_{4,4} \end{bmatrix} = \begin{bmatrix} A_{1,1} & A_{1,2} & A_{1,3} & A_{1,4} \\ A_{2,1} & A_{2,2} & A_{2,3} & A_{2,4} \\ A_{3,1} & A_{3,2} & A_{3,3} & A_{3,4} \\ A_{4,1} & A_{4,2} & A_{4,3} & A_{4,4} \end{bmatrix} \times \begin{bmatrix} B_{1,1} & B_{1,2} & B_{1,3} & B_{1,4} \\ B_{2,1} & B_{2,2} & B_{2,3} & B_{2,4} \\ B_{3,1} & B_{3,2} & B_{3,3} & B_{3,4} \\ B_{4,1} & B_{4,2} & B_{4,3} & B_{4,4} \end{bmatrix}$$

$$C_{1,1} = A_{1,1} \cdot B_{1,1} + A_{1,2} \cdot B_{2,1} + A_{1,3} \cdot B_{3,1} + A_{1,4} \cdot B_{4,1}$$
$$\ldots$$

It is important to note that matrix multiplication is *not* commutative, though it is associative and distributive over addition:

$$A \times B \neq B \times A$$
$$A \times (B + C) = A \times B + A \times C$$
$$A \times (B \times C) = (A \times B) \times C$$

Another important property of matrix multiplication is the **identity matrix**. If you multiply a scalar value by 1, it does not change. Similarly, multiplying a matrix by the identity matrix I does not change the matrix's value.

$$A = A \times I$$

The identity matrix has a number of rows and columns equal to the number of columns in A, and values within the identity matrix are all zeroes except for a diagonal of ones. Therefore, a 3×3 identity matrix looks like this:

$$I_3 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Inverse

If a matrix is multiplied by its inverse, denoted by $^{-1}$, the result is the identity matrix:

$$I = A \times A^{-1}$$

Calculation of the inverse of an arbitrary matrix is a somewhat advanced topic, so it is not covered in this book. However, it is important to know that not all matrices have an inverse, but in games a matrix with no inverse typically means a calculation has gone awry. Thankfully, the transformation matrices covered in Chapter 4 have inverses that are relatively straightforward to compute.

Transpose

A **transpose** of a matrix, denoted by ^{T}, takes each row of a matrix and converts it to a corresponding column. So the first row becomes the first column, the second row becomes the second column, and so on. Here is the transpose of a 3×3 matrix:

$$\begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix}^{T} = \begin{bmatrix} a & d & g \\ b & e & h \\ c & f & i \end{bmatrix}$$

For certain types of matrices known as **orthonormal** matrices, the inverse of the matrix is its transpose. Rotation matrices are one example of this type of matrix.

Transforming 3D Vectors by Matrices

In order to apply a matrix transformation to a vector, the two must be multiplied together. But before multiplication can occur, the vector must first be represented as a matrix. It turns out

there are two potential ways to represent a vector as a matrix: It could be a matrix with a single row or a matrix with a single column. These representations are referred to as **row-major** and **column-major**, respectively.

Given the vector $\vec{v} = \langle 1,\ 2,\ 3 \rangle$, the row-major matrix representation would be this:

$$\vec{v} = \begin{bmatrix} 1 & 2 & 3 \end{bmatrix}$$

Whereas the column-major representation would be this:

$$\vec{v} = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$$

If you have a matrix that is intended to be multiplied with row-major vectors, but instead want to multiply it by a column-major one, the matrix must be transposed. Take a look at how a 3D vector is multiplied by a 3×3 matrix first in row-major form and then in column-major form:

$$\begin{bmatrix} x' & y' & z' \end{bmatrix} = \begin{bmatrix} x & y & z \end{bmatrix} \times \begin{bmatrix} a & d & g \\ b & e & h \\ c & f & i \end{bmatrix}$$

$$x' = x \cdot a + y \cdot b + z \cdot c$$

$$y' = x \cdot d + y \cdot e + z \cdot f$$

...

$$\begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} \times \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

$$x' = a \cdot x + b \cdot y + c \cdot z$$

...

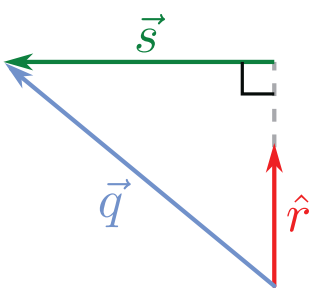
Notice how the value of x' is identical in both representations. But this is only the case because the 3×3 matrix was transposed. As with handedness, it doesn't matter whether row-major or column-major is selected, as long as it's consistent. Because most 3D game libraries utilize a row-major representation (with OpenGL being a major exception), that's what is used in this book.

Summary

Although the topic of linear algebra may not be the most exciting to prospective game programmers, the value of it is indisputable. A strong grasp of linear algebra is often the difference between someone who *wants* to program 3D video games and someone who *does* program 3D video games. We will be using both vectors and matrices at many other points throughout this book, so it is extraordinarily important that these concepts are fully digested before moving on.

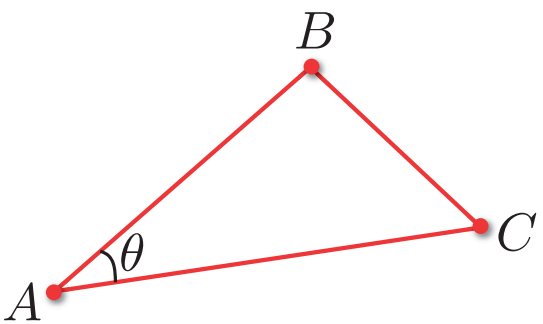
Review Questions

1. Given the vectors $\vec{a} = \langle 2,\ 4,\ 6 \rangle$ and $\vec{b} = \langle 3,\ 5,\ 7 \rangle$ and the scalar value $s = 2$, calculate the following:
   a. $\vec{a} + \vec{b}$
   b. $s \cdot \vec{a}$
   c. $\vec{a} \times \vec{b}$
2. What is a good rule of thumb for whether or not a particular vector should be normalized?
3. Given the position of the player, P, how do you *efficiently* determine whether it is closer to point A or point B?
4. Given the vectors $\vec{q}$ and $\hat{r}$ in the following diagram, solve for $\vec{s}$ using vector operations.



5. Given the following triangle, calculate θ using vectors:

$$A = \langle -1,\ 2,\ 2 \rangle$$
$$B = \langle 1,\ 2,\ 3 \rangle$$
$$C = \langle 2,\ 4,\ 3 \rangle$$



6. Using the previous triangle, calculate the surface normal *coming out of the page* in a right-handed coordinate system.
7. If the order of the cross product is reversed, what happens to the result?
8. You are tasked with implementing simple stereo sound in a top-down action game. When a sound plays in the world, you must determine whether to play the sound out of the left or right speaker based on the orientation of the sound relative to the player. How could this be solved using vector math?
9. Calculate the result of the following matrix multiplication:

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \times \begin{bmatrix} 2 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 2 \end{bmatrix}$$

10. When is the transpose of a matrix equal to its inverse?

Additional References

Lengyel, Eric. *Mathematics for 3D Game Programming and Computer Graphics (Third Edition)*. Boston: Course Technology, 2012. This book contains detailed explanations of the concepts covered in this chapter, complete with derivations and proofs. It also covers more complex math, which is beyond the scope of this book, but still gets some use by game programmers (especially those who focus on graphics).

This page intentionally left blank

CHAPTER 4

3D GRAPHICS

Although the first 3D games were released in the arcade era, it wasn't really until the mid-to-late 1990s that 3D took off. Today, nearly all AAA titles on consoles and PCs are 3D games. And as the power of smartphones increases, more and more mobile games are as well.

The primary technical challenge of a 3D game is displaying the game world's 3D environment on a flat 2D screen. This chapter discusses the building blocks necessary to accomplish this.

Basics

The first 3D games needed to implement all their rendering in software. This meant that even something as basic as drawing a line had to be implemented by a graphics programmer. The set of algorithms necessary to draw 3D objects into a 2D color buffer are collectively known as **software rasterization**, and most courses in computer graphics spend some amount of time discussing this aspect of 3D graphics. But modern computers have dedicated graphics hardware, known as the **graphics processing unit** (GPU), that knows how to draw basic building blocks such as points, lines, and triangles.

Because of this, modern games do not need to implement software rasterization algorithms. The focus is instead on giving the graphics card the data it needs to render the 3D scene in the desired manner, using libraries such as OpenGL and DirectX. And if further customization is necessary, custom micro-programs, known as **shaders**, can be applied to this data as well. But once this data is put together, the graphics card will take it and draw it on the screen for you. The days of coding out Bresenham's line-drawing algorithm are thankfully gone.

One thing to note is that in 3D graphics, it's often necessary to use approximations. This is because there simply isn't enough time to compute photorealistic light. Games are not like CG films, where hours can be spent to compute one single frame. A game needs to be drawn 30 or 60 times per second, so accuracy must be compromised in favor of performance. A graphical error that is the result of an approximation is known as a **graphical artifact**, and no game can avoid artifacts entirely.

Polygons

3D objects can be represented in a computer program in several ways, but the vast majority of games choose to represent their 3D objects with polygons—specifically, triangles.

Why triangles? First of all, they are the simplest polygon possible because they can be represented by only three **vertices**, or the points on the corners of the polygon. Second, triangles are guaranteed to lie on a single plane, whereas a polygon with four or more vertices may lie on multiple planes. Finally, arbitrary 3D objects easily tessellate into triangles without leaving holes or other deformations.

A single model, or **mesh**, is made up of a large number of triangles. The number of triangles a particular mesh has depends on the platform constraints as well as the particular use of the model. For an Xbox 360 game, a character model may have 15,000 polygons, whereas a barrel might be represented by a couple hundred. Figure 4.1 shows a mesh represented with triangles.

All graphics libraries have some way to specify the lists of triangles we want the graphics card to render. We then must also provide further information regarding how we want these list of triangles to be drawn, which is what the rest of this chapter covers.

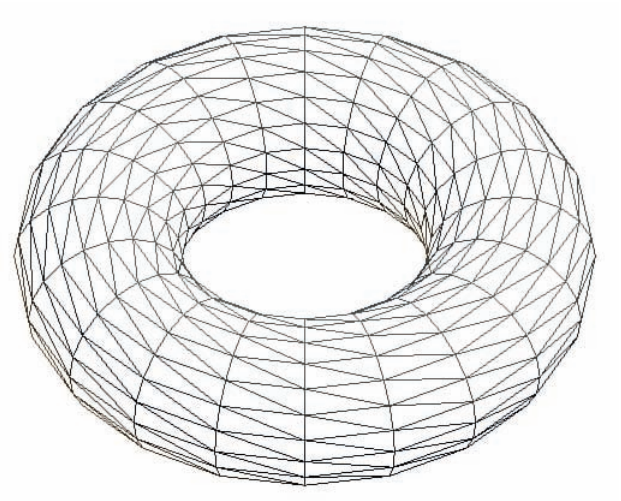


Figure 4.1 A sample mesh made with triangles.

Coordinate Spaces

A **coordinate space** provides a frame of reference to a particular scene. For example, in the Cartesian coordinate space, the origin is known to be in the center of the world, and positions are relative to that center. In a similar manner, there can be other coordinate spaces relative to different origins. In the 3D rendering pipeline, we must travel through four primary coordinate spaces in order to display a 3D model on a 2D monitor:

- Model
- World
- View/camera
- Projection

Model Space

When a model is created, whether procedurally or in a program such as Maya, all the vertices of the model are expressed relative to the origin of that model. **Model space** is this coordinate space that's relative to the model itself. In model space, the origin quite often is in the center of the model—or in the case of a humanoid, between its feet. That's because it makes it easier to manipulate the object if its origin is centered.

Now suppose there are 100 different objects in a particular game's level. If the game loaded up the models and just drew them at their model space coordinates, what would happen? Well, because every model was created at the origin in model space, every single object, including

the player, would be at the origin too! That's not going to be a particularly interesting level. In order for this level to load properly, we need another coordinate system.

World Space

This new coordinate system is known as **world space**. In world space, there is an origin for the world as a whole, and every object has a position and orientation relative to that world origin.

When a model is loaded into memory, all its vertices are expressed in terms of model space. We don't want to manually change that vertex data in memory, because it would result in poor performance and would prevent positioning multiple instances of the model at several locations in the world.

What we want instead is a way to tell the graphics card where we want the particular model to be drawn. And it's not just location: some models may need to be rotated differently, some may need to be scaled, and so on (as demonstrated in Figure 4.2). So the desired solution would be to pass the model data (in model space) to the graphics card as well as the little bit of additional data needed place the model in the world.

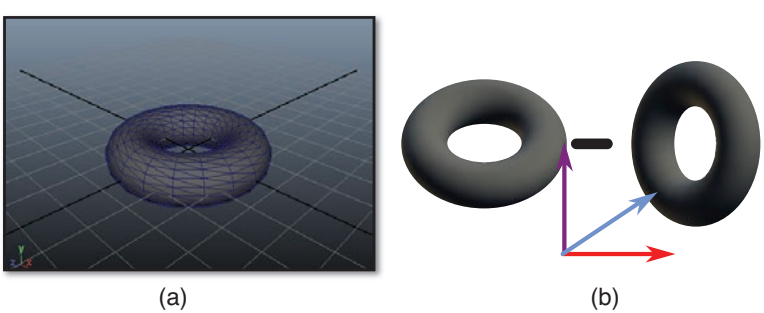


Figure 4.2 Object in model space (a), and several instances of the object positioned/oriented relative to world space (b).

It turns out that for that extra bit of data, we are going to use our trusty matrices. But first, before we can use matrices we need a slightly different way to represent the positions of the triangles, called homogenous coordinates.

Homogenous Coordinates

As mentioned earlier, sometimes 3D games will actually use 4D vectors. When 4D coordinates are used for a 3D space, they are known as **homogenous coordinates**, and the fourth component is known as the **w-component**.

In most instances, the w-component will be either 0 or 1. If $w = 0$, this means that the homogenous coordinate represents a 3D vector. On the other hand, if $w = 1$, this means that the

coordinate represents a 3D point. However, what's confusing is that in code you will typically see a class such as `Vector4` used both for vectors or points. Due to this, it is very important that the vectors are named in a sensible manner, such as this:

```
Vector4 playerPosition // This is a point
Vector4 playerFacing // This is a vector
```

Transforming 4D Vectors by Matrices

The matrices we use to transform objects will often be 4×4 matrices. But in order to multiply a vector by a 4×4 matrix, the vector has to be 4D, as well. This is where homogenous coordinates come into play.

Multiplying a 4D vector by a 4×4 matrix is very similar to multiplying a 3D vector by a 3×3 matrix:

$$\begin{bmatrix} x' & y' & z' & w' \end{bmatrix} = \begin{bmatrix} x & y & z & w \end{bmatrix} \times \begin{bmatrix} a & e & i & m \\ b & f & j & n \\ c & g & k & o \\ d & h & l & p \end{bmatrix}$$

$$x' = x \cdot a + y \cdot b + z \cdot c + w \cdot d$$
$$y' = x \cdot e + y \cdot f + z \cdot g + w \cdot h$$
$$\ldots$$

Once you understand how to transform a 4D homogenous coordinate by a matrix, you can then construct matrices in order to perform the specific transformations you want in world space.

Transform Matrices

A **transform matrix** is a matrix that is able to modify a vector or point in a specific way. Transform matrices are what allow us to have model space coordinates transformed into world space coordinates.

As a reminder, all of these transform matrices are presented in row-major representations. If you are using a library that expects a column-major representation (such as OpenGL), you will need to transpose these matrices. Although the matrices that follow are presented in their 4×4 forms, the scale and rotation matrices can have their fourth row/column removed if a 3×3 matrix is desired instead.

Scale

Suppose we have a player character model that was created to be a certain size. In game, there is a power-up that can double the size of the character. This means we need to be able to increase the size of the model in world space. This could be accomplished by multiplying every vertex in the model by an appropriate **scale matrix**, which can scale a vector or position

along any of the three coordinate axes. It looks a lot like the identity matrix, except the diagonal comprises the scale factors.

$$s(s_x, s_y, s_z) = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

If $s_x = s_y = s_z$, this is a **uniform scale**, which means the vector is transformed equally along all axes. So if all the scale values are 2, the model will double in size.

To invert a scale matrix, simply take the reciprocal of each scale factor.

Translation

A **translation matrix** moves a point by a set amount. It does not work on vectors, because the value of a vector does not depend on where we draw the vector. The translation matrix fills in the final row of the identity matrix with the x, y, and z translation values.

$$T(t_x, t_y, t_z) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ t_x & t_y & t_z & 1 \end{bmatrix}$$

For example, suppose the center of a model is to be placed at the position $\langle 5,\ 10,\ -15 \rangle$ in the world. The matrix for this translation would be as follows:

$$T(5,\ 10,\ -15) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 5 & 10 & -15 & 1 \end{bmatrix}$$

This matrix will have no effect on a vector because the homogenous representation of a vector has a w-component of 0. This means that the fourth row of the translation matrix would be cancelled out when the vector and matrix are multiplied. But for points, the w-component is typically 1, which means the translation components will be added in.

To invert a translation matrix, negate each of the translation components.

Rotation

A **rotation matrix** enables you to rotate a vector or position about a coordinate axis. There are three different rotation matrices, one for each Cartesian coordinate axis. So a rotation about the x-axis has a different matrix than a rotation about the y-axis. In all cases, the angle θ is passed into the matrix.

These rotations are known as **Euler rotations**, named after the Swiss mathematician.

$$RotateX(\theta) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$RotateY(\theta) = \begin{bmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$RotateZ(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

As an example, here is a matrix that rotates 45° about the y axis:

$$RotateY(45^\circ) = \begin{bmatrix} \cos 45^\circ & 0 & \sin 45^\circ & 0 \\ 0 & 1 & 0 & 0 \\ -\sin 45^\circ & 0 & \cos 45^\circ & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

It can be somewhat daunting to remember these different rotation matrices. One trick that helps is to assign a number to each coordinate axis: $x = 1$, $y = 2$, and $z = 3$. This is because for the rotation about the x-axis, the element at (1,1) is 1, whereas everything else in the first row/column is 0. Similarly, the y and z rotations have these values in the second and third row/column, respectively. The last row/column is always the same for all three matrices. Once you remember the location of the ones and zeros, the only thing left is to memorize the order of sine and cosine.

Rotation matrices are orthonormal, which means that the inverse of a rotation matrix is the transpose of that matrix.

Applying Multiple Transforms

It is often necessary to combine multiple transformations in order to get the desired world transform matrix. For example, if we want to position a character at a specific coordinate and have that character scale up by a factor of two, we need both a scale matrix and a translation matrix.

To combine the matrices, you simply multiply them together. The order in which you multiply them is extremely important, because matrix multiplication is not commutative. Generally, for the row-major representation, you want the world transform to be constructed as follows:

$$WorldTransform = Scale \times Rotation \times Translation$$

The rotation should be applied first because rotation matrices are about the origin. If you rotate first and then translate, the object will rotate about itself. However, if the translation is performed first, the object does not rotate about its own origin but instead about the world's origin, as in Figure 4.3. This may be the desired behavior in some specific cases, but typically this is not the case for the world transform matrix.

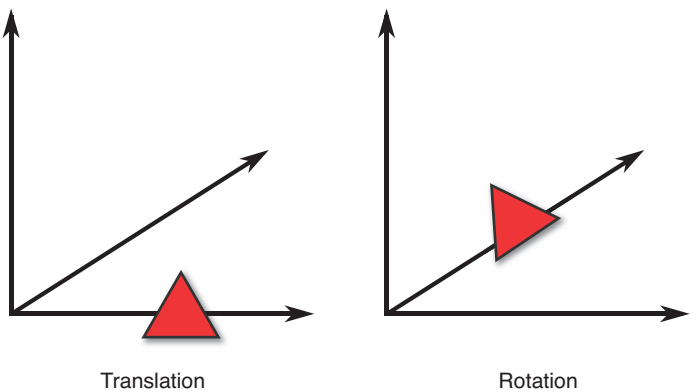


Figure 4.3 An object translated and then rotated about the world's origin instead of about its own origin.

View/Camera Space

Once all the models are positioned and oriented in the world, the next thing to consider is the location of the camera. A scene or level could be entirely static, but if the position of the camera changes, it changes what is ultimately displayed onscreen. This is known as **view** or **camera space** and is demonstrated in Figure 4.4.

So we need another matrix that tells the graphics card how to transform the models from world space into a coordinate space that is relative to the camera. The matrix most commonly used for this is a **look-at matrix**. In a look-at matrix, the position of the camera is expressed in addition to the three coordinate axes of the camera.

For a row-major left-handed coordinate system, the look-at matrix is expressed as

$$Look\text{-}At = \begin{bmatrix} L_x & U_x & F_x & 0 \\ L_y & U_y & F_y & 0 \\ L_z & U_z & F_z & 0 \\ T_x & T_y & T_z & 1 \end{bmatrix}$$

where *L* is the *left* or x-axis, *U* is the *up* or y-axis, *F* is the *forward* or z-axis, and *T* is the translation. In order to construct this matrix, these four vectors must first be calculated. Most 3D libraries provide a function that can automatically calculate the look-at matrix for you. But if you don't have such a library, it's not that hard to calculate manually.

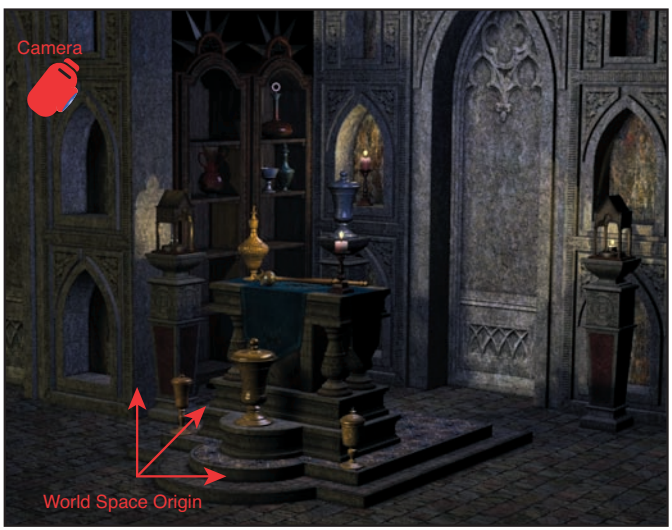


Figure 4.4 Camera space.

Three inputs are needed to construct the look-at matrix: the **eye**, or the position of the camera, the position of the target the camera is looking at, and the up vector of the camera. Although in some cases the camera's up vector will correspond to the world's up vector, that's not always the case. In any event, given these three inputs, it is possible to calculate the four vectors:

```
function CreateLookAt(Vector3 eye, Vector3 target, Vector3 Up)
   Vector3 F = Normalize(target - eye)
   Vector3 L = Normalize(CrossProduct(Up, F))
   Vector3 U = CrossProduct(F, L)
   Vector3 T
   T.x = -DotProduct(L, eye)
   T.y = -DotProduct(U, eye)
   T.z = -DotProduct(F, eye)

   // Create and return look-at matrix from F, L, U, and T
end
```

Once the view space matrix is applied, the entire world is now transformed to be shown through the eyes of the camera. However, it is still a 3D world that needs to be converted into 2D for display purposes.

Projection Space

Projection space, sometimes also referred to as screen space, is the coordinate space where the 3D scene gets flattened into a 2D one. A 3D scene can be flattened into 2D in several different ways, but the two most common are orthographic projection and perspective projection.

In an **orthographic projection**, the entire world is squeezed into a 2D image with no depth perception. This means that objects further away from the camera are the same size as objects closer to the camera. Any pure 2D game can be thought of as using an orthographic projection. However, some games that do convey 3D information, such as *The Sims* or *Diablo III*, also use orthographic projections. Figure 4.5 illustrates a 3D scene that's rendered with an orthographic projection.

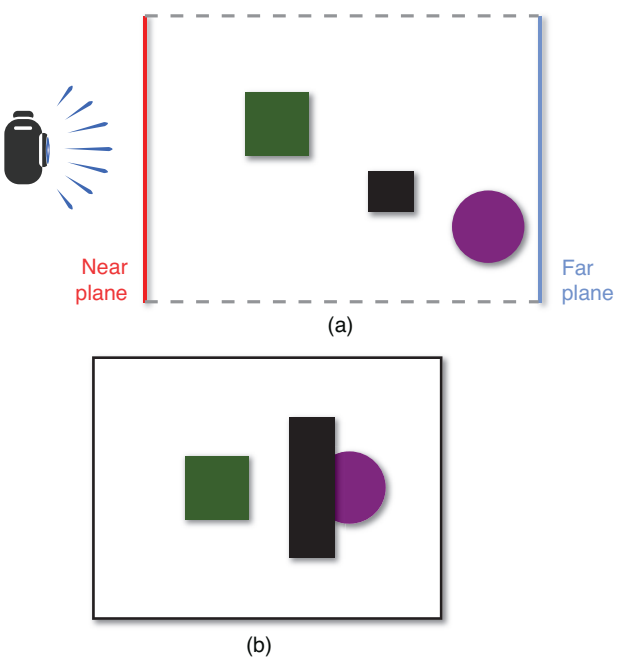


Figure 4.5 Top-down view of an orthographic projection (a), and the resulting 2D image onscreen (b).

The other type of projection commonly used is the **perspective projection**. In this projection, objects further away from the camera are smaller than closer ones. The majority of 3D games use this sort of projection. Figure 4.6 shows the 3D scene from Figure 4.5, except this time with a perspective projection.

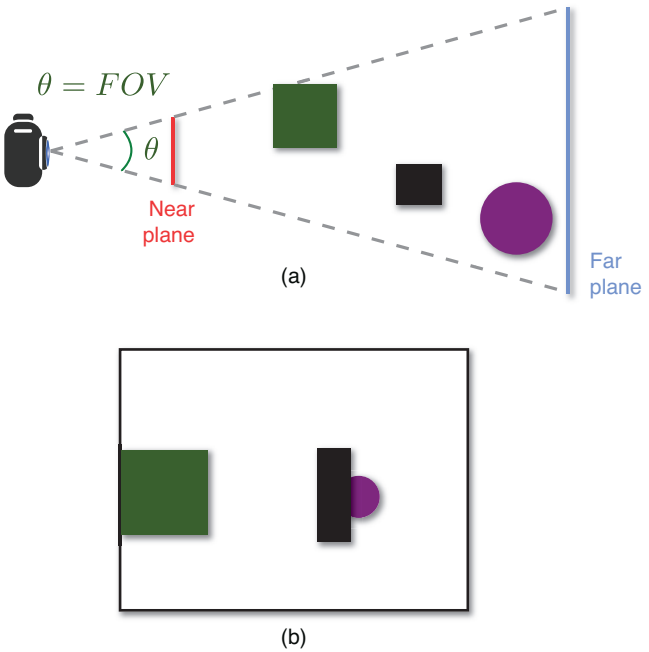


Figure 4.6 Top-down view of a perspective projection (a), and the resulting 2D image onscreen (b).

Both projections have a near and far plane. The **near plane** is typically very close to the camera, and anything between the camera and the near plane is not drawn. This is why games will sometimes have characters partially disappear if they get too close to the camera. Similarly, the **far plane** is far away from the camera, and anything past that plane is not drawn either. Games on PC will often let the user reduce the "draw distance" in order to speed up performance, and this often is just a matter of pulling in the far plane.

The orthographic projection matrix is composed of four parameters: the width and height of the view, and the shortest distance between the eye and the near/far planes:

$$Orthographic = \begin{bmatrix} \frac{2}{width} & 0 & 0 & 0 \\ 0 & \frac{2}{height} & 0 & 0 \\ 0 & 0 & \frac{1}{far - near} & 0 \\ 0 & 0 & \frac{near}{near - far} & 1 \end{bmatrix}$$

The perspective projection has one more parameter, called **field of view** (FOV). This is the angle around the camera that is visible in the projection. Changing the field of view determines how much of the 3D world is visible, and is covered in more detail in Chapter 8, "Cameras." Add in the FOV angle and it is possible to calculate the perspective matrix:

$$yScale = \cot(fov / 2)$$

$$xScale = yScale \cdot \frac{height}{width}$$

$$Perspective = \begin{bmatrix} xScale & 0 & 0 & 0 \\ 0 & yScale & 0 & 0 \\ 0 & 0 & \frac{far}{far - near} & 1 \\ 0 & 0 & \frac{-near \cdot far}{far - near} & 0 \end{bmatrix}$$

There is one last thing to consider with the perspective matrix. When a vertex is multiplied by a perspective matrix, its w-component will no longer be 1. The **perspective divide** takes each component of the transformed vertex and divides it by the w-component, so the w-component becomes 1 again. This process is actually what adds the sense of depth perception to the perspective transform.

Lighting and Shading

So far, this chapter has covered how the GPU is given a list of vertices as well as world transform, view, and projection matrices. With this information, it is possible to draw a black-and-white wireframe of the 3D scene into the 2D color buffer. But a wireframe is not very exciting. At a minimum, we want the triangles to be filled in, but modern 3D games also need color, textures, and lighting.

Color

The easiest way to represent color in a 3D scene is to use the **RGB color space**, which means there is a separate red, green, and blue color value. That's because RGB directly corresponds to how a 2D monitor draws color images. Each pixel on the screen has separate red, green, and blue sub-pixels, which combine to create the final color for that pixel. If you look very closely at your computer screen, you should be able to make out these different colors.

Once RGB is selected, the next decision is the **bit depth**, or the number of bits allocated to each primary color. Today, the most common approach is 8 bits per color, which means there are 256 possible values each for red, green, and blue. This gives a total of approximately 16 million unique colors.

Because the range for the primary colors is 0 through 255, sometimes their values will be expressed in this manner. In web graphics, for instance, #ff0000 refers to 255 for red, 0 for green, and 0 for blue. But a more common approach in 3D graphics is to instead refer to the values as a floating point number between 0.0 and 1.0. So 1.0 is the maximum value for that primary color, whereas 0.0 is an absence of it.

Depending on the game, there is also the possibility of a fourth component, known as **alpha**. The alpha channel determines the transparency of a particular pixel. An alpha of 1.0 means that the pixel is 100% opaque, whereas an alpha of 0.0 means that it's invisible. An alpha channel is required if the game needs to implement something like water—where there is some color the water gives off in addition to objects that are visible under the water.

So if a game supports full RGBA, and has 8 bits per component, it has a total of 32 bits (or 4 bytes) per pixel. This is a very common rendering mode for games. Figure 4.7 demonstrates a few different colors represented in the RGBA color space.

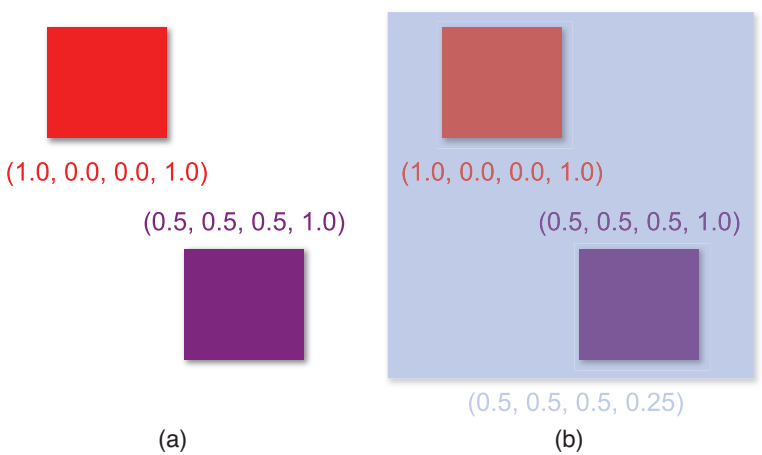


Figure 4.7 Opaque red and purple in the RGBA color space(a). A transparent blue allows other colors to be visible underneath (b).

Vertex Attributes

In order to apply color to a model, we need to store additional information for each vertex. This is known as a **vertex attribute**, and in most modern games several attributes are stored per each vertex. This, of course, adds to the memory overhead of the model and can potentially be a limiting factor when determining the maximum number of vertices a model can have.

A large number of parameters can be stored as vertex attributes. In **texture mapping**, a 2D image is mapped onto a 3D triangle. At each vertex, a **texture coordinate** specifies which part

of the texture corresponds to that vertex, as demonstrated in Figure 4.8. The most common texture coordinates are **UV coordinates**, where the x-coordinate of the texture is named u and the y-coordinate of the texture is named v.

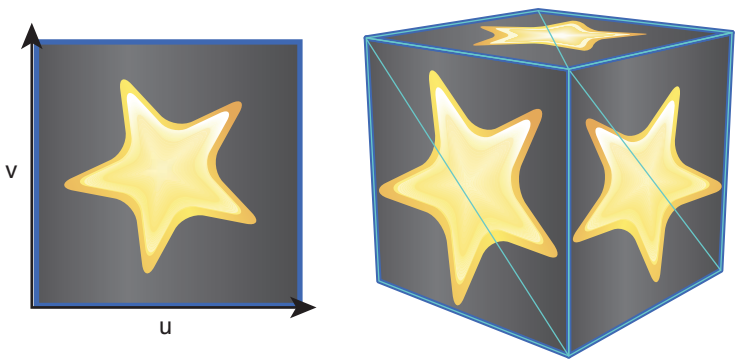


Figure 4.8 Texture mapping with UV coordinates.

Using just textures, it is possible to get a scene that looks colorful. But the problem is this scene does not look very realistic because there is no sense of light as there is in reality. Most lighting models rely on another type of vertex attribute: the **vertex normal**. Recall from Chapter 3, "Linear Algebra For Games," that we can compute the normal to a triangle by using the cross product. But how can a *vertex*, which is just a single point, have a normal?

A vertex normal can be determined in a couple of different ways. One approach is to average the normals of all the triangles that contain that vertex, as in Figure 4.9(a). This works well if you want to have smooth models, but it won't work properly when you want to have hard edges. For example, if you try to render a cube with averaged vertex normals, it will end up with rounded corners. To solve this problem, each of the cube's eight corners must be represented as three different vertices, with each of the copies having a different vertex normal per face, as in Figure 4.9(b).

Remember that a triangle technically has two normals, depending on the order in which the cross product is performed. For triangles, the order depends on the **winding order**, which can either be clockwise or counterclockwise. Suppose a triangle has the vertices A, B, and C in a clockwise winding order. This means that travelling from A to B to C will result in a clockwise motion, as in Figure 4.10(a). This also means that when you construct one vector from A to B, and one from B to C, the cross product between these vectors would point into the page in a right-handed coordinate system. The normal would point out of the page if instead A to B to C were counterclockwise, as in Figure 4.10(b). As with other things we've discussed, it doesn't matter which winding order is selected as long as it's consistent across the entire game.

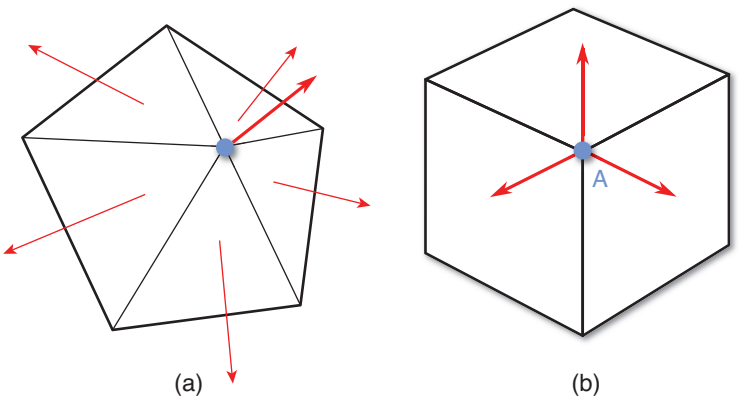


Figure 4.9 Averaged vertex normals (a). Vertex A on the cube uses one of three different normals, depending on the face (b).

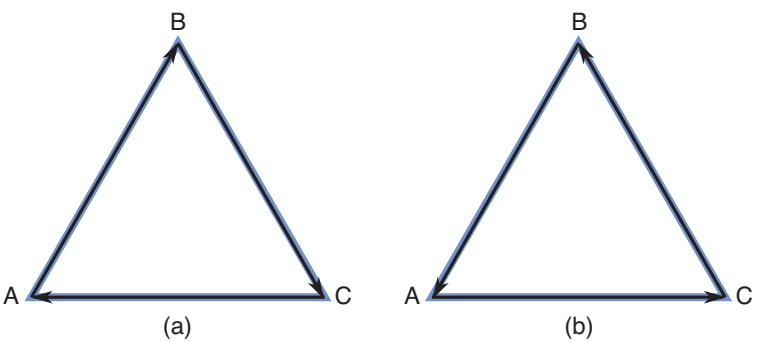
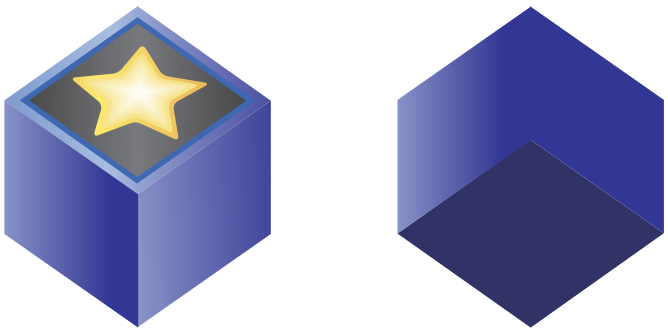


Figure 4.10 Clockwise winding order (a), and counterclockwise winding order (b).

One of the many things we might use to optimize rendering is **back-face culling**, which means triangles that are facing away from the camera are not rendered. So if you pass in triangles with the opposite winding order of what the graphics library expects, you may have forward-facing triangles disappear. An example of this is shown in Figure 4.11.

Lights

A game without lighting looks very drab and dull, so most 3D games must implement some form of lighting. A few different types of lights are commonly used in 3D games; some lights globally affect the scene as a whole, whereas other lights only affect the area around the light.



(a) (b)

Figure 4.11 A cube rendered with the correct winding order (a), and the same cube rendered with the incorrect winding order (b).

Ambient Light

Ambient light is an uniform amount of light that is applied to every single object in the scene. The amount of ambient light may be set differently for different levels in the game, depending on the time of day. A level taking place at night will have a much darker and cooler ambient light than a level taking place during the day, which will be brighter and warmer.

Because it provides an even amount of lighting, ambient light does not light different sides of objects differently. So think of ambient light as the light outside on an overcast day when there is a base amount of light applied everywhere. Figure 4.12(a) shows such an overcast day in nature.

(a) (b)

Figure 4.12 Examples in nature of ambient light (a) and directional light (b).

Directional Light

A **directional light** is a light without a position but that is emitted from a specific direction. Like ambient light, directional lights affect the entire scene. However, because directional lights come from a specific direction, they illuminate one side of objects while leaving the other side in darkness. An example of a directional light is the sun on a sunny day. The direction of the light would be a vector from the sun to the object, and the side facing the sun would be brighter. Figure 4.12(b) shows a directional light at Yellowstone National Park.

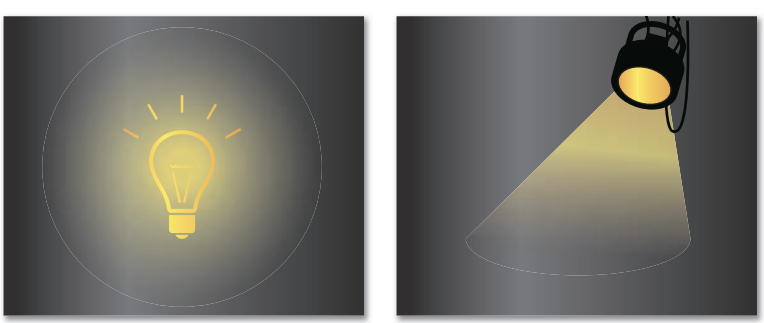
Games that use directional lights often only have one directional light for the level as a whole, representing either the sun or the moon, but this isn't always a case. A level that takes place inside a dark cave may not have any directional lights, whereas a sports stadium at night might have multiple directional lights to represent the array of lights around the stadium.

Point Light

A **point light** is a light placed at a specific point that emanates in all directions. Because they emit from a specific point, point lights also will illuminate only one side of an object. In most cases we don't want the point light to travel infinitely in all directions. For example, think of a light bulb in a dark room, as shown in Figure 4.13(a). There is visible light in the area immediately around the light, but it slowly falls off until there is no longer any more light. It doesn't simply go on forever. In order to simulate this, we can add a **falloff radius** that determines how much the light decreases as the distance from the light increases.

Spotlight

A **spotlight** is much like a point light, except instead of travelling in all directions, it is focused in a cone. To facilitate the cone, it is necessary to have an angle as a parameter of spotlights. As with point lights, spotlights will also illuminate one side of an object. A classic example of a spotlight is a theater spotlight, but another example would be a flashlight in the dark. This is shown in Figure 4.13(b).



(a) (b)

Figure 4.13 A light bulb in a dark room is an example of a point light (a). A spotlight on stage is an example of a spotlight (b).

Phong Reflection Model

Once the lights have been placed in a level, the game needs to calculate how the lights affect the objects in the scene. These calculations can be done with a **bidirectional reflectance distribution function** (BRDF), which is just a fancy way of saying how the light bounces off surfaces. There are many different types of BRDFs, but we will only cover one of the most basic: the **Phong reflection model**. It should be noted that the Phong reflection model is *not* the same thing as Phong shading (which we will cover in a moment). The two are sometimes confused, but they are different concepts.

The Phong model is considered a **local lighting model**, because it doesn't take into account secondary reflections of light. In other words, each object is lit as if it were the only object in the entire scene. In the physical world, if a red light is shone on a white wall, the red light will bounce and fill out the rest of the room with its red color. But this will not happen in a local lighting model.

In the Phong model, light is broken up into three distinctcomponents: ambient, diffuse, and specular, which are illustrated in Figure 4.14. All three components take into account both the color of the object and the color of the light affecting the object. The **ambient** component is tied to the overall illumination of the scene and can be directly tied to the ambient light. Because it's evenly applied to the entire scene, the ambient component is independent of the location of lights and the camera.

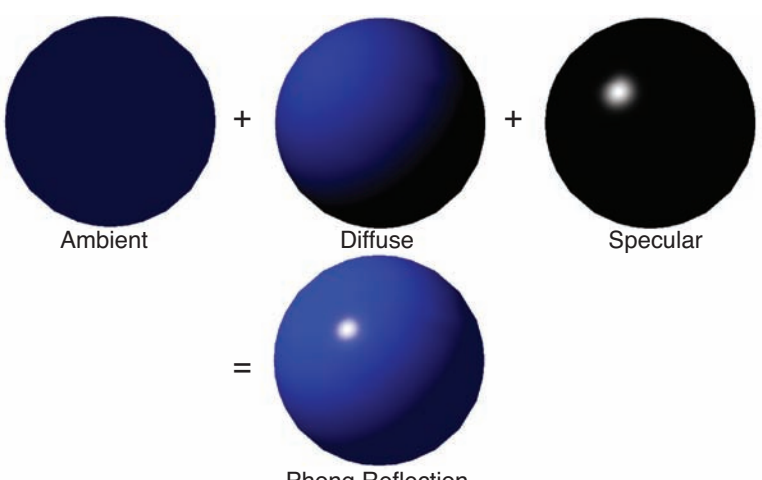


Figure 4.14 Phong reflection model.

The **diffuse** component is the primary reflection of light off the surface. This will be affected by any directional, point, or spotlights affecting the object. In order to calculate the diffuse

component, you need the normal of the surface as well as a vector from the surface to the light. But as with the ambient component, the diffuse component is not affected by the position of the camera.

The final component is the **specular** component, which represents shiny highlights on the surface. An object with a high degree of specularity, such as a polished metal object, will have more highlights than something painted in matte black. As with the diffuse component, the specular component depends on the location of lights as well as the normal of the surface. But it also depends on the position of the camera, because highlights will change position based on the view vector.

Overall, the Phong reflection model calculations are not that complex. The model loops through every light in the scene that affects the surface and performs some calculations to determine the final color of the surface (see Listing 4.1).

Listing 4.1 Phong Reflection Model

```
// Vector3 N = normal of surface
// Vector3 eye = position of camera
// Vector3 pos = position on surface
// float a = specular power (shininess)
Vector3 V = Normalize(eye - pos) // FROM surface TO camera
Vector3 Phong = AmbientColor
foreach Light light in scene
   if light affects surface
      Vector3 L = Normalize(light.pos - pos) // FROM surface TO light
      Phong += DiffuseColor * DotProduct(N, L)
      Vector3 R = Normalize(Reflect(-L, N)) // Reflection of -L about N
      Phong += SpecularColor * pow(DotProduct(R, V), a)
   end
  end
```

Shading

Separate from the reflection model is the idea of **shading**, or how the surface of a triangle is filled in. The most basic type of shading, **flat shading**, has one color applied to the entire triangle. With flat shading, the reflection model is calculated once per each triangle (usually at the center of the triangle), and this computed color is applied to the triangle. This works on a basic level but does not look particularly good.

Gouraud Shading

A slightly more complex form of shading is known as **Gouraud shading**. In this type of shading, the reflection model is calculated once per each vertex. This results in a different color potentially being computed for each vertex. The rest of the triangle is then filled by blending

between the vertex colors. So, for example, if one vertex were red and one vertex were blue, the color between the two would slowly blend from red to blue.

Although Gouraud shading is a decent approximation, there are some issues. First of all, the quality of the shading directly correlates to the number of polygons in the model. With a low number of polygons, the shading may have quite a few hard edges, as shown in Figure 4.13(a). A high-polygon model can look good with Gouraud shading, but it's at the additional memory cost of having more polygons.

Another issue is that smaller specular highlights often look fairly bad on low-polygon models, resulting in the banding seen in Figure 4.13(a). And the highlight may even be missed entirely if it's so small that it's contained only in the interior of a polygon, because we are only calculating the reflection model at each vertex. Although Gouraud shading was popular for many years, as GPUs have improved, it no longer sees much use.

Phong Shading

With **Phong shading**, the reflection model is calculated at every single pixel in the triangle. In order to accomplish this, the vertex normals are interpolated across the surface of the triangle. At every pixel, the result of the interpolation is then used as the normal for the lighting calculations.

As one might imagine, Phong shading is computationally much more expensive than Gouraud shading, especially in a scene with a large number of lights. However, most modern hardware can easily handle the increase in calculations. Phong shading can be considered a type of **per-pixel lighting**, because the light values are being calculated for every single pixel.

Figure 4.15 compares the results of shading an object with Gouraud shading and with Phong shading. As you can see, the Phong shading results in a much smoother and even shading.

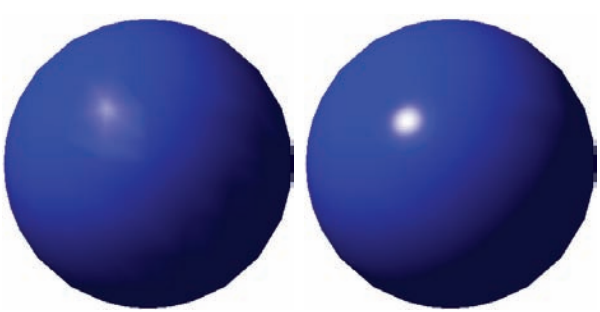


Figure 4.15 Object drawn with Gouraud shading (a), and an object drawn with Phong shading (b).

One interesting thing to note is that regardless of the shading method used, the silhouette is identical. So even if Phong shading is used, the outline can be a dead giveaway if an object is a low-polygon model.

Visibility

Once you have meshes, coordinate space matrices, lights, a reflection model, and shading, there is one last important aspect to consider for 3D rendering. Which objects are and are not visible? This problem ends up being far more complex for 3D games than it is for 2D games.

Painter's Algorithm, Revisited

As discussed in Chapter 2, "2D Graphics," the painter's algorithm (drawing the scene back to front) works fairly well for 2D games. That's because there's typically a clear ordering of which 2D sprites are in front of each other, and often the 2D engine may seamlessly support the concept of layers. But for 3D games, this ordering is not nearly as static, because the camera can and will change its perspective of the scene.

This means that to use the painter's algorithm in a 3D scene, all the triangles in the scene have to be resorted, potentially every frame, as the camera moves around in the scene. If there is a scene with 10,000 objects, this process of resorting the scene by depth every single frame would be computationally expensive.

That already sounds very inefficient, but it can get much worse. Consider a split-screen game where there are multiple views of the game world on the same screen. If player A and player B are facing each other, the back-to-front ordering is going to be different for each player. To solve this, we would either have to resort the scene multiple times per frame or have two different sorted containers in memory—neither of which are desirable solutions.

Another problem is that the painter's algorithm can result in a massive amount of **overdraw**, which is writing to a particular pixel more than once per frame. If you consider the space scene from Figure 2.3 in Chapter 2, certain pixels may have been drawn four times during the frame: once for the star field, once for the moon, once for an asteroid, and once for the spaceship.

In modern 3D games, the process of calculating the final lighting and texturing for a particular pixel is one of the most expensive parts of the rendering pipeline. If a pixel gets overdrawn later, this means that any time spent drawing it was completely wasted. Due to the expense involved, most games try to eliminate as much overdraw as possible. That's never going to happen with the painter's algorithm.

And, finally, there is the issue of overlapping triangles. Take a look at the three triangles in Figure 4.16. Which one is the furthest back?

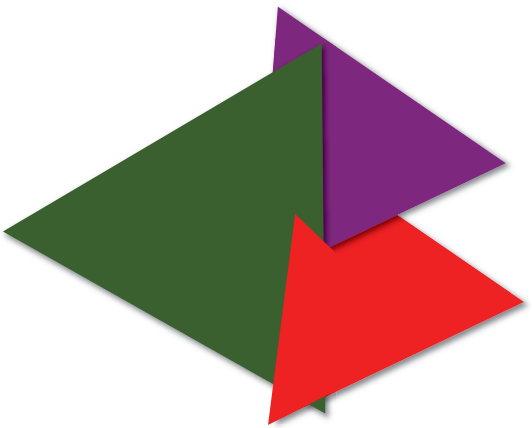


Figure 4.16 Overlapping triangles failure case.

The answer is that there is no one triangle that is furthest back. In this instance, the only way the painter's algorithm would be able to draw these triangles properly is if some of them were split into multiple triangles.

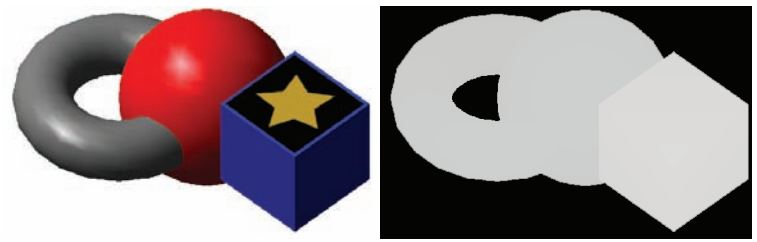
Because of all these issues, the painter's algorithm does not see much use in 3D games.

Z-Buffering

In **z-buffering**, an additional memory buffer is used during the rendering process. This extra buffer, known as the **z-buffer**, stores data for each pixel in the scene, much like the color buffer. But unlike the color buffer, which stores color information, the z-buffer (also known as the **depth buffer**) stores the distance from the camera, or **depth**, at that particular pixel. Collectively, the set of buffers we use to represent a frame (which can include a color buffer, z-buffer, stencil buffer, and so on) is called the **frame buffer**.

At the beginning of a frame rendered with z-buffering, the z-buffer is cleared out so that the depth at each pixel is set to infinity. Then, during rendering, before a pixel is drawn, the depth is computed at that pixel. If the depth at that pixel is less than the current depth value stored in the z-buffer, that pixel is drawn, and the new depth value is written into the z-buffer. A sample representation of the z-buffer is shown in Figure 4.17.

So the first object that's drawn every frame will always have all of its pixels' color and depth information written into the color and z-buffers, respectively. But when the second object is drawn, if it has any pixels that are further away from the existing pixels, those pixels will not be drawn. Pseudocode for this algorithm is provided in Listing 4.2.



Scene Z-Buffer

Figure 4.17 A sample scene and its corresponding z-buffer.

Listing 4.2 Z-Buffering

```
// zBuffer[x][y] grabs depth at that pixel
foreach Object o in scene
   foreach Pixel p in o
      float depth = calculate depth at p
      if zBuffer[p.x][p.y] > depth
         draw p
         zBuffer[p.x][p.y] = depth
      end
   end
  end
```

With z-buffering, the scene can be drawn in any arbitrary order, and so long as there aren't any objects with transparency, it will look correct. That's not to say that the order is irrelevant—for example, if the scene is drawn back to front, you would overdraw a lot of pixels every frame, which would have a performance hit. And if the scene were drawn front to back, it would have no overdraw at all. But the idea behind z-buffering is that the scene does not need to be sorted in terms of depth, which can greatly improve performance. And because z-buffering works on a per-pixel basis, and not a per-object basis, it will work well even with the overlapping triangles case in Figure 4.16.

But that's not to say z-buffering by itself solves all visibility problems. For one, transparent objects won't work with z-buffering as is. Suppose there is semi-transparent water, and under this water there's a rock. If pure z-buffering is used, drawing the water first would write to the z-buffer and would therefore prevent the rock from being drawn. To solve this issue, we first render all the opaque objects in the scene using normal z-buffering. Then we can disable depth buffer writes and render all the transparent objects, still checking the depth at each pixel to make sure we aren't drawing transparent pixels that are behind opaque ones.

As with the representation of color, the z-buffer also has a fixed bit depth. The smallest size z-buffer that's considered reasonable is a 16-bit one, but the reduced memory cost comes with some side effects. In **z-fighting**, two pixels from different objects are close to each other, but far away from the camera, flicker back and forth on alternating frames. That's because the lack of accuracy of 16-bit floats causes pixel A to have a lower depth value than pixel B on frame 1, but then a higher depth value on frame 2. To help eliminate this issue most modern games will use a 24- or 32-bit depth buffer.

It's also important to remember that z-buffering by itself cannot guarantee no pixel overdraw. If you happen to draw a pixel and later it turns out that pixel is not visible, any time spent drawing the first pixel will still be wasted. One solution to this problem is the **z-buffer pre-pass**, in which a separate depth rendering pass is done before the final lighting pass, but implementing this is beyond the scope of the book.

Keep in mind that z-buffering is testing on a pixel-by-pixel basis. So, for example, if there's a tree that's completely obscured by a building, z-buffering will still individually test each pixel of the tree to see whether or not that particular pixel is visible. To solve these sorts of problems, commercial games often use more complex **culling** or **occlusion** algorithms to eliminate entire objects that aren't visible on a particular frame. Such algorithms include binary spatial partitioning (BSP), portals, and occlusion volumes, but are also well beyond the scope of this book.

World Transform, Revisited

Although this chapter has covered enough to successfully render a 3D scene, there are some outstanding issues with the world transform representation covered earlier in this chapter. The first problem is one of memory—if the translation, scale, and rotation must be modified independently, each matrix needs to be stored separately, which is 16 floating point values per matrix, for a total of 48 floating points for three matrices. It is easy enough to reduce the footprint of the translation and scale—simply store the translation as a 3D vector and the scale as a single float (assuming the game only supports uniform scales). Then these values can be converted to the corresponding matrices only at the last possible moment, when it's time to compute the final world transform matrix. But what about the rotation?

There are some big issues with representing rotations using Euler angles, primarily because they're not terribly flexible. Suppose a spaceship faces down the z-axis in model space. We want to rotate the space ship so it points instead toward an arbitrary object at position P. In order to perform this rotation with Euler angles, you need to determine the angles of rotation. But there isn't one single cardinal axis the rotation can be performed about; it must be a combination of rotations. This is fairly difficult to calculate.

Another issue is that if a 90° Euler rotation is performed about a coordinate axis, the orientation of the other coordinate axes also changes. For example, if an object is rotated 90° about the

z-axis, the x- and y-axes will become one and the same, and a degree of motion is lost. This is known as **gimbal lock**, as illustrated in Figure 4.18, and it can get very confusing very quickly.

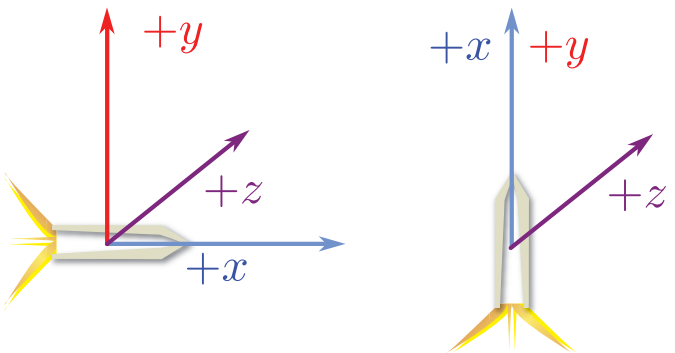


Figure 4.18 A 90° rotation about the z-axis causes gimbal lock.

Finally, there is a problem of smoothly interpolating between two orientations. Suppose a game has an arrow that points to the next objective. Once that objective is reached, the arrow should change to point to the subsequent one. But it shouldn't just instantly snap to the new objective, because an instant change won't look great. It should instead smoothly change its orientation, over the course of a second or two, so that it points at the new objective. Although this can be done with Euler angles, it can be difficult to make the interpolation look good.

Because of these limitations, using Euler angle rotations is typically not the preferred way to represent the rotation of objects in the world. Instead, a different math construct is used for this.

Quaternions

As described by mathematicians, quaternions are, quite frankly, extremely confusing. But for game programming, you really can just think of a **quaternion** as a way to represent a rotation about an arbitrary axis. So with a quaternion, you aren't limited to rotations about x, y, and z. You can pick any axis you want and rotate around that axis.

Another advantage of quaternions is that two quaternions can easily be interpolated between. There are two types of interpolations: the standard lerp and the **slerp**, or spherical linear interpolation. Slerp is more accurate than lerp but may be a bit more expensive to calculate, depending on the system. But regardless of the interpolation type, the aforementioned objective arrow problem can easily be solved with quaternions.

Quaternions also only require four floating point values to store their information, which means memory is saved. So just as the position and uniform scale of an object can be stored as a 3D vector and float, respectively, the orientation of the object can be stored with a quaternion.

For all intents and purposes, games use **unit quaternions**, which, like unit vectors, are quaternions with a magnitude of one. A quaternion has both a vector *and* a scalar component and is often written as $q = [q_v, q_s]$. The calculation of the vector and scalar components depends on the axis of rotation, $\hat{a}$, and the angle of rotation, θ:

$$q_v = \hat{a} \sin\frac{\theta}{2}$$
$$q_s = \cos\frac{\theta}{2}$$

It should be noted that the axis of rotation must be normalized. If it isn't, you may notice objects starting to stretch in non-uniform ways. If you start using quaternions, and objects are shearing in odd manners, it probably means somewhere a quaternion was created without a normalized axis.

Most 3D game math libraries should have quaternion functionality built in. For these libraries, a `CreateFromAxisAngle` or similar function will automatically construct the quaternion for you, given an axis and angle. Furthermore, some math libraries may use x, y, z, and w-components for their quaternions. In this case, the vector part of the quaternion is x, y, and z while the scalar part is w.

Now let's think back to the spaceship problem. You have the initial facing of the ship down the z-axis, and the new facing can be calculated by constructing a vector from the ship to target position P. To determine the axis of rotation, you could take the cross product between the initial facing and the new facing. Also, the angle of rotation similarly can be determined by using the dot product. This leaves us with both the arbitrary axis and the angle of rotation, and a quaternion can be constructed from these.

It is also possible to perform one quaternion rotation followed by another. In order to do so, the quaternions should be *multiplied* together, but in the reverse order. So if an object should be rotated by q and then by p, the multiplication is pq. The multiplication for two quaternions is performed using the **Grassmann product**:

$$(pq)_v = p_s q_v + q_s p_v + p_v \times q_v$$
$$(pq)_s = p_s q_s - p_v \cdot q_v$$

Much like matrices, quaternions have an inverse. Luckily, to calculate the inverse of a unit quaternion you simply negate the vector component. This negation of the vector component is also known as the **conjugate** of the quaternion.

Because there is an inverse, there must be an identity quaternion. This is defined as:

$$i_v = \langle 0,\ 0,\ 0 \rangle$$
$$i_s = 1$$

Finally, because the GPU still expects the world transform matrix to be in matrix form, the quaternion will, at some point, need to be converted into a matrix. The calculation of a matrix from a quaternion is rather complex, so it's not included in this text. But rest assured that your 3D math library most likely has a function to do this for you.

3D Game Object Representation

Once we are using quaternions, the translation, scale, and rotation can all be represented with data smaller than a single 4×4 matrix. This is what should be stored for the world transform of a 3D game object. Then, when it's time to pass world transform matrix information to the rendering code, a temporary matrix can be constructed. This means that if code needs to update the position of the game object, it only needs to change the position vector, and not an entire matrix (see Listing 4.3).

Listing 4.3 3D Game Object

```
class 3DGameObject
   Quaternion rotation
   Vector3 position
   float scale

   function GetWorldTransform()
      // Order matters! Scale, rotate, translate.
      Matrix temp = CreateScale(scale) *
                    CreateFromQuaternion(rotation) *
                    CreateTranslation(position)
      return temp
   end
  end
```

Summary

This chapter looked at many of the core aspects of 3D graphics. Coordinate spaces are an important concept that expresses models relative to the world and the camera, and also determines how we flatten a 3D scene into a 2D image. There are several different types of lights we might have in a scene, and we use BRDFs such as the Phong reflection model to determine how they affect objects in the scene. 3D engines also shy away from using the painter's algorithm and instead use z-buffering to determine which pixels are visible. Finally, using quaternions is preferred as a rotation method over Euler angles.

There is much, much more to rendering than was covered in this chapter. Most games today use advanced rendering techniques such as normal mapping, shadows, global illumination,

deferred rendering, and so on. If you are interested in studying more about rendering, there's much to learn out there.

Review Questions

1. Why are triangles used to represent models in games?
2. Describe the four primary coordinate spaces in the rendering pipeline.
3. Create a world transform matrix that translates by $\langle 2, 4, 6 \rangle$ and rotates 90° about the z-axis.
4. What is the difference between an orthographic and perspective projection?
5. What is the difference between ambient and directional light? Give an example of both.
6. Describe the three components of the Phong reflection model. Why is it considered a local lighting model?
7. What are the tradeoffs between Gouraud and Phong shading?
8. What are the issues with using the Painter's algorithm in a 3D scene? How does z-buffering solve these issues?
9. Why are quaternions preferable to Euler angles for rotation representations?
10. Construct a quaternion that rotates about the x-axis by 90°.

Additional References

Akenine-Möller, Tomas, et. al. *Real-Time Rendering (3rd Edition)*. Wellesley: AK Peters, 2008. This is considered the *de facto* reference for rendering programmers. Although this latest edition is a few years old now, it still is by far the best overview of rendering for games programmers.

Luna, Frank. *Introduction to 3D Game Programming with DirectX 11*. Dulles: Mercury Learning & Information, 2012. One thing about rendering books is they often focus on one specific API. So there are DirectX books and OpenGL books. Luna has written DirectX books since at least 2003, and when I was a budding game programmer I learned a great deal from one of his early books. So if you are planning on using DirectX for your game (remember that it's PC only), I'd recommend this book.

CHAPTER 5

INPUT

Without input, games would be a static form of entertainment, much like film or television. It is the fact that the game responds to the keyboard, mouse, controller, or other input device that enables interactivity.

This chapter takes an in-depth look at a wide variety of input devices, including those that are predominantly for mobile devices. It further covers how a high-level input system might be designed and implemented.

Input Devices

As shown in Figure 5.1, there is a host of types of input devices, some more common than others. Although a keyboard, mouse, or controller might be the most familiar input device for a PC or console gamer, on the mobile side of things touch screens and accelerometers dominate. Other more recent input devices include the WiiMote, Kinect, guitar controllers, and head-tracking devices for virtual reality headsets. Regardless of the device, however, certain core techniques are applied when the device is queried for information.

Figure 5.1 A variety of input devices.

Most forms of input can be broken down into two categories: digital and analog. A **digital** form of input is one that has a binary state: It can be either "on" or "off." For example, a key on a keyboard typically is digital—either the spacebar is pressed, or it isn't. Most keyboards do not have an in-between state. An **analog** form of input is one where there can be a range of values returned by the device. One common analog device is a joystick, which will have a range of values in two dimensions.

Whereas a keyboard may only have digital inputs, many input devices have both analog and digital components. Take, for instance, the Xbox 360 controller: The directional pad and the face buttons are digital, but the thumb sticks and triggers are analog. Note that the face buttons on a controller are not always digital, as in the case of the Playstation 2, but it's a typical implementation.

One other consideration for some types of games is that the input system may need to respond to **chords** (multiple button presses at once) or **sequences** (a series of inputs). A genre where these are prevalent is the fighting game, which often has special moves that utilize both chords and sequences. For example, in the later *Street Fighter* games, if a Blanka player presses forward

and all three punch or kick buttons at the same time (a chord), the character will quickly jump forward. Similarly, all *Street Fighter* games have the concept of a fireball, which is performed with a quarter circle forward followed by a punch button (a sequence). Handling chords and sequences is beyond the scope of this chapter, but one approach is to utilize a state machine designed to recognize the different special moves as they are performed.

Digital Input

Other than for text-based games, standard console input (such as `cin`) typically is not used. For graphical games, the console usually is disabled altogether, so it would be impossible to use standard input in any event. Most games will instead use a library that enables much more granular device queries—one possibility for a cross-platform title might be Simple DirectMedia Layer (SDL; www.libsdl.org).

With libraries such as SDL, it is possible to query the current state of a digital input device. In such systems, it usually is possible to grab an array of Booleans that describe the state of every key on the keyboard (or any other digital input device). So if the spacebar is pressed, the index corresponding to the spacebar would be set to true. Memorizing which index numbers correspond to which keys would be tiresome, so usually there is a **keycode** enumeration that maps each index to a specific name. For example, if you want to check the status of the spacebar, you might be able to check the index defined by K_SPACE.

Now suppose you are making a simple game where pressing the spacebar causes your ship to fire a missile. It might be tempting to have code similar to this:

```
if IsKeyDown(K_SPACE)
   fire missile
end
```

But the problem with this code is that even if the player quickly presses and releases the spacebar, the `if` statement will be true for multiple consecutive frames. That's because if the game runs at 60 FPS, the player would have to press and release the spacebar within the span of 16ms for it to only be detected on one single frame. Because this is unlikely to occur, each tap of the spacebar would probably fire three or four missiles on consecutive frames. Furthermore, if the player simply holds down the spacebar, a missile will fire once per frame until it's released. This might be the desired behavior, but it also might completely imbalance the game.

A simple but effective improvement is to track the state of the key on both the current frame and the previous frame. Because there are now two Boolean values, there are a total of four possible combinations, and each maps to a logically different state, as detailed in Table 5.1.

Table 5.1 Last and Current Key State Combinations

Last State	Current State	Result
False	False	Still released
False	True	Just pressed
True	False	Just released
True	True	Still pressed

So if on the last frame the spacebar was not pressed, but on the current frame it is pressed, that would correspond to the "just pressed" result. This means if the missile fire action were instead mapped to a "just pressed" spacebar action, the ship would only fire one missile per discrete button press.

This system also provides more flexibility than the previous approach. Suppose our game allows the player to charge up his missile the longer he holds down the spacebar. In this case, we would begin charging the missile on "just pressed," increase the power of the missile the longer the spacebar is "still pressed," and finally launch the missile when "just released" occurs.

In order to support these four different results, we first must declare an `enum` of possibilities:

```
enum KeyState
   StillReleased,
   JustPressed,
   JustReleased,
   StillPressed
end
```

Then, when the input is updated, the current snapshot needs to be grabbed:

```
function UpdateKeyboard()
   // lastState and currentState are arrays defined elsewhere,
   // which store the entire state of the keyboard.
   lastState = currentState
   currentState = get keyboard state
end
```

Finally, a function can be implemented that returns the correct `KeyState` value depending on the permutation:

```
function GetKeyState(int keyCode)
   if lastState[keyCode] == true
      if currentState[keyCode] == true
         return StillPressed
      else
         return JustReleased
```

```
      end
   else
      if currentState[keyCode] == true
         return JustPressed
      else
         return StillReleased
      end
   end
end
```

With this function, it is then possible to check the `KeyState` of any key. Later in this chapter, we will discuss an event-based system that builds on this. But for now, let's move on to analog devices.

Analog Input

Because analog devices have a wide range of values, it is common to have spurious input. Suppose a joystick has its x and y values represented by a signed 16-bit integer. This means both the x and y values can range from –32,768 to +32,767. If the joystick is placed down on a flat surface and the player doesn't touch it, in theory both the x and y values should be zero, because the joystick is not being touched. However, in practice the values will be *around* zero, but rarely precisely zero.

Because of this, if the raw analog input was just applied directly to the movement of a character, the character would never stay still. This means you could put down your controller yet still have your character move onscreen. To resolve this problem, most games implement some sort of **analog filtering**, which is designed to eliminate spurious input data. Figure 5.2 illustrates how this filtering might be used to implement a **dead zone**, which is an area around the center of the joystick input range that is ignored.

Figure 5.2 Joystick dead zone causes input within the inner circle to be ignored.

A very basic dead zone implementation would be to set the x and y values to zero in cases where they are relatively close to zero. If the values range between +/– 32K, perhaps a ~10% dead zone of 3,000 could be utilized:

```
int deadZone = 3000
Vector2 joy = get joystick input

if joy.x >= -deadZone && joy.x <= deadZone
   joy.x = 0
end

if joy.y >= -deadZone && joy.y <= deadZone
   joy.y = 0
end
```

However, there are two problems with this approach. First of all, the dead zone will be a square instead of a circle. This means if you press the joystick such that the x and y values of the joystick are both slightly less than the dead zone value, you won't get any movement even though you have moved the joystick beyond the 10% threshold.

Another issue is that this approach does not use the full range of values available. There is a smooth ramp from 10% speed to 100% speed, but everything below 10% is lost. We would prefer a solution where the movement can be anywhere from 0% speed to 100% speed. Or in other words, we want valid movement input to start at slightly higher than 0% speed, rather than slightly higher than 10% speed. This will require modifying the range so an "original" value of 55% speed, which is half way between 10% speed and 100% speed, is instead filtered into 50% speed.

To solve this problem, rather than treating the x and y values as individual components, we can treat the joystick input as a 2D floating point vector. Then vector operations can be performed in order to filter for the dead zone.

```
float deadZone = 3000
float maxValue = 32677
Vector2 joy = get joystick input
float length = joy.length()

// If the length < deadZone, we want no input
if length < deadzone
   joy.x = 0
   joy.y = 0
else
   // Calculate the percent between the deadZone and maxValue circles
   float pct = (length - deadZone) / (maxValue - deadZone)
```

```
    // Normalize vector and multiply to get correct final value
    joy = joy / length
    joy = joy * maxValue * pct
end
```

Let's test out the preceding pseudocode. The midpoint between 32,677 and 3,000 is ~17,835, so a vector with that length should yield a percent of ~50%. One such vector with that length would be x = 12,611 and y = 12,611. Because the length of 17,835 is greater than the dead zone of 3,000, the `else` case will be utilized and the following calculations will occur:

$$pct = (17835 - 3000) / (32677 - 3000)$$
$$pct \approx 0.5$$
$$joy(x, y) = (12611/17835, 12611/17835)$$
$$joy(x, y) \approx (0.71, 0.71)$$
$$(0.71, 0.71) * 32677 * 0.5 \approx (11600, 11600)$$

After the analog filtering is applied, the resultant vector has a length of ~16,404. In other words, a value that's 50% between 3,000 and 32,677 was adjusted to be the value 50% between 0 and 32,677, which allows us to get the full range of movement speeds from 0% all the way to 100%.

Event-Based Input Systems

Imagine you are driving a kid to an amusement park. The kid is very excited, so every minute she asks, "Are we there yet?" She keeps asking this over and over again until you finally arrive, at which point the answer to her question is yes. Now imagine that instead of one kid, you're driving a car full of kids. Every single kid is interested in going to the amusement park, so each of them repeatedly asks the same question. Not only is this annoying for the driver, it's also a waste of energy on the part of the kids.

This scenario is essentially a **polling** system in that each kid is polling for the result of the query. For an input system, a polling approach would be one where any code interested in whether the spacebar was "just pressed" would have to manually check. Over and over again on every single frame, and in multiple locations, `GetKeyState` would be called on `K_SPACE`. This not only leads to a duplication of effort, but it could also introduce bugs if the state of the keyboard changes in the middle of the frame.

If we return to the amusement park analogy, imagine once again you have that car full of kids. Rather than having the kids poll over and over again, you can employ an **event-based** or **push** system. In an event-based system, the kids "register" to an event that they care about (in this case, arrival at the amusement park) and are then notified once the event occurs. So the drive to the amusement park is serene, and once you arrive you simply notify all the kids that you've arrived.

An input system can be designed to be event based as well. There can be an overarching system that accepts registrations to particular input events. When the event occurs, the system can then notify any code that registered to said event. So all the systems that must know about the spacebar can register to a spacebar "just pressed" event, and they will all be notified once it occurs.

Note that the underlying code for an event-based input system still must use a polling mechanism. Just like how you as the driver must continue to check whether or not you've arrived at the park so you can notify the kids, the event-based input system must be polling for the spacebar so it can notify the appropriate systems. However, the difference is that the polling is now being performed in only one location in code.

A Basic Event System

Certain languages, such as C#, have native support for events. However, even if events are not natively supported by your language of choice, it is very much possible to implement an event system. Nearly every language has support for saving a function in a variable, whether it's through an anonymous function, lambda expression, function object, or function pointer. As long as functions can be saved to variables, it is possible to have a list of functions that are registered to a particular event. When that event is triggered, those registered functions can then each be executed.

Suppose you want to implement a simple event-driven system for mouse clicks. Different parts of the game code will want to be notified when a mouse click occurs and in which location on screen. In this case, there needs to be a central "mouse manager" that allows interested parties to register to the mouse click event. The declaration for this class might look something like this:

```
class MouseManager
    List functions

   // Accepts functions passed as parameters with signature of (int, int)
   function RegisterToMouseClick(function handler(int, int))
      functions.Add(handler)
   end

   function Update(float deltaTime)
      bool mouseClicked = false
      int mouseX = 0, mouseY = 0

      // Poll for a mouse click
      ...

      if mouseClicked
         foreach function f in functions
            f(mouseX, mouseY)
         end
```

```
      end
   end
end
```

Then any party interested in receiving a mouse click would be able to register a function by calling `RegisterToMouseClick` in the following manner:

```
// Register "myFunction" to mouse clicks
MouseManager.RegisterToMouseClick(myFunction)
```

Once registered, the function will be automatically called every time a mouse click occurs. The best way to make the `MouseManager` globally accessible is to use the **singleton** design pattern, which is a way to provide access to a class that has only one instance in the entire program.

The event system used for the mouse manager could easily be expanded so that it covers all button presses—it wouldn't be that much more work to allow functions to be registered to every key on the keyboard, for example. But there are a few issues with the approach used in this basic event system.

INPUT IN UNREAL

The Unreal Engine uses a fairly customizable event-driven system for player input. The key bindings are all defined in an INI file, and each key binding can be directly mapped to one or more functions written in the Unreal scripting language. For example, if there were a `Fire` script function that causes the player to fire his weapon, it could be mapped to both the left mouse button and the right trigger on a controller with the following bindings:

```
Bindings=(Name="LeftMouseButton",Command="Fire")
Bindings=(Name="XboxTypeS_RightTrigger",Command="Fire")
```

By default, these bindings get triggered when the key is "just pressed." It's also possible to have different actions occur when the key is "just released" if the `onrelease` modifier is used. For example, the following binding would cause the `StartFire` script function to be called when the left mouse button is first pressed, and then the `StopFire` function when it's released:

```
Bindings=(Name="LeftMouseButton",Command="StartFire | onrelease
StopFire")
```

The pipe operator can be used to actually map any number of script functions to one key press, so there is a great deal of flexibility. One thing that this event system is not really designed to handle, however, is menu input. For UI screens, Unreal instead provides an `UpdateInput` function that can be overloaded to create the desired behavior.

Further information on the Unreal binding system can be found on Epic's site at http://udn.epicgames.com/Three/KeyBinds.html.

One issue is that the current system has a tight coupling between specific buttons and events. That is to say interested parties must register to a specific key press (left mouse button) rather than an abstract action (such as fire weapon). What if one player wants to fire his weapon with the spacebar instead of the left mouse button? Any firing code that was registered directly to the left mouse button wouldn't work properly in this case. So an ideal input system would allow for abstract actions that can be bound to specific buttons.

Another issue is that the event system described does not allow any one registered party to prevent any other parties from receiving the event. Suppose you are making a simple RTS (such as the tower defense game in Chapter 14, "Sample Game: Tower Defense for PC/Mac"). If such a game were on a PC, it would make sense that a mouse click could be directed either at the user interface (the menus) or at the game world itself (selecting units). Because the UI is displayed on top of the game world, the UI needs to have first priority on mouse clicks. This means when a mouse click occurs, the UI must first inspect it and determine whether the click is directed at the UI. If it is, then that click event should not be dispatched to the game world. But in our current input system, both the game world and UI would receive the mouse click, which could potentially cause unintended behavior.

A More Complex System

The system covered in this section does not allow direct registration of a function to a specific event. It instead creates a list of active input actions that can then be queried by interested parties. So while perhaps it is not an event-based system in the strict definition, it is fairly similar. It also happens to be that the system outlined here is very similar to the one used in Chapter 14's tower defense game.

The basic premise of this system is that there is a map of key bindings. Recall that a map is a set of (key, value) pairs, which, depending on the type of map, may or may not be sorted. In this system, the key is the name of the binding (such as "Fire"), and the value is information about that binding such as what key it is mapped to and when it gets triggered (just pressed, still pressed, and so on). Because we want to support multiple bindings for actions (so "Fire" can be the left mouse button or the spacebar), we'd ideally select a type of map that does allow duplicate entries with the same key.

Every frame, the input system goes through the map of key bindings and determines which bindings (if any) are active. The active bindings for that specific frame are then stored in an "active" bindings map. This map of active bindings is first provided to the UI system, which will go through and determine whether there are any actions that affect the UI. If there are such actions, they will be removed from the active bindings map and processed. Any remaining bindings are then passed on to the gameplay system for processing. Figure 5.3 visualizes how the bindings are passed between the systems.

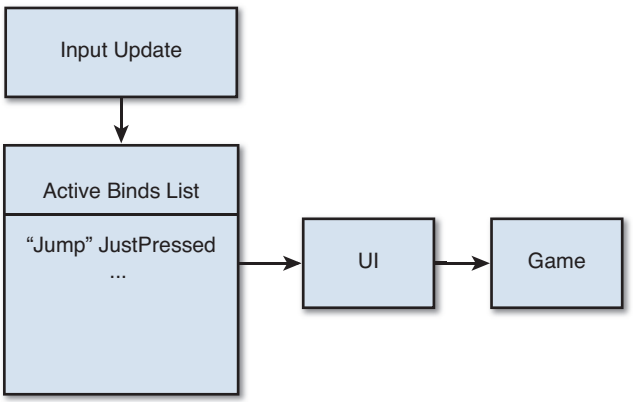


Figure 5.3 Proposed input system.

Although it might seem odd that all the bindings in the game, including actions such as "Jump," might be first processed by the UI, it provides a great deal of flexibility. For example, in a tutorial it's common to have a dialog box that only goes away once a particular action is performed. In this case, you could have the "Press ... to Jump" dialog box pause the game and only go away once the button is pressed.

The code for this system is not terribly complex and is provided in Listing 5.1. If you'd like to see a live implementation of a system very similar to this one, take a look at InputManager.cs, which is provided in the source for Chapter 14's tower defense game.

Listing 5.1 A More Complex Input Manager

```
// KeyState enum (JustPressed, JustReleased, etc.) from earlier in this
chapter
...
// GetKeyState from earlier in this chapter
...

// BindInfo is used for the "value" in the map
struct BindInfo
   int keyCode
   KeyState stateType
end

class InputManager
   // Stores all the bindings
   Map keyBindings;
   // Stores only the active bindings
   Map activeBindings;
```

```
// Helper function to add a binding to keyBindings
function AddBinding(string name, int code, KeyState type)
   keyBindings.Add(name, BindInfo(code, type))
end

// Initializes all bindings
function InitializeBindings()
   // These could be parsed from a file.
   // Call AddBinding on each binding.

   // For example, this would be a binding "Fire" mapped to
   // enter's keyCode and a KeyState of JustReleased:
   // AddBinding("Fire", K_ENTER, JustReleased)
   ...
end

function Update(float deltaTime)
   // Clear out any active bindings from last frame
   activeBindings.Clear()

   // KeyValuePair has a .key and .value member for the
   // key and value from the map, respectively.
   foreach KeyValuePair k in keyBindings
      // Use previously-defined GetKeyState to grab the state
      // of this particular key code.
      if GetKeyState(k.value.keyCode) == k.value.stateType
         // If this matches, this binding is active
         activeBindings.Add(k.key, k.value)
      end
   end

   // If there are any active bindings, send it to the UI first
   if activeBindings.Count() != 0
      // Send active bindings to UI
      ...

      // Send active bindings to rest of game
      ...
   end
 end
end
```

One inefficiency of this system is that if there are multiple bindings that use the same key, their state will be checked multiple times every frame. This could definitely be optimized, but I chose not to for the purposes of simplicity and ease of implementation.

It also would be possible to extend this system further so that arbitrary functions could be mapped to the bindings, perhaps with the option of either registering through the UI or the

game. But in the particular case of the tower defense game, doing so would be an example of over-architecting a system. If it's not really necessary for a game to have so much flexibility, there's no reason to add further complication to the code for the input manager.

Mobile Input

Most smartphones and tablets have input mechanisms that differ from the traditional keyboard, mouse, or controller. Because of this, it is fitting to have a section focused on the input varieties typically encountered on mobile devices.

Note that simply because a mobile device has a large number of input options at its disposal does not mean that most games should use them. Mobile games must be very careful not to have an overly complex control scheme, which may turn away potential players.

Touch Screens and Gestures

Touch screens enable players to interact with the device's screen using their fingers. At a basic level, if only single finger tapping is allowed in the game, it could be implemented in the same manner as mouse clicking. However, most touch devices support the idea of **multi-touch**, which means multiple fingers can be active on the screen at the same time.

Some mobile games use multi-touch to implement a **virtual controller**, which has a virtual joystick and virtual buttons that the player can interact with. This is especially popular for mobile games that are a version of an existing console game, but it can sometimes be a frustrating experience for the player because a virtual controller doesn't have the same tactile feedback as an actual controller.

Something that can only be done with a touch screen, though, is a **gesture**, which is a series of touch actions that cause a specific behavior. An example of a gesture would be the very popular "pinch-to-zoom" on iOS devices, where the user can pinch her index finger and thumb together to zoom in, or pull them apart to zoom out.

Thankfully, both the iOS and Android SDKs provide ways to easily detect system-supplied gestures, including pinch-to-zoom. In iOS, there is a set of classes that inherits from `UIGestureRecognizer`, and a function to handle said gestures can easily be added to any program. On the Android side of things, there is the `android.gesture` package, which contains several libraries that do the same.

For customized gestures, things get a bit hairier. For iOS, really the only option is to create a custom subclass of `UIGestureRecognizer` that will go through the touch events and try to detect the gesture. But in the case of the Android SDK, a very handy "Gestures Builder" app can be used to create custom gestures.

You could potentially recognize gestures in any number of ways, but one popular algorithm is the **Rubine algorithm**, which was outlined in Dean Rubine's 1991 paper "Specifying gestures by example." Although the Rubine algorithm was originally developed for pen-based gestures, it can also be applied to either single-finger gestures or each finger of a multi-finger gesture.

To utilize the Rubine algorithm, we first must create a library of gestures. The way this library is created is by drawing the desired gestures in a test program. As the gesture is drawn, 14 mathematical properties of the gesture, called **features**, are calculated. These features include elements such as the total time to draw the gesture, the total distance travelled, the area of the gesture, the midpoint of the gesture, the initial position, and so on. Figure 5.4 illustrates several of the features in the original Rubine algorithm.

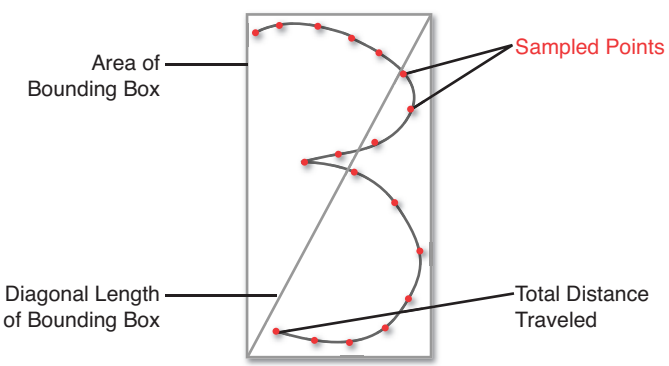


Figure 5.4 Some of the Rubine algorithm features.

Once the features have been calculated for a particular gesture, it is saved into the library. And once the library is complete, it can then be loaded by whichever program that wants to recognize said gestures. Then when the end user starts drawing a gesture, the particular gesture has the same mathematical features calculated for it. Once the user lifts the pen (or finger), the algorithm uses statistical regression to determine which gesture, if any, closely matches the user's gesture.

Although it's true that not many games will want to use advanced gestures, there certainly are some instances where they are used. For example, *Brain Age* on the Nintendo DS requires users to write in numbers for solutions to math problems. Another example might be a hangman game that's designed to teach kids how to properly write letters of the alphabet.

Accelerometer and Gyroscope

The accelerometer and gyroscope are the primary mechanisms for determining small movements on a mobile device. Early iOS and Android devices only had accelerometers, but now

essentially all mobile devices have both. Although we will discuss both alongside each other, they actually serve two distinct purposes.

The **accelerometer** detects acceleration along the coordinate space represented by the device itself at the origin. On iOS devices, the orientation of the axes is such that if you hold the device in a portrait orientation, the y-axis is up and down on the screen, the x-axis is left and right along the screen, and the z-axis is into and out of the screen. So, for example, if the device is in a portrait orientation and moves up, the acceleration would be registered along the y-axis. These axes are illustrated in Figure 5.5(a).

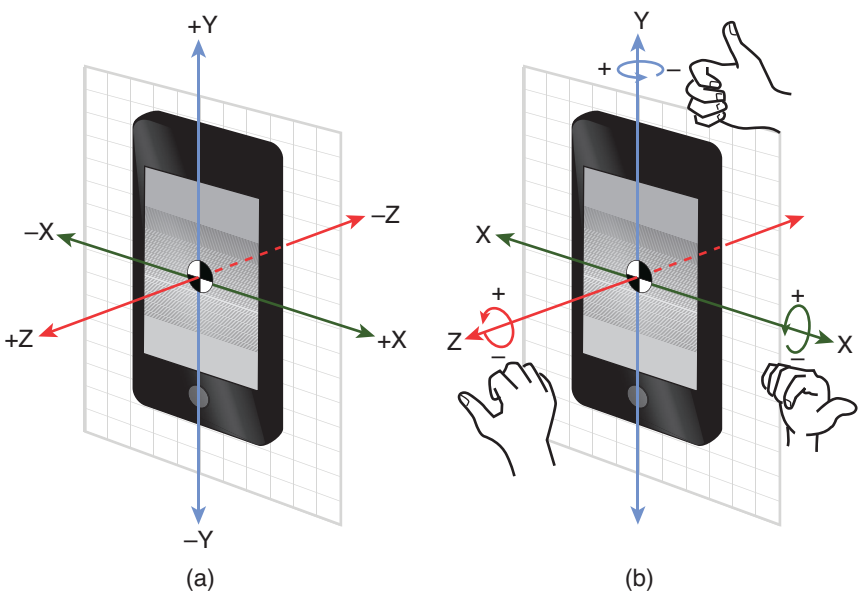


Figure 5.5 Accelerometer (a) and gyroscope (b).

Because an accelerometer measures acceleration, there is one constant that will always be applied to the device: gravity. This means that if the device is at rest, the accelerometer can roughly determine the orientation the device is held at based on which axis is being affected by gravity. So if the device is in a portrait orientation, gravity should be along the device's y-axis, whereas if it's in a landscape orientation, gravity would be along the device's x-axis.

A **gyroscope**, on the other hand, is designed to measure rotation about the device's principle axes, as illustrated in Figure 5.5(b). This means that it is possible to get very precise measurements of the device's orientation with the gyroscope. One non-gaming example that the gyroscope is well-suited for is a bubble level app that can determine whether or not a picture on the wall is level.

Trying to derive meaningful information from the raw accelerometer and gyroscope data can be challenging, so the device SDKs typically have higher-level functionality that aggregates it into easy-to-use information. For example, on iOS it is possible to use `CMDeviceMotion` to extract information such as the gravity vector, user acceleration, and **attitude** (or orientation of the device). And each of these components is provided with the usual mathematical objects—gravity and user acceleration are represented by 3D vectors, and attitude can be queried either as Euler angles, a rotation matrix, or a quaternion.

Other Mobile Input

Some types of games might expand and use additional methods of input that are only available on mobile devices. For example, an augmented reality game might use data from the camera and draw on top of objects in the physical world. Still other games might use GPS data to determine if there are any other players nearby who are playing the same game as you. Some of these input mechanisms will definitely carry over to wearable computing (such as virtual reality devices). However, even though there have been exciting developments in this area in recent years (such as the Oculus Rift), VR still has a little bit to go before it becomes a part of mainstream gaming.

Summary

Because every game must have input, it's a very fundamental concept. Both digital and analog devices exist in many different forms, but the techniques used to query them are often similar. With digital devices, we typically want to add some type of meaning to the binary "on" and "off" state. With analog devices, we often need to have some type of filtering to remove spurious input. But on a higher level, we want a way to abstract discrete input events into actions performed by the player, which is where event-based systems as well as gestures come into play.

Review Questions

1. What is the difference between a digital and an analog device? Give an example of each.
2. In input, what is a chord and what is a sequence?
3. Why are the "just pressed" and "just released" events necessary, as opposed to simply querying the state of the device?
4. What is analog filtering, and what problem does it try to solve?
5. What advantages does an event-based input system have over a polling system?

6. In a basic event-based system, how can you track which functions are registered to the event?
7. In a mouse-based game, where clicks can be processed by either the UI or the game world, what is one method you can use to ensure the UI has the first opportunity to process a click?
8. What problem does the Rubine algorithm attempt to solve? How, roughly speaking, does it function?
9. Outline the differences between an accelerometer and gyroscope.
10. What types of input mechanisms might an augmented reality game use, and for what purpose?

Additional References

Abrash, Michael. Ramblings in Valve Time. http://blogs.valvesoftware.com/abrash/. This blog, written by game industry legend Michael Abrash, features quite a few insights into the future of virtual and augmented reality.

Rubine, Dean. "Specifying gestures by example." In SIGGRAPH '91: Proceedings of the 18th annual conference on computer graphics and interactive techniques. (1991): 329–337. This is the original paper that outlines the Rubine gesture-recognition algorithm.

This page intentionally left blank

CHAPTER 6

SOUND

Though sometimes overlooked, sound is an important component of games. Whether it's to provide audio cues to gameplay situations or enhance the overall atmosphere, games lose a great deal without quality sound. To experience how important sound is, try playing one of your favorite games on mute—something is definitely lost if the sound is disabled.

This chapter first covers how source data is translated into "cues" that are played back by the game code. It then moves on to more advanced sound techniques, such as the Doppler effect, digital signal processing, and sound occlusion, that might be used in certain situations.

Basic Sound

At the most basic level, sound in a game could simply involve playing back standalone sound files at the appropriate points in time. But in many cases a single event does not necessarily correspond to a single sound. Suppose a game has a character who runs around the world. Every time the character's foot hits the ground, a footstep sound should play. If there were only one single footstep sound file played over and over again, it would very quickly be repetitive. At the very least, it would be preferable to have multiple footstep sounds that are randomly selected among every time the footstep is triggered.

An additional consideration is that there is a finite number of **channels**, or separate sounds, that can be played simultaneously. Imagine a game where a large number of enemies are running around the player—if all of them play their footstep sounds, it may very quickly use up all the available audio channels. Certain sounds are going to be far more important than enemy footstep sounds, and so there needs to be some sort of prioritization system, as well. All of these considerations, and more, very quickly lead to a situation where we need more information than what is stored in a sound file. Because of this, most games store an additional set of data that describes *how* and *in what circumstances* specific sound files should be played.

Source Data

Source data refers to the original audio files that are created by the sound designer using a tool such as Audacity (http://audacity.sourceforge.net). In the footstep scenario, there might be dozens of different source files, such as fs1.wav, fs2.wav, fs3.wav, and so on. A common approach is to store short sound effects as WAV files or another uncompressed file format, and store longer sounds, such as music or dialogue, in a compressed format such as MP3 or OGG.

When it comes to playing back these sound files in a game, there are two common approaches. It usually makes sense to preload short sound effects into memory, so when it's time to play the sound there is no time spent fetching the file from the disk. On the other hand, because compressed music or dialogue files are typically much larger in size, they are usually **streamed** off of a storage device. This means that as the sound file is being played, small segments of it are loaded on demand from the disk.

In order to load and play back source data, certain platforms have built-in sound libraries (such as CoreAudio on iOS). But for cross-platform support, OpenAL (http://kcat.strangesoft.net/openal.html) is a very popular solution.

Sound Cues

A **sound cue**, sometimes called a sound event, maps to one or more source data files. The sound cue is what is actually triggered by the game code—so rather than having code that directly plays the fs1.wav data file, there might be code that triggers a sound cue called

"footstep." The idea is that the sound cue can be a container for any number of source data files as well as store metadata about the sound as a whole.

For example, suppose there is an explosion cue. This cue should randomly trigger one of five different explosion WAV files. Additionally, because an explosion is something that can be heard from far away, there might be meta information that specifies the maximum distance the sound will be heard. It would be wise to also have a high priority assigned to the explosion cue, so even if all the channels are currently in use, the explosion is still audible. The basic layout of this cue is illustrated in Figure 6.1.

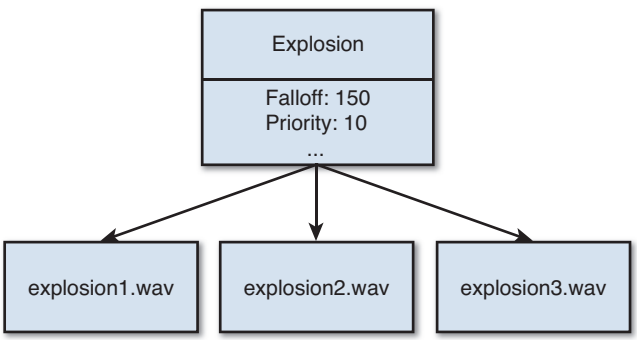


Figure 6.1 Explosion sound cue.

As for how this metadata might be stored for game use, there are a number of possibilities. One option is to store it in a JSON file, which might look something like this in the case of the explosion cue:

```
{
   "name": "explosion",
   "falloff": 150,
   "priority": 10,

   "sources":
   [
      "explosion1.wav",
      "explosion2.wav",
      "explosion3.wav"
   ]
}
```

The JSON file format is covered in further detail in Chapter 11, "Scripting Languages and Data Formats." But any format will work as long as it allows for enough flexibility. In any event, during the course of parsing the data, it can be directly mapped to a class implementation of the sound cue, like so:

```
class SoundCue
   string name
   int falloff
   int priority
   // List of strings of all the sources
   List sources;

   function Play()
      // Randomly select from sources and play
      ...
   end
end
```

The preceding system might be sufficient for many games, but for the footstep example it may not be enough. If the character can run across different surfaces—stone, sand, grass, and so on—different sound effects must play depending on what surface the character currently is on. In this case, the system needs some way to categorize the different sounds and then randomly select from the correct category based on the current surface. Or in other words, the system needs some way to **switch** between the different sets of source data based on the current surface.

The JSON file for this more advanced type of cue might look like this:

```
{
   "name": "footstep",
   "falloff": 25,
   "priority": "low",
   "switch_name": "foot_surface",

   "sources":
   [
      {
         "switch": "sand",
         "sources":
         [
            "fs_sand1.wav",
            "fs_sand2.wav",
            "fs_sand3.wav"
         ]
      },
      {
         "switch": "grass",
         "sources":
         [
            "fs_grass1.wav",
            "fs_grass2.wav",
            "fs_grass3.wav"
         ]
      }
   ]
}
```

It's certainly possible to just add this additional functionality to the `SoundCue` class. A more preferable solution, however, would be to have an `ISoundCue` interface that is implemented by both `SoundCue` and a new `SwitchableSoundCue` class:

```
interface ISoundCue
   function Play()
end

class SwitchableSoundCue implements ISoundCue
   string name
   int falloff
   int priority
   string switch_name

   // Hash Map that stores (string,List) pairs
   // For example ("sand", ["fs_sand1.wav", "fs_sand2.wav", "fs_sand3.
                           wav"])
   HashMap sources

   function Play()
      // Grab the current value for switch_name
      // Then lookup the list in the hash map and randomly play a sound
      ...
   end
end
```

In order for this implementation to work, there needs to be a global way to get and set switches. This way, the player run code can determine the surface and set the switch prior to actually triggering the footstep cue. Then the footstep cue's `Play` function can query the current value of the switch, and it will result in the correct type of footstep sound playing.

Ultimately, the key component when implementing a sound cue system is to allow for enough configurability to determine when and how sounds are played. Having as many dynamic parameters as possible will give the audio designers far more flexibility to create an immersive environment.

3D Sound

Although not an absolute requirement, 2D sound is typically positionless. This means that for most 2D games, the sound will just play equally out of the left and right speakers. Some 2D games might introduce some aspects of position into the sound, such as panning or distance-based volume reductions, but it will depend on the particular game.

For 3D sound, and 3D games by extension, position of the audio is extremely important. Most sounds are positional and have their characteristics change based on their relative

distance and orientation to the **listener**, or the virtual microphone that picks up the audio in the 3D world.

That is not to say that 3D games don't utilize 2D sounds; they are still used for elements such as user interface sounds, narration, ambient background effects, and music. But any sound effects that occur in the world itself are typically represented as 3D sounds.

Listeners and Emitters

Whereas the listener is what picks up the sound in the world, the **emitter** is what actually emits a particular sound effect. For example, if there is a fireplace that crackles, it will have a sound emitter placed at its location that plays back the crackle cue. Then based on the distance between the listener and the fireplace's sound emitter, it would be possible to determine how loud the sound should be. Similarly, the orientation of the emitter relative to the listener will determine which speaker the sound should be heard from. This is shown in Figure 6.2.

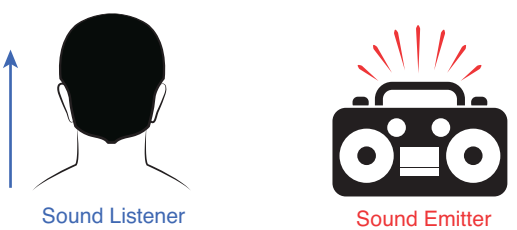


Figure 6.2 Sound listener and emitter; in this case, the sound effect should be heard on the right.

Because the listener picks up all the audio in the 3D world, determining the position and orientation of the listener is extremely important. If the listener is not set appropriately, the rest of the 3D sound system will not work properly—either sounds will be too quiet or too loud, or sounds might even come out of illogical speakers.

For many types of games, it makes sense to have the listener directly use the position and orientation of the camera. For example, in a first-person shooter, the camera is also where the player's point of reference is, so it makes perfect sense to have the listener use that same position and orientation. The same can be said for a cutscene where the camera is following a set path; the listener can just update as the camera updates.

Although it may be tempting to always have the listener track the position and orientation of the camera, there are certain types of games where this will not work properly. Take, for instance, a third-person action game. In such a game, the camera is following the main character at a distance. Suppose in one particular game, the camera's follow distance is 15 meters. What happens when a sound effect gets played right at the feet of the player character?

Well, if the listener is at the same position as the camera, the sound will seem like it's 15 meters away. Depending on the type of sound and its falloff range, it might result in a sound that's barely audible, which is odd because it's something that's happening right next to the player's character. Now if it's a sound effect that the player is triggering, we can usually identify these sounds and treat them differently. However, if it's a sound triggered by an enemy adjacent to the player, we don't have a simple recourse to the range problem.

One solution that might come to mind is to have the listener position and orientation set to that of the player character. Though this would solve the issue of sounds being 15 meters away, it introduces a major problem. Suppose an explosion occurs between the camera and the player. If the listener position and orientation inherits that of the player, this explosion will register as "behind" the listener, and therefore come out of the rear speakers. Furthermore, if the game allows the camera to rotate independently of the player, it may be possible to get into a scenario where a sound emitter that's to the left of the player (and therefore, the listener) is actually on the right part of the screen. This means that if an explosion occurs on the right part of the screen, it may actually come out of the left speaker. This seems incorrect because we expect an explosion on the right side of the screen to always come out of the right speaker, regardless of how the player is oriented.

In the end, for the melee-focused third-person game *Lord of the Rings: Conquest*, we arrived at a bit of a compromise. First, we based the orientation of the listener on the orientation of the camera, rather than on that of the player. Next, instead of placing the position of the listener precisely at the camera or precisely at the player, we placed it at a location *between* the two positions. Or in other words, the position of the listener was the result of an interpolation between the camera and player position, as in Figure 6.3. The percentage of the way between the two will vary depending on the game, but usually somewhere between 33% and 66% works well.

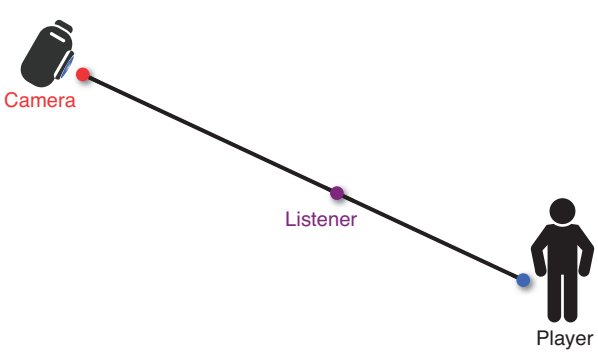


Figure 6.3 Listener position in a third-person game.

With this solution, although there still is the possibility that a sound between the camera and player will play from the rear speakers, the chance is reduced. And even though a sound at the player will not play at a distance of zero, it will play at a closer distance than it would were the listener at the camera. At the very least, there is no problem with the orientation of the listener, because that will be inherited from the camera. For some games, the correct decision is to always place the position of the listener at the camera. However, because our particular game heavily featured melee combat, this was not the case for us.

Falloff

Falloff describes how the volume of the sound decreases as it gets further away from the listener. It is possible to use any sort of function to represent falloff. However, because the unit of sound measurement, the **decibel** (dB), is a logarithmic scale, a linear decibel falloff would translate into logarithmic behavior. This sort of linear decibel function ends up often being the default method, but it certainly is not the only method.

As with point lights, it is also possible to add further parameters. Perhaps there could be an "inner" radius where the falloff function does not apply, or an "outer" radius after which the sound is automatically inaudible. Certain sound systems even allow the sound designer to create a multipart falloff function that exhibits different behavior at different distances.

Surround Sound

Certain platforms don't support the idea of surround sound—most mobile devices really only support stereo and that's about it. However, for PC or console games, it is possible to have more than just two speakers. In a **5.1 surround** system, there are a total of five regular speakers and one subwoofer, which is used for the low-frequency effects (LFE) channel.

The traditional 5.1 configuration is to place three speakers in the front and two in the back. The front channels are left, right, and center, whereas the back features only left and right. The position of the subwoofer relative to the listener doesn't matter, though placing it in the corner of a room will cause its low-frequency sounds to resonate more. Figure 6.4 illustrates a common 5.1 layout.

The neat thing about a 5.1 configuration is that it gives a much greater perception of the position of sounds. If a spaceship flies overhead in the game, the sound can pan from the back speakers to the front, so it feels like it's flying past you. Or in a horror game, it might be possible to determine if there's an enemy creeping up on you from the left or right.

But as much as it can add to the gaming experience, the reality is that a large percentage of players will not have a 5.1 surround configuration. Because of this, it is not really viable to create a game that relies on surround sound to function. Although it certainly will sound better on a $1000 home theater setup, the game still has to work on tinny built-in TV speakers or a bad pair of headphones.

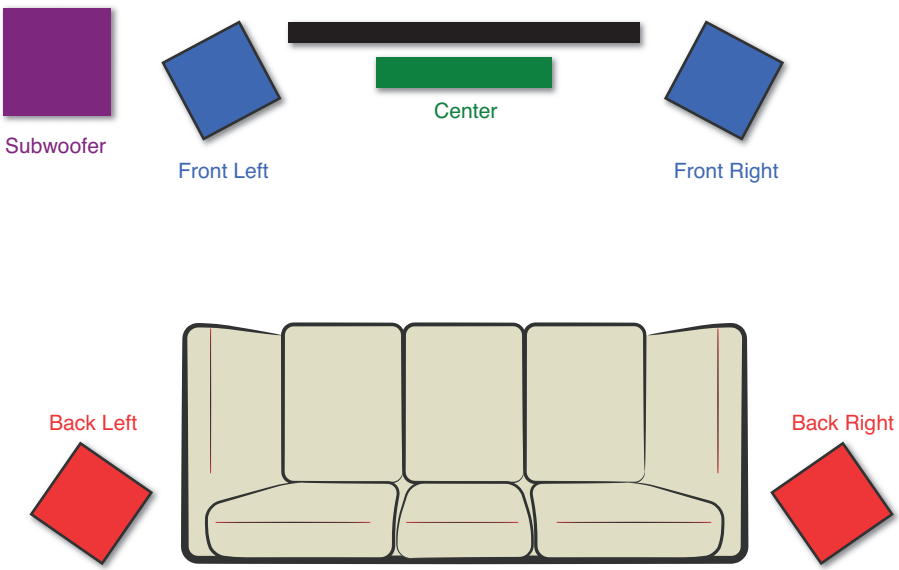


Figure 6.4 A standard 5.1 surround sound speaker configuration.

Although you can't separate front and back in a stereo configuration, this does not mean that the game still can't support positional sounds. The positions of both the listener and the emitters, as well as the falloffs and other parameters, will still affect the volume and left/right placement of the sound.

Digital Signal Processing

In a broad sense, **digital signal processing** (DSP) is the computational manipulation of a signal. In the realm of audio, DSP typically refers to taking a sound source file and modifying it on playback to sound differently. A relatively simple DSP effect would be to take a sound and increase or decrease its pitch.

It might seem like there would be no need to perform DSP effects on the fly, and rather bake such effects into the sound files that the game engine plays back. But the reason a runtime DSP effect is useful is because it can save a great deal of memory. Suppose that in a sword-fighting game, there are 20 different sound effects for swords clanging against each other. These sound effects were authored so they sound roughly like they are clanging in a wide open field. Now imagine that the game has a wide variety of locales where the sword fights can occur, in addition to the open field—it can be in a small cave, a large cathedral, and a great deal of other places.

The problem is that swords clanging in a small cave are going to sound dramatically different than in an open field. Specifically, when swords clang in a small cave, there is going to be a great deal of echo, or **reverb**. Without DSP effects, the only recourse would be to author a set of those 20 sword-clanging sounds for every single locale. If there's five such distinct locales, that means a fivefold increase, to a total of 100 sword-clanging sounds. Now if this is the case for all the combat sounds, not just the sword ones, the game may very quickly run out of memory. But if DSP effects are available, the same 20 sword-clanging sounds can be used everywhere; they just have to be modified by the effects to create the desired end result.

To actually implement DSP effects requires knowledge of linear systems and advanced mathematical operations such as the Fourier transform. Because of this, the implementation of these effects is well beyond the scope of this book. However, if you are interested in learning how to implement the effects covered in this section, check the references at the end of the chapter. In any event, even though we won't implement any of the effects, it is worthwhile to at least discuss what the most common effects represent.

Common DSP Effects

One of the most common DSP effects encountered in games is the aforementioned reverb. Any game that wants to re-create the echo of loud sounds in enclosed spaces will want to have some sort of implementation of reverb. A very popular open source reverb library is Freeverb3, which is available at http://freeverb3.sourceforge.net. Freeverb3 is an **impulse-driven system**, which means in order for it to apply the reverb effect to an arbitrary sound, it needs a source sound file that represents a specific test sound playing in the desired environment.

Another DSP effect that gets a large amount of use is the **pitch shift**, especially if Doppler shift (covered later in the chapter) is desired. A pitch shift increases or decreases the pitch of a sound by altering the frequency. Although a Doppler shift would be the most common use, another example might be in a car racing game, where the pitch of the engine might adjust depending on the number of RPMs.

Most of the other DSP effects that see use in games usually modify the range of frequencies or decibel levels output. For example, a **compressor** narrows the volume range so that very quiet sounds have their volume amplified while very loud sounds have their volume reduced. This might be used to try to normalize volume levels if there are wildly different levels in different sound files.

Another example is a **low-pass filter**, which reduces the volume of sounds with frequencies (and therefore pitches) higher than the cutoff point. This is commonly used in games that implement a "shell shock" effect when an explosion occurs near the player. To sell the effect, time is dilated, a low-pass filter is applied, and a distinct ringing sound is played.

There are quite a few other effects that might find use in a game, but these four are some of the most common effects you'll come across.

Marking Regions

It will be rare that an effect—especially with the reverb effect—will be uniformly applied to an entire level. Instead, it likely is necessary to apply the reverb only in certain regions of the level. For example, if a level has both outdoor areas and a small cave, the reverb might only be enabled inside the cave. A region can be marked in a number of ways, but one of the simplest is to use a convex polygon that lies on the ground plane.

Recall that a **convex polygon** is one where all the vertices of the polygon point outward. More specifically, all of the interior angles of a convex polygon are less than 180°. Figure 6.5 demonstrates both a convex and a concave polygon.

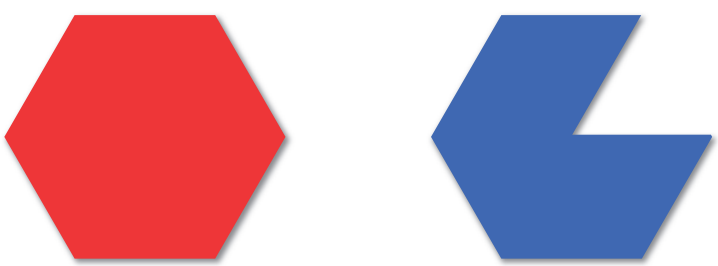


Figure 6.5 Convex (a) and concave (b) polygons.

The reason why a convex polygon is preferred is that given a point, it is relatively straightforward to determine whether or not that point is inside or outside the convex polygon. So, in our case, the convex polygon represents the region where the reverb effect should be applied. Given the position of the player, it will then be possible to determine whether the player is inside or outside that region; if the player is inside the region, the reverb should be enabled, and vice versa. The algorithm for determining whether or not a point is inside a convex polygon is covered in detail in Chapter 7, "Physics," as it has many applications beyond just DSP effects.

However, we don't want to just suddenly switch on the reverb as soon as the player enters the cave. Otherwise, the effect will be pretty jarring. This means that once the player crosses over into a marked region, we want to slowly interpolate between the reverb being off and on, to make sure that the transition feels organic.

Note that there is one big caveat to using convex polygons to mark DSP regions. If it's possible to have areas above or below other areas in the level, and those areas have different DSP effect

needs, this approach will not work. So, for example, if there is a tunnel that goes under a grassy field, a convex polygon for the tunnel would also flag the areas above the tunnel as needing the DSP effect. If this problem must be solved, instead of a convex polygon you will have to use a bounding volume of some kind, which is also covered in Chapter 7.

Other Sound Topics

Although this chapter has covered a great deal of topics related to sound, there still are a couple items that didn't really warrant entire sections but at least should be mentioned in some capacity.

Doppler Effect

If you stand on a street corner and a police car drives toward you with its sirens on, the pitch of the sound increases as the police car approaches. Conversely, once the police car passes by, the pitch of the sound decreases. This is a manifestation of the **Doppler effect**, which is demonstrated in Figure 6.6.

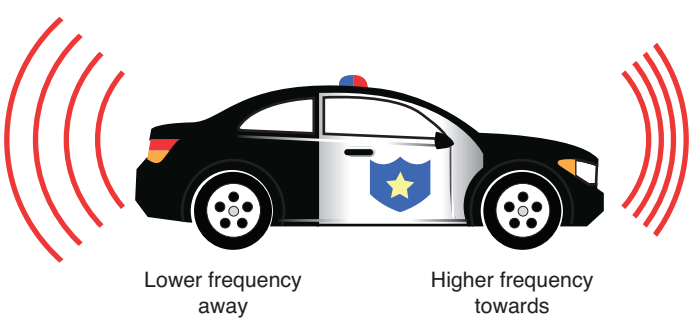


Figure 6.6 Doppler effect on a police car siren.

The Doppler effect (or Doppler *shift*) occurs because sound waves take time to travel through the air. As the police car gets closer and closer to you, it means each successive wave will arrive a little bit earlier than the previous one. This causes an increase in frequency, which leads to the heightened pitch. At the exact moment the police car is next to you, the true pitch of the sound is audible. Finally, as the car begins to travel away from you, the waves will take increasingly longer to get to you, which leads to a lowered pitch.

It's interesting to note that the Doppler effect doesn't only apply to sound waves, but it applies to any kind of wave. Most notably, for light waves, a redshift occurs if the object moves further

away and a blueshift occurs if the object moves closer. But in order for the shift to be noticeable, the object must either be travelling at exceptionally fast speeds or be exceptionally far away. This won't happen at mundane speeds on earth, so the shifts are usually only noticeable in astronomy.

In games, a dynamic Doppler effect will usually only be applied for faster moving objects, such as vehicles. It technically could also apply to something like a bullet, but because they're travelling so quickly, it's typically preferred to just play a canned bullet flyby sound. Because Doppler shift results in the pitch increasing or decreasing, it can only be dynamically implemented if a DSP pitch shift effect is available.

Sound Occlusion and Obstruction

Imagine you're living in a college dorm. Being the diligent student you are, you're hard at work studying your copy of *Game Programming Algorithms and Techniques*. Suddenly, a party starts up down the hall. The music is blaring, and even though your door is closed, the sound is so loud it's distracting. You recognize the song, but something about it sounds different. The most noticeable difference is that the bass is dominant and the higher frequency sounds are muffled. This is **sound occlusion** in action and is illustrated in Figure 6.7(a).

Sound occlusion occurs when sound does not have a direct path from emitter to listener, but rather must travel through some material to reach the listener. The predominant result of sound occlusion is that a low-pass filtering occurs, which means the volume of higher frequency sounds is reduced. That's because lower frequency waves have an easier time passing through surfaces than higher frequency ones. However, another outcome of sound occlusion is an overall reduction in volume levels of all the sounds.

Similar but different is the idea of **sound obstruction** (also known as diffraction). With sound obstruction, the sound may not have a straight line path, but is able to travel around the obstacle, as shown in Figure 6.7(b). For example, if you yell at someone on the other side of a pillar, the sound will diffract around the pillar. One interesting result of sound obstruction is that the split waves may arrive at slightly different times. So if the waves travel toward the pillar and split, the ones on the left split may arrive a little earlier than those on the right split. This means that someone standing on the other side of the pillar may experience interaural time difference, because she will hear the sound in each ear at slightly different times.

One way to detect both occlusion and obstruction is to construct a series of vectors from the emitter to an arc around the listener. If none of the vectors can get to the listener without passing through another object first, it's occlusion. If some of the vectors get through, then it's obstruction. Finally, if all of the vectors get through, it is neither. This approach is known as **Fresnel acoustic diffraction** and is illustrated in Figure 6.7(c).

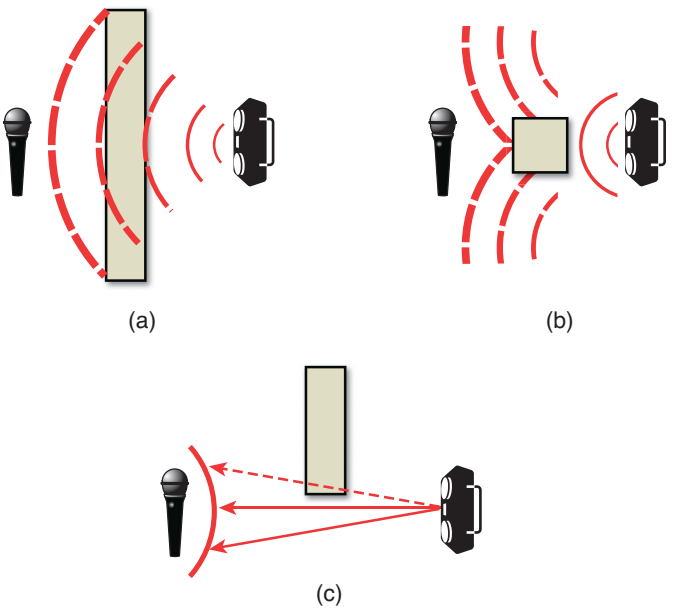


Figure 6.7 Sound occlusion (a), sound obstruction (b), and Fresnel acoustic diffraction (c).

Implementing this requires being able to determine whether or not a vector intersects with any object in the world, which we have not covered yet. However, many types of such intersections are covered in Chapter 7. Specifically, we can use ray casts to determine whether the path from emitter to listener is blocked.

Summary

Sound is an important part of any game. Typically, we want to have some sort of sound cue system that allows the use of metadata that describes how and when different sound files will be played. DSP effects such as reverb can be applied to sound files on playback. For 3D sounds, we care about the position of the listener in the world as well as the sound emitters. Finally, some games may have to implement the Doppler effect for fast-moving objects or sound occlusion/obstruction, depending on the environment.

Review Questions

1. Describe the difference between source sound data and the metadata associated with it.
2. What advantage do "switchable" sound cues provide over regular sound cues?
3. What are listeners and sound emitters?
4. When deciding on the position of the listener in a third-person action game, what problems must be taken into consideration?
5. What type of scale is the decibel scale?
6. What is digital signal processing? Give examples of three different audio DSP effects.
7. Why is it useful to be able to mark regions where DSP effects should be played?
8. What drawback does using a convex polygon have for DSP regions?
9. Describe the Doppler effect.
10. What are the differences between sound occlusion and sound obstruction?

Additional References

Boulanger, Richard and Victor Lazzarini, Eds. *The Audio Programming Book*. Boston: MIT Press, 2010. This book is a solid overview of the basics of audio signal processing and includes a large number of code examples to reinforce the concepts.

Lane, John. *DSP Filter Cookbook*. Stamford: Cengage Learning, 2000. This book is a bit more advanced than the prior one, but it has implementations of many specific effects, including compressors and low-pass filters.

This page intentionally left blank

CHAPTER 7

PHYSICS

Although not every game needs physics, the ones that do implement collision detection and movement at a minimum. Collision detection enables the game to determine whether or not two game objects intersect with each other. A variety of different algorithms can be used to detect these collisions, and the first part of this chapter covers some of the more basic ones.

The movement part of physics takes into account force, acceleration, mass, and the other properties inherit in classical mechanics to determine where objects go from frame to frame. The only way this can be done is by using calculus, specifically the numeric integration methods covered in the second part of this chapter.

Planes, Rays, and Line Segments

Before we broach the topic of how we might represent characters and other game objects in the world for the purposes of collision detection, we need to discuss a couple of additional mathematical topics useful for physics systems. You may have learned about these objects in a high school algebra course, but now that you have familiarity with linear algebra we can approach them from a slightly different perspective.

Planes

A **plane** is a flat, two-dimensional surface that extends infinitely, much like a line is a one-dimensional object that extends infinitely. In games, we may commonly use planes as abstractions for the ground or walls, though there are also other uses. A plane can be represented in a few different ways, but game programmers prefer

$$P \cdot \hat{n} + d = 0$$

where P is any arbitrary point on the plane, $\hat{n}$ is the normal to the plane, and the absolute value of d is the minimum distance between the plane and the origin.

Recall that a triangle is guaranteed to lie on a single plane. One reason the preceding plane representation is popular is because given a triangle, it is possible to quickly construct the equation of the plane that triangle is on. Once we've computed both $\hat{n}$ and d, we can then test if any arbitrary point, P, also satisfies the equation, in which case this arbitrary point is also on the plane.

Suppose we have the triangle $\triangle ABC$ expressed in a clockwise winding order (see Figure 7.1). In order to create the equation of the plane the triangle is on, we first need to calculate the normal of the triangle. Hopefully you remember that you can take the cross product between two edge vectors to determine the normal of the triangle—if you don't, you should review Chapter 3, "Linear Algebra for Games," before you go any further.

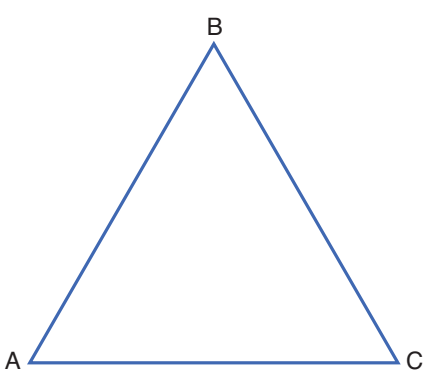


Figure 7.1 Triangle ABC in a clockwise winding order.

Don't forget that the order of the cross product matters, so it's important to look at the winding order. Because $\triangle ABC$ has a clockwise winding order, as illustrated in Figure 7.1, we want to construct vectors from A to B and B to C. Once we have these two vectors, we take the cross product between them and normalize the result to end up with $\hat{n}$.

Once we have $\hat{n}$, we need a point somewhere on the plane to solve for d, because

$$d = -P \cdot \hat{n}$$

Luckily, it turns out we have three points that we absolutely know are on the same plane as the triangle: A, B, and C. We can dot product any of these points with $\hat{n}$ and we will end up with the same value of d.

All the points on the plane will yield the same d value because it's a scalar projection: $\hat{n}$ is normalized and P is unnormalized, so what we get is the minimum distance from the origin to the plane in the direction of $\hat{n}$. This minimum distance will be the same regardless of which vertex we select, because all of them are on the same plane.

Once we've solved for $\hat{n}$ and d, we can then store these values in our `Plane` data structure:

```
struct Plane
   Vector3 normal
   float d
end
```

Rays and Line Segments

A **ray** starts at a specific point and extends infinitely in one particular direction. In games, it's typical to represent a ray using a parametric function. Recall that a parametric function is one that is expressed in terms of an arbitrary parameter, generally by the name of t. For a ray, the parametric function is

$$R(t) = R_0 + \vec{v}t$$

where R_0 is the starting point of the plane and $\vec{v}$ is the direction the ray is travelling in. Because a ray starts at a specific point and extends infinitely, for this representation of a ray to work properly, t must be greater than or equal to 0. This means that when t is 0, the parametric function will yield the starting point R_0.

A **line segment** is similar to a ray, except it has both a start and end point. We can actually use the same exact parametric function to represent a line segment. The only difference is we now also have a maximum value of t, because a line segment has a discrete end point.

Technically, a **ray cast** involves firing a ray and checking whether or not the ray intersects with one or more objects. However, what is confusing is that most physics engines, including Havok

and Box2D, use the term "ray cast" when in actuality they are using a line segment, *not* a ray. The reason for this is that game worlds usually have constraints, so it typically makes sense to use the more constrained line segment.

I admit that this terminology might be confusing. However, I wanted to be consistent with the terminology I've seen used in the game industry. So keep in mind as you continue reading the book that whenever you see the term "ray cast," *we are actually using a line segment*.

So what are these ray casts used for anyways? It turns out they're actually quite ubiquitous in 3D games. One very common example is firing a bullet that travels in a straight line. Although some games do simulate actual projectiles for their bullets, it may be appropriate to perform a ray cast instead, because the bullet travels so quickly. But there are other uses of a ray cast. Some examples include a reticule that changes red or green depending on friend or foe (covered in Chapter 10, "User Interfaces"), an AI that needs to determine whether an enemy is visible, Fresnel acoustic diffraction (covered in Chapter 6, "Sound"), and using the mouse to click objects in a 3D world (covered in Chapter 8, "Cameras"). All of these scenarios and many more use ray casts in one way or another.

Because a ray cast request uses a line segment, at a minimum we need two parameters—the start point and the end point of the ray cast:

```
struct RayCast
   Vector3 startPoint
   Vector3 endPoint
end
```

To convert `startPoint` and `endPoint` to the parametric form of a line segment is fairly straightforward. R_0 is identical to `startPoint`, and $\vec{v}$ is `endPoint - startPoint`. When the conversion is done in this manner, the t values from 0 to 1 will correspond to the range of the ray cast. That is to say, a t of 0 will yield `startPoint` whereas a t of 1 will yield `endPoint`.

Collision Geometry

In modern 3D games, it's very common for human characters to be composed of 15,000+ polygons. When a game needs to determine whether or not two characters collide, it wouldn't be terribly efficient to check against all these triangles. For this reason, most games utilize simplified **collision geometry**, such as spheres and boxes, to test for collisions. This collision geometry isn't drawn to the screen, but only used to check for collisions efficiently. In this section we cover some of the most common collision geometries used in games.

It's also worth noting that it's not uncommon for games to have multiple levels of collision geometry per object. This way, a simple collision check (such as with spheres) can be performed first to check whether there's any possibility whatsoever that two objects collided. If the simple

collision check says there might be a collision, we can then perform calculations with more complex collision geometry.

Bounding Sphere

The simplest type of collision geometry is the bounding sphere (or in 2D games, a bounding circle). A sphere can be defined with only two variables—a vector representing the center of the sphere, and a scalar for the radius of the sphere:

```
class BoundingSphere
   Vector3 center
   float radius
end
```

As illustrated in Figure 7.2, certain objects, such as an asteroid, fit well with bounding spheres. But other types of objects, including humanoid characters, end up with a lot of empty space between the edge of the character and the sphere. This means that for many types of objects, bounding spheres can have a large number of **false positives**. This happens when two game objects have intersecting collision geometry, yet the game objects themselves are not actually intersecting. False positives can be really frustrating for the player, because it will be apparent if the two game objects aren't actually intersecting when a collision is detected.

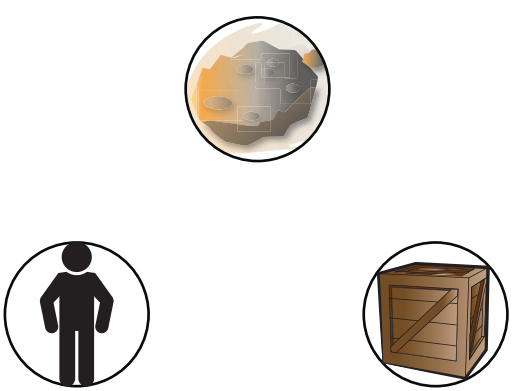


Figure 7.2 Bounding spheres for different objects.

Because of the relative inaccuracy of bounding spheres for most game objects, it usually is not appropriate to use bounding spheres as the only collision geometry. But the big advantage of bounding spheres is that instantaneous collision detection between two spheres is very simple, so this representation may be a great candidate for a first-line collision check.

Axis-Aligned Bounding Box

For a 2D game, an **axis-aligned bounding box** (or AABB) is a rectangle where every edge is parallel to either the x-axis or y-axis. Similarly, in 3D an AABB is a rectangular prism where every side of the prism is parallel to one of the coordinate axis planes. In both 2D and 3D, an AABB can be represented by two points: a minimum and a maximum point. In 2D, the minimum corresponds to the bottom-left point whereas the maximum corresponds to the top-right point.

```
class AABB2D
   Vector2 min
   Vector2 max
end
```

Because an AABB must keep its sides parallel to the coordinate axes, if an object is rotated the AABB may have to stretch, as demonstrated in Figure 7.3. But for 3D games, humanoid characters will typically only rotate about the up axis, and with these types of rotations the bounding box for the character may not change at all. Because of this, it's very common to use AABBs as a basic form of collision representation for humanoid characters, especially because collision detection between AABBs, as with spheres, is not particularly expensive.

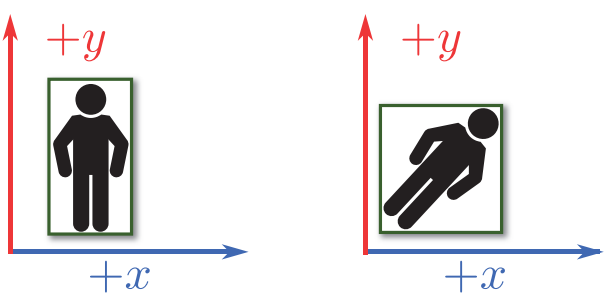


Figure 7.3 Axis-aligned bounding boxes for different orientations of a character.

Oriented Bounding Box

An **oriented bounding box** is similar to an axis-aligned bounding box, but with the parallel restrictions removed. That is to say, it is a rectangle (in 2D) or a rectangular prism (in 3D) with sides that may not necessarily be parallel to the coordinate axes/planes. The big advantage of an OBB is that it can rotate as the game object rotates; therefore, regardless of the orientation of the game object, the OBB will have the same degree of accuracy. But this increased accuracy comes with a price; compared to an AABB, the collision calculations for an OBB are far more complicated.

An OBB can be represented in a game in multiple ways, including eight vertexes or six planes. But the OBB calculations are complex enough that this book does not outline any algorithms

related to them. However, if you'd like to use them in your game, take a look at *Real-time Collision Detection* in the references at the end of this chapter.

Capsule

In 2D, a **capsule** can be thought of as an AABB with two semicircles (half circles) attached to the top and bottom. It's called a capsule because it looks very similar to a medicine capsule. If we expand the capsule to 3D, it becomes a cylinder with two hemispheres attached to the top and bottom. Capsules are also a popular form of collision representation for humanoid characters because they are slightly more accurate than an AABB, as illustrated in Figure 7.4(a).

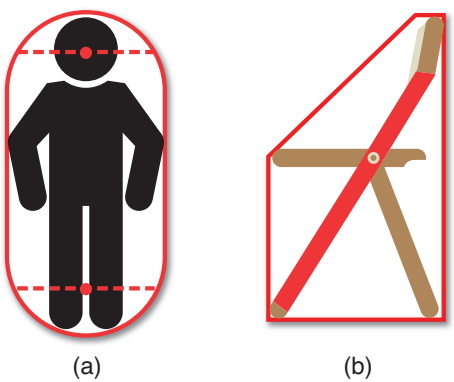


Figure 7.4 Humanoid surrounded by a capsule (a), and a chair surrounded by a convex polygon (b).

Another way to think of a capsule is as a line segment that has a radius, which actually corresponds to the representation of it in a game engine:

```
struct Capsule2D
   Vector2 startPoint
   Vector2 endPoint
   float radius
end
```

Convex Polygons

Another option is to use an **arbitrary convex polygon** (or in 3D, a **convex hull**) to represent the collision geometry. As you might expect, using an arbitrary convex polygon will be less efficient than the other options, but it also can be more accurate. Figure 7.4(b) shows a chair represented by a convex polygon. Although there still will be many false positives, the number of false positives is less than in alternative representations.

List of Collision Geometries

One last option to increase the accuracy of collision checks is to have an actual **list of collision geometries** to test against. So in the case of a human, we could have a sphere for the head, an AABB for the torso, and maybe some convex polygons for the hands/legs, and so on. By using several different types of collision geometries, we can almost completely eliminate false positives.

Although testing against a list of collision geometries is going to be faster than testing against the actual triangles of a model, it's slow enough that you may not want to do it as a primary collision check. In the case of a humanoid, you should do a first-pass collision check against an AABB or capsule. Then only if that collision check passes should you bother checking against the list of collision objects. This, of course, assumes you even need that level of accuracy—you might need such accuracy to test for bullets colliding with the player, but you don't need it to see whether or not the player is trying to walk into the wall.

Collision Detection

Now that we've covered some of the most common types of collision geometries used in games, we can take a look at actually testing for intersections between said objects. Any game that needs to respond to two objects colliding with each other must use some type of collision detection algorithm. Although some might find the amount of math in this section a bit off-putting, know that it is absolutely something that is used in a large number of games. Therefore, it's very important to understand this math—it's not covered just for the sake of being covered. This section won't cover every permutation of every geometry colliding with every other geometry, but it does go over the most common ones you might use.

Sphere versus Sphere Intersection

Two spheres intersect if the distance between their center points is less than the sum of their radii, as shown in Figure 7.5. However, computing a distance requires the use of a square root, so to avoid the square root, it is common instead to check the distance *squared* against the sum of the radii *squared*.

The algorithm for this is only a couple of lines of code, as shown in Listing 7.1. It's also extremely efficient, which is what makes using spheres a very popular basic collision detection option.

Listing 7.1 Sphere versus Sphere Intersection

```
function SphereIntersection(BoundingSphere a, BoundingSphere b)
   // Construct a vector between centers, and get length squared
   Vector3 centerVector = b.center - a.center
   // Recall that the length squared of v is the same as v dot v
```

```
    float distSquared = DotProduct(centerVector, centerVector)

    // Is distSquared < sum of radii squared?
    if distSquared < ((a.radius + b.radius) * (a.radius + b.radius))
       return true
    else
       return false
    end
  end
```

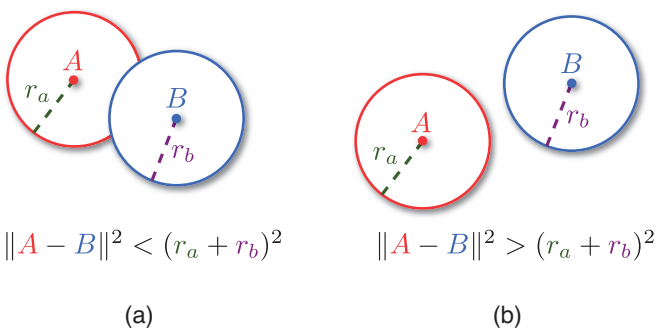


Figure 7.5 Two spheres intersect (a) and do not intersect (b).

AABB versus AABB Intersection

As with spheres, AABB intersection is not very expensive, even for 3D games. It's easier to visualize the algorithm with 2D AABBs, though, so that's what is covered here.

When checking for intersection between two 2D AABBs, rather than trying to test the cases where the two AABBs do intersect, it's easier to test the four cases where two AABBs definitely *cannot* intersect. These four cases are illustrated in Figure 7.6.

If any of the four cases are true, it means the AABBs do not intersect, and therefore the intersection function should return false. This means that the return statement should be the logical negation of the four comparison statements, as in Listing 7.2.

Listing 7.2 AABB versus AABB Intersection

```
function AABBIntersection(AABB2D a, AABB2D b)
   bool test = (a.max.x < b.min.x) || (b.max.x < a.min.x) ||
               (a.max.y < b.min.y) || (b.max.y < a.min.y)

   return !test
end
```

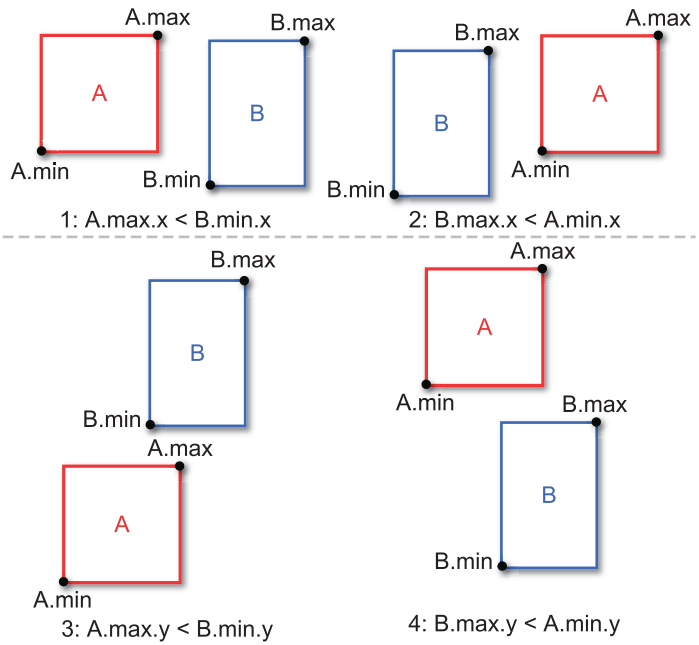


Figure 7.6 The four cases where two 2D AABBs definitely *cannot* intersect.

It's worth mentioning that AABB-AABB intersection is actually a very special application of an algorithm known as the separating axis theorem. In the general form, this theorem can actually be used on any type of convex polygon, though the details of it are beyond the scope of this book.

Line Segment versus Plane Intersection

Checking whether a line segment collides with a plane is a relatively common collision test in games. In order to understand how the algorithm for this works, it is useful to derive the formula using some linear algebra. First, we have the separate equations of a line segment and a plane:

$$R(t) = R_0 + \vec{v}t$$
$$P \cdot \hat{n} + d = 0$$

We want to find out if there is a value t such that $R(t)$ is a point on the plane. In other words, we want to determine if there is a possible value of t where $R(t)$ satisfies the P part of the plane equation. This can be done by substituting $R(t)$ for P:

$$R(t) \cdot \hat{n} + d = 0$$
$$\left(R_0 + \vec{v}t\right) \cdot \hat{n} + d = 0$$

Then it is just a matter of solving for t:

$$R_0 \cdot \hat{n} + (\vec{v} \cdot \hat{n})t + d = 0$$
$$(\vec{v} \cdot \hat{n})t = -(R_0 \cdot \hat{n} + d)$$
$$t = \frac{-(R_0 \cdot \hat{n} + d)}{\vec{v} \cdot \hat{n}}$$

Remember that for a line segment, the start point corresponds to $t = 0$, whereas the end point corresponds to $t = 1$. So when we solve for t, if we get a value that's outside of that range, it should be ignored. Specifically, a negative value means that the line segment is actually pointing *away* from the plane, as in Figure 7.7(a).

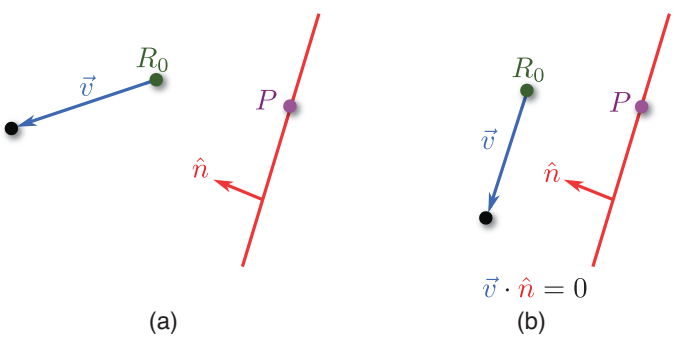


Figure 7.7 Line segment pointing away from the plane (a) and parallel to the plane (b).

Also notice how if the dot product between $\vec{v}$ and $\hat{n}$ is zero, an illegal division by zero will occur. Recall that a dot product of zero means the two vectors are perpendicular. In this particular case, this means that $\vec{v}$ is parallel to the plane and therefore cannot intersect with it, as in Figure 7.7(b). The only exception to this would be if the line segment is on the plane itself. In any event, this possibility must be taken into account to prevent division by zero.

As an added bonus, if the line segment and plane do intersect, we can simply substitute the t value to get the actual point of intersection, as shown in Listing 7.3.

Listing 7.3 Line Segment vs. Plane Intersection

```
// Return value is this struct
struct LSPlaneReturn
   bool intersects
   Vector3 point
end
```

```
// Remember "ray cast" actually refers to a line segment!
function LSPlaneIntersection(RayCast r, Plane p)
   LSPlaneReturn retVal
   retVal.intersects = false

   // Calculate v in parametric line segment equation
   Vector3 v = r.endPoint - r.startPoint

   // Check if the line segment is parallel to the plane
   float vDotn = DotProduct(v, p.normal)
   if vDotn is not approximately 0
      t = -1 * (DotProduct(r.startPoint, p.normal) + p.d)
      t /= vDotn

      // t should be between startPoint and endPoint (0 to 1)
      if t >= 0 && t <= 1
         retVal.intersects = true

         // Calculate the point of intersection
         retVal.point = r.startPoint + v * t
      end
   else
      // Test if r.startPoint is on the plane
      ...
   end

   return retVal
end
```

Line Segment versus Triangle Intersection

Suppose you want to check if a line segment representing a bullet collides with a particular triangle. The first step to solve this problem is to calculate the plane that the triangle lies on. Once you have that plane, you can see whether or not the line segment intersects with it. If they do intersect, you will have a point of intersection on the plane of the triangle. However, because a plane is infinite, we need to determine whether this point of intersection is inside or outside a triangle.

If the triangle $\triangle ABC$ is expressed in a clockwise winding order, the first step is to construct a vector from A to B. Then, construct a vector from point A to point P, the location that is being tested. If the rotation from vector $\overrightarrow{AB}$ to $\overrightarrow{AP}$ is clockwise, P is on the interior of that side of the triangle. This test can then be repeated for every other side ($\overrightarrow{BC}$ and $\overrightarrow{CA}$), and as long as it is clockwise in every instance, it is interior of every side, and therefore on the interior of the polygon as a whole. This is process is shown in Figure 7.8.

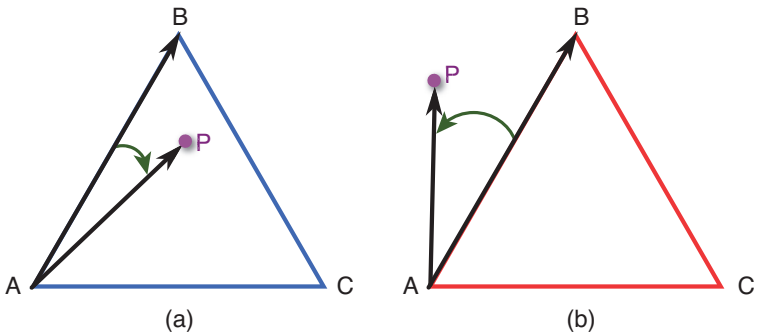


Figure 7.8 Point inside a triangle (a) and outside a triangle (b).

But how do we determine whether the rotation is clockwise or counterclockwise? Notice how in Figure 7.8(a), if we were to compute $\overrightarrow{AB} \times \overrightarrow{AP}$ in a right-handed coordinate system, the resultant vector would go into the page. Recall that in a clockwise winding order in a right-handed coordinate system, the triangle's normal also goes into the page. So in this case, the direction of the cross product result is the same as the triangle's normal vector. If two normalized vectors have a positive dot product, that means they are facing in roughly the same direction. So if you normalize the result of $\overrightarrow{AB} \times \overrightarrow{AP}$ and dot it with the normal, if it's positive, then P is on the interior of $\overrightarrow{AB}$.

This algorithm can then be applied to all the other sides of the triangle. It turns out that this works not only for triangles, but for *any* convex polygon that lies on a single plane. The pseudocode for this is provided in Listing 7.4.

Listing 7.4 Determining Whether a Point Is Inside a Convex Polygon

```
// This function only works if the vertices are provided in a
// clockwise winding order and are coplanar.
function PointInPolygon(Vector3[] verts, int numSides,
                        Vector3 point)
   // Calculate the normal of the polygon
   Vector3 normal = CrossProduct(Vector3(verts[1] - verts[0]),
                                 Vector3(verts[2] - verts[1])
   normal.Normalize()

   // Temporary variables
   Vector3 side, to, cross

   for int i = 1, i < numSides, i++
      // FROM previous vertex TO current vertex
      side = verts[i] - verts[i - 1]
      // FROM previous vertex TO point
      to = point - verts[i - 1]
```

```
        cross = CrossProduct(side, to)
        cross.Normalize()

        // This means it's outside the polygon
        if DotProduct(cross, normal) < 0
            return false
        end
    loop

    // Have to test the last side, which is from last vertex to first
    side = verts[0] - verts[numSides - 1]
    to = point - verts[numSides - 1]
    cross = CrossProduct(side, to)
    cross.Normalize()

    if DotProduct(cross, normal) < 0
        return false
    end

    // We're inside all sides
    return true
end
```

Sphere versus Plane Intersection

In a game where a ball can collide with a wall, in order to accurately model the collision it might be desirable to use sphere-plane intersection. Given that we only store $\hat{n}$ and d for a plane, the easiest way to perform this collision check is to calculate d for a hypothetical second plane with the same $\hat{n}$ that contains the center of the sphere. If the absolute value of the difference between this new plane's d and the actual plane's d is less than the sphere's radius, it means they must intersect. This process is illustrated in Figure 7.9.

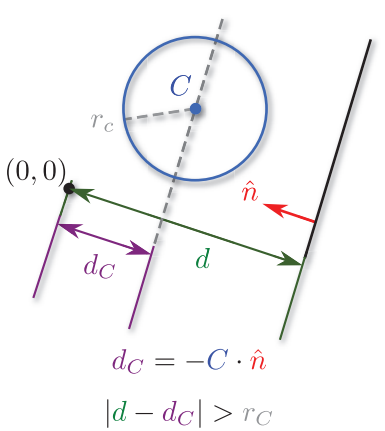


Figure 7.9 Sphere-plane intersection in a case where they don't intersect.

As with sphere-sphere intersection, the code for sphere-plane intersection is not that complex, and is outlined in Listing 7.5.

Listing 7.5 Sphere versus Plane Intersection

```
function SpherePlaneIntersection(BoundingSphere s, Plane p)
   // Calculate d for the plane with normal p.normal and
   // point on plane s.center
   float dSphere = -DotProduct(p.normal, s.center)

   // Check if we're within range
   return (abs(d - dSphere) < s.radius)
  end
```

Swept Sphere Intersection

To this point, we have covered **instantaneous** collision detection algorithms. This means the algorithm checks to see if the two objects collide on the current frame. Although this can work in many cases, there are some instances where it won't.

If a bullet is fired at a piece of paper, it's unlikely there will be a precise frame where the bullet and the paper intersect with each other. That's because the bullet is travelling so fast and the paper is so thin. This problem is fittingly known as the **bullet-through-paper problem**, and is illustrated in Figure 7.10. In order to solve this problem, a form of **continuous collision detection** (CCD) is necessary. Generic CCD is a topic that's well beyond the scope of this book, but it's worth covering at least one algorithm that does not have to worry about bullet-through-paper.

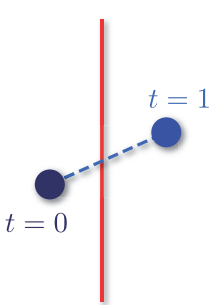


Figure 7.10 Bullet-through-paper problem.

In **swept sphere intersection**, there are two moving spheres. The inputs are the positions of both spheres during the last frame ($t = 0$) and the current frame ($t = 1$). Given these values, we can determine whether or not the two spheres collided at any point between the two frames.

So unlike the instantaneous sphere-sphere intersection, it won't miss out on any intersections that occur between frames. This is shown in Figure 7.11.

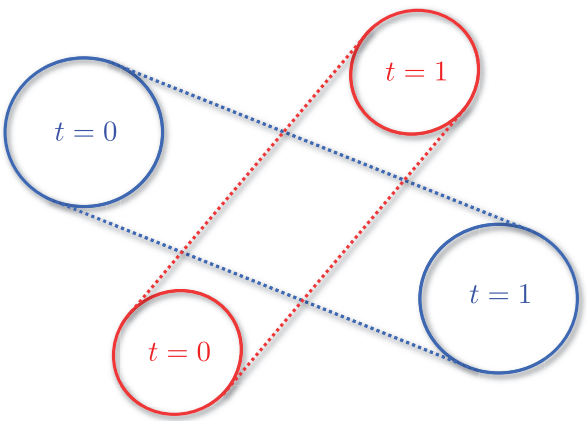


Figure 7.11 Swept sphere intersection.

You might notice that a swept sphere looks a lot like a capsule. That's because a swept sphere *is* a capsule. A swept sphere has a start point, end point, and a radius, which is exactly like a capsule. So the algorithm discussed here can also be used for "capsule versus capsule" intersection.

As with "line segment versus plane" intersection, it's useful to solve the equation first. It also turns out that solving the swept sphere equation is an extremely popular question during game programming interviews, because it requires a good grasp of many of the core linear algebra concepts that are expected from a game programmer.

Given a position of the sphere's last frame and current frame, it's possible to convert the position of the sphere into a parametric function. The process of converting it to a function equation ends up being identical to the process used for a ray cast. So given sphere P and sphere Q, we can express the position of both as separate parametric functions:

$$P(t) = P_0 + \vec{v}_p t$$
$$Q(t) = Q_0 + \vec{v}_q t$$

What we want to solve for is the value of t, where the distance between the two spheres is equal to the sum of their radii, because that is the specific point where the intersection occurs. This can be represented mathematically as follows:

$$\left\| P(t) - Q(t) \right\| = r_p + r_q$$

The problem with this equation is we need some way to get rid of the length operation. The trick to this is remembering that the length squared of a vector $\vec{v}$ is the same thing as the dot product between $\vec{v}$ and itself:

$$\left\|\vec{v}\right\|^2 = \vec{v}\cdot\vec{v}$$

So if we were to square both sides of our sphere comparison statement, we would get the following:

$$\left\|P(t)-Q(t)\right\|^2 = \left(r_p + r_q\right)^2$$
$$\left(P(t)-Q(t)\right)\cdot\left(P(t)-Q(t)\right) = \left(r_p + r_q\right)^2$$

Now that we have this equation, we need to solve for t. This process is a little complicated. First, let's substitute the full equations for $P(t)$ and $Q(t)$:

$$\left(P_0 + \vec{v}_p t - Q_0 - \vec{v}_q t\right)\cdot\left(P_0 + \vec{v}_p t - Q_0 - \vec{v}_q t\right) = \left(r_p + r_q\right)^2$$

Then we can do a little bit of factoring and grouping to make the equation a bit more palatable:

$$\left(P_0 - Q_0 + \left(\vec{v}_p - \vec{v}_q\right)t\right)\cdot\left(P_0 - Q_0 + \left(\vec{v}_p - \vec{v}_q\right)t\right) = \left(r_p + r_q\right)^2$$

To simplify it further, we can introduce a couple substitutions:

$$A = P_0 - Q_0$$
$$B = \vec{v}_p - \vec{v}_q$$
$$\left(A + Bt\right)\cdot\left(A + Bt\right) = \left(r_p + r_q\right)^2$$

Because the dot product is distributive over addition, we can then apply the FOIL (first, outside, inside, last) rule between the $\left(A + Bt\right)$ terms:

$$A\cdot A + 2\left(A\cdot B\right)t + \left(B\cdot B\right)t^2 = \left(r_p + r_q\right)^2$$

If we bring over the sum of the radii-squared terms to the left side of the equation and then apply a little more substitution, we will get an equation that may seem familiar:

$$A \cdot A + 2(A \cdot B)t + (B \cdot B)t^2 - (r_p + r_q)^2 = 0$$
$$a = (B \cdot B)$$
$$b = 2(A \cdot B)$$
$$c = A \cdot A - (r_p + r_q)^2$$
$$at^2 + bt + c = 0$$

You might recall that this type of equation can be solved using the quadratic equation, which you may have not used in years:

$$t = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

The value under the radical, known as the **discriminant**, is especially important. There are three ranges of note: less than zero, equal to zero, and greater than zero.

If the discriminant is less than zero, there is no real solution for t, which in this case means no intersection occurs. If the discriminant is equal to zero, it means that the spheres tangentially touch at exactly one value of t. If the discriminant is greater than zero, it means that the objects do fully intersect, and the smaller of the two quadratic solutions (the $-$ in $\pm$) is the t where the initial intersection occurs. All three of these scenarios are illustrated in Figure 7.12.

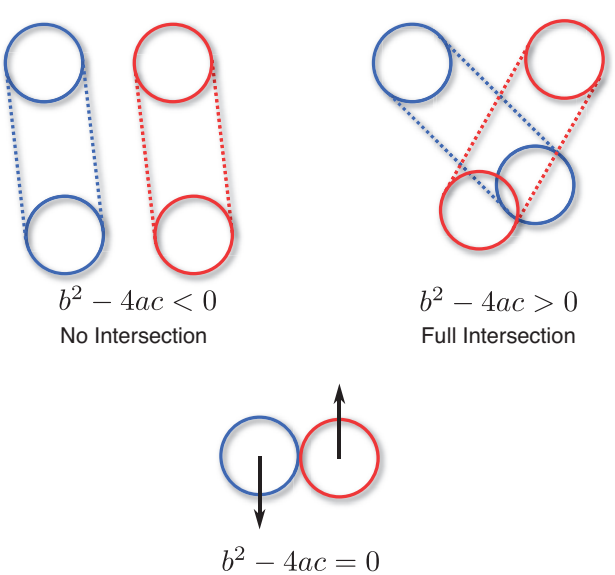


Figure 7.12 Possible values of the discriminant in swept sphere intersection.

Once we have the value of t, we can then see if the value is between the 0 and 1 range that we care about for the code implementation. Remember that a t value greater than one is at a point after the current frame, and less than zero is prior to the current frame. Therefore, values outside the range are not something the function should concern itself with. The code implementation for this is provided in Listing 7.6.

Listing 7.6 Swept Sphere Intersection

```
// p0/q0 are the spheres last frame
// p1/q1 are the spheres this frame
function SweptSphere(BoundingSphere p0, BoundingSphere q0,
                     BoundingSphere p1, BoundingSphere q1)
   // First calculate v vectors for parametric equations
   Vector3 vp = p1.center - p0.center
   Vector3 vq = q1.center - q0.center

   // Calculate A and B
   // A = P0 - Q0
   Vector3 A = p0.center - q0.center
   // B = vp - vq
   Vector3 B = vp - vq

   // Calculate a, b, and c
   // a = B dot B
   float a = DotProduct(B, B)
   // b = 2(A dot B)
   float b = 2 * (DotProduct(A, B)
   // c = (A dot A) - (rp + rq) * (rp + rq)
   float c = DotProduct(A, A) - ((q0.radius + p0.radius) *
                                 (q0.radius + p0.radius))

   // Now calculate the discriminant (b^2 - 4ac)
   float disc = b * b - 4 * a * c
   if disc >= 0
      // If we needed the value of t, we could calculate it with:
      // t = (-b - sqrt(disc)) / (2a)
      // However, this simplified function just returns true to say
      // than an intersection occurred.
      return true
   else
      // No real solutions, so they don't intersect
      return false
   end
  end
```

Collision Response

We can use any of the preceding algorithms to determine whether or not two objects did, in fact, collide. But once they do collide, what should the game do? This is the **response** to the collision. In some instances that response might be very simple: One or both objects may die or be otherwise removed from the world. A slightly more complex response might be something like a missile reducing the health of an enemy on collision.

But what if the two objects need to bounce off of each other, such as two asteroids colliding in *Asteroids*? A tempting solution might be to simply negate the velocity of the asteroids upon collision. But a couple major issues crop up with this approach. One issue is the asteroids getting stuck. Suppose there are two asteroids that intersect on a particular frame, which results in their velocities being negated. But what if they're travelling so slow that they still intersect on the next frame? Well, their velocities will get negated again and again, frame after frame, and they will be stuck in place. This is shown in Figure 7.13(a).

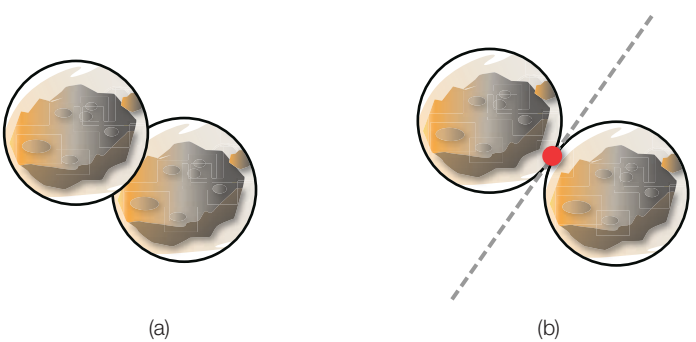


Figure 7.13 Two asteroids get stuck (a), and the plane tangential to the point of intersection (b).

To solve this first issue, we need to find the precise point in time that the asteroids intersect, even if it occurs between frames. Because the asteroids use bounding spheres, we can use swept sphere intersection to find the exact time of intersection. Once we find the time of intersection, we must roll back the position of the two spheres to that point in time. We can then apply whatever change to the velocity we want (in our naïve case, negate them) and then complete the rest of the time step with the new velocity.

However, there's still a big problem with our collision response: Negating the velocities is not the correct behavior. To see why this is the case, imagine you have two asteroids travelling in the same direction, one in front of the other. The front asteroid is travelling slower than the back asteroid, so eventually the back asteroid will catch up. This means when the two asteroids intersect, they will both suddenly start travelling in the opposite direction, which definitely is not correct.

So rather than negating the velocity, we actually want to reflect the velocity vector based on the normal of the plane of intersection. This uses the vector reflection concept discussed in

Chapter 3. If an asteroid were colliding with a wall, it would be easy enough to figure out the plane of intersection because we can make a plane out of the wall. However, in the case of two spheres touching at precisely one point, we need to find the plane that's tangential to the point of intersection, as in Figure 7.13(b).

To construct this tangential plane, we first need to determine the point of intersection. This can actually be done using linear interpolation. If we have two spheres that are touching at one point, the point ends up being somewhere on the line segment between the center of the two spheres. Where it is on that line segment depends on the radii of the spheres. If we have two instances of `BoundingSphere A` and `B` that are intersecting at precisely one point, we can calculate the point of intersection using linear interpolation:

```
Vector3 pointOfIntersection = Lerp(A.position, B.position,
                                   A.radius / (A.radius + B.radius))
```

It turns out the normal of this tangential plane is simply the normalized vector from the center of one sphere to the center of the other. With both a point on the plane and a normal, we can then create the plane that's tangential at the point of intersection. Although if our collision response is merely a reflection of the velocity, we only really need to compute the normal of the plane.

With this reflected velocity, our asteroid collision will look much better, though it still will seem weird because the asteroids will reflect and continue at the same speed as before. In reality, when two objects collide there is a **coefficient of restitution**, or a ratio between the speed after and before a collision:

$$C_R = \frac{\text{Relative speed after collision}}{\text{Relative speed before collision}}$$

In an **elastic** collision ($C_R > 1$), the relative speed afterward is actually higher than the relative speed beforehand. On the other hand, an **inelastic** collision ($C_R < 1$) is one where the relative speed becomes lower. In the case of asteroids, we likely want an inelastic collision, unless they are magical asteroids.

A further consideration is angular dynamics, which are briefly covered at the end of the chapter. But hopefully this section gave you some idea on how to implement a more believable collision response.

Optimizing Collisions

All of the collision detection algorithms we covered allow for testing between pairs of objects to see if they collide. An optimization question that might come up is what can be done if there is a large number of objects to test against? Suppose there are 10,000 objects in the world, and we want to check whether the player collides with any of them. A naïve solution would perform

10,000 collision checks: one between the player and each of the objects. That's not terribly efficient, especially given that the vast majority of the objects are going to be far away.

So what games must do is try to partition the world in some way such that the player only needs to test against a subset of the objects. One such type of partitioning for a 2D game is called a **quadtree**, because the world is recursively split into quarters until each node in the tree only references one particular object, as shown in Figure 7.14.

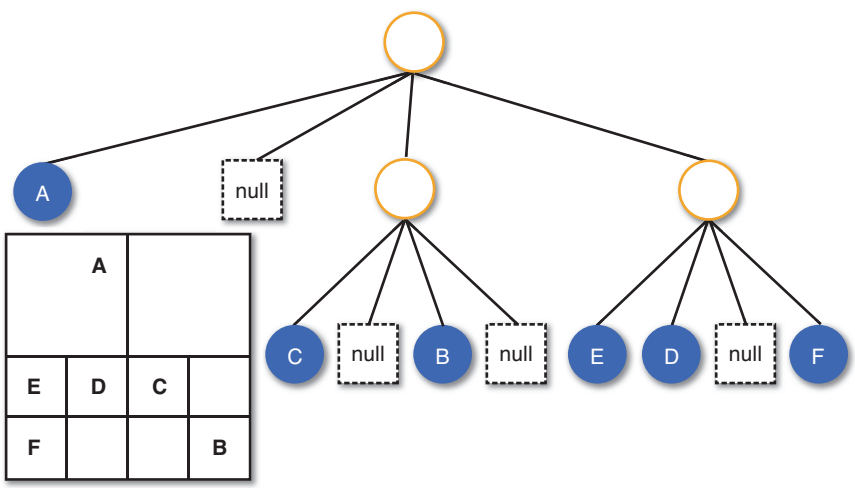


Figure 7.14 A quadtree, where letters represent objects in the world.

When it's time to do the collision check, the code would first check which of the original quarters intersect against the player, potentially eliminating three fourths of the objects instantly. Then a recursive algorithm could continue testing until it finds all of the nodes that potentially intersect with the player. Once there's only a handful of objects left, they could each be tested against the player using the normal collision checks.

A quadtree isn't the only partitioning method available; there are others, including binary spatial portioning (BSP) and octrees (which are the 3D version of quadtrees). Most of the algorithms partition the world spatially, but others partition it with some other grouping heuristic. Partitioning algorithms could easily be an entire chapter in a book, and it turns out there are multiple chapters on this topic in the aforementioned *Real-time Collision Detection*.

Physics-Based Movement

If a game requires objects to move around in the world, some form of physics will be used to simulate the movement. Newtonian physics (also known as classical mechanics) was formulated

by Isaac Newton, among others, in the seventeenth century. The vast majority of games utilize Newtonian physics, because it is an excellent model so long as the objects are not moving close to the speed of light. Newtonian physics has several different components, but this section focuses on the most basic one: **linear mechanics**, which is movement without any rotational forces applied.

I want to forewarn you before going further into this topic that classical mechanics is an extremely comprehensive topic—that's why there is an entire college-level class built around it. Of course, I can't fit everything within the context of this book (and honestly, there are enough physics textbooks out there already). So I had to be very selective of which topics I covered—I wanted to talk about topics that had a high change of being included in a game written by someone who's in the target audience for this book.

Linear Mechanics Overview

The two cornerstones of linear mechanics are force and mass. **Force** is an influence that can cause an object to move. Force has a magnitude and a direction, and therefore it is represented by a vector. **Mass** is a scalar that represents the quantity of matter contained in an object. For mechanics, the primary relevant property is that the higher the mass, the more difficult it is to move the object.

If enough force is applied to an object, in theory it would eventually start accelerating. This idea is encapsulated by Newton's second law of motion:

$$F = m \cdot a$$

Here, F is force, m is mass, and a is the acceleration. Because force is equal to mass times acceleration, it's also true that acceleration is force divided by mass. Given a force, this equation is what a game will use in order to calculate the acceleration.

Typically, we will want to represent the acceleration as a function over time, $a(t)$. Now acceleration understandably has a relationship between velocity and position. That relationship is that the derivative of the position function ($r(t)$) is the velocity function ($v(t)$) and the derivative of the velocity function is the acceleration function. Here it is in symbolic form:

$$v(t) = \frac{dr}{dt}$$
$$a(t) = \frac{dv}{dt}$$

But this formulation is not terribly valuable for a game. In a game, we want to be able to apply a force to an object and from that force determine the acceleration over time. Once we have the acceleration, we then want to determine the velocity over time. Finally, with the velocity in

hand, we can then determine the position of the object. All of these are need to be applied over the current time step.

So in other words, what we need is the opposite of the derivative, which you might recall is the integral. But it's not just any type of integration we're interested in. You are likely most familiar with symbolic integration, which looks something like this:

$$\int \cos(x)\,dx = \sin(x) + C$$

But in a game, symbolic integration is not going to be very useful, because it's not like we are going to have a symbolic equation to integrate in the first place. Rather, on an arbitrary frame we want to apply the acceleration over the time step to get the velocity and the position. This means using **numeric integration**, which is an *approximation* of the symbolic integral applied over a specific time step. If you ever were required to calculate the area under a curve using the midpoint or trapezoidal rule, you calculated a numeric integral. This section covers a few of the most common types of numeric integration used in games.

Issues with Variable Time Steps

Before we cover any of the numeric integration methods, it's important to discuss one big gotcha for any physics-based movement. Once you are using numeric integration, you more or less *cannot* use variable time steps. That's because the accuracy of numeric integration is wholly dependent on the size of the time step. The smaller the time step, the more accurate the approximation.

This means that if the time step changes from frame to frame, the accuracy of the approximation would also change from frame to frame. If the accuracy changes, the behavior will also change in very noticeable ways. Imagine you are playing a game where the character can jump, as in *Super Mario Bros*. You're playing the game, and Mario is jumping at a constant speed. Suddenly, the frame rate drops low and you notice that Mario can jump much further. The reason this happens is that the percent error in numeric integration gets higher with a lower frame rate, so the jump arc becomes more exaggerated. This is demonstrated with the jumping character in Figure 7.15. This means a player on a slower machine will be able to jump further than a player on fast machine.

For this reason, any game that uses physics to calculate the position should not use a variable time step for physics calculations. It's certainly possible to code a game where the physics is running at a different time step than the rest of the game, but that's a little bit more advanced. For now, you could instead utilize the frame rate limiting approach outlined in Chapter 1, "Game Programming Overview."

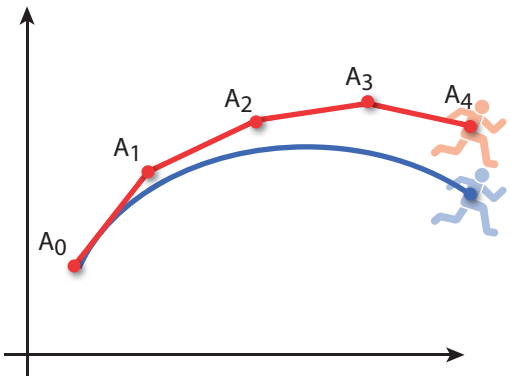


Figure 7.15 Different jump arcs caused by different sized time steps.

Calculating the Force

Numeric integration will allow us to go from acceleration to velocity, and then velocity to position. But in order to compute the acceleration, we need force and mass. There are multiple types of forces to consider. Some forces, such as gravity, are constantly applied to an object. Other forces may instead be **impulses**, or forces applied for a single frame.

For example, a jump might be caused first by applying an impulse force to begin the jump. But once the jump starts, it's the force of gravity that'll bring the character back down to the ground. Because multiple forces can be acting on an object at the same time, the most common approach in a game is to sum up all the force vectors affecting an object and divide this by the mass to determine the current acceleration:

```
acceleration = sumOfForces / mass
```

Euler and Semi-Implicit Euler Integration

The simplest type of numeric integration is **Euler integration**, which is named after the famed Swiss mathematician. In Euler integration, first the new position is calculated by taking the old position and adding to it the old velocity multiplied by the time step. Then the velocity is similarly calculated using the acceleration. A simple physics object that uses this in its update function is shown in Listing 7.7

Listing 7.7 Euler Integration in Physics Object

```
class PhysicsObject
   // List of all the force vectors active on this object
   List forces
   Vector3 acceleration, velocity, position
   float mass
```

```
    function Update(float deltaTime)
        Vector3 sumOfForces = sum of forces in forces
        acceleration = sumOfForces / mass

        // Euler Integration
        position += velocity * deltaTime
        velocity += acceleration * deltaTime
    end
end
```

Although Euler integration is very simple, it does not really exhibit very much accuracy. One big issue is that the position calculation is using the old velocity, not the new velocity after the time step. This results in a propagation of error that causes the approximation to diverge further and further as time continues.

A simple modification to basic Euler integration is to swap the order of the position and velocity calculations. What this means is that the position is now updating based on the new velocity, not the old velocity. This is **semi-implicit Euler integration**, and it ends up being reasonably more stable, to the point that respectable physics engines such as Box2D utilize it. However, if we want further accuracy we have to explore more complex numerical integration methods.

Velocity Verlet Integration

In **velocity Verlet integration**, first the velocity at the time step's midpoint is calculated. This average velocity is then used to integrate the position over the entire time step. Next, the acceleration is updated based on the force and mass, and finally that new acceleration is applied to calculate the velocity at the end of the time step. The implementation of velocity Verlet is shown in Listing 7.8.

Listing 7.8 Velocity Verlet Integration

```
function Update(float deltaTime)
    Vector3 sumOfForces = sum of forces in forces

    // Velocity Verlet Integration
    Vector3 avgVelocity = velocity + acceleration * deltaTime / 2.0f
    // Position is integrated with the average velocity
    position += avgVelocity * deltaTime
    // Calculate new acceleration and velocity
    acceleration = sumOfForces / mass
    velocity = avgVelocity + acceleration * deltaTime / 2.0f
end
```

By essentially using both the average velocity and average accelerations when performing the integrations, the velocity Verlet method ends up with a much greater accuracy than both

Euler methods. And the computational expense, although certainly greater than the Euler approaches, still is not that steep.

Other Integration Methods

A number of other integration methods might be used in a game, though they are a bit more complex. Perhaps one of the most popular advanced methods is the fourth-order Runge-Kutta method. The way this essentially works is by using Taylor approximation in order to approximate a solution to the differential equations representing the movement. This method is inarguably more accurate than both Euler and velocity Verlet, but it's also slower. It might make sense for a game where very accurate integration is required (such as for car physics), but for most games it's probably overkill.

Angular Mechanics

Angular mechanics is rotational physics. For example, you might need this type of physics when you have an object tied to a string and rotating around another object. Just as linear mechanics has mass, force, acceleration, velocity, and position, angular mechanics has moment of inertia, torque, angular acceleration, angular velocity, and the angular position (or angle). The equations for angular mechanics are a little bit more complex than linear mechanics, but not by much. As with linear mechanics, you will also have to numerically integrate if you want to simulate angular mechanics in your game. Some types of games that have movement, such as a billiards game, absolutely need angular mechanics to function properly. But this is not as prevalent as games that rely on linear mechanics, so I choose not to cover this topic in greater detail.

Physics Middleware

Recall from Chapter 1 that middleware is a code library that implements a solution to a common problem. Because of the level of complexity of physics and how vast the problem is, it is extremely common for game companies to utilize a middleware package for physics in lieu of implementing it themselves.

The most popular commercial physics middleware for 3D games without question is Havok Physics. If you go through a list of the top AAA games from the last 10 years, you'll find that a staggering number of them utilize Havok. Although Havok is designed for commercial game engines, there is a version of it for smaller developers.

An alternative that's considered industrial strength is PhysX, whose claim to fame is that it was selected as the default physics system used by Unreal Engine 3 (and newer versions of Unity). The library is available free as long as you have less than $100,000 in yearly revenue.

For 2D physics, the most popular solution by far is the open source Box2D (http://box2d.org). The core library is written in C++, but there are ports to many different languages, including C#, Java, and Python. It's pretty much the *de facto* 2D physics engine, and many 2D physics-based games, including *Angry Birds*, use it.

Summary

This chapter was a lengthy overview of using physics in 2D and 3D games. Collision geometry, such as spheres and boxes, is used to simplify intersection calculations. There are many potential intersection tests, including AABB vs. AABB and line segment vs. plane, and we looked at the implementation of many of them. Remember that some intersection tests, such as swept sphere, are continuous in that they can detect intersections between frames. Finally, movement in games is most commonly implemented through the use of linear mechanics. This means that we must use numeric integration methods such as Euler integration to compute position from velocity and velocity from acceleration.

Review Questions

1. What is the difference between an axis-aligned bounding box (AABB) and an oriented bounding box (OBB)?
2. What plane equation is most commonly used in games? What does each variable in said equation represent?
3. What is a parametric equation? How can you represent a ray via a parametric equation?
4. How can you *efficiently* determine whether or not two spheres intersect?
5. What is the best approach to determine whether or not two 2D AABBs intersect?
6. In line segment vs. plane intersection, what does a negative t value represent?
7. What is the difference between instantaneous and continuous collision detection?
8. In the swept sphere solution, the discriminant can be negative, zero, or positive. What do all three of these different cases represent?
9. Why is it a bad idea to use a variable time step when using numeric integration methods?
10. How does velocity Verlet integration work conceptually?

Additional References

Ericson, Christer. *Real-time Collision Detection*. San Francisco: Morgan Kaufmann, 2005. This book is the end-all be-all book when it comes to collision detection. It covers many different types of collision geometry and all the possible permutations between them. Be warned that this book has a large amount of math, so you might want to make sure you are comfortable with the math in this book before picking it up.

Fielder, Glenn. Gaffer on Games – Game Physics. http://gafferongames.com/game-physics/. This blog covers several different topics, but the physics articles are of particular note. Fielder covers topics including how to implement fourth-order Runge-Kutta as well as angular dynamics and springs.

This page intentionally left blank

CHAPTER 8

CAMERAS

The camera determines the player's point of view in the 3D game world. Many different types of cameras see use in games, and selecting the type of camera is a basic design decision that typically is made early in development.

This chapter covers the major types of cameras used in games today and the calculations that must happen behind the scenes to get them to work. It also takes a deeper look at the perspective projection, which was first covered in Chapter 4, "3D Graphics."

Types of Cameras

Before we dive into different implementations of cameras, it's worthwhile to discuss the major types of 3D cameras used in games. This section doesn't cover every type of camera that conceivably could be used, but it does cover many of the most common varieties.

Fixed and Non-Player Controlled Cameras

Strictly speaking, a **fixed camera** is one that's in exactly the same position at all times. This type of stationary camera typically is only used in very simple 3D games. The term "fixed camera" is usually also extended to refer to systems where the camera goes to a predefined location depending on where the player character is located. As the player character moves through the level, the position of the camera will jump to new spots once the character passes a threshold. The locations and thresholds of the camera are typically set by the designer during creation of the level.

Fixed cameras were once extremely popular for survival horror games such as the original *Resident Evil*. Because the camera can't be controlled by the player, it enables the game designer to set up scenarios where frightening enemies can be hidden behind corners that the camera is unable to see. This added to the atmosphere of tension that these titles were built around. A top-down view of a sample scene with a couple of fixed camera positions is shown in Figure 8.1.

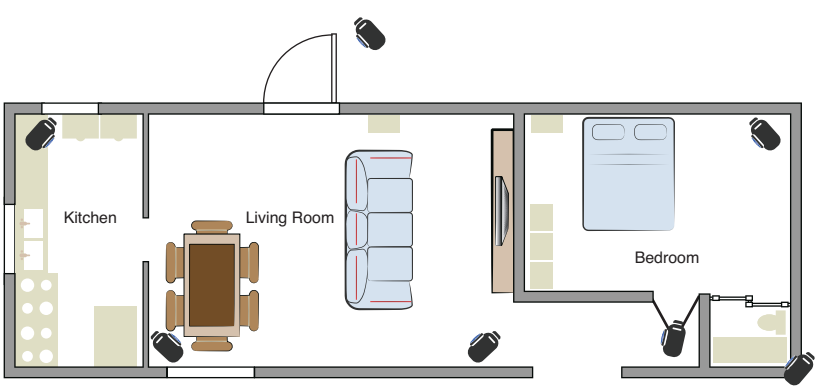


Figure 8.1 Top-down view of a sample scene, and where fixed cameras might be placed.

Whereas the designer might see a fixed camera as an opportunity to create tension, some players find the lack of camera control frustrating. This is perhaps one of the reasons why the modern iterations of the *Resident Evil* series have abandoned a fixed-camera system for more dynamic ones. Some modern games use cameras where the player still does not have any control, but the camera dynamically moves as the player moves through the level. One series that uses this type of system is *God of War*.

A detailed implementation of a fixed-camera system is not covered in this book, but it is one of the easier types of cameras to implement. One possible approach would be to have a list of convex polygons that correspond to a particular camera location. Then, as the player moves, the camera system could check which convex polygon the player is in (using the point in polygon algorithm covered in Chapter 7, "Physics") and select the camera that's associated with said convex polygon.

First-person Camera

A **first-person camera** is one that is designed to give the perspective of the player character moving through the world, as in Figure 8.2. Because the camera is from the character's perspective, first-person camera is often referred to as the most immersive type of game camera. First-person cameras are very popular for the appropriately named first-person shooter, but they also see use in other games such as *Skyrim*.

Figure 8.2 First-person camera in *Quadrilateral Cowboy*.

A common approach in first-person games is to place the camera roughly at eye level, so other characters and objects line up at the expected height. However, this is a problem for first-person games that want to show the hands of the player, as is common in many of them. If the camera is at eye level, when the character looks straight forward they wouldn't see their hands. A further problem is that if a regular character model were drawn for the player, you would likely get odd graphical artifacts from the camera being nearly "inside" the model.

To solve both of these issues, most first-person games will not draw a normal character model. Instead, a special model that only has the arms (and maybe legs) is drawn at an anatomically incorrect location. This way, the player can always see what is in their hands, even though they are looking straight forward. If this approach is used, special consideration must be taken when the game allows the player to see their reflection. Otherwise, the player would see the reflection of their avatar's hands floating in the air, which might be frightening.

Follow Cameras

A **follow camera** is one that follows behind a target object in one way or another. This type of camera can be used in a wide variety of genres—whether a racing game, where it follows behind a car, as in the *Burnout* series, or a third-person action/adventure game such as *Uncharted*. Because follow cameras can be used in so many different types of games, there's a great deal of variety. A drawing of a follow camera behind a car is shown in Figure 8.3.

Figure 8.3 A follow camera behind a car.

Some follow cameras very rigidly trail the target, whereas others have some amount of springiness. Some allow the player to manually rotate the camera while in follow mode, whereas others don't. Some might even let the player instantly flip the camera to observe what's behind them. There are many different ways this type of camera may ultimately be implemented.

Cutscene Cameras

An increasing number of games have **cutscenes**—that is, scenes that *cut away* from the actual gameplay in order to advance the story of the game in some manner. To implement a cutscene in a 3D game, at a minimum there needs to be a system to specify a series of fixed cameras while the animation plays. But most cutscenes use more cinematic devices such as panning and moving the camera, and in order to do this, some sort of spline system is typically also utilized, which we'll discuss later in the chapter.

Perspective Projections

As discussed in Chapter 4, a perspective projection is one that has depth perception. That is to say, as objects become further and further away from the camera, they appear smaller and smaller. We briefly touched on the parameters that drive the perspective projection in Chapter 4, but now it's time to look at this topic in more detail.

Field of View

The amount of the world that is visible at any one time is expressed as an angle known as the **field of view** (FOV). As humans, our pair of eyes affords us nearly a 180° field of view, though not with an equal amount of clarity. The binocular field of view, which is what can be seen with both eyes simultaneously, is approximately 120°. The rest of our field of view is in the peripheral, which is good at detecting movement but not necessarily able to effectively focus.

Field of view is an important consideration when setting up the perspective projection because an incorrect field of view can lead to eye strain, or worse, motion sickness for certain players. To see how this comes into play, let's first take a look at the field of view for a console gamer playing on an HDTV.

The recommended viewing distance for an HDTV varies depending on who you ask, but THX recommends taking the diagonal measurement and multiplying it by 1.2. So a 50" television should be viewed from approximately 60" (or 5'). With this distance, the television has a **viewing angle** of approximately 40°, which means that the TV occupies 40° of the viewer's field of view. Figure 8.4(a) demonstrates a television at the THX recommended distance.

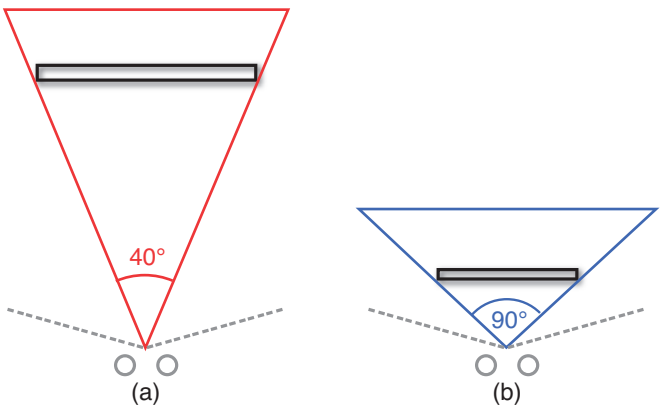


Figure 8.4 50" HDTV with a 40° viewing angle (a), and PC monitor with a 90° viewing angle (b).

In this configuration, as long as the field of view for the game is greater than or equal to 40°, all but the most sensitive of players should not have an issue viewing it. This is one of the reasons why for console games it is very common to have a field of view of approximately 65°.

But what happens if this game is instead played on a PC? In a PC configuration, the player is typically going to have the monitor occupying a much greater percentage of their binocular field of view. It is very common to have a configuration where the viewing angle of the monitor is 90° or more, as shown in Figure 8.4(b). If the game has a 65° field of view in a setup where the viewing angle is 90°, the discrepancy between the virtual field of view and the viewing angle may cause some players discomfort. That's because the game world appears to be more narrow than what the brain is expecting to be visible in a 90° view. Because of this issue, it is a good practice for PC games to allow the player to select their preferred field of view in the settings.

If the field of view is increased, it will also increase what can be seen in the game world. Figure 8.5 illustrates a sample scene—first viewed with a 65° field of view and then viewed with a 90° field of view. Notice how in the wider field of view, more of the scene is visible. Some might argue that players who use a wider field of view will have an advantage over those using a narrower one, especially in a multiplayer game. But as long as the field of view is limited to say, 120°, I'd argue that the advantage really is fairly minimal.

If the field of view becomes too large, it may create the **fisheye effect**, where the edges of the screen become distorted. This is similar to what happens when an ultra-wide angle lens is used in photography. Most games do not allow the player to select a field of view high enough to actually distort this much, but certain game mods do implement this behavior.

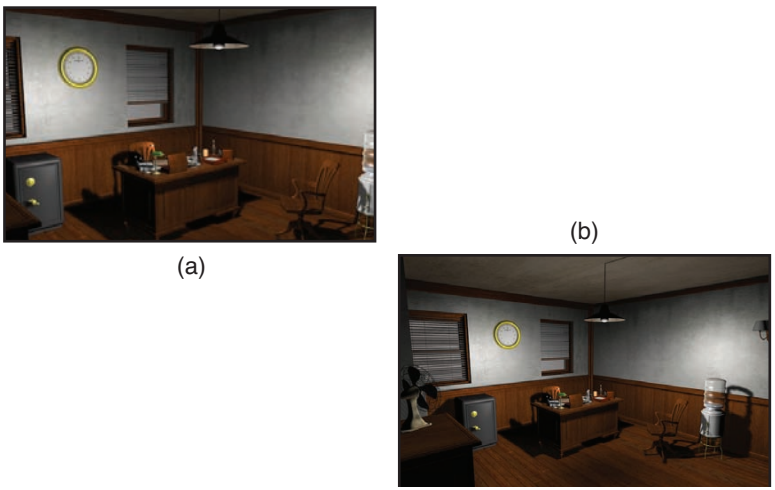


Figure 8.5 Sample scene shown with a 65° field of view (a) and a 90° field of view (b).

Aspect Ratio

The **aspect ratio** is the ratio between the width and height of the view into the world. For a game that runs in full screen, this will typically be the ratio between the width and height of the resolution at which the display device is running. One exception to this is a game that has a split-screen multiplayer mode; in this scenario, there are going to be multiple views into the world (or **viewports**), and in this instance the aspect ratio of the viewport may be different from the aspect ratio of the screen. Irrespective of this, however, the three most prevalent aspect ratios for games are 4:3, 16:9, and 16:10.

The classic 1024×768 resolution (which means it's 1024 pixels across by 768 pixels down) is a 4:3 aspect ratio. But the number of display devices that use a 4:3 aspect ratio has decreased over time; today, almost all new displays sold are not 4:3, but rather 16:9. The standard HD resolution of 720p (which is 1280×720) is an example of a 16:9 aspect ratio, and HDTV is why 16:9 is by far the most prevalent aspect ratio in use today. 16:10 is an aspect ratio that only sees use for certain computer monitor displays, but the number of computer monitors supporting this aspect ratio is decreasing over time.

An issue that comes up when you need to support both 4:3 and 16:9 aspect ratios is determining which aspect ratio is able to see more of the game world. The preferred (and most common) solution is to have a wider horizontal field of view in 16:9 mode, which means more of the world will be visible than in 4:3. But some games, most notably *BioShock* at initial release, go with the opposite approach, where the 16:9 mode actually has a narrower vertical field of view than the 4:3 mode. Both approaches are shown in Figure 8.6.

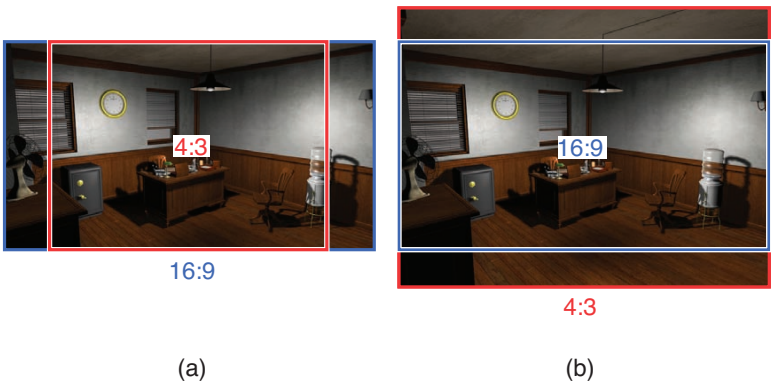


Figure 8.6 16:9 has a wider horizontal FOV (a) and a narrower vertical FOV (b).

Camera Implementations

Now that we've covered the basics of different types of cameras, as well as the basics of the perspective projection, let's take a closer look at some of the types of camera implementations possible.

Basic Follow Camera

In a basic follow camera, the camera always follows directly behind a particular object with a prescribed vertical and horizontal follow distance. For example, a basic follow camera that's tracking a car would follow as shown in Figure 8.7.

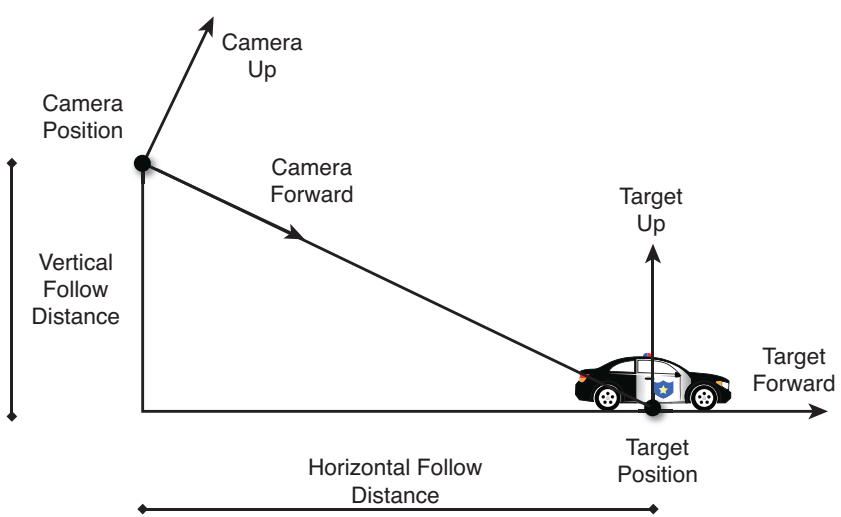


Figure 8.7 Basic follow camera tracking a car.

Recall that to create a look-at matrix for the camera, three parameters are required: the eye (or position of the camera), the target the camera is looking at, and the camera's up vector. In the basic follow camera, the eye is going to be a set vertical and horizontal offset from the target. Once this is calculated, we can then calculate the remaining parameters to pass to `CreateLookAt`:

```
// tPos, tUp, tForward = Position, up, and forward vector of target
// hDist = horizontal follow distance
// vDist = vertical follow distance
function BasicFollowCamera(Vector3 tPos, Vector3 tUp, Vector3 tForward,
                           float hDist, float vDist)
   // Eye is offset from the target position
   Vector3 eye = tPos - tForward * hDist + tUp * vDist

   // Camera forward is FROM eye TO target
   Vector3 cameraForward = tPos - eye
   cameraForward.Normalize()

   // Cross products to calculate camera left, then camera up
   Vector3 cameraLeft = CrossProduct(tUp, cameraForward)
   cameraLeft.Normalize()
   Vector3 cameraUp = CrossProduct(cameraForward, cameraLeft)
   cameraUp.Normalize()

   // CreateLookAt parameters are eye, target, and up
   return CreateLookAt(eye, tPos, cameraUp)
end
```

Although the basic follow camera will work to track the target around in the world, it will seem very rigid. There is no give to it because it always stays at the same distance from the target. When turning, the basic follow behavior makes it hard to tell whether it's the object turning or the world turning around the object. Furthermore, it may be difficult to get a sense of speed because the distance doesn't vary at all based on the speed the object is travelling. For all of these reasons, this basic follow camera is rarely used "as is" in games. Although it provides a simple solution, the aesthetics of it are not quite there.

One simple addition that will improve the sense of speed is to make the horizontal follow distance a function of the speed the target is travelling. So perhaps at rest the horizontal follow distance could be 100 units, but when the target is travelling at full speed, the horizontal follow distance could increase to 200. This simple change will fix the sense of speed with the basic follow camera, but won't fix the overall stiffness.

Spring Follow Camera

With a **spring follow camera**, rather than the camera position instantly changing based on the orientation of the position and the target, the camera adjusts over the course of several frames.

The way this is accomplished is by having both an ideal and actual camera position. The ideal camera position updates instantly every frame, and can be determined with the same calculations as the basic follow camera (perhaps along with the horizontal follow distance function). The actual camera position then lags behind a little bit and follows along the ideal position, which creates a much more fluid camera experience.

The way this is implemented is via a virtual spring connecting the ideal and the actual camera positions. When the ideal camera position changes, the spring is decompressed. If the ideal camera position stops changing, over time the spring will compress all of the way and the ideal and actual camera positions will be one and the same. This configuration with the spring, ideal, and actual camera positions is shown in Figure 8.8.

Figure 8.8 A spring connects the camera's ideal and actual positions.

The stiffness of the spring can be driven by a spring constant. The higher the constant, the stiffer the spring, which means the closer the camera stays to the ideal position. Implementing the spring follow camera requires the camera velocity and actual camera position to be persistent from frame to frame; therefore, the simplest implementation is to use a class. The algorithm roughly works by first calculating the acceleration of the spring based on the constants. Then acceleration is numerically integrated to determine the camera velocity, and then once more to determine the position. The full algorithm, as presented in Listing 8.1, is similar to the spring camera algorithm presented in *Game Programming Gems 4*.

Listing 8.1 Spring Camera Class

```
class SpringCamera
   // Horizontal and vertical follow distance
   float hDist, fDist
   // Spring constant: higher value means stiffer spring
   // A good starting value will vary based on the desired range
   float springConstant
   // Dampening constant is based on the above
   float dampConstant

   // Vectors for velocity and actual position of camera
   Vector3 velocity, actualPosition

   // The target game object the camera follows
   // (Has position, forward vector, and up vector of target)
   GameObject target
```

```
// The final camera matrix
Matrix cameraMatrix

// This helper function computes the camera matrix from
// the actual position and the target
function ComputeMatrix()
   // Camera forward is FROM actual position TO target
   Vector3 cameraForward = target.position - actualPosition
   cameraForward.Normalize()

   // Cross to calculate camera left, then camera up
   Vector3 cameraLeft = CrossProduct(target.up, cameraForward)
   cameraLeft.Normalize()
   Vector3 cameraUp = CrossProduct(cameraForward, cameraLeft)
   cameraUp.Normalize()

   // CreateLookAt parameters are eye, target, and up
   cameraMatrix = CreateLookAt(actualPosition, target.position,
                               cameraUp)
end

// Initializes the constants and camera to initial orientation
function Initialize(GameObject myTarget, float mySpringConstant,
                    float myHDist, float myVDist)
   target = myTarget
   springConstant = mySpringConstant
   hDist = myHDist
   vDist = myVDist

   // Dampening constant is calculated from spring constant
   dampConstant = 2.0f * sqrt(springConstant)

   // Initially, set the actual position to the ideal position.
   // This is like the basic follow camera eye location.
   actualPosition = target.position - target.forward * hDist +
                    target.up * vDist

   // The initial camera velocity should be all zeroes
   velocity = Vector3.Zero

   // Set the initial camera matrix
   ComputeMatrix()
end

function Update(float deltaTime)
   // First calculate the ideal position
   Vector3 idealPosition = target.position - target.forward * hDist
                           + target.up * vDist
```

```
        // Calculate the vector FROM ideal TO actual
        Vector3 displacement = actualPosition - idealPosition
        // Compute the acceleration of the spring, and then integrate
        Vector3 springAccel = (-springConstant * displacement) -
                              (dampConstant * velocity)
        velocity += springAccel * deltaTime
        actualPosition += velocity * deltaTime

        // Update camera matrix
        ComputeMatrix()
    end
end
```

The biggest advantage of a spring camera is that when the target object turns, the camera will have a bit of a delay before it starts turning as well. This results in a sense of turning that's much more aesthetically pleasing than a basic follow camera. And even though the spring camera looks much better than the basic follow camera, it does not require too many additional calculations.

Orbit Camera

An **orbit camera** is one that orbits around a target object. The simplest way to implement orbiting is to store the camera's position as an offset from the target, as opposed to storing the world space position of the camera. That's due to the fact that rotations are always applied with the assumption that the rotation is about the origin (which hopefully you recall from Chapter 4). So if the camera position is stored as an offset, any rotations applied will act as if the target object is at the origin, which in the case of orbiting is precisely what we want. This type of orbit camera is demonstrated in Figure 8.9.

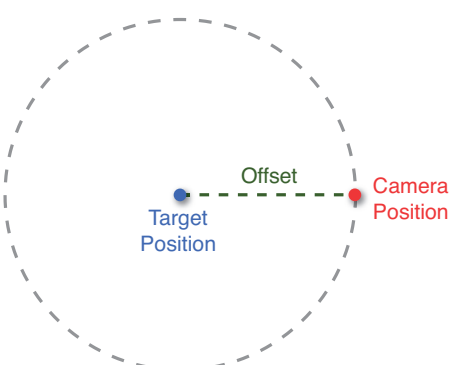


Figure 8.9 Orbit camera.

The typical control scheme for orbiting enables the player to both yaw and pitch, but not roll. Because the input scheme will typically provide the yaw and pitch as separate values, it's best to implement the orbiting as a two-part rotation. First, rotate the camera up and then offset about the world up axis (for yaw), then rotate the camera up and offset about the camera's left vector (for pitch). You need to rotate the camera up in both rotations in order to get the correct behavior when the camera is below the target.

As for how the rotation should be implemented, the system presented in Listing 8.2 uses quaternions, though it would be possible to implement it with matrices instead. I should also note that there is an entirely different approach to implementing an orbit camera using spherical coordinates, but I find the approach presented here a bit easier to follow.

Listing 8.2 Orbit Camera

```
class OrbitCamera
   // Up vector for camera
   Vector3 up
   // Offset from target
   Vector3 offset
   // Target object
   GameObject target
   // Final camera matrix
   Matrix cameraMatrix

   // Initializes the initial camera state
   function Initialize(GameObject myTarget, Vector3 myOffset)
      // In a y-up world, the up vector can just be the Y axis
      up = Vector3(0,1,0)

      offset = myOffset
      target = myTarget

      // CreateLookAt parameters are eye, target, and up
      cameraMatrix = CreateLookAt(target.position + offset,
                                  target.position, up)
   end

   // Updates based on the incremental yaw/pitch angles for this frame
   function Update(float yaw, float pitch)
      // Create a quaternion that rotates about the world up
      Quaternion quatYaw = CreateFromAxisAngle(Vector3(0,1,0), yaw)
      // Transform the camera offset and up by this quaternion
      offset = Transform(offset, quatYaw)
      up = Transform(up, quatYaw)

      // The forward is target.position - (target.position + offset)
      // Which is just -offset
```

```
        Vector3 forward = -offset
        forward.Normalize()
        Vector3 left = CrossProduct(up, forward)
        left.Normalize()

        // Create quaternion that rotates about camera left
        Quaternion quatPitch = CreateFromAxisAngle(left, pitch)
        // Transform camera offset and up by this quaternion
        offset = Transform(offset, quatPitch)
        up = Transform(up, quatPitch)

        // Now compute the matrix
        cameraMatrix = CreateLookAt(target.position + offset,
                                    target.position, up)
    end
end
```

In some games, it may be desirable to have the spring camera behavior as well as the ability to manually orbit the camera. Both methods can certainly be combined to enable following and orbiting, but the implementation of this is left as an exercise for the reader.

First-person Camera

With a first-person camera, the position of the camera is always a set vertical offset from the player character's position. So as the player moves through the world, the camera remains at the player position plus the vertical offset. But even though the camera offset doesn't change, the target location can change independently. That's because most first-person games support looking around and turning while not changing the physical location of the character. This configuration is shown in Figure 8.10.

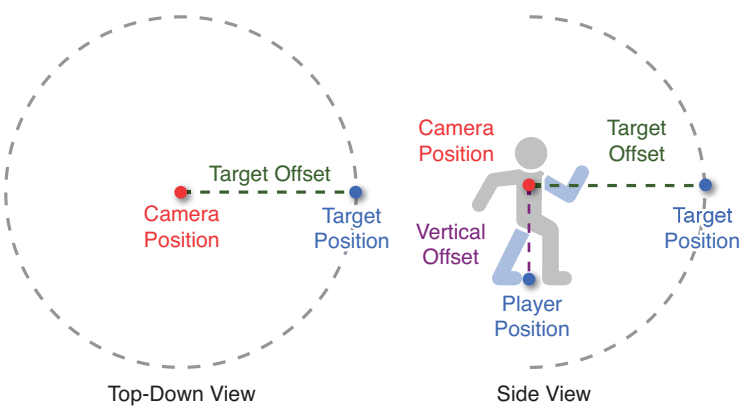


Figure 8.10 First-person camera implementation.

The implementation for the rotation ends up being similar to the rotation of the orbit camera; the main difference is that it is the target offset that's being rotated instead of a camera offset. But there are some additional differences with this type of camera. One is that when the target offset is rotated about the world up, the avatar must also be rotated in the same manner. Another change is that typically the pitch angle is limited to be no more than a certain value.

Due to these modifications, for a first-person camera it's common to store the pitch and yaw as the total angles, as opposed to the delta amount per frame. This means that the rotation will always be applied to the initial target offset, as opposed to an incremental rotation, as was done with the orbit camera. An implementation of a first-person camera is provided in Listing 8.3.

Listing 8.3 First-person Camera

```
class FirstPersonCamera
   // Camera offset from player character position
   // For a y-up world, the vector would be (0, value, 0)
   Vector3 verticalOffset
   // Target position offset from camera
   // For a z-forward world, vector would be (0, 0, value)
   Vector3 targetOffset
   // Total yaw and pitch angles
   float totalYaw, totalPitch
   // Player that the camera is attached to
   GameObject Player
   // Actual camera matrix
   Matrix cameraMatrix

   // Initializes all the camera parameters
   function Initialize(GameObject myPlayer, Vector3 myVerticalOffset,
                       Vector3 myTargetOffset)
      player = myPlayer
      verticalOffset = myVerticalOffset
      targetOffset = myTargetOffset

      // Initially, there's no extra yaw or pitch
      totalYaw = 0
      totalPitch = 0

      // Calculate camera matrix
      Vector3 eye = player.position + verticalOffset
      Vector3 target = eye + targetOffset
      // For y-up world
      Vector3 up = Vector3(0, 1, 0)
      cameraMatrix = CreateLookAt(eye, target, up)
   end
```

```
    // Updates based on the incremental yaw/pitch angles for this frame
    function Update(float yaw, float pitch)
        totalYaw += yaw
        totalPitch += pitch

        // Clamp the total pitch based on restrictions
        // In this case, 45 degrees (approx 0.78 rad)
        totalPitch = Clamp(totalPitch, -0.78, 0.78)

        // The target offset is before any rotations,
        // actual is after rotations.
        Vector3 actualOffset = targetOffset

        // Rotate actual offset about y-up for yaw
        Quaternion quatYaw = CreateFromAxisAngle(Vector3(0,1,0),
                                                 totalYaw)
        actualOffset = Transform(actualOffset, quatYaw)

        // Calculate left for pitch
        // Forward after yaw just the actualOffset (normalized)
        Vector3 forward = actualOffset
        forward.Normalize()
        Vector3 left = CrossProduct(Vector3(0, 1, 0), forward)
        left.Normalize()

        // Rotate actual offset about left for pitch
        Quaternion quatPitch = CreateFromAxisAngle(left, totalPitch)
        actualOffset = Transform(acutalOffset, quatPitch)

        // Now construct the camera matrix
        Vector3 eye = player.position + verticalOffset
        Vector3 target = eye + actualOffset
        // In this case we can just pass in world up, since we can never
        // rotate upside-down.
        cameraMatrix = CreateLookAt(eye, target, Vector3(0, 1, 0))
    end
end
```

Spline Camera

Though the mathematical definition is a bit more specific, a **spline** can be thought of as a curve that is defined by a series of points that lie on the curve. Splines are popular in games because they enable an object to smoothly interpolate across the curve over time. This can be very useful for cutscene cameras, because it means it is possible to have a camera follow along a predefined spline path.

There are many different types of splines, but one of the simplest is the **Catmull-Rom** spline. This type of spline enables interpolation between two adjacent points provided there is a control point before and after the two active ones. For example, in Figure 8.11, P_1 and P_2 are the active control points (at $t = 0$ and $t = 1$, respectively) and P_0 and P_3 are the control points prior to and after. Even though the figure shows only four points, there is no practical limit. As long as there is a control point before and after, the path can go on indefinitely.

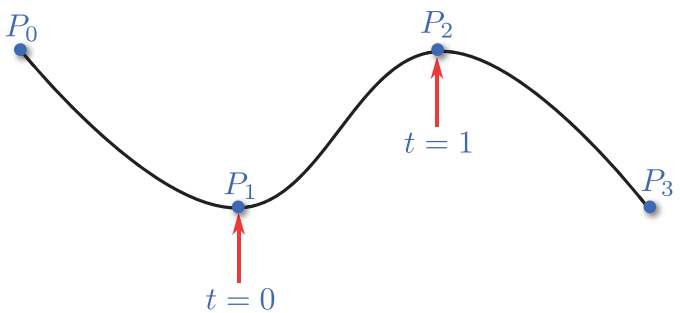


Figure 8.11 Catmull-Rom spline with the minimum four points.

Given this set of four control points, it is then possible to compute the position on the spline at any t value between 0 and 1 using the following equation:

$$P(t) = 0.5 \cdot ((2 \cdot P_1) + (-P_0 + P_2) \cdot t + (2 \cdot P_0 - 5 \cdot P_1 + 4 \cdot P_2 - P_3) \cdot t^2 + (-P_0 + 3 \cdot P_1 - 3 \cdot P_2 + P_3) \cdot t^3)$$

Note that this equation only works if there is uniform spacing between the control points. In any event, with this equation in hand, a simple Catmull-Rom spline class that supports an arbitrary number of points could be implemented as follows:

```
class CRSpline
   // Vector (dynamic array) of Vector3s
   Vector controlPoints

   // First parameter is the control point that corresponds to t=0
   // Second parameter is the t value
   function Compute(int start, float t)
      // Check that start - 1, start, start + 1, and start + 2
      // all exist
      ...
```

```
        Vector3 P0 = controlPoints[start - 1]
        Vector3 P1 = controlPoints[start]
        Vector3 P2 = controlPoints[start + 1]
        Vector3 P3 = controlPoints[start + 2]

        // Use Catmull-Rom equation to compute position
        Vector3 position = 0.5*((2*P1)+(-P0+P2)*t+
                           (2*P0-5*P1+4*P2-P3)*t*t+
                           (-P0+3*P1-3*P2+P3)*t*t*t)

        return position
    end
end
```

It is also is possible to compute the tangent at any t between 0 and 1 using the same equation. First, compute the position at t like you normally would. Then, compute the position at t plus a small Δt. Once you have both positions, you can then construct a vector from $P(t)$ to $P(t+\Delta t)$ and normalize it, which will approximate the tangent vector. In a way, this approach can be thought of as numeric differentiation.

If a camera is following the spline, the tangent vector corresponds to the facing vector of the camera. Provided that the spline does not allow the camera to be flipped upside down, once you have both the position and the facing vector, the camera matrix can be constructed. This is shown with the spline camera implementation in Listing 8.4.

Listing 8.4 Spline Camera

```
class SplineCamera
    // Spline path the camera follows
    CRSpline path
    // Current control point index and t value
    int index
    float t
    // speed is how much t changes per second
    float speed
    // Actual camera matrix
    Matrix cameraMatrix

    // Calculates the camera matrix given current index and t
    function ComputeMatrix()
        // Eye position is the spline at the current t/index
        Vector3 eye = path.Compute(index, t)
        // Get point a little bit further ahead to get the target point
        Vector3 target = path.Compute(index, t + 0.05f)
        // For y-up world
        Vector3 up = Vector3(0, 1, 0)
```

```
        cameraMatrix = CreateLookAt(eye, target, up)
    end

    function Initialize(float mySpeed)
        // The initial index should be 1 (because 0 is P0 initially)
        index = 1
        t = 0.0f
        speed = mySpeed

        ComputeMatrix()
    end

    function Update(float deltaTime)
        t += speed * deltaTime

        // If t > = 1.0f, we should move to the next control point
        // This code assumes that the speed won't be so fast that we
        // might move forward two control points in one frame.
        if t >= 1.0f
            index++
            t = t - 1.0f
        end

        // Should check that index+1 and index+2 are valid points
        // If not, this spline path is complete
        ...

        ComputeMatrix()
    end
end
```

Camera Support Algorithms

The implementations covered in the preceding section should provide enough information to get a camera that works on a basic level. But basic operation is only a small part of implementing a successful camera. In order to implement a camera that is useful for the player, additional support algorithms should be considered.

Camera Collision

Camera collision strives to solve a very common problem with many types of cameras: when there is an opaque object between the camera and the target. The simplest (though not the best) way to fix this is by performing a ray cast from the target position to the camera position. If the ray cast collides with an object, the camera can be moved to a position that's in front of

the object that's blocking the camera, as shown in Figure 8.12. Unfortunately, this behavior can be jarring; a more robust solution is to make a physics object representing the camera, but that's beyond the scope of this section.

Figure 8.12 Camera blocked by an object (a), and camera moved up so it's no longer blocked (b).

Another consideration is what should be done when the camera becomes too close to the target object. Recall that the near plane of the camera is a little bit in front of the eye, which means a close camera could cause part of the target object to disappear. A popular solution to this problem is to hide or alpha out the target if the camera gets too close.

The alpha solution sometimes is also applied to the general problem of the camera being blocked by an object. So rather than moving the camera forward, the game may just alpha out the object that's blocking it. Several third-person action games utilize this method.

Picking

Picking is the capability to click or tap to select an object in the 3D world. Picking is commonly used in RTS games, such as the tower defense game in Chapter 14, "Sample Game: Tower Defense for PC/Mac." In that particular game, picking is implemented in order to enable selection of the various towers (and locations where towers can be placed) in the game world. Although picking is not strictly a camera algorithm, it does tie in with the camera and projection.

Recall that to transform a given point in world space into projection (or screen) space, it must be multiplied by the camera matrix followed by the projection matrix. However, the position of the mouse (or touch) is expressed as a 2D point in screen space. What we need to do is take that 2D point that's in screen space and transform it back into world space, which is known as an **unprojection**. In order to perform an unprojection, we need a matrix that can do the opposite transformation. For a row-major representation, this means we need the inverse of the camera matrix multiplied by the projection matrix:

$$unprojection = (camera \times projection)^{-1}$$

But a 2D point can't be multiplied by a 4×4 matrix, so before the point can be multiplied by this matrix, it must be converted to homogenous coordinates. This requires z- and w-components

to be added to the 2D point. The z-component is usually set to either 0 or 1, depending on whether the 2D point should be converted to a point on the near plane or the far plane, respectively. And since it's a point, the w-component should always be 1. The following `Unproject` function takes a 4D vector, because it assumes that it already has its z- and w-components set appropriately:

```
function Unproject(Vector4 screenPoint, Matrix camera, Matrix projection)
   // Compute inverse of camera * projection
   Matrix unprojection = camera * projection
   unprojection.Invert()

   return Transform(screenPoint, unprojection)
end
```

The `Unproject` function can then be used to calculate two points: the mouse position unprojected onto the near plane ($z = 0$) and the mouse position unprojected onto the far plane ($z = 1$). These two points in world space can then be used as the start and end points of a ray cast that tests against all of the selectable objects in the world. Because there can potentially be multiple objects the ray cast intersects with, the game should select the closest one. Figure 8.13 shows the basic premise behind picking.

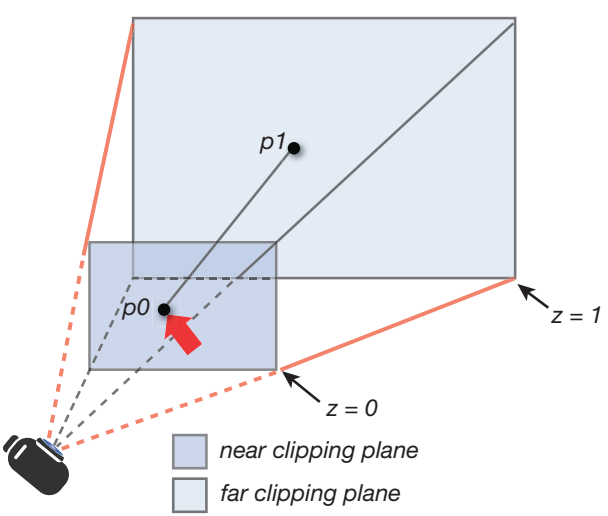


Figure 8.13 Picking casts a ray from the near to far plane and selects an object that intersects with the line segment.

Summary

The camera is a core component of any 3D game. You potentially might use lots of different types of cameras, and this chapter covered implementations of many of them. A first-person camera gives an immersive view from the eye of a character. A follow camera might be used to chase behind a car or a player. An orbiting camera rotates around a specific target, and a spline camera might be used for cutscenes. Finally, many games might need to implement picking in order to allow the player to click on and select objects.

Review Questions

1. What does field of view represent? What issues can arise out of a narrow field of view?
2. How can you set up a basic camera that follows a target at a set horizontal and vertical distance?
3. In what way does a spring follow camera improve upon a basic follow camera?
4. When implementing an orbit camera, how should the camera's position be stored?
5. How does a first-person camera keep track of the target position?
6. What is a Catmull-Rom spline?
7. What is a spline camera useful for?
8. A follow camera is consistently being blocked by an object between the camera and the target. What are two different ways this can be solved?
9. In an unprojection, what do z-components of 0 and 1 correspond to?
10. How can unprojection be used to implement picking?

Additional References

Haigh-Hutchinson, Mark. *Real-Time Cameras*. Burlington: Morgan Kaufmann, 2009. This book provides a comprehensive look at many different types of game cameras, and was written by the creator of the excellent camera systems in *Metroid Prime*.

CHAPTER 9

ARTIFICIAL INTELLIGENCE

This chapter covers three major aspects of artificial intelligence in games: pathfinding, state-based behaviors, and strategy/planning. Pathfinding algorithms determine how non-player characters (NPCs) travel through the game world. State-based behaviors drive the decisions these characters make. Finally, strategy and planning are required whenever a large-scale AI plan is necessary, such as in a real-time strategy (RTS) game.

"Real" AI versus Game AI

In traditional computer science, much of the research on artificial intelligence (AI) has trended toward increasingly complex forms of AI, including genetic algorithms and neural networks. But such complex algorithms see limited use in computer and video games. There are two main reasons for this. The first issue is that complex algorithms require a great deal of computational time. Most games can only afford to spend a fraction of their total frame time on AI, which means efficiency is prioritized over complexity. The other major reason is that game AI typically has well-defined requirements or behavior, often at the behest of designers, whereas traditional AI is often focused on solving more nebulous and general problems.

In many games, AI behaviors are simply a combination of state machine rules with random variation thrown in for good measure. But there certainly are some major exceptions. The AI for a complex board game such as chess or Go requires a decision tree, which is a cornerstone of traditional game theory. But a board game with a relatively small number of possible moves at any one time is entirely different from the majority of video games. Although it's acceptable for a chess AI to take several seconds to determine the optimal move, most games do not have this luxury. Impressively, there are some games that do implement more complex algorithms in real time, but these are the exception and not the rule. In general, AI in games is about perception of intelligence; as long as the player perceives that the AI enemies and companions are behaving intelligently, the AI system is considered a success.

It is also true that not every game needs AI. Some simple games, such as *Solitaire* and *Tetris*, certainly have no need for such algorithms. And even some more complex games may not require an AI, such as *Rock Band*. The same can be said for games that are 100% multiplayer and have no **NPCs** (non-player characters). But for any game where the design dictates that NPCs must react to the actions of the player, AI algorithms become a necessity.

Pathfinding

Pathfinding is the solution to a deceptively simple problem: Given points A and B, how does an AI *intelligently* traverse the game world between them?

The complexity of this problem stems from the fact that there can be a large set of paths between points A and B but only one path is objectively the best. For example, in Figure 9.1, both the red and blue paths are potential routes from point A to B. However, the blue path is objectively better than the red one, simply because it is shorter.

So it is not enough to simply find a single valid path between two points. The ideal pathfinding algorithm needs to search through all of the possibilities and find the absolute best path.

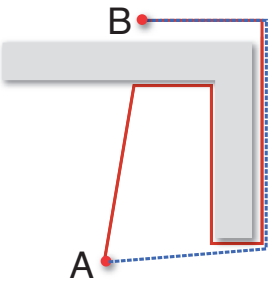


Figure 9.1 Two paths from A to B.

Representing the Search Space

The simplest pathfinding algorithms are designed to work with the **graph** data structure. A graph contains a set of **nodes**, which can be connected to any number of adjacent nodes via **edges**. There are many different ways to represent a graph in memory, but the most basic is an adjacency list. In this representation, each node contains a list of pointers to any adjacent nodes. The entire set of nodes in the graph can then be stored in a standard container data structure. Figure 9.2 illustrates the visual and data representations of a simple graph.

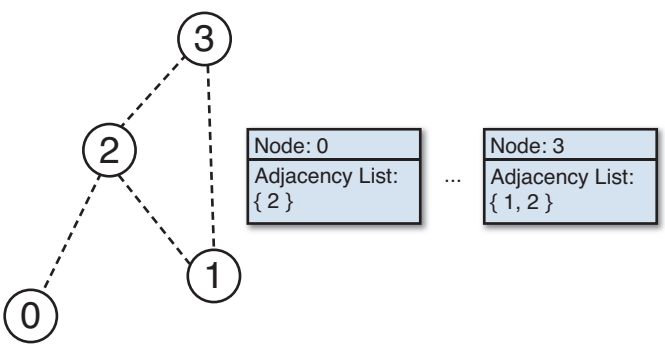


Figure 9.2 A sample graph.

This means that the first step to implement pathfinding in a game is to determine how the world will be represented as a graph. There are a several possibilities. A simple approach is to partition the world into a grid of squares (or hexes). The adjacent nodes in this case are simply the adjacent squares in the grid. This method is very popular for turn-based strategy games in the vein of *Civilization* or *XCOM* (see Figure 9.3).

However, for a game with real-time action, NPCs generally don't move from square to square on a grid. Because of this, the majority of games utilize either path nodes or navigation meshes. With both of these representations, it is possible to manually construct the data in a level editor.

But manually inputting this data can be tiresome and error prone, so most engines employ some automation in the process. The algorithms to automatically generate this data are beyond the scope of this book, though more information can be found in this chapter's references.

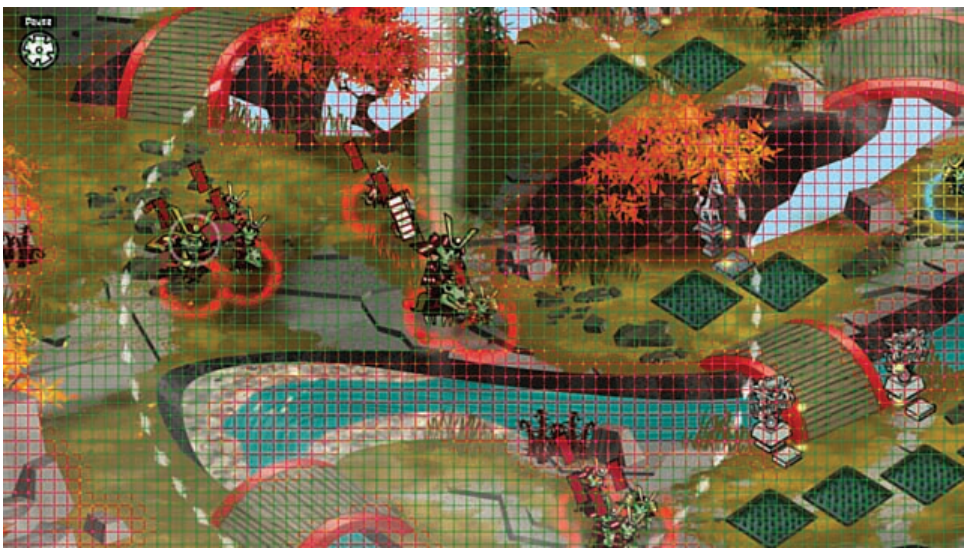


Figure 9.3 *Skulls of the Shogun* with its AI debug mode enabled to illustrate the square grid it uses for pathfinding.

Path nodes first became popular with the first-person shooter (FPS) games released by id Software in the early 1990s. With this representation, a level designer places path nodes at locations in the world that the AI can reach. These path nodes directly translate to the nodes in the graph. Automation can be employed for the edges. Rather than requiring the designer to stitch together the nodes by hand, an automated process can check whether or not a path between two relatively close nodes is obstructed. If there are no obstructions, an edge will be generated between these two nodes.

The primary drawback with path nodes is that the AI can only move to locations on the nodes or edges. This is because even if a triangle is formed by path nodes, there is no guarantee that the interior of the triangle is a valid location. There may be an obstruction in the way, so the pathfinding algorithm has to assume that any location that isn't on a node or an edge is invalid.

In practice, this means that when path nodes are employed, there will be either a lot of unusable space in the world or a lot of path nodes. The first is undesirable because it results in less believable and organic behavior from the AI, and the second is simply inefficient. The more nodes and more edges there are, the longer it will take for a pathfinding algorithm to arrive at a solution. With path nodes, there is a very real tradeoff between performance and accuracy.

An alternative solution is to use a **navigation mesh**. In this approach, a single node in the graph is actually a convex polygon. Adjacent nodes are simply any adjacent convex polygons. This means that entire regions of a game world can be represented by a small number of convex polygons, resulting in a small number of nodes in the graph. Figure 9.4 compares how one particular room in a game would be represented by both path nodes and navigation meshes.

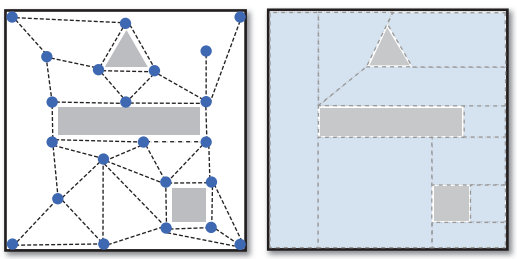


Figure 9.4 Two representations of the same room.

With a navigation mesh, any location contained inside one of the convex polygon nodes can be trusted. This means that the AI has a great deal of space to move around in, and as a result the pathfinding can return paths that look substantially more natural.

The navigation mesh has some additional advantages. Suppose there is a game where both cows and chickens walk around a farm. Given that chickens are much smaller than cows, there are going to be some areas accessible to the chickens, but not to the cows. If this particular game were using path nodes, it typically would need two separate graphs: one for each type of creature. This way, the cows would stick only to the path nodes they could conceivably use. In contrast, because each node in a navigation mesh is a convex polygon, it would not take very many calculations to determine whether or not a cow could fit in a certain area. Because of this, we can have only one navigation mesh and use our calculations to determine which nodes the cow can visit.

Add in the advantage that a navigation mesh can be entirely auto-generated, and it's clear why fewer and fewer games today use path nodes. For instance, for many years the Unreal engine used path nodes for its search space representation. One such Unreal engine game that used path nodes was *Gears of War*. However, within the last couple of years the Unreal engine was retrofitted to use navigation meshes instead. Later games in the *Gears of War* series, such as *Gears of War 3*, used navigation meshes. This shift corresponds to the shift industry-wide toward the navigation mesh solution.

That being said, the representation of the search space does not actually affect the implementation of the pathfinding algorithm. As long as the search space can be represented by a graph,

pathfinding can be attempted. For the subsequent examples in this section, we will utilize a grid of squares in the interest of simplicity. But the pathfinding algorithms will stay the same regardless of whether the representation is a grid of squares, path nodes, or a navigation mesh.

Admissible Heuristics

All pathfinding algorithms need a way to mathematically evaluate which node should be selected next. Most algorithms that might be used in a game utilize a **heuristic**, represented by $h(x)$. This is the estimated cost from one particular node to the goal node. Ideally, the heuristic should be as close to the actual cost as possible. A heuristic is considered **admissible** if the estimate is guaranteed to always be less than or equal to the actual cost. If a heuristic overestimates the actual cost, there is decent probability that the pathfinding algorithm will not discover the best route.

For a grid of squares, there are two different ways to compute the heuristic, as shown in Figure 9.5. **Manhattan distance** is akin to travelling along city blocks in a sprawling metropolis. A particular building can be "five blocks away," but that does not necessarily mean that there is only one route that is five blocks in length. Manhattan distance assumes that diagonal movement is disallowed, and because of this it should only be used as the heuristic if this is the case. If diagonal movement is allowed, Manhattan distance will often overestimate the actual cost.

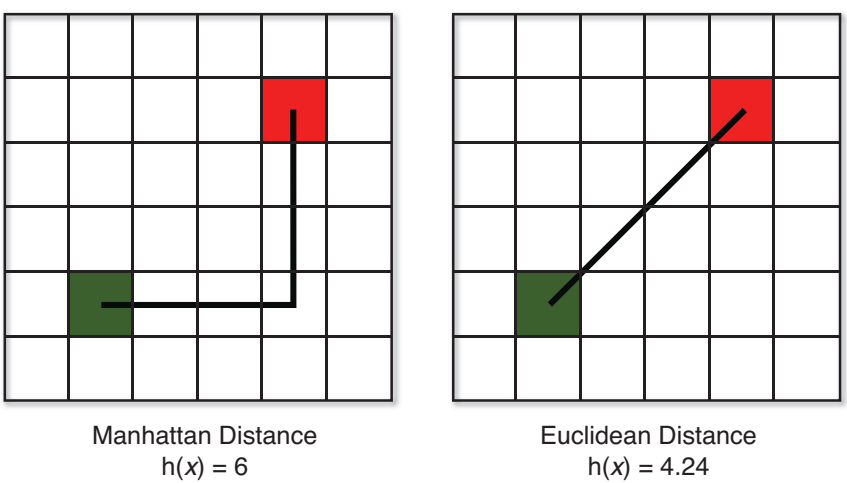


Figure 9.5 Manhattan and Euclidean heuristics.

In a 2D grid, the calculation for Manhattan distance is as follows:

$$h(x) = |start.x - end.x| + |start.y - end.y|$$

The second way to calculate a heuristic is **Euclidean distance**. This heuristic is calculated using the standard distance formula and estimates an "as the crow flies" route. Unlike Manhattan distance, Euclidean distance can also be employed in situations where another search space representation, such as path nodes or navigation meshes, is being utilized. In our same 2D grid, Euclidean distance is

$$h(x) = \sqrt{(start.x - end.x)^2 + (start.y - end.y)^2}$$

Greedy Best-First Algorithm

Once there is a heuristic, a relatively basic pathfinding algorithm can be implemented: the **greedy best-first search**. An algorithm is considered greedy if it does not do any long-term planning and simply picks what's the best answer "right now." At each step of the greedy best-first search, the algorithm looks at all the adjacent nodes and selects the one with the lowest heuristic.

Although this may seem like a reasonable solution, the best-first algorithm can often result in sub-optimal paths. Take a look at the path presented in Figure 9.6. The green square is the starting node, the red square is the ending node, and grey squares are impassable. The arrows show the greedy best-first path.

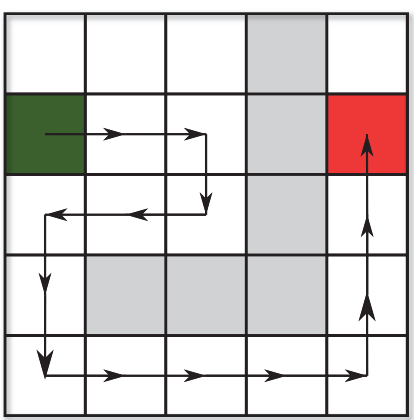


Figure 9.6 Greedy best-first path.

This path unnecessarily has the character travel to the right because, at the time, those were the best nodes to visit. An ideal route would have been to simply go straight down from the starting node, but this requires a level of planning the greedy-best first algorithm does not exhibit. Most games need better pathfinding than greedy best-first can provide, and because of this it does not see much use in actual games. However, the subsequent pathfinding algorithms

covered in this chapter build on greedy best-first, so it is important you first understand how this algorithm is implemented before moving on.

First, let's look at the data we need to store for each node. This data will be in addition to any adjacency data needed in order to construct the graph. For this algorithm, we just need a couple more pieces of data:

```
struct Node
   Node parent
   float h
end
```

The `parent` member variable is used to track which node was visited prior to the current one. Note that in a language such as C++, the parent would be a pointer, whereas in other languages (such as C#) classes may inherently be passed by reference. The `parent` member is valuable because it will be used to construct a linked list from the goal node all the way back to the starting node. When our algorithm is complete, the parent linked list will then be traversed in order to reconstruct the final path.

The float `h` stores the computed $h(x)$ value for a particular node. This will be consulted when it is time to pick the node with the lowest $h(x)$.

The next component of the algorithm is the two containers that temporarily store nodes: the open set and the closed set. The **open set** stores all the nodes that currently need to be considered. Because a common operation in our algorithm will be to find the lowest $h(x)$ cost node, it is preferred to use some type of sorted container such as a binary heap or priority queue for the open set.

The **closed set** contains all of the nodes that have already been evaluated by the algorithm. Once a node is in the closed set, the algorithm stops considering it as a possibility. Because a common operation will be to check whether or not a node is already in the closed set, it may make sense to use a data structure in which a search has a time complexity better than $O(n)$, such as a binary search tree.

Now we have the necessary components for a greedy best-first search. Suppose we have a start node and an end node, and we want to compute the path between the two. The majority of the work for the algorithm is processed in a loop. However, before we enter the loop, we need to initialize some data:

```
currentNode = startNode
add currentNode to closedSet
```

The current node simply tracks which node's neighbors will be evaluated next. At the start of the algorithm, we have no nodes in our path other than the starting node. So it follows that we should first evaluate the neighbors of the starting node.

In the main loop, the first thing we want to do is look at all of the nodes adjacent to the current node and then add some of them to the open set:

```
do
   foreach Node n adjacent to currentNode
      if closedSet contains n
         continue
      else
         n.parent = currentNode
         if openSet does not contain n
            compute n.h
            add n to openSet
         end
      end
   loop //end foreach
```

Note how any nodes already in the closed set are ignored. Nodes in the closed set have previously been fully evaluated, so they don't need to be evaluated any further. For all other adjacent nodes, the algorithm sets their parent to the current node. Then, if the node is not already in the open set, we compute the $h(x)$ value and add the node to the open set.

Once the adjacent nodes have been processed, we need to look at the open set. If there are no nodes left in the open set, this means we have run out of nodes to evaluate. This will only occur in the case of a pathfinding failure. There is no guarantee that there is always a solvable path, so the algorithm must take this into account:

```
if openSet is empty
   break //break out of main loop
end
```

However, if we still have nodes left in the open set, we can continue. The next thing we need to do is find the lowest $h(x)$ cost node in the open set and move it to the closed set. We also mark it as the current node for the next iteration of our loop.

```
currentNode = Node with lowest h in openSet
remove currentNode from openSet
add currentNode to closedSet
```

This is where having a sorted container for the open set becomes handy. Rather than having to do an $O(n)$ search, a binary heap will grab the lowest $h(x)$ node in $O(1)$ time.

Finally, we have the exit case of the loop. Once we find a valid path, the current node will be the same as the end node, at which point we can finish the loop.

```
until currentNode == endNode //end main do...until loop
```

If we exit the `do...until` loop in the success case, we will have a linked list of parents that take us from the end node all the way back to the start node. Because we want a path from the

start node to the end node, we must reverse it. There are several ways to reverse the list, but one of the easiest is to use a stack.

Figure 9.7 shows the first two iterations of the greedy best-first algorithm applied to the sample data set. In Figure 9.7(a), the start node has been added to the closed set, and its adjacent nodes (in blue) are added to the open set. Each adjacent node has its $h(x)$ cost to the end node calculated with the Manhattan distance heuristic. The arrows point from the children back to their parent. The next part of the algorithm is to select the node with the lowest $h(x)$ cost, which in this case is the node with $h = 3$. It gets marked as the current node and is moved to the closed set. Figure 9.7(b) then shows the next iteration, with the nodes adjacent to the current node (in yellow) added to the open set.

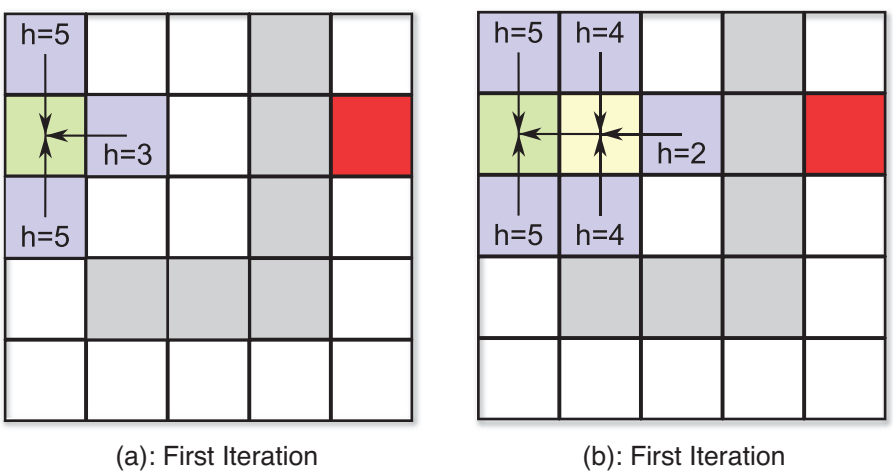


Figure 9.7 Greedy best-first snapshots.

Once the goal node (in red) is added to the closed set, we would then have a linked list from the end node all the way back to the starting node. This list can then be reversed in order to get the path illustrated earlier in Figure 9.6.

The full listing for the greedy best-first algorithm follows in Listing 9.1. Note that this implementation makes the assumption that the $h(x)$ value cannot change during the course of execution.

Listing 9.1 Greedy Best-First Algorithm

```
curentNode = startNode
add currentNode to closedSet
do
   // Add adjacent nodes to open set
```

```
    foreach Node n adjacent to currentNode
       if closedSet contains n
          continue
       else
          n.parent = currentNode
          if openSet does not contain n
             compute n.h
             add n to openSet
          end
       end
    loop

    // All possibilities were exhausted
    if openSet is empty
       break
    end
    // Select new current node
    currentNode = Node with lowest h in openSet
    remove currentNode from openSet
    add currentNode to closedSet
 until currentNode == endNode

 // If the path was solved, reconstruct it using a stack reversal
 if currentNode == endNode
    Stack path
    Node n = endNode
    while n is not null
       push n onto path
       n = n.parent
    loop
 else
    // Path unsolvable
 end
```

If we want to avoid the stack reversal to reconstruct the path, another option is to instead calculate the path from the goal node to the start node. This way, the linked list that we end up with will be in the correct order from the start node to the goal node, which can save some computation time. This optimization trick is used in the tower defense game in Chapter 14, "Sample Game: Tower Defense for PC/Mac" (though it does not implement greedy-best first as its pathfinding algorithm).

A* Pathfinding

Now that we've covered the greedy best-first algorithm, we can add a bit of a wrinkle to greatly improve the quality of the path. Instead of solely relying on the $h(x)$ admissible heuristic, the **A* algorithm** (pronounced A-star) adds a **path-cost** component. The path-cost is the actual

cost from the start node to the current node, and is denoted by $g(x)$. The equation for the total cost to visit a node in A* then becomes:

$$f(x) = g(x) + h(x)$$

In order for us to employ the A* algorithm, the `Node` struct needs to additionally store the $f(x)$ and $g(x)$ values, as shown:

```
struct Node
   Node parent
   float f
   float g
   float h
end
```

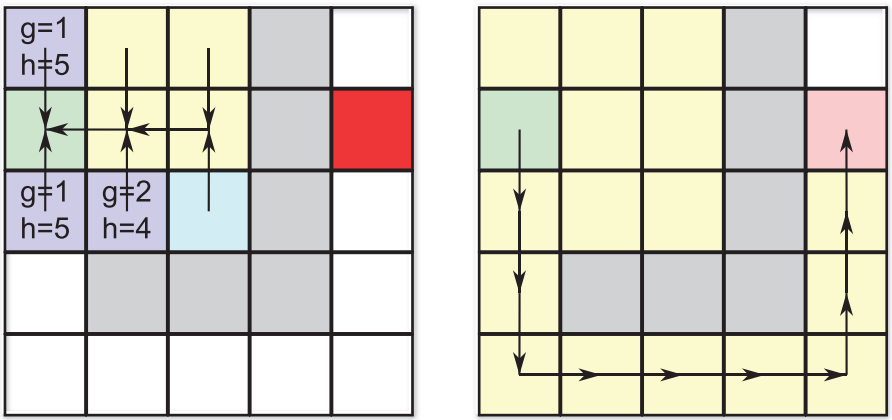
When a node is added to the open set, we must now calculate all three components instead of only the heuristic. Furthermore, the open set will instead be sorted by the full $f(x)$ value, as in A* we will select the lowest $f(x)$ cost node every iteration.

There is only one other major change to the code for the A* algorithm, and that is the concept of **node adoption**. In the best-first algorithm, adjacent nodes always have their parent set to the current node. However in A*, adjacent nodes that are already in the open set need to have their value evaluated to determine whether the current node is a superior parent.

The reason for this is that the $g(x)$ cost of a particular node is dependent on the $g(x)$ cost of its parent. This means that if a more optimal parent is available, the $g(x)$ cost of a particular node can be reduced. So we don't want to automatically replace a node's parent in A*. It should only be replaced when the current node is a better parent.

In Figure 9.8(a), we see the current node (in teal) checking its adjacent nodes. The node to its left has $g = 2$. If that node instead had teal as its parent, it would have $g = 4$, which is worse. So in this case, the node in teal has its adoption request denied. Figure 9.8(b) shows the final path as computer by A*, which is clearly superior to the greedy best-first solution.

Apart from the node adoption, the code for the A* algorithm ends up being very similar to best-first, as shown in Listing 9.2.



(a): Current Node Adoption Fails (b): Final A* Route

Figure 9.8 A* path.

Listing 9.2 A* Algorithm

```
currentNode = startNode
add currentNode to closedSet
do
   foreach Node n adjacent to currentNode
      if closedSet contains n
         continue
      else if openSet contains n // Check for adoption
         compute new_g // g(x) value for n with currentNode as parent
         if new_g < n.g
            n.parent = currentNode
            n.g = new_g
            n.f = n.g + n.h // n.h for this node will not change
         end
      else
         n.parent = currentNode
         compute n.h
         compute n.g
         n.f = n.g + n.h
         add n to openSet
      end
   loop

   if openSet is empty
      break
   end
```

```
    currentNode = Node with lowest f in openSet
    remove currentNode from openSet
    add currentNode to closedSet
until currentNode == endNode
// Path reconstruction from Listing 9.1.
...
```

The tower defense game covered in Chapter 14 provides a full implementation of the A* algorithm over a hex-based grid.

Dijkstra's Algorithm

One final pathfinding algorithm can be implemented with only a minor modification to A*. In **Dijkstra's algorithm**, there is no heuristic estimate—or in other words:

$$f(x) = g(x) + h(x)$$
$$h(x) = 0$$
$$\therefore f(x) = g(x)$$

This means that Dijkstra's algorithm can be implemented with the same code as A* if we use zero as the heuristic. If we apply Dijkstra's algorithm to our sample data set, we get a path that is identical to the one generated by A*. Provided that the heuristic used for A* was admissible, Dijkstra's algorithm will always return the same path as A*. However, Dijkstra's algorithm typically ends up visiting more nodes, which means it's less efficient than A*.

The only scenario where Dijkstra's might be used instead of A* is when there are multiple valid goal nodes, but you have no idea which one is the closest. But that scenario is rare enough that most games do not use Dijkstra's; the algorithm is mainly discussed for historical reasons, as Dijkstra's algorithm actually *predates* A* by nearly ten years. A* was an innovation that combined both the greedy best-first search and Dijkstra's algorithm. So even though this book covers Dijkstra's through the lens of A*, that is not how it was originally developed.

State-Based Behaviors

The most basic AI behavior has no concept of different states. Take for instance an AI for *Pong*: It only needs to track the position of the ball. This behavior never changes throughout the course of the game, so such an AI would be considered **stateless**. But add a bit more complexity to the game, and the AI needs to behave differently at different times. Most modern games have NPCs that can be in one of many states, depending on the situation. This section discusses how to design an AI with states in mind, and covers how these state machines might be implemented.

State Machines for AI

A **finite state machine** is a perfect model to represent the state-based behavior of an AI. It has a set number of possible states, conditions that cause transitions to other states, and actions that can be executed upon entering/exiting the state.

When you're implementing a state machine for an AI, it makes sense to first plan out what exactly the different behaviors should be and how they should be interconnected. Suppose we are implementing a basic AI for a guard in a stealth-based game. By default, we want the guard to patrol along a set path. If the guard detects the player while on patrol, it should start attacking the player. And, finally, if in either state the guard is killed, it should die. The AI as described should therefore have three different states: Patrol, Attack, and Death.

Next, we need to determine which transitions are necessary for our state machine. The transition to the Death state should be apparent: When the guard's killed, it'll enter the Death state. As for the Attack state, the only time it should be entered is if the player is spotted while the guard is patrolling. Combine these criteria together, and we can draw a simple flow chart for our state machine, as in Figure 9.9.

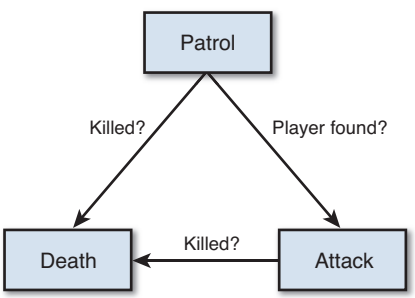


Figure 9.9 Basic stealth AI state machine.

Although this AI would be functional, it is lacking compared to the AI in most stealth games. Suppose the guard is patrolling and hears a suspicious sound. In our current state machine, the AI would go about its business with no reaction. Ideally, we would want an "Investigate" state, where the AI starts searching for the player in that vicinity. Furthermore, if the guard detects the player, it currently always attacks. Perhaps we instead want an "Alert" state that, upon being entered, randomly decides whether to start attacking or to trigger an alarm.

If we add in these elements, our state machine starts to become more complex, as in Figure 9.10.

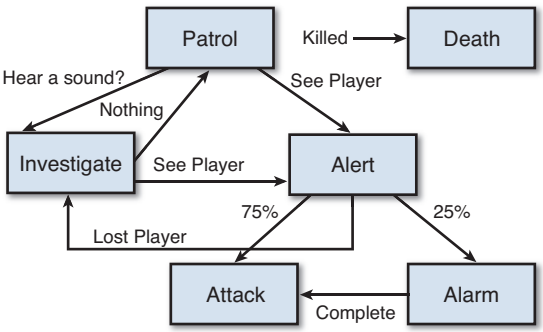


Figure 9.10 More complex stealth AI state machine.

Notice how the Death state has been detached from the others. This was done simply to remove some arrows from the flow chart, because any state can transition into the Death state when the AI is killed. Now, we could add even more complexities to our AI's behavior if we so desired. But the principles of designing an AI's behavior are the same irrespective of the number of states.

GHOST AI IN PAC-MAN

At first glance, it may seem like the AI for the arcade classic *Pac-Man* is very basic. The ghosts appear to either chase after the player or run from the player, so one might think that it's just a binary state machine. However, it turns out that the ghost AI is relatively complex.

Toru Iwatani, designer of *Pac-Man*, spoke with Susan Lammers on this very topic in her 1986 book *Programmers at Work*. He "wanted each ghostly enemy to have a specific character and its own particular movements, so they weren't all just chasing after Pac-Man... which would have been tiresome and flat."

Each of the four ghosts have four different behaviors to define different target points in relation to *Pac-Man* or the maze. The ghosts also alternate between phases of attacking and dispersing, with the attack phases increasing proportionally as the player progresses to further and further levels.

An excellent write up on the implementation of the *Pac-Man* ghost AI is available on the *Game Internals* blog at http://tinyurl.com/238l7km.

Basic State Machine Implementation

A state machine can be implemented in several ways. A minimum requirement is that when the AI updates itself, the correct update action must be performed based on the current state of the AI. Ideally, we would also like our state machine implementation to support enter and exit actions.

If the AI only has two states, we could simply use a Boolean check in the AI's `Update` function. But this solution is not very robust. A slightly more flexible implementation, often used in simple games, is to have an enumeration (`enum`) that represents all of the possible states. For instance, an `enum` for the states in Figure 9.9 would be the following:

```
enum AIState
   Patrol,
   Death,
   Attack
end
```

We could then have an `AIController` class with a member variable of type `AIState`. In our `AIController`'s `Update` function, we could simply perform the current action based on the current state:

```
function AIController.Update(float deltaTime)
   if state == Patrol
      // Perform Patrol actions
   else if state == Death
      // Perform Death actions
   else if state == Attack
      // Perform Attack actions
   end
end
```

State changes and enter/exit behaviors could then be implemented in a second function:

```
function AIController.SetState(AIState newState)
   // Exit actions
   if state == Patrol
      // Exit Patrol state
   else if state == Death
      // Exit Death state
   else if state == Attack
      // Exit Attack state
   end

   state = newState

   // Enter actions
   if state == Patrol
```

```
      // Enter Patrol state
   else if state == Death
      // Enter Death state
   else if state == Attack
      // Enter Attack state
   end
end
```

There are a number of issues with this implementation. First and foremost, as the complexity of our state machine increases, the readability of the `Update` and `SetState` functions decrease. If we had 20 states instead of the three in this example, the functions would end up looking like spaghetti code.

A second major problem is the lack of flexibility. Suppose we have two different AIs that have largely different state machines. This means that we would need to have a different `enum` and controller class for each of the different AIs. Now, suppose it turns out that both of these AIs do share a couple of states in common, most notably the Patrol state. With the basic state machine implementation, there isn't a great way to share the Patrol state code between both AI controllers.

One could simply copy the Patrol code into both classes, but having nearly identical code in two separate locations is never a good practice. Another option might be to make a base class that each AI inherits from and then "bubble up" the majority of the Patrol behavior to this base class. But this, too, has drawbacks: It means that any subsequent AI that needs a Patrol behavior also must inherit from this shared class.

So even though this basic implementation is functional, it is not recommended for more than the simplest of AI state machines.

State Design Pattern

A **design pattern** is a generalized solution to a commonly occurring problem. The seminal book on design patterns is the so-called "Gang of Four" *Design Patterns* book, which is referenced at the end of this chapter. One of the design patterns is the State pattern, which allows "an object to alter its behavior when its internal state changes."

This can be implemented using class composition. So the `AIController` "has-a" `AIState` as a member variable. Each specific state then subclasses `AIState`. Figure 9.11 shows a rough class diagram.

The `AIState` base class could then be defined as follows:

```
class AIState
   AIController parent
   function Update(float deltaTime)
```

```
    function Enter()
    function Exit()
end
```

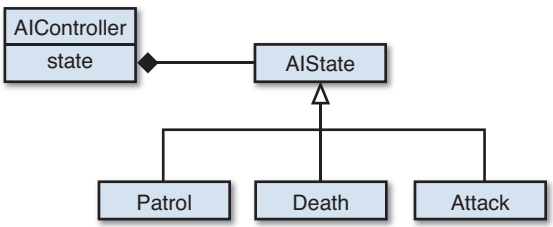


Figure 9.11 State design pattern.

The parent reference allows a particular instance of an `AIState` to keep track of which `AIController` owns it. This is required because if we want to switch into a new state, there has to be some way to notify the `AIController` of this. Each `AIState` also has its own `Update`, `Enter`, and `Exit` functions, which can implement state-specific actions.

The `AIController` class could then have a reference to the current `AIState` and the required `Update` and `SetState` functions:

```
class AIController
   AIState state
   function Update(float deltaTime)
   function SetState(AIState newState)
end
```

This way, the `AIController`'s `Update` function can simply call the current state's `Update`:

```
function AIController.Update(float deltaTime)
   state.Update(deltaTime)
end
```

With the design pattern implementation, the `SetState` function is also much cleaner:

```
function AIController.SetState(AIState newState)
   state.Exit()
   state = newState
   state.Enter()
end
```

With the state design pattern, all state-specific behavior is moved into the subclasses of `AIState`. This makes the `AIController` code much cleaner than in the previous implementation. The State design pattern also makes the system far more modular. For instance, if we wanted to use the Patrol code in multiple state machines, it could be dropped in relatively

easily. And even if we needed slightly different Patrol behavior, the Patrol state could be subclassed.

Strategy and Planning

Certain types of games require AI that is more complex than state-based enemies. In genres such as real-time strategy (RTS), the AI is expected to behave in a manner similar to a human player. The AI needs to have an overarching vision of what it wants to do, and then be able to act on that vision. This is where strategy and planning come into play. In this section, we will take a high-level look at strategy and planning through the lens of the RTS genre, though the techniques described could apply to other genres as well.

Strategy

Strategy is an AI's vision of how it should compete in the game. For instance, should it be more aggressive or more defensive? In an RTS game, there are often two components of strategy: micro and macro strategy. **Micro** strategy consists of per-unit actions. This can typically be implemented with behavioral state machines, so it is nothing that requires further investigation. **Macro** strategy, however, is a far more complex problem. It is the AI's overarching strategy, and it determines how it wants to approach the game. When programming AIs for games such as *StarCraft*, developers often model the strategy after top professional players. An example of a macro strategy in an RTS game might be to "rush" (try to attack the other player as quickly as possible).

Strategies can sometimes seem like nebulous mission statements, and vague strategies are difficult to program. To make things more tangible, strategies are often thought in terms of specific goals. For example, if the strategy is to "tech" (increase the army's technology), one specific goal might be to "expand" (create a secondary base).

> ### Note
>
> Full-game RTS AI strategy can be extremely impressive when implemented well. The annual Artificial Intelligence and Interactive Digital Entertainment (AIIDE) conference has a *StarCraft* AI competition. Teams from different universities create full-game AI bots that fight it out over hundreds of games. The results are quite interesting, with the AI displaying proficiency that approaches that of an experienced player. More information as well as videos are available here at www.starcraftaicompetition.com.

A single strategy typically has more than one goal. This means that we need a prioritization system to allow the AI to pick which goal is most important. All other goals would be left on the

back burner while the AI attempts to achieve its highest priority one. One way to implement a goal system would be to have a base `AIGoal` class as such this:

```
class AIGoal
   function CalculatePriority()
   function ConstructPlan()
end
```

Each specific goal would then be implemented as a subclass of `AIGoal`. So when a strategy is selected, all of that strategy's goals would be added to a container that is sorted by priority. Note that a truly advanced strategy system should support the idea of multiple goals being active at once. If two goals are not mutually exclusive, there is no reason why the AI should not attempt to achieve both goals at the same time.

The heuristic for calculating the priority in `CalculatePriority` may be fairly complex, and will be wildly different depending on the game rules. For instance, a goal of "build air units" may have its priority reduced if the AI discovers that the enemy is building units that can destroy air units. By the same token, its priority may increase if the AI discovers the enemy has no anti-air capabilities.

The `ConstructPlan` function in `AIGoal` is responsible for constructing the **plan**: the series of steps that need to be followed in order to achieve the desired goal.

Planning

Each goal will need to have a corresponding plan. For instance, if the goal is to expand, then the plan may involve the following steps:

1. Scout for a suitable location for an expansion.
2. Build enough units to be able to defend the expansion.
3. Send a worker plus the defender units to the expansion spot.
4. Start building the expansion.

The plan for a specific goal could be implemented by a state machine. Each step in the plan could be a state in the state machine, and the state machine can keep updating a particular step until the condition is met to move on to the next state. However, in practice a plan can rarely be so linear. Based on the success or failure of different parts of the plan, the AI may have to modify the ordering of its steps.

One important consideration is that the plan needs to periodically assess the goal's feasibility. If during our expansion plan it is discovered that there are no suitable expansion locations, then the goal cannot be achieved. Once a goal is flagged as unachievable, the overarching strategy also needs to be reevaluated. Ultimately, there is the necessity of having a "commander" who will decide when the strategy should be changed.

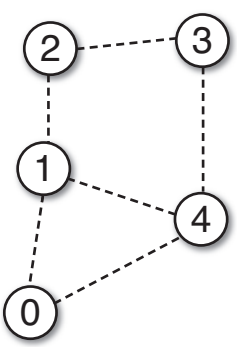
The realm of strategy and planning can get incredibly complex, especially if it's for a fully featured RTS such as *StarCraft*. For further information on this topic, check the references at the end of this chapter.

Summary

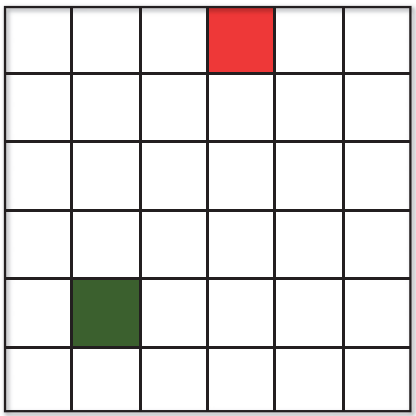
The goal of a game AI programmer is to create a system that appears to be intelligent. This may not be what a researcher considers "true" AI, but games typically do not need such complex AIs. As we discussed, one big problem area in game AI is finding the best path from point A to point B. The most common pathfinding algorithm used in games is A*, and it can be applied on any search space that can be represented by a graph (which includes a grid, path nodes, or navigation meshes). Most games also have some sort of AI behavior that is controlled by a state machine, and the best way to implement a state machine is with the State design pattern. Finally, strategy and planning may be utilized by genres such as RTS in order to create more organic and believable opponents.

Review Questions

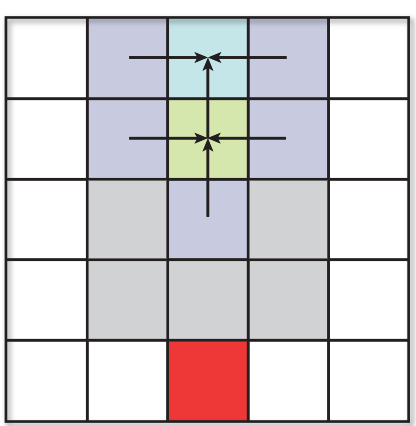
1. Given the following graph, what are the adjacency lists for each node?



2. Why are navigation meshes generally preferred over path nodes?
3. When is a heuristic considered admissible?
4. Calculate the Manhattan and Euclidean distances between the red and green nodes in the following diagram:



5. The A* algorithm is being applied to the following data. Nodes adjacent to the current node have been added, so the algorithm needs to select a new current node. Calculate the $f(x)$ cost of each node in the open set (in blue) using a Manhattan heuristic, and assume diagonal travel is not allowed. Which node is selected as the new current node?



6. If the A* algorithm has its heuristic changed such that $h(x) = 0$, which algorithm does this mimic?
7. Compare A* to Dijkstra's algorithm. Is one preferred over the other, and why?
8. You are designing an AI for an enemy wolf. Construct a simple behavioral state machine for this enemy with at least five states.
9. What are the advantages of the state design pattern?
10. What is a strategy, and how can it be broken down into smaller components?

Additional References

General AI

Game/AI (http://ai-blog.net/**).** This blog has articles from several programmers with industry experience who talk about many topics related to AI programming.

Millington, Ian and John Funge. *Artificial Intelligence for Games* (2nd Edition). Burlington: Morgan Kaufmann, 2009. This book is an algorithmic approach to many common game AI problems.

***Game Programming Gems* (Series)**. Each volume of this series has a few articles on AI programming. The older volumes are out of print, but the newer ones are still available.

***AI Programming Wisdom* (Series)**. Much like the *Game Programming Gems* series, except 100% focused on AI for games. Some of the volumes are out of print.

Pathfinding

Recast and Detour (http://code.google.com/p/recastnavigation/**).** An excellent open source pathfinding library written by Mikko Mononen, who has worked on several games including *Crysis*. Recast automatically generates navigation meshes, and Detour implements pathfinding within the navigation mesh.

States

Gamma, Eric et. al. *Design Patterns: Elements of Reusable Object-Oriented Software*. Boston: Addison-Wesley, 1995. Describes the state design pattern as well as many other design patterns. Useful for any programmer.

Buckland, Mat. *Programming Game AI By Example*. Plano: Wordware Publishing, 2005. This is a general game AI book, but it also has great examples of state-based behavior implementations.

CHAPTER 10

USER INTERFACES

A game typically has two main components to its user interface: a menu system and an in-game heads-up display (HUD). The menu system defines how the player gets in and out of the game—including changing modes, pausing the game, selecting options, and so on. Some games (especially RPGs) may also have menus to manage an inventory or upgrade skills.

The HUD includes any elements that give additional information to the player as he or she is actually playing the game. This can include a radar, ammo count, compass, and an aiming reticule. Although not all games will have a HUD (and some might have the option to disable the HUD), the vast majority have at least a basic one.

Menu Systems

A well-implemented menu system should provide for flexibility in a lot of different ways—there should be no limit to the number of elements and distinct screens, and it should be easy to quickly add new submenus. At the same time, it must be implemented in a manner that centralizes as much common functionality as possible. This section discusses what must be taken into account to implement a solid menu system; many of the techniques discussed here are also utilized in the tower defense game in Chapter 14, "Sample Game: Tower Defense for PC/Mac."

Menu Stack

The menu system for a typical console game might start with the platform-mandated "Press Start" screen. Once the user presses Start, he enters the main menu. Perhaps he can go into Options, which brings up an options menu, or maybe he can take a look at the credits or instructions on how to play. Typically, the player is also provided a way to exit the current menu and return to a previous one. This sort of traditional menu flow is illustrated in Figure 10.1.

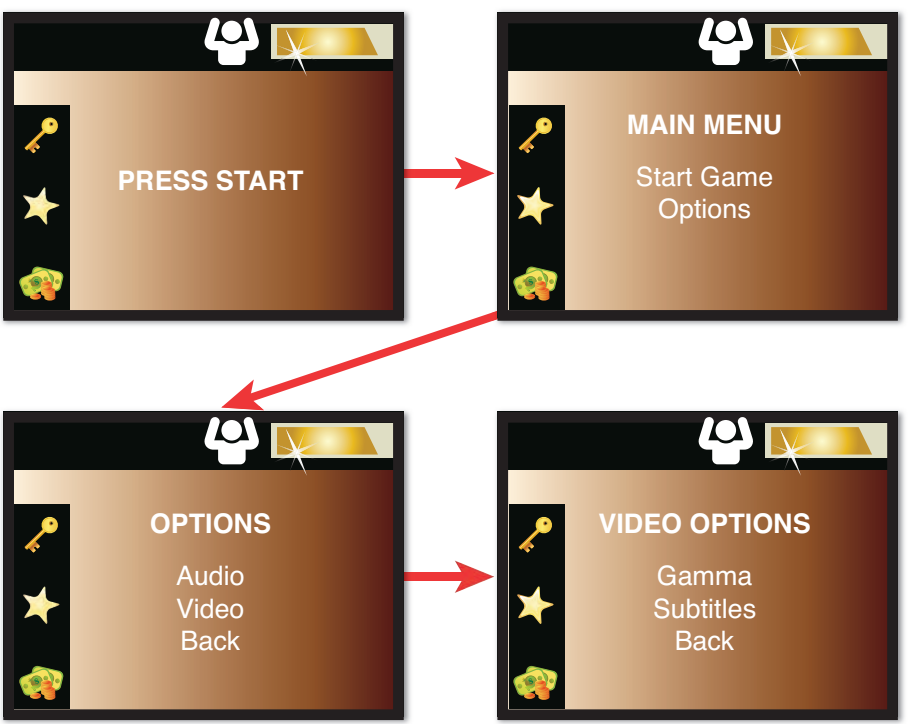


Figure 10.1 Sample console game menu flow.

One way to ensure menu flow can always return to the base menu is by utilizing the stack data structure. The "top" element on the stack is the currently active menu item, and going to a new menu involves pushing that menu onto the stack. Returning to the previous menu involves popping the current menu off the stack. This can be further modified so that multiple menus can be visible at once—for example, a dialog box can appear on top of a particular menu if there's a need to accept/reject a request. In order to do this, the menus would need to be rendered from the bottom to the top of the stack.

To store all the menus in one stack, you will need some sort of base class from which all menus derive. This base class may store information such as the title of the menu and a linked list of buttons (or sub-elements that the menu has). I used an approach similar to this to implement the menu system for *Tony Hawk's Project 8*, and it's also the approach that's used for the overall UI in Chapter 14's tower defense game. One aspect of this system that's not quite a standard stack is the operation to allow the stack to be cleared and have a new element pushed on top. This is necessary when going from the main menu into gameplay, because you typically don't want the main menu to still be visible while in gameplay.

Buttons

Almost every menu system has buttons that the user can interact with. For a PC or console game, there needs to be two visual states for a button: unselected and selected. This way, the currently selected button is clear to the user. The simplest way to signify a button is selected is by changing its color, but sometimes the texture might change or the size of the button increases. Some PC/console menus also use a third state to denote when a button is being pressed, but this third state is not required by any means. However, on a touch device a common approach is to have only the default (unselected) state and a pressed state. This way, when the user taps on the button, it will change to a different visual state.

If the menu can only be navigated with a keyboard or controller, supporting buttons is fairly straightforward. A particular menu screen could have a doubly linked list of buttons, and when the user presses the appropriate keys (such as up and down), the system can unselect the current button and select the next one. Because the buttons are in a doubly linked list, it's easy to go back or forward, and also wrap around if the user goes past the first or last element in the list.

Usually when a button is pressed, the user expects something to happen. One way to abstractly support this is to have a member variable in the button class that's a function. This will vary based on the language, but it could be an action (as in C#), a function pointer (as in C), or a lambda expression (as in C++). This way, when a new button is created, you can simply associate it with the correct function and that function will execute when the button is pressed.

If the game also supports menu navigation via the mouse, a little bit of complexity has to be added to the system. Each button needs to have a **hot zone**, or 2D bounding box that represents where the mouse can select a particular button. So as the user moves the mouse around the menu, the code needs to check whether the new position of the mouse intersects with the hot zone of any buttons. This approach is used for menu navigation in Chapter 14's game, and is illustrated in Figure 10.2.

Figure 10.2 A debug mode shows the hot zones of the main menu buttons in Chapter 14's game.

It's also possible to allow the user to seamlessly switch between mouse and keyboard navigation of the menus. One common way to accomplish this is to hide the mouse cursor (and ignore the mouse position) when the user presses a keyboard navigation button. Then, as soon as the mouse is moved again, mouse selection is activated once more. This type of system is further extended by games that support a keyboard/mouse as well as a controller. In these cases, it might also be designed so that the menu navigation tips change based on whether the controller or the keyboard/mouse is active.

Typing

A typical usage case for a computer game is to allow the player to type in one or more words while in a menu. This might be for the purposes of a high score list, or maybe so the player can type in a filename to save/load. When a traditional program supports typing in words, it's usually done via standard input, but as previously covered in Chapter 5, "Input," standard input typically is not available for a game.

The first step to allow typing is to start with an empty string. Every time the player types in a letter, we can then append the appropriate character to said string. But in order to do this, we need to determine the correct character. Recall that Chapter 5 also discussed the idea of virtual keys (how every key on the keyboard corresponds to an index in an `enum`). So, for example, `K_A` might correspond to the "A" key on the keyboard. Conveniently, in a typical system, the letter virtual keys are sequential within the `enum`. This means that that `K_B` would be one index after `K_A`, `K_C` would be an index after `K_B`, and so on. It just so happens that the ASCII character "B" is also sequentially after the ASCII character "A." Taking advantage of this parallel allows us to implement a function that converts from a letter key code to a particular character:

```
function KeyCodeToChar(int keyCode)
   // Make sure this is a letter key
   if keyCode >= K_A && keyCode <= K_Z
      // For now, assume upper case.
      // Depending on language, may have to cast to a char
      return ('A' + (char)(keyCode - K_A))
   else if keyCode == K_SPACE
      return ' '
   else
      return ''
   end
end
```

Let's test out this code with a couple examples. If `keyCode` is `K_A`, the result of the subtraction should be 0, which means the letter returned is simply "A." If instead `keyCode` is `K_C`, the subtraction would yield 2, which means the function would return A + 2, or the letter "C." These examples prove that the function gives the results we expect.

Once we've implemented this conversion function, the only other code that needs to be written is the code that checks whether any key from `K_A` to `K_Z` was "just pressed." If this happens, we convert that key to its appropriate code and append said character to our string. If we wanted to, we could also further extend `KeyCodeToChar` to support upper- and lowercase by defaulting to lowercase letters, and only switching to uppercase letters if the Shift key is also down at the time.

HUD Elements

The most basic **HUD** (heads-up display) is one that displays elements such as the player's score and number of lives remaining. This type of HUD is relatively trivial to implement—once the main game world has been rendered, we only need to draw on top of it some text or icons that convey the appropriate information. But certain games utilize more complex types of elements, including waypoint arrows, radars, compasses, and aiming reticules. This section takes a look at how some of these elements might be implemented.

Waypoint Arrow

A waypoint arrow is designed to point the player to the next objective. One of the easiest ways to implement such an arrow is to use an actual 3D arrow object that is placed at a set location onscreen. Then as the player moves around in the world, this 3D arrow can rotate to point in the direction that the player needs to travel. This type of waypoint arrow has been used in games such as *Crazy Taxi*, and an illustration of such an arrow is shown in Figure 10.3.

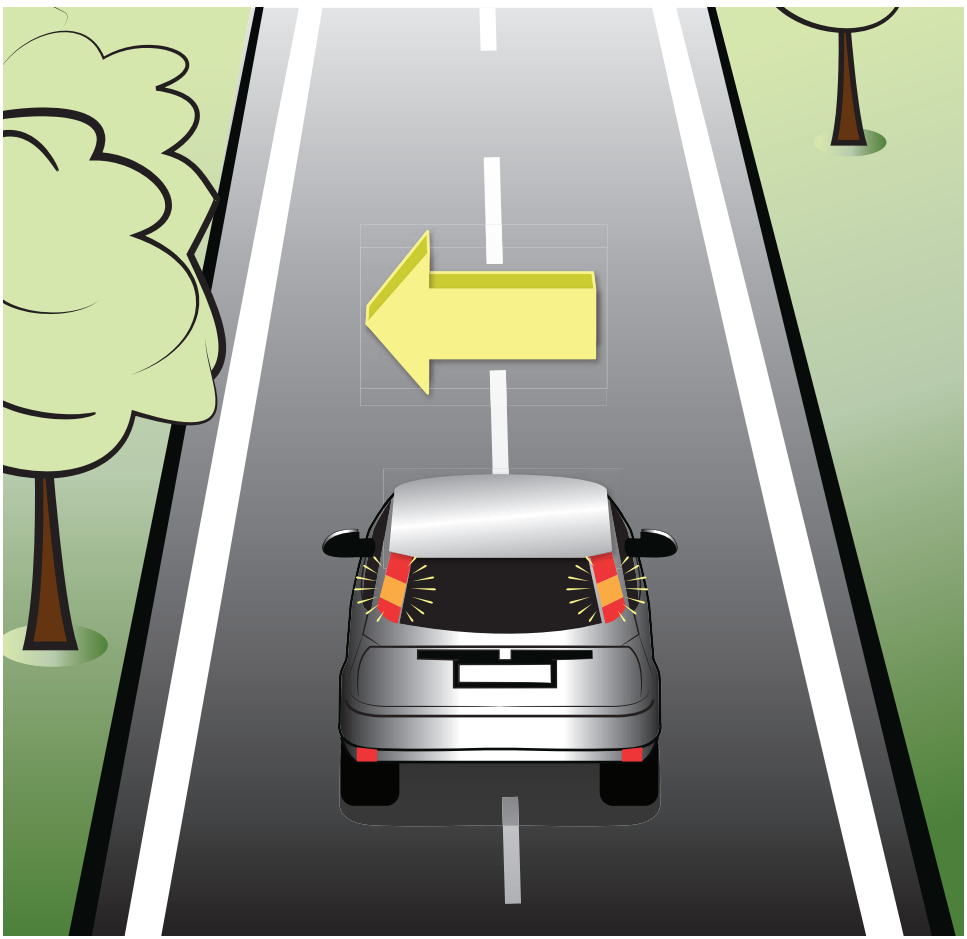
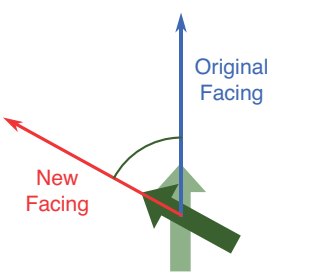


Figure 10.3 A driving game where a waypoint arrow instructs the player to turn left.

The first step in implementing this type of waypoint arrow is to create an arrow model that points straight forward when no rotation is applied to it. Then there are three parameters to keep track of during gameplay: a vector that represents the facing of the arrow, the screen space location of the arrow, and the current waypoint the arrow should be tracking.

The facing vector should first be initialized to the axis that travels into the screen. In a traditional left-handed coordinate system, this will be the +z-axis. The screen space location of the arrow corresponds to where we want the arrow to be onscreen. Because it's a screen space position, it will be an (x, y) coordinate that we need to convert into a 3D world space coordinate for the purposes of rendering the 3D arrow in the correct spot. In order to do this, we will need to use an unprojection, much like in the "Picking" section of Chapter 8, "Cameras." Finally, the waypoint position is the target at which the arrow will point.

In order to update the waypoint arrow, in every frame we must construct a vector *from* the player's position *to* the target; once normalized, this will give us the direction the arrow should be facing. Then it's a matter of using the dot product and cross product to determine the angle and axis of rotation between the original (straight into the screen) facing vector and the new facing vector. The rotation can then be performed with a quaternion, using a lerp if we want the arrow to more smoothly rotate to the target. Figure 10.4 shows the basic calculations that must be done to update the waypoint arrow.



$$\theta = acos(old \cdot new)$$

$$axis = old \times new$$

Figure 10.4 Waypoint arrow calculations, assuming old and new are normalized.

An implementation of this type of waypoint arrow is provided in Listing 10.1. There are a couple of things to keep in mind with this implementation. First of all, it assumes that the camera is updated prior to the waypoint arrow being updated. That's because it's the only way to really guarantee that the waypoint arrow is always at the same spot on the screen. Otherwise, when the camera changes, the waypoint arrow will always be one frame behind.

Furthermore, the waypoint arrow must be rendered with z-buffering disabled and after all the other 3D objects in the world have been rendered. This ensures that the arrow is always visible, even if there is a 3D object that should technically be in front of it.

Listing 10.1 Waypoint Arrow

```
class WaypointArrow
   // Stores the current facing of the arrow
   Vector3 facing
   // 2D position of the arrow on screen
   Vector2 screenPosition
   // Current waypoint the arrow points at
   Vector3 waypoint
   // World transform matrix used to render the arrow
   Matrix worldTransform

   // Computes world transform matrix from given position/rotation
   function ComputeMatrix(Vector3 worldPosition, Quaternion rotation)
      // Scale, rotate, translate (but we don't have a scale this time)
      worldTransform = CreateFromQuaternion(rotation) *
                       CreateTranslation(worldPosition)
   end

   // Gets world position of the 3D arrow based on screenPosition
   function ComputeWorldPosition()
      // In order to do the unprojection, we need a 3D vector.
      // The z component is the percent between the near and far plane.
      // In this case, I select a point 10% between the two (z=0.1).
      Vector3 unprojectPos = Vector3(screenPosition.x,
                                     screenPosition.y, 0.1)

      // Grab the camera and projection matrices
      ...

      // Call Unproject function from Chapter 8
      return Unproject(unprojectPos, cameraMatrix, projectionMatrix)
   end

   function Initialize(Vector2 myScreenPos, Vector3 myWaypoint)
      screenPosition = myScreenPos
      // For left-handed coordinate system with Y up
      facing = Vector3(0, 0, 1)
      SetNewWaypoint(myWaypoint)

      // Initialize the world transform matrix
      ComputeMatrix(ComputeWorldPosition(), Quaternion.Identity)
   end

   function SetNewWaypoint(Vector3 myWaypoint)
      waypoint = myWaypoint
   end
```

```
    function Update(float deltaTime)
       // Get the current world position of the arrow
       Vector3 worldPos = ComputeWorldPosition()

       // Grab player position
       ...
       // The new facing of the arrow is the normalized vector
       // from the player's position to the waypoint.
       facing = waypoint - playerPosition
       facing.Normalize()

       // Use the dot product to get the angle between the original
       // facing (0, 0, 1) and the new one
       float angle = acos(DotProduct(Vector3(0, 0, 1), facing))
       // Use the cross product to get the axis of rotation
       Vector3 axis = CrossProduct(Vector3(0, 0, 1), facing)
       Quaternion quat
       // If the magnitude is 0, it means they are parallel,
       // which means no rotation should occur.
       if axis.Length() < 0.01f
          quat = Quaternion.Identity
       else
          // Compute the quaternion representing this rotation
          axis.Normalize()
          quat = CreateFromAxisAngle(axis, angle)
       end

       // Now set the final world transform of the arrow
       ComputeMatrix(worldPos, quat)
    end
end
```

Aiming Reticule

An aiming reticule is a standard HUD element used in most first- and third-person games that have ranged combat. It allows the player to know where he is aiming as well as to acquire further information regarding the target (such as whether the target is a friend or foe). Whether the reticule is a traditional crosshair or more circular, the implementation ends up being roughly the same. In fact, the implementation is extremely similar to mouse picking as discussed in Chapter 8.

As with a mouse cursor, with an aiming reticule there will be a 2D position on the screen. We take this 2D position and perform two unprojections: one at the near plane and one at the far plane. Given these two points, we can perform a ray cast from the near plane point to the far plane point. We can then use the physics calculations discussed in Chapter 7, "Physics," to generate a list of all the objects the ray cast intersects with. From the resultant list, we want

to select the first object that the ray cast intersects with. In a simple game, it may be the case that the first object the ray cast intersects with is also the one whose position is closest to the near plane point. But with larger objects, that may not always be the case. So for more complex games, we might need to actually compare the points of intersection between the ray and each object.

Once we know which object the aiming ray intersects with, we can then check to see whether it's a friendly, foe, or an object that can't be targeted. This then determines what color the reticule should be rendered in—most games use green to signify a friendly, red to signify a foe, and white to signify that the object is not a target (though this coloring scheme is not particularly usable for a player who happens to be red-green color blind). Figure 10.5 illustrates a couple of different scenarios with an aiming reticule in a hypothetical game.

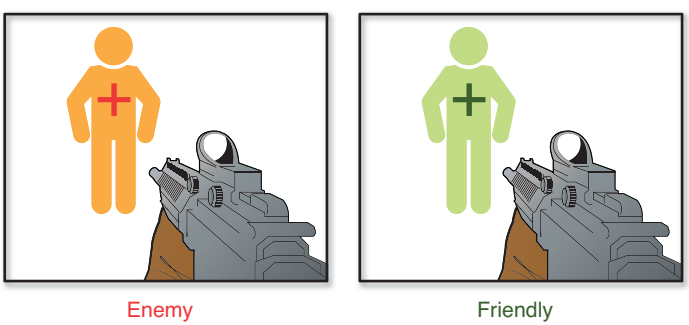


Figure 10.5 An aiming reticule changes colors based on the target.

Radar

Some games have a radar that displays nearby enemies (or friendlies) who are within a certain radius of the player. There are a couple of different radar variations; in one variation, anyone within the proper radius will show up on the radar. In another variation, enemies only show up on the radar if they have recently fired a weapon. Regardless of the variation, however, the implementation ends up being roughly the same. A sample screenshot of a game with a radar is shown in Figure 10.6.

Two main things must be done in order to get a radar to work. First, we need a way to iterate through all of the objects that could show up on the radar, and check whether or not they are within the range of the radar. Then any objects that are within range of the radar must be converted into the appropriate 2D offset from the center of the radar in the UI. Both when calculating the distance and converting into a 2D offset, we want to ignore the height component. This means we are essentially projecting the radar objects onto the plane of the radar, which sounds harder than it actually is in practice.

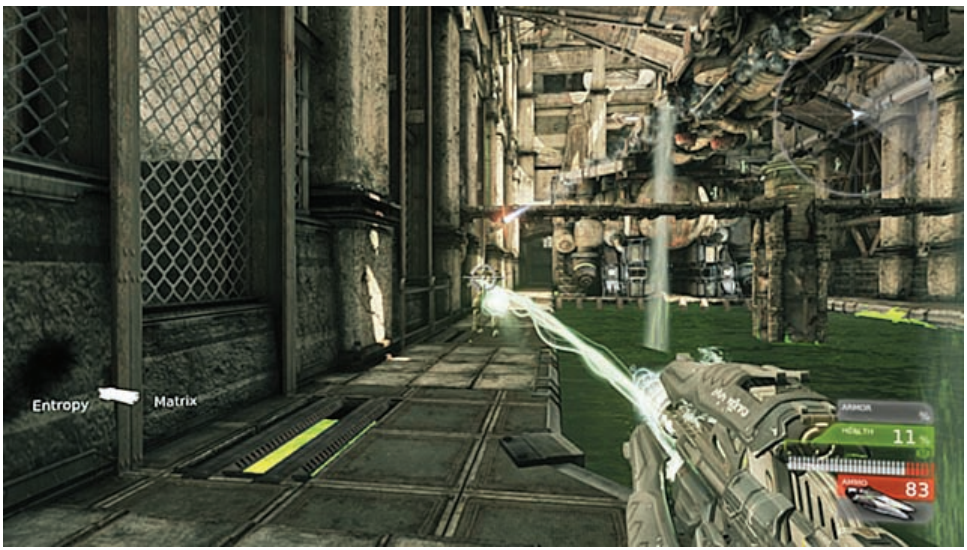


Figure 10.6 Radar (in the top right corner) in *Unreal Tournament 3*.

But before we do these calculations, it's worthwhile to declare a `struct` for a **radar blip**, or the actual dot that represents an object on the radar. This way, it would be possible to have different size and color blips depending on the target in question.

```
struct RadarBlip
   // The color of the radar blip
   Color color = Color.Red
   // The 2D position of the radar blip
   Vector2 position
   // Scale of the radar blip
   float scale = 1.0f
end
```

We could also inherit from `RadarBlip` to create all sorts of different types of blips for different enemies if we so desired. But that's more of an aesthetic decision, and it's irrelevant to the calculations behind implementing the radar.

For the actual radar class, there are two main parameters to set: the maximum distance an object can be detected in the game world, and the radius of the radar that's displayed onscreen. With these two parameters, once we have a blip position it is possible to convert it into the correct location on the screen.

Suppose a game has a radar that can detect objects 50 units away. Now imagine there is an object 25 units straight in front of the player. Because the object positions are in 3D, we need to first convert both the position of the player and the object in question into 2D coordinates for

the purposes of the radar. If the world is y-up, this means the radar's plane is the xz-plane. So to project onto the plane, we can ignore the y-component because it represents the height. In other words, we want to take a 3D point that's (x, y, z) and map it instead to a 2D point that's essentially (x, z), though a 2D point's second component will always be referred to as "y" in the actual code.

Once both the player and object's positions are converted into 2D coordinates projected onto the plane of the radar, we can construct a vector from the player to the object, which for clarity we'll refer to as vector $\vec{a}$. Although $\vec{a}$ can be used to determine whether or not an object is within range of the radar (by computing the length of $\vec{a}$), there is one problem with this vector. Most game radars rotate as the player rotates, such that the topmost point on the radar (90° on a unit circle) always corresponds to the direction the player is facing. So in order to allow for such functionality, $\vec{a}$ must be rotated depending on the player's facing vector. This issue is illustrated in Figure 10.7.

Figure 10.7 A top-down view of the player facing to the east (a), so east should correspond to the topmost point on the radar (b).

To solve this problem, we can take the normalized facing vector of the player and project it, too, onto the plane of the radar. If we then take the dot product between this projected vector and the "forward" vector of the radar onscreen, which will usually be $\langle 0, 1 \rangle$, we can determine the angle of rotation. Then, much as in the "Rotating a 2D Character" sample problem in Chapter 3, "Linear Algebra for Games," we can convert both vectors into 3D vectors with a z-component of 0 and use the cross product to determine whether the rotation should be performed clockwise or counterclockwise. Remember that if the cross product between two converted 2D vectors yields a positive z-value, the rotation between the two vectors is counterclockwise.

Armed with the angle of rotation and whether the rotation should be clockwise or counter-clockwise, we can use a 2D rotation matrix to rotate $\vec{a}$ to face the correct direction. A 2D rotation matrix only has one form, because the rotation is always about the z-axis. Assuming row-major vectors, the 2D rotation matrix is

$$Rotation2D(\theta) = \begin{bmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{bmatrix}$$

In this case, the θ we pass in will be either positive or negative, depending on whether the rotation should be counterclockwise (positive) or clockwise (negative), which we know based on the cross product result.

Returning to our example, if the vector $\vec{a}$ is the vector from the player to an object 25 units in front of the player, after the rotation is applied we should end up with $\vec{a} = \langle 0,\ 25 \rangle$. Once we have this 2D vector, we then divide each component of it by the maximum distance of the radar to get the vector that represents where the blip would be positioned if the radar were a unit circle—so in this case, $\langle 0,\ 25 \rangle / 50 = \langle 0,\ 0.5 \rangle$. We then take the radius of the radar on the screen and multiply each component of our 2D vector by it. So if the radar onscreen had a radius of 200 pixels, we would end up with the 2D vector $\langle 0,\ 100 \rangle$, which represents the offset from the center of the radar to where the blip should be positioned. Figure 10.8 shows what the radar would look like for our example of an enemy 25 units directly in front of the player.

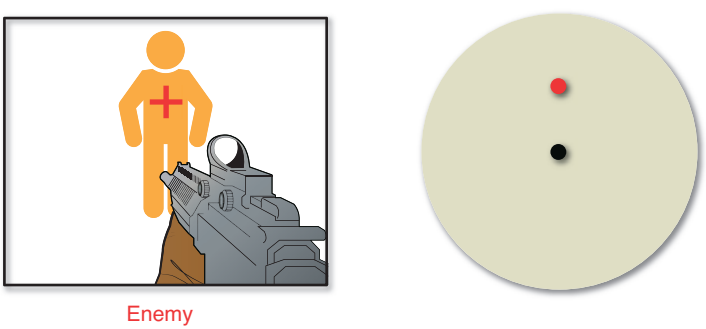


Figure 10.8 An enemy 25 units directly in front on a radar with a max range of 50.

These calculations can then be applied to every object that's within range in order to determine which blips to draw. A full implementation of a radar is provided in Listing 10.2.

Listing 10.2 Radar System

```
class Radar
   // Range of radar in game world units
   float range
   // (x,y) center of radar on screen
   Vector2 position
   // Radius of radar on screen
   float radius
   // Radar background image
```

```
ImageFile radarImage
// List of all active RadarBlips
List blips

// Initialize function sets range, center, radius, and image
...

function Update(float deltaTime)
   // Clear out the List of blips from last frame
   blips.Clear()
   // Get the player's position
   ...
   // Convert playerPosition to a 2D coordinate.
   // The below assumes a y-up world.
   Vector2 playerPos2D = Vector2(playerPosition.x, playerPosition.z)

   // Calculate the rotation that may need to be applied to the blip
   // Get the players normalized facing vector
   ...
   // Convert playerFacing to 2D
   Vector2 playerFacing2D = Vector2(playerFacing.x, playerFacing.z)
   // Angle between the player's 2D facing and radar "forward"
   float angle = acos(DotProduct(playerFacing2D, Vector2(0,1)))
   // Convert 3D vector so we can perform a cross product
   Vector3 playerFacing3D = Vector3(playerFacing2D.x,
           playerFacing2D.y, 0)
   // Use cross product to determine which way to rotate
   Vector3 crossResult = CrossProduct(playerFacing3D,
                                      Vector2(0,1,0))
   // Clockwise is -z, and it means the angle needs to be negative.
   if crossResult.z < 0
      angle *= -1
   end

   // Determine which enemies are in range
   foreach Enemy e in gameWorld
      // Convert Enemy's position to 2D coordinate
      Vector2 enemyPos2D = Vector2(e.position.x, e.position.z)
      // Construct vector from player to enemy
      Vector2 playerToEnemy = enemyPos2D - playerPos2D
      // Check the length, and see if it's within range
      if playerToEnemy.Length() <= range
         // Rotate playerToEnemy so it's oriented relative to
         // the player's facing (using a 2D rotation matrix).
         playerToEnemy = Rotate2D(angle)
         // Make a radar blip for this Enemy
         RadarBlip blip
```

```
            // Take the playerToEnemy vector and convert it to
            // offset from the center of the on-screen radar.
            blip.position = playerToEnemy
            blip.position /= range
            blip.position *= radius
            // Add blip to list of blips
            blips.Add(blip)
         end
      loop
   end

   function Draw(float deltaTime)
      // Draw radarImage
      ...

      foreach RadarBlip r in blips
         // Draw r at position + blip.position, since the blip
         // contains the offset from the radar center.
         ...
      loop
   end
end
```

This radar could be improved in a few different ways. First of all, we may want to have the radar not only show enemies, but allies as well. In this case, we could just change the code so that it loops through both enemies and allies, setting the color of each `RadarBlip` as appropriate. Another common improvement for games that have a sense of verticality is to show different blips depending on whether the enemy is above, below, or on roughly the same level as the player. In order to support this, we can compare the height value of the player to the height value of the enemy that's on the radar, and use that to determine which blip to show.

A further modification might be to only show blips for enemies that have fired their weapons recently, as in *Call of Duty*. In order to support this, every time an enemy fires, it might have a flag temporarily set that denotes when it is visible on the radar. So when iterating through the enemies for potential blips, we can also check whether or not the flag is set. If the flag isn't set, the enemy is ignored for the purposes of the radar.

Other UI Considerations

Other UI considerations include support for multiple resolutions, localizations, UI middleware, and the user experience.

Supporting Multiple Resolutions

For PC games, it's rather common to have a wide range of screen resolutions—the most popular for new monitors is 1920×1080, but other common resolutions still see use, such as 1680×1050. This means that the total number of pixels the user interface has to play with can vary from monitor to monitor. But supporting multiple resolutions isn't only in the realm of computer games. For example, both the Xbox 360 and PS3 require games to support traditional CRT televisions in addition to widescreen ones (though the Xbox One and PS4 do not support older televisions, so games for these platforms may not have to worry about more than one resolution).

One way to support multiple resolutions is to avoid using specific pixel locations, which are known as **absolute coordinates**. An example of an absolute coordinate would be (1900, 1000), or the precise pixel that a UI element is drawn at. The problem with using this type of coordinate is that if the monitor is only running at 1680×1050, a UI element at (1900, 1000) would be completely off the screen.

The solution to this type of problem is to instead use **relative coordinates**, or coordinates that are relative to something else. For example, if you want something in the bottom-right corner of the screen, you might place an element at (–100, –100) *relative* to the bottom-right corner. This means that the element would be placed at (1820, 980) on a 1920×1080 screen, whereas it would be placed at (1580, 950) on a 1680×1050 screen (as illustrated in Figure 10.9).

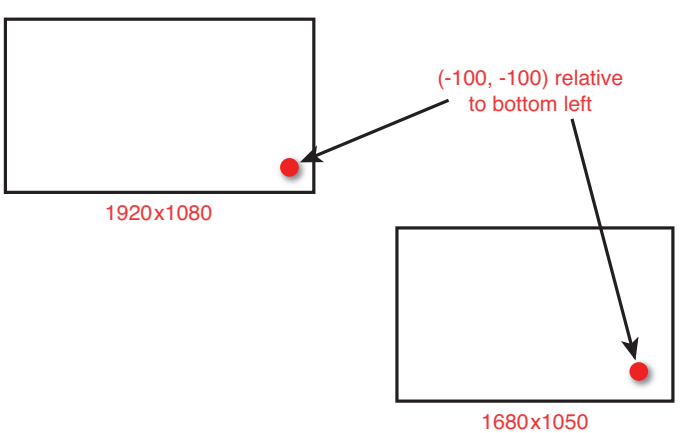


Figure 10.9 A UI element is positioned relative to the bottom-right corner of the screen.

Relative coordinates can be expressed relative to key points on the screen (usually the corners or the center of the screen), or even relative to other UI elements. Implementing this second approach is a bit more complex, and beyond the scope of this chapter.

One refinement might also be to scale the size of UI elements depending on the resolution. The reason for this is that at very high resolutions, the UI might just become too small to be usable. So at higher resolutions, the UI can be scaled up so it's more easily seen by the player. Some MMORPGs even have a UI scale slider that allows the player to adjust the UI with a high degree of granularity. If scaling is to be supported, it's doubly important that relative coordinates are utilized.

Localization

Although it may be acceptable for smaller games to support only one language (usually English), most commercial games typically need to support more than one language. **Localization** is the process of adding support for these additional languages. Because many menus and HUD elements have text, it's important to take localization into account when designing user interface systems. Even if a game does not need to be localized, it's a bad idea to have onscreen text hard-coded into source files because it makes it more difficult for nonprogrammers to modify. But if the game is to be localized, staying away from hard-coded text is particularly critical.

The simplest solution to the text localization problem is to have an external file that stores all of the text in the game. This external file could be in XML, JSON, or the like (a more in-depth discussion of different file formats can be found in the following chapter). This could then map to a simple dictionary that has a key identifying the particular string. So whenever the code needs to display text onscreen, it will request the text from the dictionary using the appropriate key. This means that instead of having a button hard-coded to display the text "Cancel," it may instead query the dictionary for the string associated with "ui_cancel." In this scenario, supporting a new language might only require creating a different file to populate the dictionary from. This style of approach is utilized by the tower defense game example in Chapter 14.

Unfortunately, supporting languages that use different character sets can complicate the matter. The traditional ASCII character set (which is what's used by the `char` type in most languages) can only support the English alphabet. There's no support for accent marks, let alone any support for other writing systems such as Arabic and simplified Chinese.

LAST-MINUTE LOCALIZATION

On one project I worked on, all of the user interface was created using a scripting language. Throughout development, no one really considered localization. The result of this was that all of the text that was to be displayed onscreen was hard-coded into the scripts. When it was time to localize, no one wanted to manually go through and change all the strings in the script files into keys. So an idea was hatched: the English strings in the scripts were made into the keys for the localization dictionary, and then the values were set in external files for each language.

This worked, but there was one little problem: If someone changed a string in the script file, not knowing there was now a localization system, that string would no longer function properly. That's because the localization file would not have the new English string listed in it. This caused some headaches on the project, and certainly is not a recommended approach for most games. It's a cautionary tale as to why planning for localization is important, even if there are no immediate plans to localize a particular game.

Most games that need to support different writing systems end up using the **Unicode** character set. There are multiple ways to encode Unicode characters, but the most popular is arguably UTF-8, which is a variable-width encoding. Standard ASCII characters are only one byte in UTF-8, whereas Unicode characters can be two to six bytes. Due to this variable width, assumptions about the size of strings in memory cannot be made with UTF-8 because the size will vary from language to language.

But changing the text strings and character encoding are not the only things to consider. One issue is that certain languages have lengthier words than others. One language that is commonly localized for is German; it turns out that the average German word is longer than the average English word (I could not find a reputable article on the exact numbers, but I think 20%–25% longer is a decent rule of thumb). This means that if there's a UI element that barely fits the text in English, it has a good chance of not fitting in other languages. These types of issues can be frustrating to fix, because they may require changing art assets months after they were to have been finalized. But this is one problem that consistently comes up late in the localization process. Figure 10.10 illustrates a button in English and a few other languages (translations courtesy of Google).

Figure 10.10 A Cancel button and the problems with localizing it.

In addition to text and voice over, one other aspect of localization is localizing the actual content for certain countries. For example, any World War II game must be careful not to infringe

on German laws, which forbid the usage of certain symbology related to the Third Reich. Other countries may have stricter laws on violence, which may require reducing or removing blood and/or objectionable scenes. And, finally, the most difficult type of localization error to catch is instances where idiomatic phrases or other references might be okay in the American culture, but confusing or perhaps even offensive in other cultures. These types of localization problems extend well beyond the realm of user interfaces, but they are something to keep in mind when preparing a game for another region.

UI Middleware

Recall from Chapter 1, "Game Programming Overview," that middleware is an external code library that's used to simplify certain parts of development. No discussion of user interfaces for games, especially AAA ones, would be complete without discussing the predominant middleware in this category: Autodesk Scaleform. Some games that have used Scaleform include *Star Wars: The Old Republic* and *BioShock: Infinite*. Although Scaleform is very popular for large-scale commercial games, it's not as popular for smaller games because it's not a free solution.

The premise behind Scaleform is that it allows artists and content creators to use Adobe Flash to design the entire layout of the UI. This way, a programmer does not have to spend time manually setting up the locations of all the elements. Much of the UI programming can also be done in ActionScript, which is a JavaScript-esque scripting language that Flash uses. Using Scaleform does mean that some time will have to be spent integrating it into the overall game engine, but once that is complete the UI can go through many iterations without any system-level changes.

User Experience

One very important aspect of UIs that this chapter has not discussed at all is **user experience** (or UX), which is the reaction a user has while actually using the interface. An example of a poorly designed game UI is one where the user feels like he or she has to make several button presses to perform simple actions, which is notoriously a problem in certain console RPGs. If you are tasked with not only programming but also designing the UI, it's critical to take into account UX. However, because this topic is not specifically related to programming, I have relegated any further discussion on this topic to the references.

Summary

Game UIs can vary a great deal—a MMORPG such as *World of Warcraft* might have dozens of UI elements on screen at once, whereas a minimalistic game may eschew the UI almost entirely. Often, the UI is the player's first interaction with the game as he or she navigates through the initial menu system. But the UI is also featured during gameplay through the HUD, using elements such as an aiming reticule, radar, and waypoint arrow to provide important information to the player. Although experienced programmers often express a dislike of programming

UI systems, a well-implemented UI can greatly improve the player's overall experience. A poorly implemented UI, however, can ruin an otherwise enjoyable game.

Review Questions

1. What are the benefits of using a "menu stack"?
2. What property of letter key codes can be taken into account when implementing a function to convert from key codes to characters?
3. When implementing a waypoint arrow, how does one determine the position of the 3D arrow in world space?
4. How are the waypoint arrow calculations for the axis and angle of rotation performed?
5. Describe the calculations that allow for an aiming reticule to determine whether a friend or foe is being targeted.
6. When implementing a radar, how is the 3D coordinate of an object converted into a 2D radar coordinate?
7. What is the difference between absolute and relative coordinates?
8. Why should UI text *not* be hard-coded?
9. What problem does the Unicode character set solve?
10. What is user experience (UX)?

Additional References

Komppa, Jari. "Sol on Immediate Mode GUIs." http://iki.fi/sol/imgui/. An in-depth tutorial on the implementation of different elements of a game UI using C and SDL.

Quintans, Desi. "Game UI By Example: A Crash Course in the Good and the Bad." http://tinyurl.com/d6wy2yg. This relatively short but informative article goes over examples of games that had good and bad UIs, and takeaways that should be considered when designing your own game's UI.

Spolsky, Joel. *User Interface Design for Programmers*. Berkeley: Apress, 2001. This programmer's approach to designing UI is not specifically written for games, but it does provide interesting insights into how to create an effective UI.

CHAPTER 11

SCRIPTING LANGUAGES AND DATA FORMATS

When writing code that drives gameplay, it is increasingly common to utilize a scripting language. This chapter explores the advantages and disadvantages of using a scripting language, as well as several of the potential language options.

A further consideration is how data that describes the level and other properties of the game should be stored. As with scripting languages, there are several options for representing data, both binary and text based.

Scripting Languages

For many years, games were written entirely in assembly. That's because extracting a high performance out of those earlier machines required a level of optimization that was only possible with assembly. But as computers became more powerful, and games became more complex, it made less and less sense. At a certain point, the opportunity cost of developing a game entirely in assembly was not worth it, which is why all game engines today are written in a high-level programming language such as C++.

In a similar vein, as computers have continued to improve, more and more games have shifted away from using C++ or similar languages for gameplay logic. Many games now write much of their gameplay code in a **scripting language**, which is an even higher-level language such as Lua, Python, or UnrealScript.

Because script code is easier to write, it is often possible for designers to work directly on the scripts. This gives them a much greater ability to prototype ideas without needing to dive into the engine code. Although certain core aspects of AAA games such as the physics or rendering system still are written in the core engine's language, other systems such as the camera and AI behavior might be in script.

Tradeoffs

Scripting languages aren't a panacea; there are tradeoffs that must be considered before using one. The first consideration is that the performance of script is simply not going to match the performance of a compiled language such as C++. Even when compared to a JIT- or VM-based language such as Java or C#, an interpreted scripting language such as Lua or Python will not have competitive performance. That's because an **interpreted language** reads in the text-based code on demand, rather than compiling it in advance. Certain scripting languages do provide the option to compile into an intermediary; although this still won't be as fast as a natively compiled language, it usually will be faster than an interpreted language.

Because there is still a performance gap, code that must be high performance typically should not be implemented in script. In the case of an AI system, the pathfinding algorithm (for example, A*) should be high performance and therefore should not be written in script. But the state machines that drive the AI behavior can absolutely be written in script, because they typically will not require complex computations. Some other examples of what might be in script versus in C++ for a sample game are listed in Table 11.1.

Table 11.1 Scripting Language Usage in Sample Game

C++	Scripting Language
Rendering engine	Camera logic
AI (pathfinding)	AI (state machines)
Physics systems	General gameplay logic
File loading	User interface

One big advantage of a scripting language is that it can allow for more rapid iteration during development. Suppose that for a particular game, the AI state machines are written in C++. While playing the game, an AI programmer notices that one enemy is acting incorrectly. If the state machines are written in C++, the programmer may have many tools to diagnose the problem, but typically cannot fix it while the game is running. Although Visual Studio does have an "edit and continue" feature in C++, in practice this will only work in certain situations. This means that usually the programmer must stop the game, modify the code, build the executable, restart the game, and finally see if the problem went away.

But if this same scenario is encountered in a game where the AI state machines are written in script, it may be possible to reload the AI's script dynamically and have the fix occur while the game is still running. This ability to dynamically reload scripts can be a huge productivity increase.

Returning to the C++ version of the AI behavior, suppose that there is a bug in a guard's AI where sometimes a bad pointer is accessed. If you've programmed in C++ before, you know that accessing a bad pointer will usually result in a crash. If this bug keeps popping up all the time, the game will crash often. But if the state machine was instead written in script, it would be possible to have only the AI for that particular character stop functioning while the rest of the game continues to run. This second scenario is far more preferable than the first.

Furthermore, because scripts are files that are separate from the executable, it makes distributing changes much easier. On a large project, building the executable can take several minutes, and the resultant file might approach 100MB. This means that whenever there's a new version, whoever wants it might have to download that entire file. On the other hand, if a scripting language is used, the user may only have to download a couple kilobytes of data, which is going to be a much faster process. This is especially helpful when deploying patches in a retail scenario, but it can also be useful during development.

Because of these productivity advantages, a good rule of thumb is that if the performance of the system is not mission-critical, it may be beneficial to write the system in script. There is, of course, the overhead of adding a scripting system to the game itself, but the time spent on that can be easily reclaimed if it leads to increased productivity for several members of the development team.

Types of Scripting Languages

If you decide to use a scripting language for some gameplay logic, the next logical question is, which scripting language? Two different approaches need to be considered: using an existing scripting language, such as Lua or Python, and using a custom scripting language designed specifically for the game engine in question. Some examples of custom languages include UnrealScript and QuakeC.

The advantage of using an existing language is that it typically will be much less work, because writing a script parser can be a time-consuming and error-prone endeavor, even when you're using tools that aid the process. Furthermore, because a custom language has a much smaller number of users than existing languages, there is a high likelihood of errors persisting in the language over long periods of time. See the following sidebar, "An Error in Operator Precedence," for one such instance.

AN ERROR IN OPERATOR PRECEDENCE

What is the result of the following arithmetic operation?

```
30 / 5 * 3
```

According to one proprietary scripting language I worked with, the answer is `2`. That's because the language in question incorrectly assigned multiplication a higher precedent than division. So the preceding code was being interpreted as follows:

```
30 / (5 * 3) = 2
```

However, because division and multiplication have equal precedent, the expression should instead be computed as this:

```
(30 / 5) * 3 = 18
```

This particular bug persisted in the scripting language for over a decade. No one had ever mentioned the problem to the programmer in charge of the scripting language, probably because programmers have a tendency to add lots of parentheses to mathematical expressions.

Once the bug was found, it was easy enough to fix. But it's a cautionary tale of what can happen with a custom scripting language.

The other consideration is that established languages typically have been designed and developed by people who are extremely knowledgeable about compiler and virtual machine design—far more knowledgeable on the topic than most game programmers.

But the downside is that existing scripting languages may not always be well-suited for game applications. There may be performance or memory allocation concerns with the way the language is designed, or it may be difficult to bridge the code between the scripting language and the engine. A custom language, on the other hand, can be designed such that it is optimized specifically for game usage—even to the point where the language has features that are very game specific, as in UnrealScript.

One last consideration with existing scripting languages is that it's easier to find developers familiar with them. There are many programmers who might already know a language such as

Lua or Python, but it's less likely that someone will know a language proprietary to a specific company or engine.

Lua

Lua is a general-purpose scripting language that's perhaps the most popular scripting language used in games today. Some examples of games that have used Lua include *World of Warcraft*, *Company of Heroes*, and *Grim Fandango*. One reason why it's so popular for games is because its interpreter is very lightweight—the reference C implementation is approximately 150KB in memory. It also has the relatively simple capability to set up **native bindings**, where functions called by Lua can execute code in C/C++. It also supports multitasking, so it's possible to have many Lua functions running at the same time.

Syntactically, the language does have some similarities with C-family languages, though there are also several differences. Semicolons at the end of lines are optional, and braces are not used for control statements. An intriguing aspect of Lua is that the only complex data type it supports is the table, which can be used in a variety of different ways, including as an array, a list, a set, and so on. This is demonstrated in the following bit of code:

```
-- Comments use --
-- As an array
-- Arrays are 1-index based
t = { 1, 2, 3, 4, 5 }
-- Outputs 4
print( t[4] )

-- As a dictionary
t= { M="Monday", T="Tuesday", W="Wednesday" }
-- Outputs Tuesday
print( t[T] )
```

Even though Lua is not an object-oriented language, with the clever use of tables it is absolutely possible to implement OOP. This technique is used frequently because OOP makes sense for many usage cases in games.

UnrealScript

UnrealScript is a strictly object-oriented language designed by Epic for the Unreal Engine. Unlike many scripting languages, UnrealScript is compiled. Because it's compiled, it has better performance than an interpreted language, but it does mean that reloading a script at runtime is not supported. Games powered by Unreal typically use UnrealScript for almost all of their gameplay. For games using the full engine (and not just the free UDK), native bindings allow for UnrealScript functions to be implemented in C++. An example of this is shown in Figure 11.1.

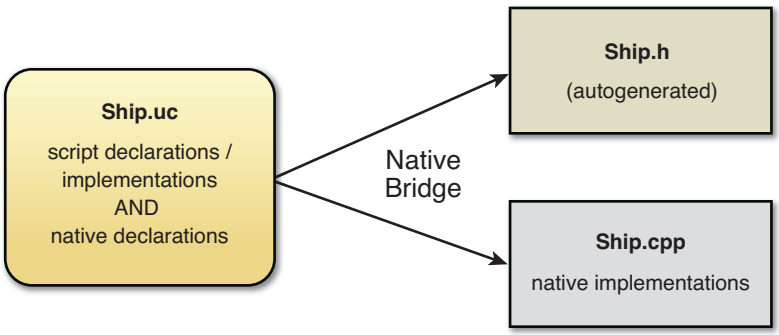


Figure 11.1 UnrealScript native bindings of a Ship class.

Syntactically, UnrealScript looks very much like C++ or Java. Because it's strictly object oriented, every class derives from `Object` in one way or another, and nearly every class representing a character on the screen is derived from `Actor`. One very unique feature of UnrealScript is built-in support for states. It is possible to have different function overloads depending on the state, which makes it very easy to set up state machines for AI behavior, for example. In the code snippet that follows, a different `Tick` function (Unreal's object update function) is called depending on the current state of the class:

```
// Auto means this state is entered by default
auto state Idle
{
   function Tick(float DeltaTime)
   {
      // Update in Idle state
      ...

      // If I see an enemy, go to Alert state
      GotoState('Alert');
   }
Begin:
   `log("Entering Idle State")
}

state Alert
{
   function Tick(float DeltaTime)
   {
      // Update in Alert state
      ...
   }
Begin:
   `log("Entering Alert State")
}
```

Visual Scripting Systems

An increasing number of game engines now implement **visual scripting systems** that look a lot like flow charts. These systems are typically used for setting up level logic. The idea is that rather than requiring level designers to write any text-based code, it instead is easier to set up the actions that occur with a flow chart. For example, if a level designer wants to have enemies spawn when the player opens a door, this might be done with a visual scripting system. In the Unreal Engine, the visual scripting system used is called Kismet; a screenshot of it in action is shown in Figure 11.2.

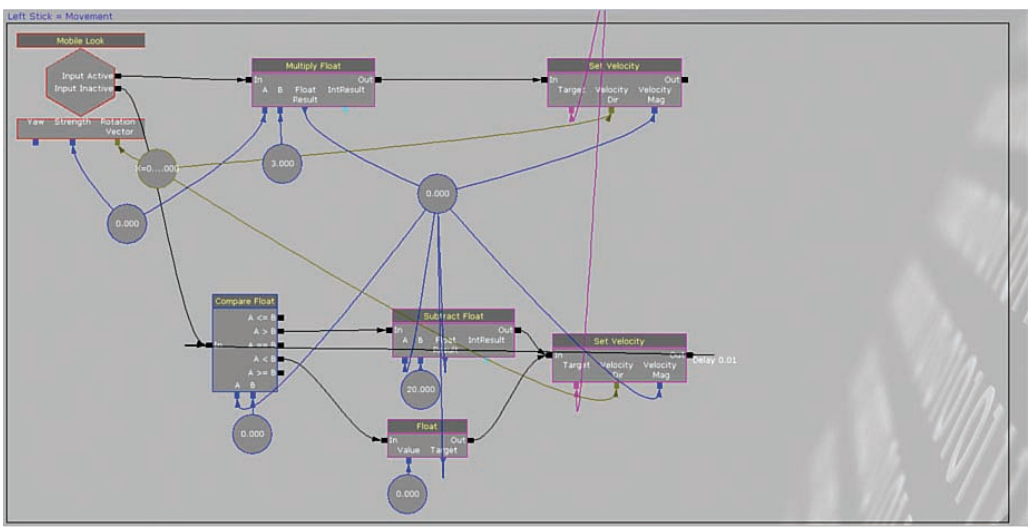


Figure 11.2 Kismet visual scripting system.

Implementing a Scripting Language

Although the full implementation of a scripting language is well beyond the scope of this book, it's worthwhile to at least discuss the main components necessary to implement one. This is both to give a sense of how complex the problem is and to at least provide some guidance on how to start off on the right track.

The basics of implementing a custom scripting language is very similar to that of creating a general compiler; therefore, many of the topics covered in this section can be studied in much further detail in a book focused on compilers (such as the one detailed in the references). Learning at least the basics of how a compiler works is important, even if you aren't going to implement one, because it's what makes high-level programming possible. Without compiler

theory, everyone would still be writing code in assembly, which at the very least would mean there would be far fewer qualified programmers.

Tokenization

The first step necessary to read in a language is to take the stream of text and break it down into a series of **tokens** such as identifiers, keywords, operators, and symbols. This process is known as **tokenization**, or more formally as **lexical analysis**. Table 11.2 shows a simple C file broken down into its tokens.

Table 11.2 Simple C File Tokenized

C Code	Tokens

`int main(void)`
`{`
`  return 0;`
`}`
	`int`
	`main`
	`(`
	`void`
	`)`
	`{`
	`return`
	`0`
	`;`
	`}`

Although it certainly is possible to write a tokenizer (also known as a scanner or lexer) by hand, it definitely is not recommended. Manually writing such code is extremely error prone simply because there are so many cases that must be handled. For example, a tokenizer for C++ must be able to recognize that only one of these is the actual `new` keyword:

```
newnew
_new
new_new
_new_
new
```

Instead of manually writing a tokenizer, it is preferable to use a tool such a flex, which generates a tokenizer for you. The way flex works is you give it a series of matching rules, known as **regular expressions**, and state which regular expressions match which tokens. It will then automatically generate code (in the case of flex, in C) for a tokenizer that emits the correct tokens based on the given rules.

Regular Expressions

Regular expressions (also known as regex) have many uses beyond tokenization. For example, most IDEs can perform a search across code files using regular expressions, which can be a very handy way to find specific types of sequences. Although general regular expressions can become rather complex, matching patterns for a scripting language only require using a very small subset of regular expressions.

The most basic regular expression would be to match specific keywords, which must be the same series of characters every time. In order to match this, the regex is simply the series of characters with or without quotes:

```
// Matches new keyword
new

// Also matches new keyword
"new"
```

In a regular expression, there are also operators that have special meaning. The `[]` operator means any character contained within the brackets will be matched. This can also be combined with a hyphen to specify any character within a range. Here's an example:

```
// Matches aac, abc, or acc
a[abc]c

// Matches aac, abc, acc, ..., azc
a[a-z]c

// You can combine multiple ranges...
// Matches above as well as aAc, ..., aZc
a[a-zA-Z]c
```

The `+` operator means "one or more of a particular character," and the `*` operator means "zero or more of a particular character." These can be combined with the `[]` operator to create expressions that can define most of the types of tokens in a particular language:

```
// Matches one or more number (integer token)
[0-9]+

// Matches a single letter or underscore, followed by zero or more
// letters, numbers, and underscores (identifier in C++)
[a-zA-Z_][a-zA-Z0-9_]*
```

In any event, once a list of tokens and the regular expressions corresponding to them is created, this data can then be fed to a program such a flex in order to automatically generate a tokenizer. Once a script is fed into this tokenizer and broken down into tokens, it's time to move on to the next step.

Syntax Analysis

The job of **syntax analysis** is to go through the stream of tokens and ensure that they conform to the rules of the language's grammar. For example, an `if` statement must have the appropriate number and placement of parentheses, braces, a test expression, and a statement to execute. As the syntax of the script is verified, an **abstract syntax tree** (AST) is generated, which is a tree-based data structure that defines the layout of the entire program. An AST for a simple math expression is illustrated in Figure 11.3.

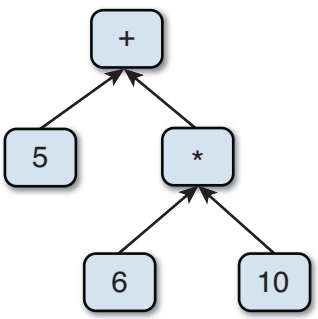


Figure 11.3 AST for 5 + 6 * 10.

Notice how if the tree in Figure 11.3 were traversed with a post-order traversal (left child, right child, parent), the result would be `5 6 10 * +`, which happens to correspond how the infix expression `5 + 6 * 10` would be written in postfix notation. This is not by chance—it turns out that postfix expressions are much easier to compute since they work so well with a stack. Ultimately, all ASTs (whether mathematical expressions or otherwise) will be traversed in a post-order manner after syntax analysis.

But in order to traverse the AST, we first must generate one. The first step toward generating an AST is to define the grammar of the language. A classic way to define grammars for computer languages is to use **Backus-Naur Form**, which is often abbreviated as BNF. BNF is designed to be relatively succinct; the grammar for a calculator that can do integral addition and subtraction might be defined as follows:

```
<integer>    ::== [0-9]+
<expression> ::== <expression> "+" <expression>
                | <expression> "-" <expression>
                | <integer>
```

The `::==` operator means "is defined as," the `|` operator means "or," and `<>` are used to denote grammar rule names. So the preceding BNF grammar says that an expression can be either an expression plus another expression, or an expression minus another expression, or an integer. Thus, `5 + 6` would be valid because 5 and 6 are both integers, which means they can both be expressions, which in turn means they can be added together.

As with tokenization, there are tools you can use to help with syntax analysis. One such tool is bison, which allows for C/C++ actions to be executed whenever a grammar rule is matched. The general usage of the action is to have it create the appropriate node that goes into the AST. For instance, if an addition expression were matched, an addition node could be generated that has two children: one each for the left and right operands.

This means it's best to have a class corresponding to each possible node type. So the addition/subtraction grammar would need to have four different classes: an abstract base expression class, an integer node, an addition node, and a subtraction node. This hierarchy is shown in Figure 11.4.

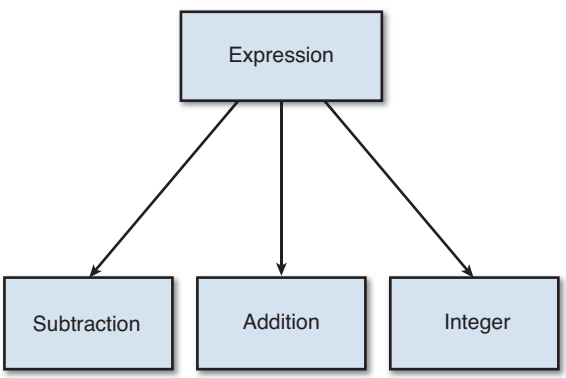


Figure 11.4 Class hierarchy for basic addition/subtraction expressions.

Code Execution or Generation

Once an AST is generated from the script file, there are two potential courses of action, both of which require using a post-order traversal. For a more traditional compiler, the goal of the AST traversal would be to generate the actual code that will be run on the target machine. So when C++ code is compiled, the AST is traversed and code is generated in assembly that is designed specifically to work on the target platform.

If the game's scripting language is compiled, such as UnrealScript, then it would definitely make sense to perform the code-generation step. But if the language is interpreted, code generation is unnecessary. Instead, it would be preferable to just traverse the AST and execute any actions that the nodes represent.

For the addition/subtraction example, the nodes could be defined in the following manner:

```
abstract class Expression
   function Execute()
end
```

```
class Integer inherits Expression
   // Stores the integral value
   int value

   // Constructor
   ...

   function Execute()
      // Push value onto calculator's stack
      ...
   end
end

class Addition inherits Expression
   // LHS/RHS operands
   Expression lhs, rhs

   // Constructor
   ...

   function Execute()
      // Post-order means execute left child, then right child,
      // then myself.
      lhs.Execute()
      rhs.Execute()

      // Add top two values on calculator's stack, push result back on
      ...
   end
end

// Subtraction node identical to Addition, except it subtracts
...
```

With the preceding classes, it would then be possible to calculate the result of the AST by calling `Execute` on the root node. Once that call returns, the lone value on the stack will correspond to the result of the operation.

For example, the expression `5 + 6` would have an AST where the root node is an `Addition` class, the left child is an `Integer` class with a value of `5`, and the right child is an `Integer` class with a value of `6`. So if `Execute` is called on that root `Addition` node, it would first call `Execute` on `5`, which will push `5` onto the calculator stack. Then `Execute` on `6` pushes `6` onto the calculator stack. Finally, `Execute` on the root mode will add the top two values on the stack, which gives `11`, and push that back on, giving the final result of the operation.

Of course, if the scripting language in question supports far more than just basic arithmetic, the contents of the `Execute` functions will become more complex. But this basic principle of a post-order traversal carries over regardless of the complexity of the language or node type.

Data Formats

Another decision that must be made during game development is how the data describing elements such as levels and other game properties should be stored. For a very simple game, you might be able to get away with having most of the data hard-coded, but it's not an ideal solution. By having data stored in external files, you allow for nonprogrammers to edit it. This also opens up the possibility of creating tool programs (such as a level editor) that allow the data to be more easily manipulated.

When you're deciding on a data format, the first decision is whether or not the data should be binary or text-based. A **binary** file is one that contains data that's mostly unreadable by a human. If you open up a binary file in a text editor, it will be a stream of random characters. An example of a binary format is PNG or any other image file format. A **text-based** file, on the other hand, is a format that is composed of standard ASCII characters and therefore can be read by a human. As with the decision on whether or not a scripting language should be used, there are tradeoffs between both approaches. Ultimately, it will make sense to store some data in a binary format while storing other data in a text-based one.

Tradeoffs

The primary advantage of using a binary file is that it's going to be a smaller file, and also one that can be loaded more quickly. Rather than having to spend time parsing in text and converting it into the appropriate in-memory format, it's often possible to directly load in chunks of the binary file without any conversion. Because efficiency is so important for games, it's on that merit alone that binary typically wins out for files that are going to contain large amounts of information, such as the layout of a level.

However, the extra speed does come with a cost. One big disadvantage of binary files is that they don't work as well with a source control system (if you aren't familiar with source control, you should check Appendix B, "Useful Tools for Programmers"). The reason is that it can be very hard to figure out what has changed between two versions of a binary file.

For example, suppose a game stores its level data in a binary format (such as the .pkg files that Unreal uses). Now further suppose that a designer goes into the level editor and makes ten different changes to the level. Without testing the level, the designer then decides to submit the updated level. For whatever reason, this level actually does not load after the changes. At this point, to fix the game so that the level can load, the only course of action is to revert to the previous version of the file, thus losing all ten changes. But what if only one of the ten changes was bad? In a binary file, there's typically no way to go through and actually change the one part of it that's bad, so all the changes have to be undone.

Granted, this scenario does expose some other problems—why is the designer checking in something without testing it, and why can the level editor save bad data? But the fact of the

matter is that between two revisions of this hypothetical binary level file format, it's going to be very hard to tell what has changed without loading the level.

For a text-based file format, however, it would be very easy to look at two different versions of the file and see the differences between them. This means that if there were ten changes to a level, with one not working, it would be possible to go through and actually remove the specific change that's breaking the level.

There's also the fact that a text-based file format can be more easily edited by end users. For something such as configuring the keyboard input, it may be preferable to have a text file so it's very easy to change in a standard text editor. But on the other hand, this may be a disadvantage for the level data, because it would be easy for a player to change the behavior of the game and break it.

One final option is to use both text-based and binary representations for the same data. During development, being able to easily track changes is extremely helpful, so all level and game object data could be stored in text. Then, for the release version, we could introduce a **bake** step that converts the text into a more efficient binary format. These binary files would never be saved into source control, and the developers would only ever modify the text versions. This way, we can get the advantages of a text-based file format for development as well as the advantages of a binary format for release. Because it has the benefits of both, this approach is very popular. The only concern here is one of testing—the development team needs to be extra vigilant to ensure that any changes to the text format do not cause bugs to appear in the binary format.

Binary Formats

For storing game data, more often than not a binary file format is going to be a custom format. There really are a lot of different ways to store the data, which also will depend on the language/framework. If it's a C++ game, sometimes the easiest way to save data is to directly write the contents of a class to the output file, which is a process called **serialization**. But there are some hurdles to consider—for example, any dynamically allocated data in the class is going to have pointers, so you need to actually perform a deep copy and reconstruct that data dynamically. However, the overall design of a binary file format is a bit too advanced for this book.

INI

The simplest text-based file format is likely **INI**, which is often used for setting files that an end user might want to change. An INI file is broken down into sections, and each section has a list of keys and values. For example, graphics settings in an INI file might look like this:

```
[Graphics]
Width=1680
```

```
Height=1050
FullScreen=true
Vsync=false
```

Although INI files work well for very simple data, they become very cumbersome with more complex data. This wouldn't be a great file format to use for the layout of a level, for instance, because INI does not support any idea of nesting parameters and sections.

Because INI is so simple and widely used, a large number of easy-to-use libraries is available. In C/C++, one particular library I'm a fan of is minIni, because it works on a wide variety of platforms and is very easy to use.

XML

XML, or Extensible Markup Language, is a file format that is an expansion of the concept of HTML. Rather than using set HTML tags such as `<html>`, `<head>`, `<body>`, and so on, in XML you can have any custom tags and attributes you want. So to the uninitiated, it might look a lot like HTML, even though it's not. *The Witcher 2* used XML extensively to store all the parameters of the different items that could be found in the game. For example, here's an entry that stores the stats of a particular sword:

```
<ability name="Forgotten Sword of Vrans _Stats">
   <damage_min mult="false" always_random="false" min="50" max="50"/>
   <damage_max mult="false" always_random="false" min="55" max="55"/>
   <endurance mult="false" always_random="false" min="1" max="1"/>
   <crt_freeze display_perc="true" mult="false" always_random="false"
    min="0.2" max="0.2" type="critical"/>
   <instant_kill_chance display_perc="true" mult="false"
    always_random="false" min="0.02" max="0.02" type="bonus"/>
   <vitality_regen mult="false" always_random="false" min="2" max="2"/>
</ability>
```

One big criticism of XML is that it requires a lot of additional characters to express the data. There are many < and > symbols, and you always need to qualify each parameter with the name, quotes, and so on. You also always have to make sure there are matching closing tags. All this combined leads to larger text files.

But one big advantage of XML is that it allows for the enforcement of sticking to a **schema**, or a set of requirements on which tags and properties must be used. This means that it can be easy to validate an XML file and make sure it conforms and declares all the necessary parameters.

As with the other common file formats, a lot of different parsers support loading in XML. The most popular one for C/C++ is arguably TinyXML (and its companion for C++, ticpp). Some languages also have built-in XML parsing; for instance, C# has a `System.Xml` namespace devoted to XML files.

JSON

JSON, or JavaScript Object Notation, is a newer file format than INI or XML, but it's also one that's gained a great deal of popularity in recent years. Although JSON is more commonly used for transmitting data over the Internet, it certainly is possible to use it as a lightweight data format in a game. Recall that we used JSON to store the sound metadata in Chapter 6, "Sound." There are several third-party libraries that can parse JSON, including libjson (http://libjson.sourceforge.net) for C++ and JSON.NET for C# and other .NET languages.

Depending on the type of data being stored, a JSON file may be both smaller and more efficient to parse than a comparable XML file. But this isn't always the case. For example, if we were to take the sword data from *The Witcher 2* and instead store it in a JSON file, it actually ends up being larger than the XML version:

```
"ability": {
   "name": "Forgotten Sword of Vrans _Stats",
   "damage_min": {
      "mult": false, "always_random": false, "min": 50, "max": 50
   },
   "damage_max": {
      "mult": false, "always_random": false, "min": 55, "max": 55
   },
   "endurance": {
      "mult": false, "always_random": false, "min": 1, "max": 1
   },
   "crt_freeze": {
      "display_perc": true, "mult": false, "always_random": false,
         "min": 0.2, "max": 0.2, "type": "critical"
   },
   "instant_kill_change": {
      "display_perc": true, "mult": false, "always_random": false,
      "min": 0.2, "max": 0.2, "type": "bonus"
   },
   "vitality_regen": {
      "mult": false, "always_random": false, "min": 2, "max": 2
   }
}
```

Even if you try to reduce the sizes of both the JSON and XML files by removing all extraneous whitespaces and carriage returns, in this particular instance the JSON file still ends up being slightly larger than the XML one. So although JSON can be a great text-based file format in some situations, in the case of *The Witcher 2*, the developers likely made the correct decision to use XML instead.

Case Study: UI Mods in *World of Warcraft*

Now that we've explored scripting languages and text-based data representations, let's take a look at a system that uses both: user interface modifications in *World of Warcraft*. In most games, the UI is whatever the designers and programmers implemented, and you're stuck with it. Although there are often some options and settings, most games do not allow configuration of the UI beyond that. However, in an MMORPG, players have a lot of different preferences—for instance, some players might want the enemy health to be shown in the center of the screen, whereas others might want it in the top right.

What Blizzard wanted to do is create a UI system that was heavily customizable. They wanted a system that not only allowed the placement of elements to be changed, but for entirely new elements to be added. In order to facilitate this, they required creating a system that heavily relied on scripting languages and text-based data.

The two main components of a UI mod in *World of Warcraft* are the layout and the behavior. The layout of interfaces, such as placement of images, controls, buttons, and so on, is stored in XML. The behavior of the UI is then implemented using the Lua scripting language. Let's look at both aspects in more detail.

Layout and Events

The layout of the interface is entirely XML driven. It's set up with base widgets such as frames, buttons, sliders, and check boxes that the UI developer can use. So if you want a menu that looks like a built-in window, it's very much possible to declare an interface that uses the appropriate elements. The system goes a step further in that one can inherit from a particular widget and overwrite certain properties. Therefore, it would be possible to have a custom button XML layout that derives from the base one and maybe modifies only certain parameters.

It's also in the XML file where the add-on specifies whether there are any particular events to register. In a *WoW* add-on, many different events can be registered, some specific to widgets and some specific to the game as a whole. Widget-based events include clicking a button, opening a frame, and sliding a slider. But the real power of the event system is the plethora of game events. Whenever your character deals or takes damage or you chat with another player, check an item on an auction house, or perform one of many different actions, a game event is generated. It is possible for a UI add-on to register to any of these events. For example, a UI add-on that determines your DPS (damage per second) can be implemented by tracking the combat damage events.

Behavior

The behavior of each interface add-on is implemented in Lua. This allows for quick prototyping of behavior, and it's further possible to reload the UI while the game is running. Because they utilize a table-based inheritance system, mods can implement member functions that override parent systems. The bulk of the code for most add-ons is focused on processing the events and doing something tangible with that information. Every event that is registered must have corresponding Lua code to handle it.

Issue: Player Automation

Although the *WoW* UI system was extremely successful at launch, there were some issues that arose. One concern during development was the idea of **player automation**, or the player being able to automate the gameplay with a UI mod. In order to combat this, the system was designed such that a UI mod could only ever perform one action per key press. Therefore, if you pressed the spacebar, at most, the add-on would only be able to cast a single spell.

Although this prevented full automation, it still was possible to make an add-on that did most of the work. For example, a healer character could use an add-on that performed this logic every time the spacebar is pressed:

1. Iterate through all players in the group.
2. Find the player with the lowest hit points and target him or her.
3. Based on the player's health, cast the appropriate healing spell.

Then a player could run through a level and continuously spam the spacebar. If the logic is sound enough, it will work to keep the whole group alive with minimal skill on the part of the healer. To solve this problem, Blizzard changed the system so that UI mods could not cast any spells at all. Although this didn't entirely eliminate automation (as there are always external third-party programs that try to do so), it eliminated the use of UI mods to greatly streamline gameplay.

Issue: The UI Breaking

Another issue with the UI system is that because core behavior of the interface can be easily modified by players, it was very much possible to render the game unplayable. The most common way this happened was when new major patches were released. Often, these releases would also coincide with changes to the UI API, which meant if you were running older mods, they could potentially break the entire interface.

This became problematic for Blizzard's technical support, because every time a new patch was released they were inundated with people who could no longer attack, chat, or cast spells, often due to broken UI mods. Although there was an option to disable out-of-date add-ons, most players did not use the feature because most older mods worked in new patches.

The solution ultimately became to add a new "secure" code path that represented any calls from the built-in UI. These secure calls handled all of the basic functionality, such as attacking and chatting, and add-ons were no longer allowed to override these functions. Instead of overriding them, they had to hook onto the functions, which means that after a secure function executes, the custom hook has an opportunity to execute.

Conclusion

Ultimately, the UI add-on system for *World of Warcraft* was extremely successful, so much so that most MMORPGs released since have employed similar systems. The number of add-ons created by the community is enormous, and some became so successful that Blizzard incorporated elements into the core game interface. Ultimately, the design of a system that was so flexible and powerful would not have been possible without the use of both scripting languages and a text-based file format.

Summary

Scripting languages have seen an increase in usage in games due to the improvements in productivity. Because of this, many gameplay programmers now spend much of their time programming in script. Some games might use an existing scripting language such as Lua, whereas others might use a language custom-built for their game engine. If a custom scripting language must be implemented, many of the techniques used to create a compiler must be employed, including lexical and syntax analysis.

As for the question of how to store game data, there are tradeoffs between text-based and binary formats. A text-based format makes it much easier to glean changes between revisions, whereas a binary format will almost always be more efficient. Some sample text-based formats this chapter covered included INI, XML, and JSON. Finally, in the *World of Warcraft* UI we saw a system that combines both a scripting language and text-based data files.

Review Questions

1. What are the advantages and disadvantages of using a scripting language in a game?
2. What are three examples of systems that make sense to implement with a scripting language?
3. Give an example of a custom scripting language. What advantages and disadvantages might this have over a general scripting language?
4. What is a visual scripting system? What might it be used for?

5. Break down the following C++ statement into its corresponding tokens:
```
int xyz = myFunction();
```
6. What does the following regular expression match?
```
[a-z][a-zA-Z]*
```
7. What is an abstract syntax tree and what is it used for?
8. Describe the tradeoffs between a binary and text-based file format for game data.
9. What file format might you use to store basic configuration settings, and why?
10. What are the two main components of creating an interface mod in *World of Warcraft*?

Additional References

Aho, Alfred, et. al. *Compilers: Principles, Techniques, and Tools* (2nd Edition). Boston: Addison-Wesley, 2006. This revised edition of the classic "Dragon Book" goes through many of the main concepts behind designing a compiler. Much of this knowledge can also be applied to the implementation of a custom scripting language.

"The SCUMM Diary: Stories behind one of the greatest game engines ever made." Gamasutra. http://tinyurl.com/pypbhp8. This extremely interesting article covers the development of the SCUMM engine and scripting language, which powered nearly all of the classic LucasArts adventure games.

CHAPTER 12

NETWORKED GAMES

Networked games allow multiple players to connect over the Internet and play together. Some of the most popular games in recent years—whether *Halo*, *Call of Duty*, or *World of Warcraft*—either support networking or are exclusively networked games. Playing with or against another human provides an experience that even today cannot be matched by an AI.

Implementing networked games is a complex topic, and there are entire books on the subject. This chapter covers the basics of how data is transmitted over a network and how games must be architected differently in order to support networking.

Protocols

Imagine you are mailing a physical letter via the postal service. At a minimum, there is an envelope that has the addresses that define both where the letter is from and where it is going. Usually there also is some sort of stamp affixed to the letter, as well. Inside the envelope is the actual data you wanted to transmit—the letter itself. A **packet** can be thought of a digital envelope that is sent over a network. A packet has addresses and other relevant information in its **header**, and then the actual data **payload** it's sending out.

For envelopes, there is a fairly standardized method of addressing. The from address goes in the top-left corner, the destination address is in the middle right, and the stamp goes in the top-right corner. This seems to be the norm in most countries. But for networked data transmission, there are several different **protocols**, or rules that define how the packet must be laid out and what must happen in order to send it. Networked games today typically use one of two protocols for gameplay: TCP or UDP. Some games also might use a third protocol, ICMP, for some limited non-gameplay features. This section discusses these different protocols and when they might see use.

IP

IP, or **Internet Protocol**, is the base protocol that must be followed to send any data over the Internet. Every protocol mentioned in this chapter, whether ICMP, TCP, or UDP, must additionally follow Internet Protocol in order to transmit data. This is even if the data is going to be transmitted over a local network. This is due to the fact that in a modern network, all machines on a local network will be assigned a specific local address that can be used to identify a particular local machine via IP.

The Internet Protocol header has a large number of fields, as shown in Figure 12.1. I won't cover what most of the elements in the header mean, but countless resources online go over what each and every part of this header (and all the other headers in this chapter) refers to.

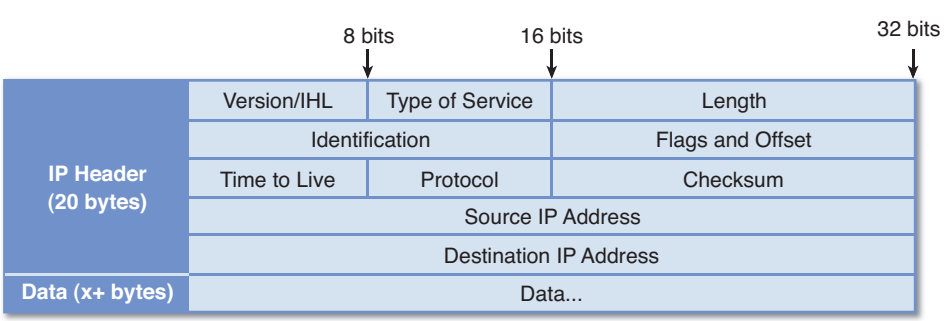


Figure 12.1 IPv4 header.

Two versions of Internet Protocol are in wide use: v4 and v6. An IPv4 address is a set of four 8-bit numbers (called **octets**) separated by a period. So, for example, an address on a local network might be 192.168.1.1. Because IPv4 addresses are 32 bit, there are roughly four billion possible combinations. This might have seemed like a huge number of addresses when IPv4 was first created in 1977, but in the modern day when even kitchen appliances connect to the Internet, four billion just was not enough.

To combat this problem, IPv6 was developed. It uses 128-bit addresses, so it can support an astounding 2^{128} addresses. Because the addresses take up four times the number of bytes, the IPv6 header must be different by necessity. One downside of IPv6 addresses is they are much more difficult to memorize because the full form is several hexadecimal numbers separated by semicolons, as in 2001:0db8:85a3:0042:1000:8a2e:0370:7334. IPv6 was officially "launched" on June 6, 2012, and at some point in the future, IPv4 will be deprecated. For now, most computers on the Internet are reachable via both IPv4 and IPv6 addresses. However, if your platform and ISP support it, switching on IPv6 is recommended.

ICMP

ICMP, or **Internet Control Messaging Protocol**, is not really designed to transmit large amounts of data over the network. Because of this, it cannot be used to send game data. That being said, there is one aspect of ICMP that is relevant when programming a multiplayer game: the ability to **echo** packets. Through ICMP, it is possible to send a packet to a specific address and have that packet be directly returned to the sender via the use of the echo request and echo reply ICMP modes. This echo functionality can be leveraged in order to measure the amount of time it takes for a packet to travel a round trip between two computers.

Knowing the round trip time, or **latency**, is useful in scenarios where there are multiple servers that a player can connect to. If the game measures the latency to all the possible servers, it will then be able to select the server with the lowest latency. This process of measuring the latency to a particular address is known as a **ping**.

When an echo request is sent, the recipient takes the packet's payload and sends it all back in an appropriate echo reply packet. Because the ICMP header does not have any timestamp information, prior to sending out the echo request, the sender must determine the current timestamp and save it in the payload. When the payload comes back via the echo reply, the difference between the timestamp upon send and receipt can be computed, giving the round-trip time. This process is usually repeated a couple of times to get an average latency. The overall layout of an echo request packet is shown in Figure 12.2.

The checksum in the ICMP packet is used to make sure there wasn't any packet corruption in transmission, and is calculated from the values in the ICMP header and payload data. If a recipient gets an ICMP packet with a mismatched checksum, the packet gets rejected. Because the checksum is based on all the other data in the packet, it must be computed once the packet is

ready to transmit. The identifier and sequence values are specific to echo requests—the identifier is usually unique per the machine, and the sequence value should be increased every time a new echo request is sent.

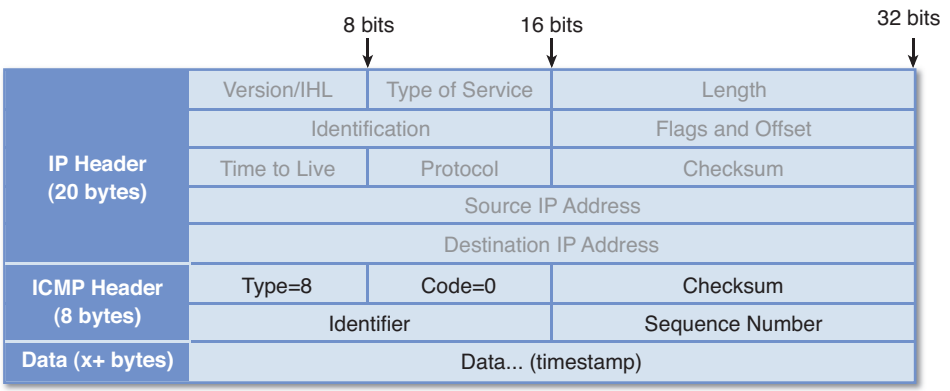


Figure 12.2 ICMP echo request packet.

TCP

Transmission Control Protocol (TCP) is one of two methods a game might use to transmit data over the network. TCP is a connection-based protocol that provides for guaranteed delivery of all packets in the order they were sent. Guaranteed delivery may sound great initially, but as we'll discuss, TCP is typically not as popular for games as the alternative protocol, UDP.

TCP is connection based, which means that before any data can be transmitted between two computers, they must first connect to each other. The way this connection is enforced is via a handshaking process. The computer requesting the connection sends a request to the target computer, telling it specifically how it wants to connect, and then the receiver confirms the request. This confirmation is further confirmed by the initial requester, completing the three step handshake process.

Once a TCP connection has been established, it is possible to send data between the two computers. As mentioned, all packets sent via TCP are guaranteed. The way this works is when a packet is sent via TCP, the receiver sends an acknowledgement of receipt. If the sender does not get the acknowledgement after a certain period of time (the **timeout**), it will resend the packet. The sender will continue sending that packet until it finally receives an acknowledgement, as illustrated in Figure 12.3.

It turns out that TCP guarantees not only that all packets will be received, but that they will be received in the order they were sent. For example, suppose that three packets are sent in the order A, B, and C. If packets A and C arrive, but B doesn't, the receiver cannot process packet C until it gets packet B. So it has to wait for packet B to be resent, and then once it finally is

received, it'll be able to move on. Depending on the **packet loss**, or percentage of packets that don't make it through, this could really slow down the transmission of data, as shown in Figure 12.4.

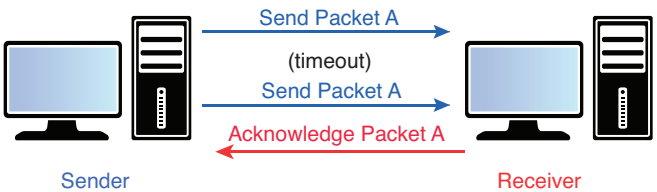


Figure 12.3 In TCP, the sender keeps sending a packet until it receives an acknowledgement.

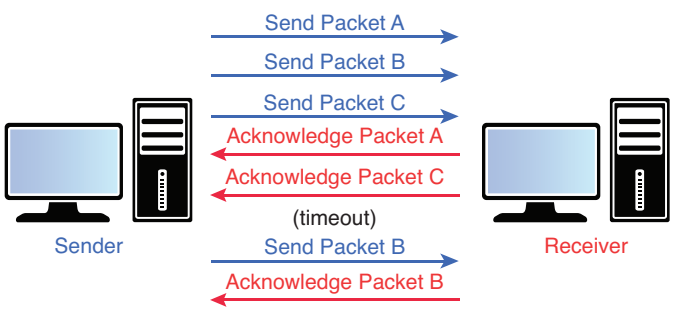


Figure 12.4 TCP packets received out of order.

For a game, guaranteed ordering is usually an unnecessary bottleneck. What if packets A, B, and C in the preceding example contain information regarding the position of a specific player: that is, first the player was at position A, then at position B, and finally at position C. Once position C is received, the game really does not care about position B, as the player is no longer there. But with TCP, the game will be prevented from reading in position C until position B is received first, which is definitely not a desirable scenario.

There is one other helpful aspect of TCP to consider. All networks have an MTU, or **maximum transmission unit**, which determines the size limit for packets. If you attempt to send a packet that's larger than the MTU, it will not go through. Thankfully, TCP is designed such that the operating system will automatically break up large blocks of data into the correct-sized packets. So if you need to download a 1MB file from the Web, the process of breaking it down into the appropriate-sized packets that must be received in the correct order is abstracted away from the application programmer when TCP is utilized.

In most scenarios, it probably will not be necessary to transmit such large amounts of data during gameplay. But there are some fringe uses where the automatic packet creation is helpful—for example, if a game supports custom maps, one issue is that new players may attempt to

join a game session where they do not have the custom map. Using TCP, it would be possible to painlessly send that custom map, regardless of the size, to a new player trying to join the game.

That being said, for actual gameplay there are really only a few types of games that consistently use TCP. For a turn-based game, it makes sense to use TCP because all of the information sent over the network is presumably relevant, because it will define what actions a player performed on a particular turn. Another genre where TCP might see some use is in MMOs; most notably *World of Warcraft* utilizes TCP for all of its data transmission. That's because all of the actions in *WoW* are strictly ordered, so it makes sense to use a protocol that has this ordering built in. But games that have more real-time action, such as FPS or any action games, typically do not use TCP/IP. This is because the forced guarantee of all packets might be to the potential detriment of the game's performance.

Because TCP is a fairly complex protocol, the TCP header is as large as the IP header (20 bytes). The overall layout of the header is shown in Figure 12.5.

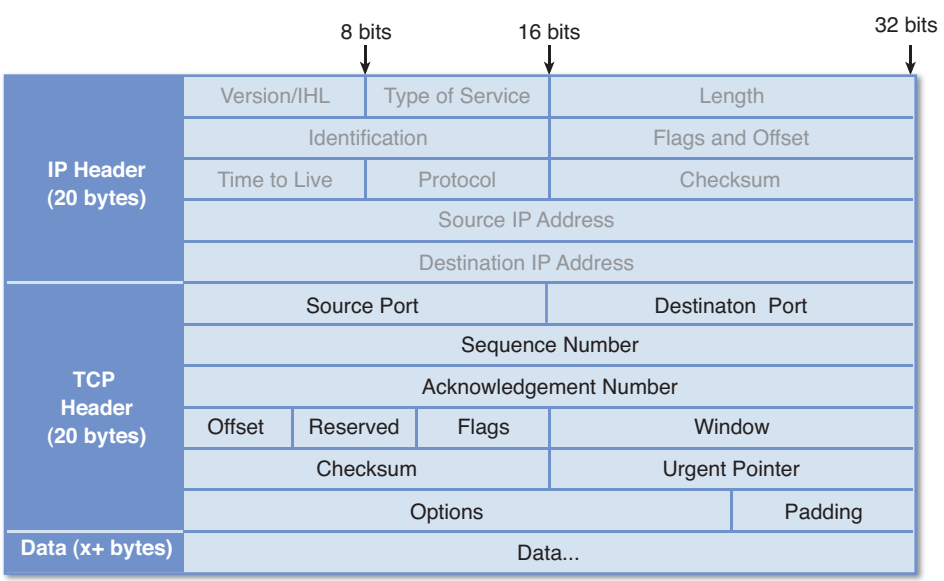


Figure 12.5 TCP header.

As with the IP header, I won't cover every single element of the TCP header. However, one important concept bears mentioning—the **port**. Imagine an office has 50 employees at a particular address. In this scenario, it makes sense to have a total of 50 mailboxes, one for each employee. The receptionist will get the daily delivery of mail from the postal service, and then place each employee's mail in his or her respective mailbox. Without a mailbox, it would be very frustrating for employees to find their mail because they would just have to go through an unsorted bin of every employee's mail.

The same type of sorting is applied to the port when sending packets over the network. For instance, most web servers accept connections on port 80, which means data received on any other port can be ignored by the web server. There is a long list of Internet services and the ports they typically use, though the assignments are not strictly enforced. That being said, you probably don't want to have your game use port 80 because a conflict could occur. There are roughly 65,000 ports to choose from, so although some do have standardized uses, most systems will have tens of thousands of ports available at any particular point in time. The game should select a port that it knows is not being used by any other program. Note that the source and destination port does not have to be the same, so although the web server will accept connections on port 80, the client connecting to the server can be on a different port.

UDP

User Datagram Protocol (UDP) is a connectionless, unreliable protocol. This means that you can just start sending UDP packets to a specific destination without first establishing a connection. And because it is an unreliable protocol, there is no guarantee that a packet will be received, no guarantee in what order packets will be received, and no functionality for acknowledgement of receipt. Because UDP is a much simpler protocol, the header for it is much smaller than TCP, as shown in Figure 12.6.

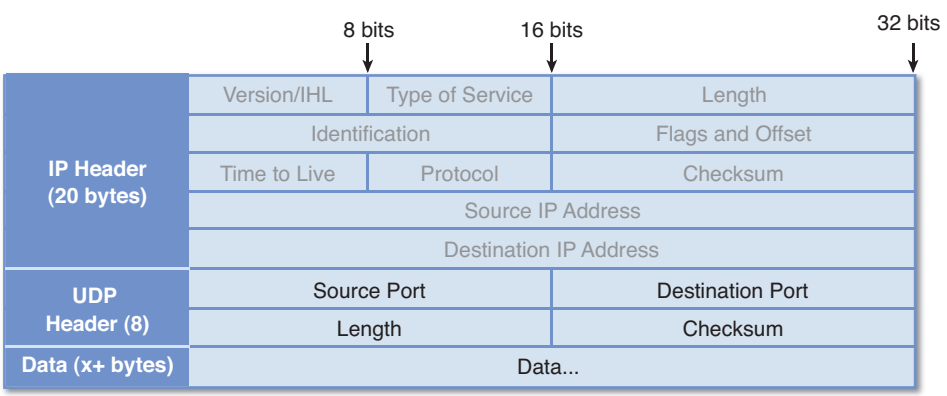


Figure 12.6 UDP header.

As with TCP, UDP also supports roughly 65,000 ports. The UDP ports are separate from the TCP ports, so there is no overlap if both UDP and TCP happen to be using the same port. Because UDP has no reliability, UDP transmissions can be much more efficient than TCP ones. But it also can be problematic in that there is no built-in way of knowing whether or not a packet actually was received. Although there might be some data that's not necessarily mission critical (such as position information of opponents), there is going to be some data that is important to keeping

the game state consistent. If you're playing a multiplayer FPS and you fire your weapon, that information is important and its receipt must be guaranteed by the server and/or other players.

A tempting solution might be to use TCP for mission-critical information, and then UDP for the other data that's not as important. But as shown in an INET 97 proceedings paper (mentioned in the references), TCP's guarantee system running at the same time as UDP increases the number of UDP packets that are lost. A further issue is that even though the movement data may not be critical, and thus it makes sense to send it via UDP, we still need some way of knowing the order of the packets. Without any knowledge of ordering, if position B is received after position C, the player will incorrectly be moved to an old location.

Most games tackle this problem by using UDP, but then adding on a custom layer of reliability for the packets that need it. This additional layer can just be added at the start of the UDP data section—think of it as a custom protocol header added on. The minimum amount of data needed for basic reliability is a **sequence number**, which tracks which packet number it is, and then a bitfield for acknowledgements. By using a bitfield, a particular packet can acknowledge several packets at once, rather than needing to individually acknowledge each packet. This system also has flexibility, because if something truly does not need reliability or sequence information, it can be sent without that, as well. The actual implementation of this system is beyond the scope of this book, but a good starting point is the *Tribes* whitepaper in the references.

As mentioned before, UDP is the predominant protocol used for real-time games. Almost all FPS, action-oriented, RTS, and any other networked games that have time-sensitive information use UDP. It is also for this reason that almost all networking middleware designed for games (such as RakNet) only supports UDP.

Network Topology

Topology determines how the various computers in a networked game session are connected to each other. Although there are a large number of possible configurations, most games support one of two models: server/client or peer-to-peer. As with many decisions, there are tradeoffs and benefits of both approaches.

Server/Client

In a **server/client** model, there is one central computer (the **server**) that all the other computers (the **clients**) communicate with. Because the server is communicating with every client, in this model it is desirable to have a server with more bandwidth and processing power than the clients. For example, if a client sends 10KBps of data over the network, in an eight-player game that means the server must be able to receive 80KBps of data. This type of model is often referred to as a hub-and-spoke model, because the server is the central hub that all of the clients connect to, as in Figure 12.7.

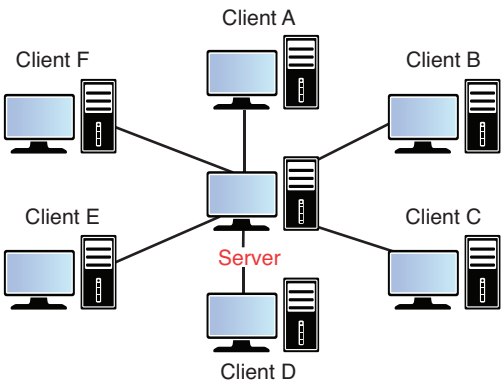


Figure 12.7 Server/client model.

The server/client model is the most popular network topology used in networked games today. Most FPS, action games, MMOs, strategy games, turn-based games, and so on use a server/client setup. Of course there are some exceptions, as with everything, but by and large networked games employ server/client.

In the typical implementation of server/client, the server is considered **authoritative**, which means that it is required to verify most major client actions. Suppose a networked game allows players to throw dodge balls at other players; once a player gets hit by the dodge ball, that player is eliminated from that round. With an authoritative server, when a player wants to throw the dodge ball, the request goes to the server, which verifies that it is a valid action. Then the server will simulate the trajectory of the dodge ball and check every frame for whether or not any client was hit by the dodge ball. If a client gets hit, that client will then be notified by the server that they are out of the round.

The reason for the server verification is twofold. The first reason is that the server is going to have the most up-to-date position information of all the clients. One client throwing the dodge ball at another client might think they definitely hit that client, but that might be because they did not have fully up-to-date information regarding the target's position. Furthermore, if the client can eliminate another player in the dodge ball game without being verified by the server, it would be quite possible for a programmer to make a cheat program that eliminates the other players. Cheating will be discussed in further detail later in this chapter.

Because of the authoritative nature of the server, implementing gameplay code for a server/client game is more complex than in a single-player game. In a single-player game, if the spacebar fires a missile, the same code that detects the spacebar can create and fire a missile. But in a server/client game, the spacebar code must create a fire request packet that is sent to the server, and then the server notifies all the other players about the existence of the missile.

Because it's a dramatically different approach, for games that intend to support both multiplayer and single-player modes, the best practice is to implement single player as a special case of multiplayer. This is done by default in many game engines, including the id Software engines. What this means is that single-player mode actually has the game running a separate server and client on the same machine at once. The advantage of making single player a special case of multiplayer is that any gameplay code that works in single player should also work in multiplayer. Otherwise, a network programmer may have to spend a great deal of time converting client-authoritative gameplay code to work in a server-authoritative setting.

If we go back to our sample dodge ball game, let's imagine we want players to be able to lead targets. That is to say, they need some way of knowing in which direction the opposing players are travelling in order to make an educated throw that they expect to hit the target with. In the best case, a client can expect to receive updates regarding the position of opposing players from the server once every quarter of a second. Now imagine if the client only updated the positions of the opponents when the new position data is received from the server. That would mean that every quarter of a second, the opposing players are going to teleport to a new location. Imagine trying to aim a dodge ball in this scenario—it sure doesn't sound like a very satisfying game to play!

To combat this, most games will implement some sort of **client prediction**, or the client making educated guesses of what happens in between updates from the server. In the case of movement, the way client prediction could work is that the server could send the position of the opposing players as well as the *velocity* of the opposing players. Then, for the frames between the server updates, the client can extrapolate the position of the opposing players based on their last known position and velocity, as demonstrated in Figure 12.8.

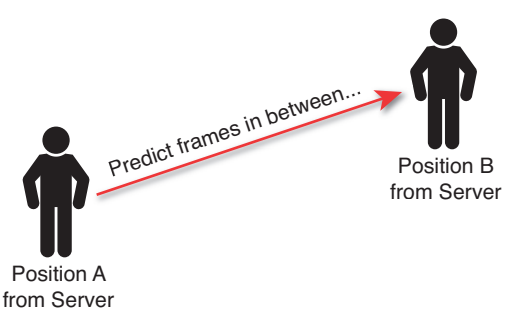


Figure 12.8 Client prediction extrapolates the position of the opponents in between server updates.

As long as the updates are frequent enough, the client should have a fairly accurate representation of the opposing players at all times. However, if the updates are infrequent due to a bad connection, the client prediction will be incorrect. Because the server is the final authority, the client must ultimately fix any discrepancies between the prediction and the actual positions. But if the prediction works well enough, it will appear smooth and seamless to the player.

This concept can also be extended to actions performed on the local end. If we want the gameplay to feel very smooth, as soon as a player presses the spacebar to throw the dodge ball, the throwing animation should begin onscreen. If the client waits until the server confirms the dodge ball throw, it might seem like the game is being unresponsive. The solution to this is that behind the scenes, while the server is actually verifying that the throw is valid, the local client can begin animating the dodge ball anyway. If it turns out that the throw was illegal, the client can correct the problem. But so long as the player is properly synchronized, the perception will be that the game is extremely responsive when dodge balls are being thrown.

Even though the server/client model is a very popular approach, there are some issues to consider. First of all, some games allow a single computer to be both the server and the client. In this case, there is a perceptible **host advantage**, which means that the player who's both the server and the client will have all their actions instantaneously processed by the server.

One game where this problem cropped up was in the original *Gears of War*. What happened was that the shotgun damage was calculated with instantaneous ray casts. So what if two players both fire the shotgun at precisely the same time—with one player as the host, and the other player as a regular client? Well, time and time again, the host would always win the battle of shotguns. This lead to matches where everyone would try to get the shotgun because it was just so powerful. This issue was patched in fairly short order, but it was one of the reasons for the early dominance of the shotgun.

Another issue with the server/client model is that if the server crashes, the game immediately ends as all clients will lose communication with the server. It very difficult to migrate over to a new server (because all the clients will have incomplete information), so there is no potential for corrective action. This means that if a player is the host and is losing, that player can simply quit the game to boot all the other players, which isn't very fun. But the flip side is that in a server/client model, if one client has really bad latency, it does not greatly affect the experience of any of the other players.

In any event, in order to prevent the issues associated with a host, many server/client games only support **dedicated servers**. In most cases, this means that the servers are set up in a specific location (often in a data center), and all players are required to connect to these servers (no one can be a host). Although it's true that players with faster connections will still have an advantage over players with slower connections, any real advantage can be largely mitigated by having the servers at a third-party site. However, a downside of running dedicated servers in this manner is the increased cost of needing to deploy said servers.

Peer-to-Peer

In a **peer-to-peer** model, every client is connected to every other client, as shown in Figure 12.9. This means that there is a symmetric performance and bandwidth requirement for all of the clients. Because there is no single authority in peer-to-peer, there are a few possibilities:

each client is simply an authority over their own actions, each client is an authority over another client, or each client simulates the entire world.

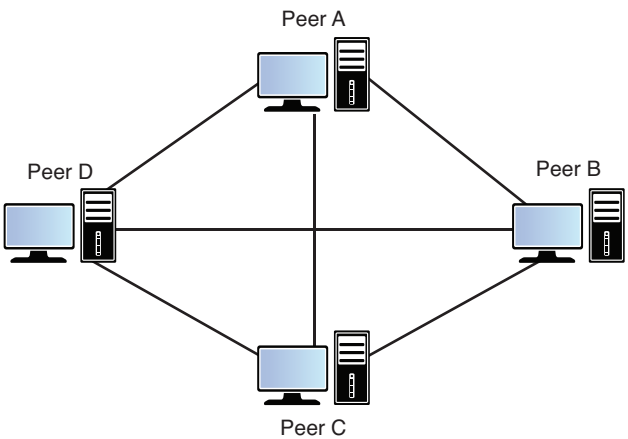


Figure 12.9 Peer-to-peer model.

One genre that commonly uses peer-to-peer is the RTS. The usual model used is a so-called **lockstep** model, where the network update is broken down into discrete "turns" of roughly 150ms–200ms. When any input actions are performed, the commands are stored in a queue and are only executed once the turn ends. This is why when you play a multiplayer game of *StarCraft II*, commands sent to units aren't instantly processed—there is a perceptible delay before the units actually respond to the command because they are waiting for the end of the lockstep turn.

Because the input commands are sent over the network, every client in the RTS game is actually simulating all of the units. They are processing the inputs as if every player were local. This also makes it possible to save all your opponents' commands and view them in an instant replay after the match is over.

The lockstep method results in the clients all being tightly synchronized with each other, and at no point will one player be ahead of another. Of course, there is a downside to this synchronization as well—if one client starts lagging, every other client must wait until that player catches up. But the lockstep approach is very popular for RTS games because it allows for a relatively small amount of data to be transmitted over the network. Rather than sending all of the unit information, the game is only sending a relatively small number of actions per minute, as even the best RTS player is not going to have more than 300–400 commands per minute.

Because in a peer-to-peer setup, every peer might be simulating part of the game world, the game state must be 100% deterministic. This can make it difficult to have gameplay based on

chance; if a Zealot in *StarCraft II* could do a variable amount of damage based on virtual dice rolls, this might result in one peer simulating a round of combat differently than another peer. This doesn't mean that random elements are entirely impossible in a peer-to-peer game (as in *Company of Heroes*), but that further effort must be taken to guarantee synchronization between peers.

Though RTS is arguably the main genre that sees peer-to-peer use, there are definitely other games that do so. One notable example is the *Halo* series, which uses a peer-to-peer model for its multiplayer. But that being said, it's fair to say that peer-to-peer does not see nearly as much use in games as server/client does.

Cheating

A major consideration in any networked game is to ensure that the players have an even playing field. It's the unfortunate truth that some players will do whatever they can to win, even if it involves breaking the rules of the game. In a networked multiplayer game, there are many ways rules can potentially be broken, and so whenever possible, networked games must be designed to prevent players from cheating.

Information Cheats

An **information cheat** is one where a player is able to get additional information that players normally are not intended to have. Suppose you are playing a game such as *Team Fortress 2*, which has a character that can use stealth to hide from other players. Normally, when the spy character goes into stealth mode, that character becomes invisible to all opposing players. But we already know that every so often, the server is going to be sending positional information of the other players. If the server is still sending the location information of the hidden character, it would not be that challenging for an experienced hacker to keep the spy visible, even in stealth mode.

The easiest way to stop information cheats is by limiting what information is available. In the case of the spy character, the solution would be to have the server stop sending position information for a character when in stealth mode. In this manner, even if a hacker figured out how to get the spy character to stay visible, all that would do is enable him to see the last location the spy was at prior to entering stealth mode. It still is cheating, but it typically won't provide that much of an advantage.

But sometimes the solution isn't quite that simple. For example, one cheat that's very common in RTS games is the so-called map hack, which allows a player to see the entire map, including all of the opposing players. This can be a huge advantage in an RTS, because it allows a player to not only see the enemies' movements, but also see what they are building. Because RTS

games are built around the idea of certain units countering certain other units, this can be an insurmountable advantage.

Unfortunately, the solution of not sending the information can't work in the case of the map hack, simply due to the way the lockstep model works. Because each peer simulates the entire game world, it necessitates knowing where all the enemy units are at all times. So the only real solution in this scenario is to attempt to institute cheating countermeasures. In the case of *StarCraft II*, Blizzard's countermeasure is a cheat detection program called Warden. What Warden attempts to do is detect cheat programs that are running while the game is running, and if one is detected the account is flagged as a cheater. Periodically, Blizzard will then go through and ban all of the players who were marked as cheaters.

But the fact of the matter is that no cheat-detection program is unbeatable—this leads to almost a cat and mouse game where Blizzard updates Warden, cheaters get caught, and then cheat programs modify themselves so that Warden can't detect them anymore. One popular countermeasure to Warden is to have a "kill switch" in the cheat program that exits immediately if a Warden update is detected, but such a system can only be successful for so long. In any event, Blizzard and any company that wants to have a serious competitive networked game must be very vigilant about preventing cheating.

Game State Cheats

Whereas information cheats can give one player an unfair advantage over the other players, a **game state cheat** can outright break the game. Once a player is able to modify the game state, that player can very quickly make the game unplayable. One scenario in which this can especially occur is when there is a host. If a player controls the server, it can be fairly difficult to prevent that player from cheating.

Suppose in the dodge ball game, the host sends out a message to every opponent that they were hit by a dodge ball and therefore eliminated from the round. Well, because the server is the authority, that could mean the game ends with the host as the winner. If third-party controlled dedicated servers can't be used, there are a couple potential solutions. One would be to use one of the aforementioned cheat-detection programs; the other would be to add some client verification of the server commands. If the client sees that several players are eliminated at the same time, which is impossible, it can potentially trigger some code that says the server is cheating.

Man-in-the-Middle Attack

One of the most nefarious ways to cheat is to place a machine in between two machines that are communicating in order to intercept all the packets. This is known as a **man-in-the-middle attack**, and it's especially an issue when plain text information is sent over the network. This

style of attack is the primary reason why whenever you connect to a financial institution online, you use HTTPS instead of HTTP.

With HTTP, all data is sent over the network in plain text. This means that if you submit your username and password to a website that uses HTTP, it is trivial for the man-in-the-middle to steal that information. But with HTTPS, all the data is encrypted, which makes it *nearly* impossible for the man-in-the-middle to access the information. There are certain vulnerabilities that have been exposed in the HTTPS protocol, but practically speaking it should not be a big concern for most users.

In the case of a game, however, the biggest advantage of the man-in-the-middle attack is that it allows the cheating to happen on a machine that isn't playing the game. This means that nearly all cheat-detection programs will be rendered impotent—they simply have no way of knowing that the packets are being intercepted and modified. A further concern is that having access to the packet information in general might allow a hacker to discover additional vulnerabilities in the game that can then be exploited.

One solution to this type of attack is to encrypt literally every packet so that only the server can decrypt and understand the packet. But this is probably overkill for most game information, and it does have the added overhead of needing to encrypt everything. But at the very least, whenever a game requires players to log in (such as in an MMO), some sort of encryption must be utilized to ensure that players don't have their accounts stolen.

Summary

Networking is a very deep concept, and this chapter covered the important basics. The cornerstone of any networked game is the protocols that allow transmission of data over the network. Although TCP has guaranteed reliability, most games choose to use the more efficient UDP. Once the protocol is selected, the next major decision is what type of network topology will be utilized. Although the majority of real-time games today use a server/client model, most RTS games still use peer-to-peer. The topology will greatly influence how the networked game has to perform every update. Finally, an issue that every multiplayer game programmer has to face at one time or another is cheaters, and there are a number of countermeasures that can be employed to combat them.

Review Questions

1. What is the Internet Protocol? What is the difference between IPv4 and IPv6?
2. What is ICMP and how might it be used for a game?
3. What are the main characteristics of the TCP protocol?

4. What is a port in networking?
5. Why is the UDP protocol generally more preferable for real-time games?
6. What is a server/client model, and how does it differ from peer-to-peer?
7. What does client prediction entail? How does it enhance the gameplay experience for players?
8. Briefly describe how the peer-to-peer "lockstep" model works for RTS games.
9. Give an example of an information cheat and how it might be combated.
10. What is a man-in-the-middle attack, and how can it be prevented?

Additional References

Frohnmayer, Mark and Tim Gift. "The TRIBES Engine Networking Model." This classic paper outlines an implementation of adding reliability and data streaming to UDP, as used in *Tribes*.

Sawashima, Hidenari et al. "Characteristics of UDP Packet Loss: Effect of TCP Traffic." Proceedings of INET 97. This article goes into a very technical discussion as to why TCP acknowledgements can induce packet loss in UDP.

Steed, Anthony and Manuel Oliveira. *Networked Graphics*. Burlington: Morgan Kaufmann, 2009. This book provides an in-depth look at all the different layers involved with programming a networked game.

CHAPTER 13

SAMPLE GAME: SIDE-SCROLLER FOR IOS

A side-scroller is a game where the action is viewed from the side. An endless scroller is a variation in which the game world continues to scroll until the player loses. An example of a very popular endless scroller is *Jetpack Joyride*.

In this chapter, we take a look at a sample implementation of a 2D space-based endless scroller called *Ship Attack*. The code for this game was written in Objective-C using the cocos2d framework, and is for iOS devices.

Overview

This chapter is intended to be read while reviewing the source code for the game, which is available on the book's website at http://gamealgorithms.net/source-code/shipattack/. Because this game is for iOS, in order to run the code you need a Mac (running Mountain Lion or higher) as well as Xcode (which can be downloaded for free from the App Store). The code unfortunately cannot be tested on a Windows machine.

The control scheme for *Ship Attack* is fairly simple. Hold the iPhone horizontally and slide your left thumb up and down on the left part of the screen in order to move the ship. To fire, hold down your right thumb on the right part of the screen. The objective of the game is to avoid enemy projectiles while destroying them with your lasers.

The sprites for *Ship Attack* were created by Jacob Zinman-Jeanes and are used under the CC-By license. A link to the sprites is provided on the book's website. These sprites were packed into a sprite sheet using TexturePacker. For more information on sprite sheets, refer back to the section on it in Chapter 2, "2D Graphics."

Now, before we review the source code in detail, let's first take a look at the technologies used to create *Ship Attack*, shown in Figure 13.1.

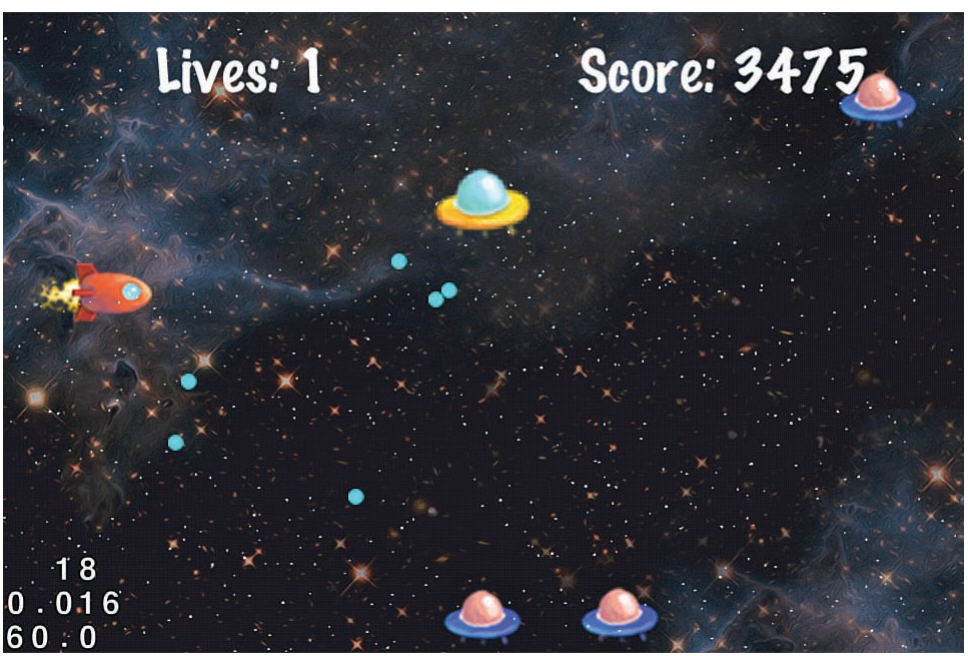


Figure 13.1 Screenshot of *Ship Attack*.

Objective-C

The Objective-C language was originally developed in the early 1980s with the goal of adding object-oriented concepts to C. It was developed around roughly the same time as C++, though at different laboratories by different researches. Objective-C really was not used very much until NeXT, a company started by Steve Jobs, licensed the language for their new platform. Although NeXT never really gained that much popularity, as part of Jobs' return to Apple in 1996, NeXT came with him. The NeXT operating system was then used as the base for Mac OS X, and later the iOS operating system. Today, all GUI programs developed for the Mac and any iOS device must use Objective-C on at least some level.

There are some notable differences been C++ and Objective-C, most notably Objective-C technically does not have the concept of member functions. Rather, classes are sent messages and then respond to them—this means that all messages must be passed and processed dynamically, whereas in C++ member functions are only dynamic if the virtual keyword is utilized. Even though they aren't technically member functions, during this chapter I will still refer to them as such because the terminology is more familiar.

The syntax for message passing in Objective-C might be a little odd for someone used to the C++/Java language style. To call a member function in C++, one might use the following:

```
myClass->myFunction(arg);
```

But to pass an equivalent message in Objective-C, the syntax is this instead:

```
[myClass myMessage: arg];
```

While browsing through the code, you may also notice that function declarations have either a `+` or `-` in front of them. A `+` signifies the method is a class method (which you can think of as a static function in C++), whereas a `-` signifies the method is an instance method (or a regular member function). Here's an example of each:

```
// Class method (essentially a static function)
+(void) func1;

// Instance method (regular member function)
-(void) funct2;
```

If you find yourself confused by the syntax of Objective-C, note that it is possible to limit the usage of Objective-C in an iOS app. This is because it plays nicely with C++ code, with a few restrictions. So it is absolutely possible to write 90% of the code for an app in C++ and then only use Objective-C where it must interface with the operating system. This is the recommended approach for a mobile game that you might want to later release on the Android platform, because it will reduce the amount of code that must be rewritten. However, because *Ship Attack* uses the coco2d framework, it is written 100% in Objective-C.

cocos2d

Cocos2d (available at www.cocos2d-iphone.org) is a 2D game framework that targets both the Mac and iOS platforms. Because it's only for these two platforms, its interfaces are all written in Objective-C. Cocos2d provides a lot of built-in functionality for 2D sprites, so it makes creating 2D games fairly painless.

A core concept in cocos2d is the base `CCNode` class, which represents a **scene node**, or an object that has a position and can be drawn onscreen. From `CCNode` there are several children, including `CCScene` (an overall scene), `CCLayer` (a specific layer in a scene), and `CCSprite` (a sprite object). At a minimum, a cocos2d game must have one `CCScene` and one `CCLayer`, though most games will have more than this.

A `CCScene` can be thought of as a specific state of the game. For example, in *Ship Attack* there is a main menu scene as well as a gameplay scene. A `CCScene` must be associated with at least one `CCLayer`, though it's possible to have several `CCLayers` with different z-ordering. The gameplay scene has three different layers: a far-away nebula, a main object layer, and a faster scrolling star field. You normally also might have a separate layer for the UI, but because this game is so simple it did not seem necessary to separate the UI.

`CCSprite` is used for all of the moving and interactive objects in the game. This includes the player's ship, lasers, enemy projectiles, and enemy flying saucers. Rather than storing each sprite in an individual file, I used sprite sheets, which were generated via TexturePacker. Conveniently, cocos2d can automatically import sprite sheets created by TexturePacker, so it makes the process of loading and animating said sprites relatively straightforward.

One drawback of using cocos2d is that it's limited to only the two Apple platforms. However, a C++ port of the library called cocos2d-x is now available that supports a long list of platforms, including all the major mobile and desktop ones. Although the class and function names are almost identical in cocos2d-x, there are some differences in how they are used simply due to language differences. For example, cocos2d-x has STL at its disposal, whereas cocos2d must use `NSMutableArray` and the like. For *Ship Attack*, I chose to use base cocos2d because less documentation is available for the newer cocos2d-x. However, as more books on cocos2d-x enter the market, this will become less of an issue.

Code Analysis

Now that we've covered the technologies behind *Ship Attack*, we can take a deeper look at the source code and how its systems are implemented. Overall, the code for this game is definitely simpler than the code for the game in the following chapter. This makes sense because the mechanics of *Ship Attack* are far simpler than that of the tower defense game in Chapter 14, "Sample Game: Tower Defense for PC/Mac."

If you browse all the folders in the project, you might see a huge number of files, most of which are in the libs directory. But all of these source files are the base cocos2d files; they are not files that were created specifically for *Ship Attack*. Therefore, don't bother looking through those source files. You are only interested in the handful of files that are in the main shipattack directory.

AppDelegate

The `AppDelegate` class contains `didFinishLaunchingWithOptions`, which is the (rather verbose) entry point of any iOS program. For the purposes of a cocos2d game, almost all of the code related to the `AppDelegate` is auto-generated. The `AppDelegate` created by the template normally launches into a scene called `HelloWorld`. For this game, I've modified it so that the first scene it goes into is instead the main menu.

The only other change made to the auto-generated `AppDelegate` code was to enable multitouch, which is what allows the game to respond to multiple finger presses. Without multitouch, the control scheme for this game would not function properly, because one finger controls movement and the other controls firing.

MainMenuLayer

`MainMenuLayer` has a class method called `scene` that returns a `CCScene` that contains `MainMenuLayer` as its only `CCLayer`. This is the recommended approach in cocos2d when a scene has only one layer. The layer itself has very little code in it. All it does is show the title of the game and creates a menu with one option (to start the game). Menus can be set up in cocos2d using `CCMenu` and `CCMenuItem`, and a particular menu item can have a segment of code that is executed when the item is selected. In this case, when the Start Game option is selected, the game transitions into the `GameplayScene`.

GameplayScene

As you might expect, the `GameplayScene` is the scene representing the main gameplay. If you look at this file, you'll see that there are three different layers: two `ScrollingLayers` and the `ObjectLayer`. The two `ScrollingLayers` are used for the nebula and star field, whereas the `ObjectLayer` is where all the objects in the world (as well as the score display) reside.

`GameplayScene` really does not do much else other than have an update function that gets called every frame. This function, in turn, updates the layers. Finally, a sprite frame cache loads in the sprite sheet and allows other classes to grab the correct sprites.

ScrollingLayer

`ScrollingLayer` is a generic class I created that allows for two or more screen-sized segments to scroll infinitely. To construct a `ScrollingLayer`, you pass in an array that lists out all of the background sprite names, as well as the speed (pixels/second) at which the layer should scroll. To simplify things, `ScrollingLayer` assumes that there is enough memory to load in all of the backgrounds at the same time.

The update function is where most of the work happens in this file. What it does is shift the positions of all the background sprites by however many pixels have been traversed on a particular frame. Once a background sprite fully scrolls off the screen, it has its position moved to be all the way to the right of any other background sprites. This way, when it's time for a particular image to appear onscreen again, it will slide in properly.

When I originally implemented the nebula scrolling layer, there was an issue with a **seam** (a one pixel vertical line) visible when the nebula wrapped back to the beginning (see Figure 13.2). After poring over the code I became frustrated because I did not see any issue with the implementation of `ScrollingLayer`. However, one of my technical reviewers pointed out that the issue was actually in the nebula image files I was using. It turned out that one of the files had a vertical line of gray pixels that was causing the issue. Once the grey pixels were eliminated from the image file, the nebula scrolled perfectly. Let this be a lesson that you should always double-check your content and make sure it's not causing any issues!

Figure 13.2 A seam is visible between scrolling layers in an earlier version of *Ship Attack*.

In any event, *Ship Attack* uses two layers scrolling at different speeds in order to create a parallax effect (as covered in Chapter 2). The star field scrolls much more quickly than the nebula, thus giving the perception that the star field is closer than the nebula.

Ship

One thing you might notice about this game's code is there isn't a class that's called "GameObject." Instead, the code uses `CCSprite` (a cocos2d class that represents a sprite object) as the base class for objects in the game. So for the case of the player's ship, the `Ship` class inherits directly from `CCSprite`, as opposed to any other intermediary class.

The `init` function for `Ship` sets up a looping animation for the `Ship` using **actions**, which comprise a cocos2d system for having a `CCNode` do something. For example, there are actions to run an animation, move to a specific location, and so on. These actions very much simplify certain behaviors because it's possible to give a command and hand it off to cocos2d.

All the `update` function does in `Ship` is move the ship vertically toward the target point, which is the point where the left thumb of the player is. As the player moves his thumb, the target point changes, which in turn causes the ship to move toward the target point. The movement to the thumb is not instant so as to prevent teleporting. But at the same time it's fast enough that the player should feel that the controls are responsive.

Projectiles

The `Projectile` class inherits from `CCSprite` and is the base class used for both the ship's lasers as well as the enemy's projectiles. The `update` function for `Projectile` is extremely simple. First, it updates the position of the projectile using Euler integration (which was discussed in Chapter 7, "Physics"). Then, it checks to see whether or not the projectile has moved outside the bounds of the screen. Once a projectile moves outside the bounds, its `despawn` function is called.

Two classes inherit from `Projectile`: The first is `Laser` (which is the ship's laser) and the second is `EnemyProj` (which is an enemy projectile). These inherited classes only have to do two things: set the correct sprite and overload the `despawn` function that removes the particular projectile from the `ObjectLayer`.

Enemy

The `Enemy` class is the most complex of the object classes. Right now, the enemy supports two different types of behaviors: a figure-eight path (which is performed by the larger, yellow flying saucers) and a "post" route (which the regular blue enemies use). The two functions that set up these paths are `setupFigure8Route` and `setupPostRoute`.

Note that for both routes, I utilize actions to set up the entire path. When an `Enemy` is spawned by `ObjectLayer`, it sets up the correct route for the enemy and then the actions will dictate what it does. In the case of the post route, the enemies will move toward the middle of the screen and then exit from either the top or the bottom. In the case of the figure eight, the enemy will endlessly perform the figure eight until it's finally defeated.

The firing patterns of the two types of enemies are also different. The post route enemy will only fire once in the negative x-direction, whereas the figure-eight enemy will fire a projectile at the position of the player. This behavior is contained in the `fire` function, which uses some basic vector math to determine the direction of the projectile.

The other functions in `Ship` are more or less helper functions. One sets up the correct animation set, depending on the color of the ship, and the other generates a list of `CCMoveTo` actions when given a list of points to travel toward.

ObjectLayer

The `ObjectLayer` is the main layer that controls gameplay, and you'll notice a lot of member variables in this class. Most notably, it has a pointer to the `Ship` and arrays for the player's lasers, the enemies, and the enemy projectiles. Beyond that, several parameters are used to control the basic behavior of the game, such as the speed of the ship, the respawn time, the interval between enemy spawns, and so on. These variables can be modified in order to tweak the difficulty of the game.

The update function in `ObjectLayer` is responsible for updating all of the game objects that are currently active. So it loops through the arrays that contain said objects and executes each object's update member function. Once all of the objects have been updated, the code performs some collision checks. Using AABBs (which were covered in Chapter 7), the code first checks to see if the active lasers intersect with any enemies. If an enemy is hit by a laser, the laser dies and the enemy loses some health. If the enemy's health hits zero, it dies and is removed from the world. Similarly, the lasers are checked for collisions against the enemy projectiles, and both the laser and the projectile get destroyed if a collision is detected. Finally, the update function checks to see if any enemy projectiles are colliding with the player's ship, in which case the player loses a life.

The enemy spawning is configured so there's a `spawnEnemy` function that's called every X seconds. That X value decreases as the player gets further and further in the game. When an enemy is spawned, it's placed at a random y-position using `arc4random`, which is a high-quality random number generator. The game also tracks how many enemies it has spawned, and after a certain number of enemies a boss enemy will spawn. A boss enemy is one of the larger enemies in yellow that performs a figure eight.

In cocos2d, a layer can be configured to receive touch events. This is done with the `setTouchEnabled` call toward the top of `init`. Once this is configured, you can then have

functions that get called based on specific touch events. When the player first initiates a touch, a call to `ccTouchBegan` is triggered. The code in this function checks whether the touch is on the left or the right part of the screen. A touch event on the left part of the screen is saved into `m_MoveTouch`, because it's a touch that's trying to control ship movement. A touch on the right part of the screen is instead saved into `m_FireTouch`.

We save these off because the same touch object will be used as the player moves his finger across the screen. Every time an active touch has its position changed, the `ccMoveTouch` function gets called. Notice how the code in this function checks to see if the touch that moved is the one that controls the ship movement—if the position of `m_MoveTouch` changes, that should modify the target position of the ship, as well. All other touches are ignored because we don't care about the positional change of any touch that's not the one tied to player movement.

0nce the player lifts his finger and the touch is no longer active, `ccTouchEnded` is called. In the case of `ObjectLayer`, all that it does is check to see whether it was the `m_MoveTouch` or `m_FireTouch` that ended and clear out that touch object reference, as appropriate.

Finally, there are functions in `ObjectLayer` whose purpose is to track the state of the game. These include functions that grant the player points for killing enemies as well as keeping track of the number of lives the player has. If the player's life total hits zero, the game ends and a "game over" menu is displayed.

Exercises

Now that you've taken a look at the code, here are a couple features that could be added to this game:

1. **More Enemy Types**

   Having only two types of enemies makes the game fairly repetitive. The simplest way to add more enemy types would be to add an additional route function to the `Enemy` class. Then this route could be activated based on a variety of conditions. You could add more enemy images by dropping the files into Resources/sprites and then loading the space.tps file into TexturePacker. From TexturePacker, you can then click "publish" to generate new sprite sheets.

2. **Have Structured Waves**

   If you play a classic wave-based shoot-'em-up game such as *1942*, you will notice set wave patterns where different types of enemies approach at different points in time. If you replay a level over and over again, you will notice the waves of enemies are always the same. This is more interesting than the approach used in *Ship Attack*, where enemies only spawn on a set timer.

In order to have structured waves, you want to add some sort of class to manage the spawning of waves. You also need some way to define the timing of enemies spawning. One way to d this would be to have a list of all the enemy spawns, essentially detailing the point in time, the type of enemy, and the position of the spawn. The spawn manager then could keep track of the overall elapsed time and spawn the appropriate enemies.

Summary

Our discussion of *Ship Attack* should have given you an idea of how to implement a simple-yet-interesting 2D side-scroller. Because it's a 2D game, it employs many of the techniques discussed in Chapter 2. This discussion also touched on a bit of physics (which was covered in Chapter 7) through the use of Euler integration as well as AABB collision detection. From the basic building blocks provided by this game, it would certainly be possible to create more complex and interesting versions.

CHAPTER 14

SAMPLE GAME: TOWER DEFENSE FOR PC/MAC

The tower defense genre has been around for many years, but its popularity really took off in custom maps for games such as *StarCraft* and *Warcraft III*. It's such a popular genre that today there are many tower defense games on nearly every gaming platform.

This chapter takes a look at a sample implementation of a futuristic tower defense game called __*Defense*. It is fully rendered in 3D, though the gameplay occurs on a 2D plane. The code for this game was written in C# using the XNA/MonoGame frameworks.

Overview

Before you read this chapter, you should visit the book's website and download the source code for this game. The link to the source code, along with instructions on how to build it, can be found at http://gamealgorithms.net/source-code/defense/. Once you've got the game running and have played it, you're ready to continue on. But before we dive into the code analysis, let's take a brief look at the different technologies used to make *__Defense* (pictured in Figure 14.1).

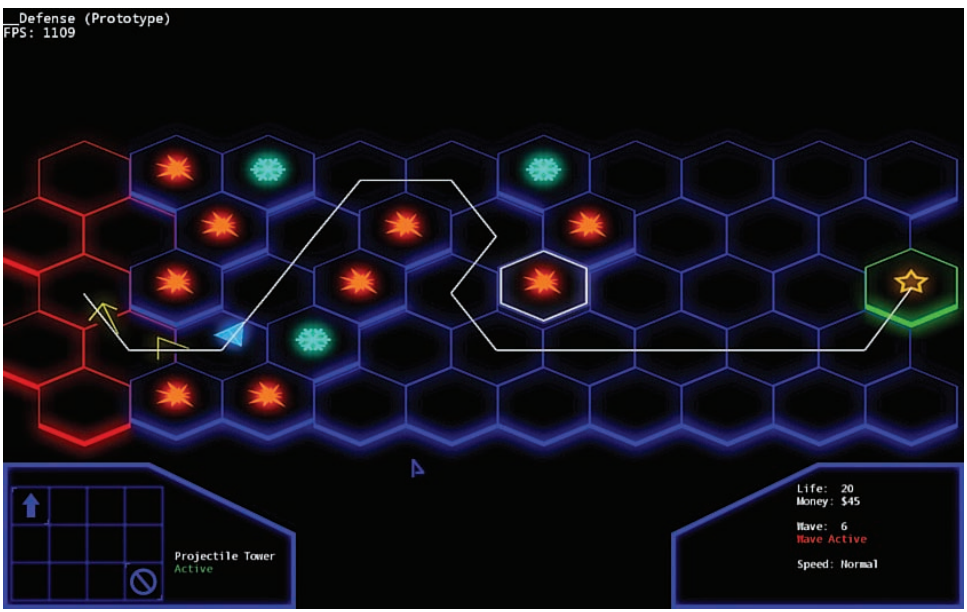


Figure 14.1 *__Defense* in action.

C#

Microsoft developed **C#** (pronounced *C sharp*) in order to have a language that was similar to C++ or Java, but enabled rapid development of Windows applications on the .NET framework. Like Java, C# is not compiled directly into native code, but into a byte code that can then run on a virtual machine. At first, there was only a virtual machine that could run on PCs, but now there are open-source implementations of the C# virtual machine on many different platforms.

Though there initially were many similarities between C# and Java, over time the languages have diverged pretty dramatically. But because it is syntactically very much related to both C++ and Java, the language is still relatively easy to pick up for someone experienced in either of these languages. Over time, C# has become the language of choice for rapidly developing GUI apps for Windows. Because of this and other reasons, C# is a very popular choice for game

development tools—engines such as the newer versions of Frostbite (used for the *Battlefield* series) use C# heavily in their toolchain.

Most of the language constructs in C# should be immediately familiar to a C++ programmer, but a couple of aspects are worth mentioning. First of all, there (usually) is no memory management in C#. All basic types and structs are always allocated on the stack, and all classes are always allocated on the heap. The language is garbage collected, so you don't have to worry about leaking memory—once an object is no longer referenced, it will be deleted at some point in the future. An "unmanaged" mode in C# does allow for lower-level access, but it's not recommended unless it's for a very specific reason.

There is one unique feature of the language that's used quite a bit in the code for this game. Normal rules of class encapsulation state that all data in a class should be private and only be accessible via getter or setter functions. So in C++, a 2D vector class might be implemented like this:

```
class Vector2
{
public:
   int GetX() { return x; }
   void SetX(int value) { x = value; }
   int GetY() { return y; }
   void SetY(int value) { y = value; }
private:
   int x,y;
};
```

Then if you had an instance of a `Vector2`, in order to set the x-value to 10, you would call the `SetX` function like so:

```
Vector2 v;
v.SetX(10);
```

The annoying thing about getter and setter functions, though, is that they are typically quite verbose. To solve this, C# added **properties**, which are basically getter and setter functions, but the fact that they are functions is hidden from the code that uses them. Properties are accessed syntactically as if they were variables, but they still have all the protection that creating getter/setter functions can provide. Therefore, the preceding class in C# could be expressed as follows:

```
class Vector2
{
   private int x, y;
   public int X
   {
      get { return x; }
      set { x = value; }
   }
```

```
    // Same for Y property
    ...
}
```

Then the code using the `Vector2` would look like this:

```
Vector2 v = new Vector();
v.X = 10; // Automatically calls get property on X
```

The neat thing about a property is it follows the syntax of a normal variable, even though it secretly isn't one. There are quite a few other unique features in C#, as well, but the majority of them do not really see use in the code for this game. When going through the code, if you find yourself struggling with C#-isms, you can find many references online for learning C# coming from a C++ or Java background.

XNA

XNA is a 2D and 3D game library for C# that was created by Microsoft for the development of games that work on the PC, Xbox 360, and Windows Phone 7. PC development is completely free, but in order to develop for the Xbox 360, there is a $100/year subscription (though students can get one year free at http://www.dreamspark.com/). However, paying the fee also allows you to potentially release your game on the Xbox Live Indie Games store. There have been some major commercial successes that used the XNA framework. One of the most popular is the indie game *Bastion*, which launched on the Xbox 360 in 2011.

I personally think XNA is a great framework for first learning how to program games (especially 3D ones). It handles the more tedious aspects of programming a game—such as creating a window, initializing the graphics library, and most importantly, loading in art files. In XNA, writing code that can load a 3D model, initialize the camera/projection matrices, and then render the model can be done in only a handful of lines. That's because XNA has built-in support for loading a large number of media files, including textures, models, sounds, and even certain types of videos.

Although it certainly might be easier to prototype a 3D game in a full engine such as Unity or UDK, the problem with such engines is that they do most of the programming heavy lifting for you. For instance, if you want to use pathfinding in one of those engines, you can use the built-in pathfinding and never have to implement A* once. But in XNA, if you want to have pathfinding, you have to implement it yourself, which means you will have to understand the algorithm well.

Unfortunately, Microsoft has decided to abandon XNA, and they no longer will be providing updates to the framework. But even though XNA has been orphaned by Microsoft, the framework still lives on in MonoGame.

MonoGame

MonoGame is an open-source and cross-platform implementation of the XNA framework. It uses all of the same class and namespace names that XNA does, so it really is a very quick process to convert an XNA game to MonoGame. In the case of *__Defense*, it took maybe 15 minutes to get the XNA project to work in MonoGame. The first two versions of MonoGame only supported 2D games, but as of version 3.0 (released in March 2013), the framework now supports most of the 3D functionality, as well.

What's great about MonoGame is that because it's written in a cross-platform manner, it works on a large number of platforms. At the time of writing, it runs on Windows, Windows 8 Metro, Mac, Linux, iOS, and Android (though the mobile versions are not free). The aforementioned *Bastion* was ported from XNA to MonoGame, and that's how it was released on so many additional platforms, including Mac, Linux, iOS, and even the Chrome Web Store. In the future, it may even be possible to develop Xbox One games using MonoGame, depending on how Microsoft implements their system for development on retail units.

One important aspect of the XNA framework that hasn't quite yet been fully implemented by MonoGame is the ability to convert assets from their original format to the custom format that XNA/MonoGame uses. So in order to prepare assets for use in MonoGame, you have to build them using XNA and then copy those files into the appropriate directory for the MonoGame project. But the MonoGame developers are actively working on a content pipeline for MonoGame, so by the time you're reading this, it may no longer be an issue. But for now, if you want to change any of the models or textures for this game, you have to build the assets in the XNA solution before it will work in the MonoGame ones.

Code Analysis

Compared to the sample game in the last chapter, there's a lot more code to go through for *__Defense*. But that makes sense because the mechanics and interface for this game are far more involved than *Ship Attack*. If you try to just jump in and go through the files without reading this section, it may be difficult to grasp it all. The best approach is to consult this section while perusing through the code.

Settings

Normally, it's best if settings are stored in external files for ease of editing (as in Chapter 11, "Scripting Languages and Data Formats"). Because this game is relatively simple, it felt a bit unnecessary to write the code for parsing in settings from a file. However, I didn't want to just have all the parameters hard-coded all over the place, so I ended up consolidating most of them into three source files.

Balance.cs has all of the parameters that affect the balance of the game. So parameters such as the number of waves, number of enemies, health of enemies, and anything that adjusts the difficulty of the game are stored in Balance.cs. Any settings that don't particularly affect the balance of the game (such as whether the game is full screen and the camera speed) are in GlobalDefines.cs. Finally, DebugDefines.cs is mostly unused, but has a few parameters related to debugging the game.

Singleton

A **singleton** is a class that's globally accessible and has only one instance in the entire program. It's one of the most commonly used design patterns and often one of the first classes to implement in a new engine. Although you could try to implement a singleton using a global class, that's typically not the best way to do so. For this game's code, the singleton implementation is in Patterns/Singleton.cs.

The way `Singleton` works is it's a templated class that has a static instance of the templated type, and a static `Get` function that returns the static instance. It sounds a little complicated, but the end result is that in order for a class to be a singleton, all it must do is inherit from `Singleton` like this:

```
public class Localization : Patterns.Singleton<Localization>
```

That makes the `Localization` class a singleton that can be globally accessible anywhere via the `Get` function, as shown here:

```
Localization.Get().Text("ui_victory");
```

Several classes in this game are singletons, including `GraphicsManager`, `InputManager`, `PhysicsManager`, `SoundManager`, and `GameState`. It might seem like the camera would also be a good candidate for a singleton. However, the camera isn't a singleton because if the game were a split-screen game, there could potentially be multiple cameras. In order to try to keep the code flexible, avoiding assumptions like this that may not be true for every game is typically a good idea.

Game Class

Game1.cs has the main game class, but it really doesn't have very much code. `Initialize` is called when the game is first loaded, and it simply instantiates some of the singletons. `Update` and `Draw` are called every frame, and they just call `Update` and `Draw` on the appropriate singletons. So overall, there really is not much happening in this file.

One small thing worth noting, though, is the following code snippet from `Update`:

```
if (fDeltaTime > 0.1f)
{
   fDeltaTime = 0.1f;
}
```

This forces the minimum frame rate to be 10 FPS. The reason for this is if the game is paused in the debugger, when it's unpaused the elapsed time might be several seconds. By limiting the minimum FPS to 10, it makes sure that there's never an elapsed time that's so long that the behavior of the game becomes unstable.

Game State

The overall state of the game is controlled by `GameState` (in GameState.cs), which is why it's the largest file in the game. It's a state machine, but because the main menu and pause state have so little behavior, almost all of the code in `GameState` is for controlling the overall gameplay. Because so much of the code is for one particular state, I did not use the state design pattern (as in Chapter 9, "Artificial Intelligence"). But if a particular game has a lot of different states (such as different game modes, for instance), it probably would be a good idea to refactor this to use the design pattern instead.

Anything related to the overall state of the game, such as the amount of money, the health, the time until the next wave, and so on, are also stored in `GameState`. So the majority of the member variables in `GameState` are very game specific, but there are some that aren't. One is the linked list of all of the active game objects in the world. All new game objects must go through the `SpawnGameObject` function in `GameState`, as that's what adds it to the game world. Then, in `UpdateGameplay`, all of the game objects are updated provided that the game is not paused.

There are a couple of wrinkles worth mentioning in `UpdateGameplay`. First of all, it doesn't actually iterate on the game object linked list, but on a copy of the game object linked list. This is to allow for the fact that some game objects might need to remove game objects (including themselves) during the update. In C#, the `foreach` does not allow for modification of the container during the loop, which is why the copy is made. Although this may seem very inefficient, keep in mind that classes are always passed by reference in C#. So copying the list is just doing a shallow copy of pointer data, which isn't really that computationally expensive. The other notable code in this function is that the delta time that `UpdateGameplay` receives is not always the delta time that is passed to all of the game objects. This is what allows the player to change the speed of the game using the +/- keys.

`GameState` also has an instance of the `Camera` class that it keeps track of. Another important member variable in this class is the UI stack, which is a stack of all the currently active user interface screens. The UI system will be discussed in further detail later in this chapter.

Game Objects

The base game object class is in Objects/GameObject.cs. In *__Defense*, all game objects are both drawable and updatable, so there isn't a complex hierarchy of interfaces to implement for different categories of game objects. The world transform matrix is built from a vector for position, a float for scale, and a quaternion for the rotation, as discussed in Chapter 4, "3D Graphics." All game objects have a bounding sphere, and any game objects that also want to have an AABB (such as the Tiles) can set the `m_bUseAABB` Boolean to true in their constructor.

The `Tower` class (Objects/Tower.cs) inherits from `GameObject` and has all the behavior specific to towers. Most notably, it has functions for building and upgrading towers, as well as changing the texture depending on whether or not the tower is active. Because the game currently has two types of towers, it makes sense that there are two children of `Tower`: `ProjectileTower` and `SlowTower`. The custom behavior for each of these towers is more or less implemented in its `Update` functions.

The `Tile` class that inherits from `GameObject` really doesn't have much functionality—it just provides for being able to attach a `Tower` onto a `Tile`. It may seem like overkill to have the `Tile` as a separate game object, but this was done so that the height of the tiles can dynamically change. This still would have been possible even if the tiles weren't game objects, but it definitely was a simpler implementation to make them a subclass of `GameObject`. The children class of `Tile` (`TileRed` and `TileGreen`) were intended to have different functionality than the base `Tile`, but they don't really anymore (except red tiles can't be built on).

`Projectile`s are spawned by `ProjectileTower`, and all they do is travel toward the enemies and deal damage when they get to them. The movement is done with a lerp (discussed in Chapter 3, "Linear Algebra for Games"), and they just slowly get closer and closer to the enemy's position. Originally, the collision between the projectile and the enemy was done using an instantaneous bounding sphere check (like in Chapter 7, "Physics"). But the problem with this was that the higher level enemies had such huge bounding spheres that the projectile would almost instantly collide and disappear. So to make the game more challenging, the code was changed so that irrespective of the enemy size, the projectile has to make it to the center of the enemy to deal damage.

Both towers use functions that iterate through all the enemies in the world. In the case of the `ProjectileTower`, it finds the closest enemy, whereas the `SlowTower` finds all the enemies within range. Both of these functions use brute-force algorithms to determine which enemies to affect. One way to solve this issue would be through the use of a partitioning algorithm such as a quadtree, which was mentioned in Chapter 7. But implementing such an algorithm is

beyond the scope of this book, so for simplicity the code here sticks to the brute-force system. However, if *__Defense* were to be a commercial game, we likely would want to eliminate the brute-force algorithm for performance reasons.

The `Enemy` game object has some custom behavior that allows it to move along a given path through the world. It has access to a linked list of nodes representing the path that's generated by the `Pathfinder`. In the `Update` function, it uses linear interpolation to move between two nodes; once it reaches a node, it then continues to the next node on the path. The facing of the enemy is computed using quaternions in the `SetDirection` function, and a couple of assorted functions allow for enemies being snared (as in Figure 14.2) or killed.

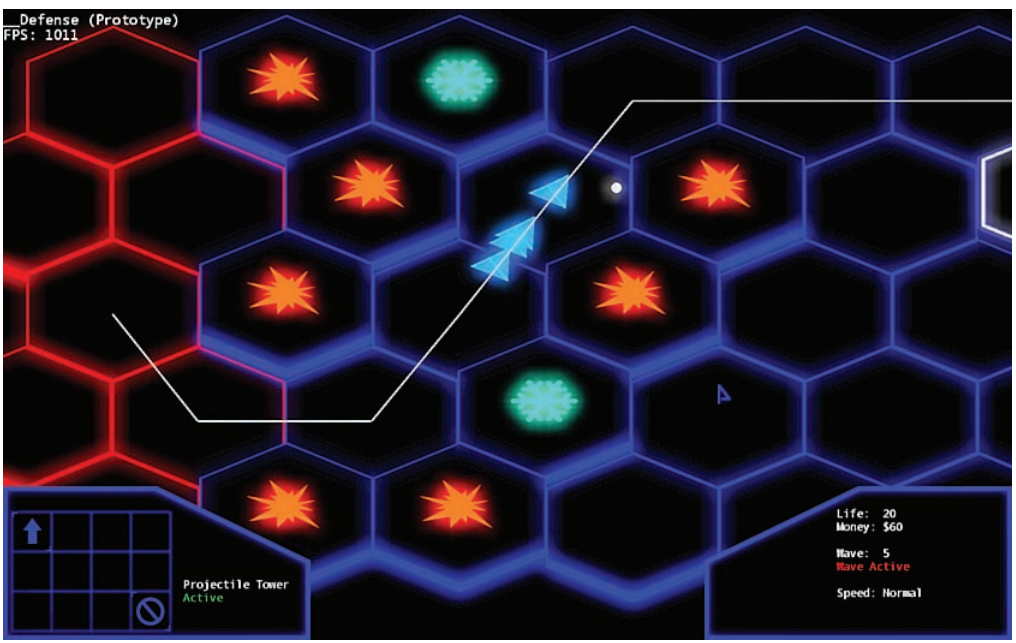


Figure 14.2 Enemies in blue are snared.

Level

The level data is all processed in Level.cs. Right now, the level layout is hard-coded in the `TileData` array that's in the `LoadLevel` function. This is definitely not ideal, but it would have been an unnecessary complexity for this game to actually load in the data from a file. The rest of the `LoadLevel` function basically loops through the array and places the tiles in the correct spots. Because the tiles are hexagons, the calculations for the center position are a bit more complex than they would be for square tiles. There is one other function here, called `Intersects`, that will be discussed when we cover the physics of the game.

Timer

The `Timer` class (in Utils/Timer.cs) is a utility class that allows for registration of functions that get called after a period of time. Because each game object has an instance of a `Timer` class, it becomes a handy way to have delayed behavior. Rather than needing to add many floating point variables to track how much time is remaining to call a particular function, you can simply add a timer, like so:

```
m_Timer.AddTimer("HideError", duration, ClearErrorMessage, false);
```

The first parameter for the `AddTimer` function is just a unique name for the timer, the second is how long until the timer fires, the third is the name of the function to call, and the last parameter is whether or not the timer is looping. For example, it's possible to set up a function to be called once per second using the `Timer` class.

Pathfinding

Pathfinding (in the appropriately named Pathfinder.cs) is implemented using the A* algorithm (which is discussed in Chapter 9). Because enemies always spawn from the same location, the pathfinding code does not recalculate the path for every enemy. Instead, the `Pathfinder` has a global path that it saves as the optimal path for new enemies to take. This optimal path is shown in game as the white line segments.

In tower defense, it's important to not allow the player to fully block off the path of the enemies. If there were no path, the enemies would just get stuck as soon as they spawned, and the game would not be challenging in the slightest. To solve this problem, before a new tower is built, `GameState` checks with the `Pathfinder` to make sure that the new tower will not make the path unsolvable. If the path would become unsolvable, the build request is denied and the game shows an appropriate error message to the player. If the path is solved, the game builds the tower and sets the global path to the new path. Figure 14.3 shows a location where tower placement would be denied because it fully blocks the path.

If the tower location is valid, the global path will change. But this new tower will potentially invalidate the path of any enemy that's active at the time. In order to fix this issue, the `ComputeAStar` function can also be called on an individual enemy. So when the global path changes due to a new tower, any active enemies will recompute their new best path. It is possible that some enemies might end up fully blocked by the tower being built, but that won't trivialize the entire game, so the system doesn't worry about it. It turns out it's actually possible to get the game into an unwinnable state if you box in an enemy so that it's only in range of snare towers and can never die, but you really have to try to cause this bug to occur.

If you look at `ComputeAStar` closely, you'll notice that it actually searches for the path *from* the goal node *to* the start node. That's because if the path were computed the other way, the result would be a linked list of nodes from the end goal back to the start goal, which isn't what the enemies need. So to eliminate the need to reverse the path, it's instead computed backward.

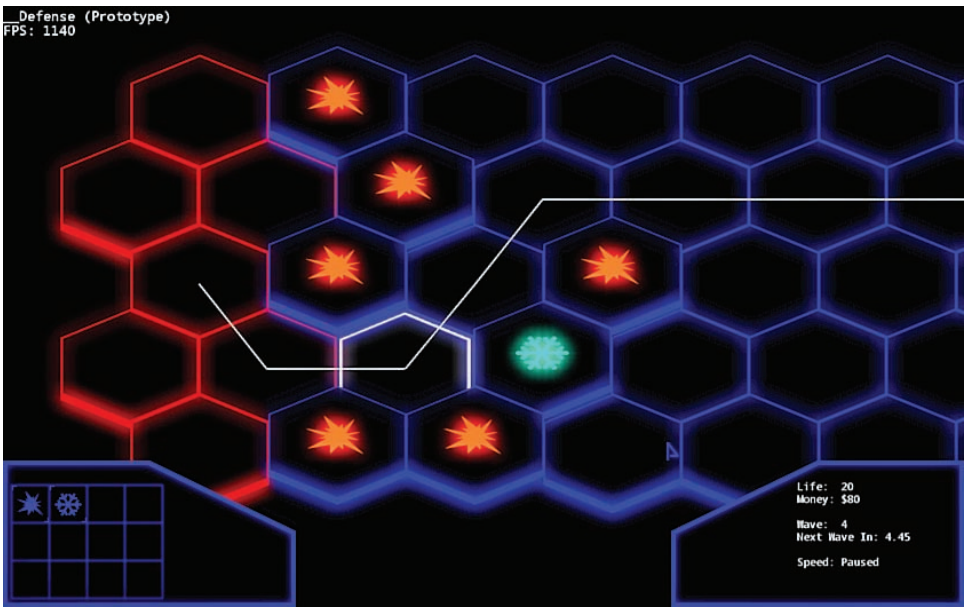


Figure 14.3 The game won't allow building a tower on the selected tile (outlined in white) because it blocks the path.

Camera and Projection

The game uses an orthographic projection, which was covered in Chapter 4. This means that there is no sense of depth because objects further away from the camera are not smaller. As for the camera (in Camera.cs), it's fairly simple in this game. There really is not much functionality beyond the ability to pan the camera around the level. In order to pan, both the eye position and the target position of the camera are changed by the same amount, which is passed to the camera via the `AddToPan` function. The camera can also zoom in and out if the mouse scroll wheel is used. Finally, there's also some code for animating the camera to a specific location, though it's currently not in use.

This game also uses mouse picking (covered in Chapter 8, "Cameras") in order to enable the player to click and select a tile. Even though picking is a camera-related algorithm, the code for this is in `InputManager.CalculateMouseRay`. In order to perform picking, you need to know both the position of the mouse in screen space as well as the camera matrix. So the choice was to either pass in the position of the mouse to the camera or have the input manager access the camera matrix. Because the drawing code also needs the camera matrix, and the camera matrix was already globally accessible through `GameState`, the picking code ended up in `InputManager`.

Input

Everything related to the keyboard and mouse input is in InputManager.cs. The keyboard binding system is very similar to the "more complex system" discussed in Chapter 5, "Input." In the `InitializeBindings` function, an abstract action such as `Pan_Left` is bound to a specific key and a specific state of that key (either just pressed, just released, or held).

Then every frame, the `UpdateKeyboard` function is called, which makes a linked list of all the actions that are active on that particular frame. This list is first passed to the UI, which can process any actions it cares about. The remaining actions are then sent to `GameState` for processing. Right now, the only actions `GameState` processes are the bindings related to panning because it has access to the `Camera` class.

The mouse is handled separately from the keyboard (in `UpdateMouse`), but with the same idea. A mouse click is first sent to the UI (to see if a button was clicked), and then if the UI does not care about the click, it's sent to the `GameState`. The `GameState` checks the mouse clicks to select the appropriate `Tile` when one is clicked. Because it's also possible to pan with the mouse (instead of using the keyboard), `UpdateMouse` also does some checks to see if the mouse is on the edge of the screen, which signifies a pan must occur.

Physics

Physics is only really used in this game for the purpose of collision detection. Due to this, all the `PhysicsManager` does is generate a model space bounding sphere or bounding box for a particular model. Because the model space bounding data will never change for the same model, it caches it in a sorted list. So every time a particular mesh (such as "Miner") is requested, it checks to see if it already previously computed the bounding sphere or box. This way, every time an enemy spawns, there's no need to recalculate the model space bounds.

In terms of actual collision detection, it's really only used to check the picking ray against all of the tiles in the world. XNA provides built-in `Ray`, `BoundingSphere`, and `BoundingBox` (AABB) classes. All of these classes can check for intersections with each other, so I did not have to implement any of the intersection tests that were covered in Chapter 7. The intersection between the picking ray and the tiles occurs in the `Level.Intersects` function. This function basically creates a list of all the tiles that intersect with the ray and then selects the tile that's closest to the start position of the ray.

Localization

As mentioned in Chapter 10, "User Interfaces," it's typically not a good idea to hard-code text strings that are displayed onscreen because it makes it more complex to translate the game to other languages. For this game, all text strings that are displayed onscreen (other than the

debug FPS) are stored in the Languages/en_us.xml file. If you open up the file, you'll see the format is relatively simple, with the data for each entry stored inside `<text>` tags, like this:

```
<text id='ui_err_money'>You don't have enough money.</text>
```

This string can then be grabbed by its ID in code:

```
Localization.Get().Text("ui_err_money")
```

The code for the `Localization` class is not terribly complex. On load, it parses in the XML and then populates a dictionary with the key/value pairs. The `Text` function then just performs a lookup into the dictionary. With this system, it would be relatively easy to translate the game to another language by loading in a different XML file.

Graphics

The `GraphicsManager` is responsible for most of the rendering logic. When it's first initialized, if the game is set to full-screen mode, it determines the current desktop resolution and sets the game's resolution to that. Otherwise, it uses the resolution that's specified in `GlobalDefines`. But the most important function of the `GraphicsManager` class is to handle rendering of all the game objects.

A game object can specify in its constructor which draw order group it's in. There are three possibilities: background, default, and foreground. Each draw order group has its own list in `GraphicsManager`, so the game objects are added to the appropriate list when they are added to the world. The purpose of the different groups is to allow for specific objects to always be in front of or behind the objects that are in the default group (which is nearly all objects).

Drawing normally is done with z-buffering, which if you remember from Chapter 4 means that it only draws a particular pixel if there's nothing in front of it that has already been drawn. However, the draw function in `GraphicsManager` only enables z-buffering for the default group. First, z-buffering is disabled and the background group is drawn. This group can be used for a **skybox**, foe example, which is a cube that encompasses the entire world and has a texture of the sky (or space). Then z-buffering is enabled and the default group, which contains the vast majority of game objects, is drawn. Finally, z-buffering is disabled once more and the foreground objects are drawn. This group allows for objects such as a first-person model to be drawn without worrying about clipping into other objects.

There's some other functionality in `GraphicsManager`, but it's not as critical. You'll find a few small helper functions for things such as drawing 2D and 3D lines as well as drawing a filled rectangle. Finally, the bloom effect on the XNA version causes the tiles to glow. The effect is

implemented as per a Microsoft sample (http://xbox.create.msdn.com/en-US/education/catalog/sample/bloom). However, the details of how the bloom effect and multiple render targets work is beyond the scope of this book, and it's not too important anyway because it's just a graphical enhancement.

Sound

Because this game only has a handful of 2D sounds, the `SoundManager` class is extraordinarily simple. All it does is load in WAV files into a dictionary, where the key is the cue name, and the value is the wave file. Then sounds can be played anywhere with the `PlaySoundCue` function:

```
SoundManager.Get().PlaySoundCue("Alarm");
```

This isn't a fabulous implementation, however. Most notably, sounds will not pause when the game is paused. Because of this, you would never want to actually release a game with a sound system like the one here. But it works on a very basic level.

User Interface

The UI system ended up being rather complex for this game, most notably because of the tooltips. But before we look at the tooltip code, let's look at the basic layout of the UI system. There is a `UIScreen` class (in UI/UIScreen.cs) that represents a particular menu or HUD. So, for example, the main menu is in `UIMainMenu` while the HUD that's shown during gameplay is in `UIGameplay`. The system supports multiple `UIScreen`s visible at once, which is handled by the UI stack in `GameState`. So, for example, the `UIGameplay` and `UIPauseMenu` screens are both visible at the same time when the game is paused. Only the top-most screen on the stack receives any input, however, which prevents actions such as clicking on buttons in the build grid while the game is paused.

`UIScreen`s can have any number of buttons (UI/Button.cs) on them. Every frame, the active `UIScreen` receives the location of the mouse cursor and can highlight a button as appropriate. If a click occurs, it will then call whatever function is registered to be called when that particular `Button` is clicked. Buttons can be initialized in three different ways: They can be text-only buttons, image-only buttons, or buttons that are "hot zones" (meaning they show tooltips, but can't be clicked). For text-only and image-only buttons, the constructor requires specifying a callback function that gets called when the button is clicked. Right now, the menu system does not support keyboard navigation because it was designed for a PC with a mouse, though it's certainly a feature that could be added.

Although the `UIScreen` system isn't that complicated, the tooltip system ended up being much more complex. There were a few reasons it ended up that way. First of all, by default, text rendering in XNA cannot wrap to a specific width. You give it a string and it'll simply draw that string until the text ends. For a tooltip, it's important that it wraps around nicely to a specific

maximum width. Second, I wanted to be able to arbitrarily change the font and color in the middle of a text string to help certain information pop out on the tooltips. This is also something that XNA does not provide by default. To solve these two problems, I ended up using the extremely useful `MarkupTextEngine`, which was released by "astroboid" under the Microsoft Permissive License (http://astroboid.com/2011/06/markup-text-rendering-in-xna.html). It worked marvelously to solve those two problems, though the code for it might be a bit difficult to understand.

That wasn't the end of the tooltip complexity, though. I also wanted to be able to dynamically change the text inside of tooltips. I didn't want the tooltip text to just be fixed (as in "This costs $500") because then whenever any values were changed in Balance.cs, the tooltip text would have to change also. Instead, I wanted to be able to replace placeholders in the tooltips with actual values. So instead of "This costs $500," the tooltip string would be "This costs ${0}," with the {0} getting replaced with the correct value. And to even make it more complex, I wanted to be able to colorize things depending on the situation. So if a building costs $500, but you don't have $500, I wanted the text to be red so it pops out that there isn't enough money to purchase the building. The end result of the tooltip system is shown in Figure 14.4.

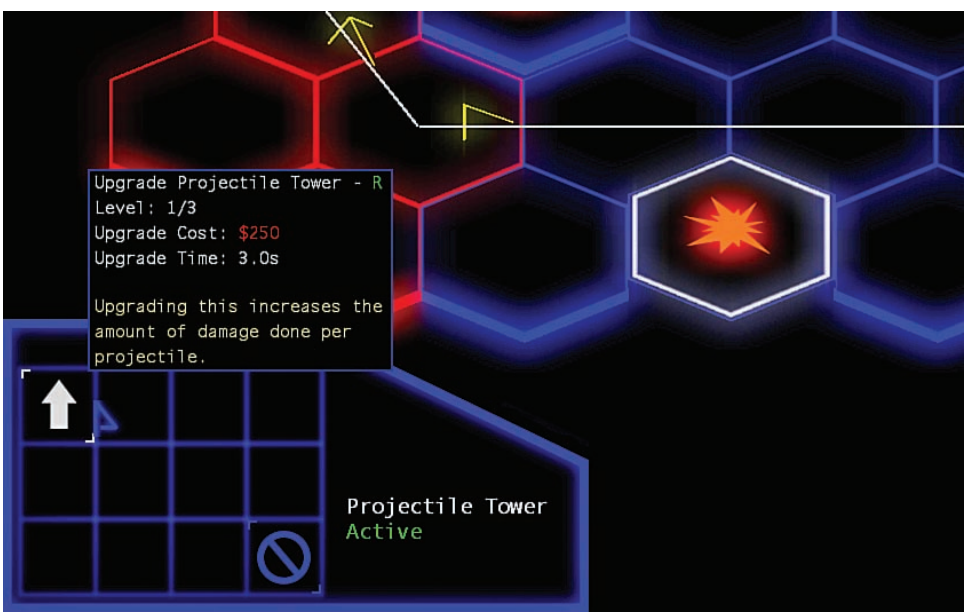


Figure 14.4 A tooltip in action.

This is where the `Tooltip` class comes into play. To initialize a tooltip, you provide the key to the localized string, the screen space position of where the bottom-left corner of the tooltip should be, and a list of `TipData`. The `TipData` list is what gives the `Tooltip` class some

information on what exactly it needs to replace the placeholder values with. This is so it knows that, for example, {0} refers to the monetary cost of the tower. This information is what all of the `CreateTowerBuildCost` and similar functions generate.

Where it gets a bit messy is in the `InitializeGrid` function in `UIGameplay`. The grid is the series of buttons in the bottom-left part of the HUD, which are the buttons that allow you to build all the different towers and such. Because there are multiple grid buttons, and some complex tooltips for said grid buttons, the function to create the tooltips ends up being pretty long. If the game had a lot more grid buttons, it would be very impractical to initialize the buttons in this manner. This ideally would be something that's also stored in an external data file. But for this game, it was simpler to implement in the manner it was done.

Exercises

Now that we've reviewed the code for *__Defense*, here are some ideas on what could be done to improve the game.

1. **Add More Waves/Rebalance the Game**

   One idea would be to add more waves and make the game more challenging. All of the data related to the game's balance is in Balance.cs. The information about the number and difficulty of waves is stored in the `Waves` array. Adding more waves is just a matter of adding more elements to the array and increasing the `TotalWaves` variable. To change the difficulty or add more levels of enemies, modify the `Enemies` array.

2. **Add a New Tower**

   Add a new "lightning" tower that chains between multiple targets, decreasing the damage as it jumps between chains. To get a new tower functioning on a basic level will require touching several files. First, you need to add a new `LightningTower` class and add an entry to the `eTowerType` enum. The chain effect requires finding the nearest enemy within range (much like the `ProjectileTower` does), but once it finds the first enemy, it needs to continue and find the next enemy in the chain. Then, `GameState.TryBuild` needs to be updated to handle the new tower type. Finally, the grid needs to be updated in `UIGameplay` so that it's possible to build the new tower (it would also be a good idea to add a keybind for this, as well).

   Once the new tower is functional, there are a few things left to make it pretty. You'll want to add new custom tooltip text in en_us.xml, and also create a couple of new textures. Take a look at the textures in XNAContent/Buildings and XNAContent/UI to get an idea of what the textures should look like. You need to reference these new textures in `LightingTower` and `UIGameplay`, respectively. Remember that if you do make new textures, only the XNA project will actually build them. So even if you want to use them in a MonoGame project, they first must be built in the XNA one.

3. **Create an External Balance File**

   Although the Balance.cs file works, it would be better if all of that data was stored in an external file. Create a JSON file that stores all of the balance information, and then load it in when the game is loaded. To serialize/deserialize the JSON, the most popular library in C# is JSON.NET. Once you get this working, you could move even more of the data (such as level data or the button grid) into separate JSON files.

Summary

The sample game in this chapter was a culmination of many of the algorithms and techniques discussed in the earlier chapters. A game object model was implemented (as in Chapter 1, "Game Programming Overview"), and the objects construct their world transform matrix from a scale, rotation, and translation (as in Chapter 4). Objects have AABBs (from Chapter 7) and are collided against with a ray for mouse picking (as in Chapter 8). A* pathfinding is used to determine the paths of the enemies as they travel toward the home base (as in Chapter 9). Finally, both the input and UI systems implemented for *__Defense* are very similar to the systems discussed in Chapters 5 and 10, respectively. Hopefully, the discussion in this chapter has given you a sense of how you can make a more complex game now that you've completed all the chapters of this book.

This page intentionally left blank

APPENDIX A

ANSWERS TO REVIEW QUESTIONS

This appendix presents you with the answers to the review questions at the end of Chapters 1–12.

Chapter 1: Game Programming Overview

1. The early consoles were programmed in assembly language because they had an extraordinarily small amount of memory and processing power. A high-level language would have added too much overhead, especially because compilers at the time were not nearly as optimized as they are today. Furthermore, the early consoles typically did not have development tools that would have been able to facilitate high-level language usage, even if it were feasible.
2. Middleware is a code library written by a third party to solve a specific problem in games. For example, Havok is a physics library that many game companies use in lieu of implementing their own physics systems.
3. There are many possible answers to this question. Here is one:

   For *Galaga*, there are two inputs to process: the joystick, which allows the player's ship to move left or right, and the fire button. The majority of the work in the "update world" phase of the game loop would be to spawn and then simulate the AI for the enemy ship squads that come in. Furthermore, the game needs to detect collisions between the enemy projectiles and the player, as well as detect the player projectiles against the enemy. The game needs to keep track of how many waves have gone by and increment levels as appropriate. The only outputs for this game are video and audio.
4. Other outputs in a traditional game loop might include sound, force feedback, and networking data.
5. If the rendering portion takes roughly 30ms, and updating the world takes 20ms, a traditional game loop would take 50ms per frame. However, if the rendering is split into its own separate thread, it can then be completed in parallel with the world updates. In this case, the overall time per frame should be reduced to 30ms.
6. Input lag is the time delay between pressing a button and seeing the result of that button press onscreen. The multithreaded game loop increases input lag because the rendering must be one frame behind the gameplay, which adds one additional frame of lag.
7. Real time is the amount of time that has elapsed in the real world, whereas game time measures time elapsed in the game world. There are many instances where game time might diverge—for instance "bullet time" slow-mo would have game time be slower than real time.
8. Here is the code as a function of delta time:

```
position.x += 90 * deltaTime
position.y += 210 * deltaTime
```

9. To force a 30 FPS frame rate, at the end of each iteration of the loop, check how much time has elapsed for the iteration. If it's less than 33.3ms, then the code should wait

until 33.3ms has elapsed before continuing to the next iteration. If the elapsed time is greater than 33.3ms, then one option is to try to skip rendering on the next frame, in an attempt to catch up.

10. The three primary categories are objects that are drawn and updated, objects that are only updated, and objects that are only drawn. Objects that are drawn and updated covers most objects in the game, including the player, characters, enemies, and so on. Objects that are only updated include the camera and invisible design triggers. Objects that are only drawn include static meshes, such as trees in the background.

Chapter 2: 2D Graphics

1. The electron gun starts aiming at the top-left corner of the screen and draws across one scan line. It then aims down one line and starts at the left again, repeating this process until all the scan lines have been drawn. VBLANK is the amount of time it takes for the electron gun to reposition its aim from the bottom-right corner all the way back to the top-left corner.
2. Screen tearing occurs when the graphical data changes while the electron gun is drawing one frame. The best way to avoid screen tearing is to use double buffering, and to only swap buffers during VBLANK.
3. If a single-color buffer is used, it means that in order to prevent screen tearing, all rendering must be completed during the relatively short VBLANK period. But with double-buffering, the only work that must be completed during VBLANK is the swapping of buffers, and there is much more time to render the scene.
4. The painter's algorithm is a rendering technique where the scene is rendered back to front. This is similar to the way a painter might paint a canvas. The painter's algorithm works relatively well for 2D scenes.
5. Sprite sheets can net huge memory savings because it is possible to pack in sprites more tightly than it might be with individual sprite files. There also can be performance gains from the fact that with sprite sheets, source textures do not have to be changed as frequently in the hardware.
6. In single-direction scrolling, the camera's position should update when the player has advanced past the midpoint of the screen.
7. A doubly linked list would work best, because it would be possible to point to both the previous background segment and the next background segment.
8. A tile set contains all the possible images that can be used for the tiles. Each tile in the tile set has a corresponding ID. The tile map, on the other hand, lists out the IDs of the tile layout of a particular scene or level.
9. By storing the animation FPS as a member variable, it makes it possible to alter the speed of an animation as it is playing. For example, a run animation could start playing more rapidly as the character moves more quickly.

10. An isometric view displays the world from a slight angle, so there is a greater sense of depth.

Chapter 3: Linear Algebra for Games

1. Solutions:

a. $\langle 5,\ 9,\ 13 \rangle$

b. $\langle 4,\ 8,\ 12 \rangle$

c. $\vec{c} = \vec{a} \times \vec{b}$

$c_x = a_y b_z - a_z b_y = 4 \cdot 7 - 6 \cdot 5 = -2$

$c_y = a_z b_x - a_x b_z = 6 \cdot 3 - 2 \cdot 7 = 4$

$c_z = a_x b_y - a_y b_x = 2 \cdot 5 - 4 \cdot 3 = -2$

$\vec{c} = \langle -2,\ 4,\ -2 \rangle$

2. If you only care about the direction of a vector, it is probably worthwhile to normalize this vector. If you need both the direction and magnitude of the vector, you should not normalize.

3. $\left|\left|A - P\right|\right|^2 < \left|\left|B - P\right|\right|^2$

4. Project $\vec{q}$ onto $\hat{r}$ to get $\vec{r} = \hat{r}\left(\hat{r} \cdot \vec{q}\right)$, which extends all the way to the start of $\vec{s}$. The vector $\vec{s}$ goes from $\vec{r}$ to $\vec{q}$, so $\vec{s} = \vec{q} - \vec{r}$.

Substitute for $\vec{r}$ to get the final solution: $\vec{s} = \vec{q} - \hat{r}\left(\hat{r} \cdot \vec{q}\right)$.

5. Solution:

$\vec{u} = B - A = \langle 2,\ 0,\ 1 \rangle$

$\vec{v} = C - A = \langle 3,\ 2,\ 1 \rangle$

Normalize $\vec{u}$ and $\vec{v}$.

$\hat{u} \approx \langle 0.89,\ 0,\ 0.45 \rangle$

$\hat{v} \approx \langle 0.80,\ 0.53,\ 0.26 \rangle$

Solve for θ using the dot product.

$\theta = \arccos\left(\hat{u} \cdot \hat{v}\right) \approx 0.59 \text{ rad } \textit{or } 34^\circ$.

6. This is $\hat{v} \times \hat{u} \approx \langle 0.23,\ -0.12,\ -0.47 \rangle$.

You should normalize this, for a final value of $\langle 0.43,\ -0.23, -0.86 \rangle$.

7. This will return the normal that faces in the opposite direction.
8. Take the player's forward vector and cross it with the vector from the player to the sound effect. If the z value of the cross is positive, the sound should come out of the left speaker. Otherwise, it should come out of the right.
9. Result:

$$\begin{bmatrix} 2 & 4 & 6 \\ 8 & 10 & 12 \\ 14 & 16 & 18 \end{bmatrix}$$

10. When the matrix is orthonormal.

Chapter 4: 3D Graphics

1. Triangles are the simplest type of polygon and can be represented with only three vertices. Furthermore, they are guaranteed to lie on a single plane.
2. The four primary coordinate spaces in the rendering pipeline are model space, world space, view/camera space, and projection space. Model space is the coordinate space relative to the origin point of the model (which is often set in a modeling program). World space is the coordinate system of the level, and all objects in the world are positioned relative to the world space origin. View space is the world seen through the eyes of the camera. Projection space is the coordinate space where the 3D view has been flattened into a 2D image.
3. The order of multiplication should be rotation matrix multiplied by the translation matrix:

$$\begin{aligned} &RotateZ\left(45^\circ\right) \times Translate\left(2,\ 4,\ 6\right) \\ &= \begin{bmatrix} \cos 45^\circ & -\sin 45^\circ & 0 & 0 \\ \sin 45^\circ & \cos 45^\circ & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 2 & 4 & 6 & 1 \end{bmatrix} \\ &\approx \begin{bmatrix} 0.71 & -0.71 & 0 & 0 \\ 0.71 & 0.71 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 2 & 4 & 6 & 1 \end{bmatrix} \\ &\approx \begin{bmatrix} 0.71 & -0.71 & 0 & 0 \\ 0.71 & 0.71 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 2 & 4 & 6 & 1 \end{bmatrix} \end{aligned}$$

4. An orthographic projection doesn't have depth perception, which means objects further away from the camera are not smaller than objects closer to the camera. On the other hand, a perspective projection does have true 3D depth perception.
5. Ambient and directional lights both globally affect the scene. However, ambient light is uniformly applied to the entire scene. On the other hand, directional lights come from one specific direction, and because of this they will not illuminate objects on all sides.
6. The three components of the Phong reflection model are ambient, diffuse, and specular. The ambient component uniformly applies to the entire object. The diffuse component depends on the position of lights in relation to the surface, and is the primary reflection of light. The specular components are the shiny highlights on the surface, and depend on both the light position and the position of the camera. Phong reflection is considered a local lighting model because each object is lit as if it were the only object in the scene.
7. Gouraud shading is per-vertex, whereas Phong shading is per-pixel. Phong shading looks much better on low-poly models because of this, but it also ends up being more computationally expensive.
8. The biggest issue with using the painter's algorithm in a 3D scene is that the process of sorting the scene back to front is time consuming, especially when given the fact that the camera positions can change dramatically. It also can potentially cause too much overdraw, which is when a pixel is drawn over during the process of rendering a frame. Finally, overlapping triangles may have to be broken down into smaller triangles for the painter's algorithm. Z-buffering solves these problems by doing a depth check at each pixel prior to drawing it—if there is a pixel already drawn with a closer depth than the current pixel, the current pixel does not get drawn. This allows the scene to be drawn in an arbitrary order.
9. Euler rotations only work on the coordinate axes, whereas quaternions can rotate about any arbitrary axis. Euler angles can also suffer from gimbal lock, and they are difficult to interpolate between.
10. Solve for the vector and scalar components:

$$q_v = \langle 1,\ 0,\ 0 \rangle \cdot \sin\frac{90^\circ}{2} \approx \langle 0.71,\ 0,\ 0 \rangle$$

$$q_s = \cos\frac{90^\circ}{2} \approx 0.71$$

Chapter 5: Input

1. A digital device is one that is binary—it can be either in an "on" state or an "off" state. One example of a digital device is a key on the keyboard. An analog device, on the

other hand, is one that has a range of values. An example of an analog device is a joystick.

2. Chords involve pressing multiple buttons at the same time in order to get a particular action to occur, whereas sequences are a series of button presses.
3. Without "just pressed" and "just released" events, button presses will be detected over the course of multiple frames. For example, if you press the spacebar quickly, at 60 FPS it is likely the spacebar will be detected as pressed for several frames, which in turn will trigger the spacebar's action for multiple consecutive frames. On the other hand, if only "just pressed" is used, only the first frame where the spacebar is pressed will trigger the action.
4. Analog filtering attempts to solve the problem of spurious, low-value inputs. For example, with a joystick it is rare to get precisely (0,0) when the joystick is at rest. Instead, a range of values around zero will be detected. Filtering will make sure that the range of values around zero are considered equivalent to zero input.
5. With a polling system, there is a possibility that different points in the code will get different results if a key press changes in the middle of the frame. Furthermore, a polling system may unnecessarily query the same button multiple times in multiple places. This can lead to code duplication. An event system, on the other hand, tries to centralize all the input so that interested parties can register with the centralized system. This also guarantees that a particular key will have only one value on one frame.
6. As long as the programming language supports storing a function (or a pointer to a function) in a variable, it is possible to simply store a list of all functions registered to a particular event.
7. One method that can ensure the UI has the first opportunity to process events is to first pass the container of triggered events to the UI. The UI can then respond to any events it chooses to, and then remove those events from the container. The remaining events are then passed to the game world code.
8. The Rubine algorithm is a pen-based gesture-recognition algorithm designed to recognize user-drawn gestures. The algorithm has a two-step process. First, a library of known gestures is created that stores several mathematical features of the gesture. Then, as the user draws a gesture, the mathematical properties of that gesture are compared against the library to determine if there is a match.
9. An accelerometer measures acceleration along the primary coordinate axes, whereas a gyroscope measures rotation about said axes.
10. An augmented reality game might use the camera in order to "augment" the world, or alternatively use the GPS in order to detect nearby players or obstacles.

Chapter 6: Sound

1. Source sounds are the actual audio files created by a sound designer. These are usually WAV files for sound effects and some sort of compressed format for longer tracks. The metadata describes how and in what context the source sound files should be played.
2. Switchable sound cues allow you to have different sets of sounds that play in different circumstances. For example, footsteps sound differently depending on the surface the character is walking on. With a switch, it would be possible to switch between different footstep source sounds, depending on the surface.
3. The listener is a virtual microphone that picks up all the sound that plays in the world, whereas the emitter is the object that actually emits the sound. In a first-person shooter (FPS), the listener might be at the camera, while an emitter might be tied to every non-player character (NPC), as well as other objects that may give off sound (such as a fireplace).
4. For a third-person game, both the position and orientation of the listener are very important. The orientation should always be camera relative, not player relative. If it's player relative, you could have a scenario where an explosion that occurs on the right part of the screen comes out of the left speaker, which is not desirable. The position of the listener is often placed somewhere between the camera and the player position.
5. The decibel scale is a logarithmic scale. Going from 0 to -3 dB cuts the volume of the sound roughly in half.
6. Digital signal processing involves taking a signal and transforming it in some way. For audio, this is often used to alter a sound on playback. One example is reverb, which creates an echo. Another example would be a pitch shift, which increases or decreases the pitch of the sound. Lastly, a compressor would normalize the volume levels by increasing the volume of quieter sounds and decreasing the volume of louder sounds.
7. Oftentimes, a DSP effect such as a reverb only needs to affect part of the level. Therefore, it is valuable to be able to mark which regions within the level are affected by DSP.
8. Using a convex polygon to mark DSP might not work if the level has some parts that are above or below other parts. For example, if a field has a tunnel running under it, a convex polygon surrounding the tunnel would incorrectly also affect when the player runs above the tunnel.
9. The Doppler effect occurs when a sound emitter is quickly moving toward or away from you. As it moves toward you, the pitch increases, and as it moves away, the pitch decreases. This is most commonly seen with sirens on emergency vehicles.
10. Sound occlusion is when the sound waves must go through a surface to get to the listener. This results in a low-pass filter effect, which means that high frequency sounds become harder to hear. In obstruction, although there is no a direct path to the listener, the sound can go around the surface.

Chapter 7: Physics

1. An axis-aligned bounding box has sides that must be parallel to the coordinate axis (or coordinate planes, in the case of 3D), which makes calculations with AABBs a bit easier to compute. An oriented bounding box does not have this restriction.
2. $P \cdot \hat{n} + d = 0$
   P is an arbitrary point on the plane, $\hat{n}$ is the normal to the plane, and d is the minimum distance from the plane to the origin. In a game engine, we will typically store $\hat{n}$ and d in our plane data structure.
3. A parametric equation is one where the function is defined in terms of an arbitrary parameter (most often t).

   A ray can be represented by $R(t) = R_0 + \vec{v}t$.
4. Two spheres intersect if the distance between their centers is less than the sum of the radii. However, because the distance requires a square root, it is more efficient to check if the distance *squared* is less than the sum of the radii *squared*.
5. The best approach for 2D AABB-AABB intersection is to check for the four cases where the AABBs definitely cannot intersect.
6. In line segment vs. plane intersection, a negative t value means that the segment is facing away from the plane.
7. An instantaneous collision detection algorithm only checks if the two objects are actively colliding on the current frame, whereas a CCD algorithm will also find collisions between frames.
8. For swept sphere, if the discriminant is negative, it means the two spheres do not collide. If it's equal to zero, it means there is a t such that the two spheres tangentially intersect. Finally, if it's positive, it means the two spheres fully intersect, and the smaller solution to the quadratic is the first point of intersection.
9. The accuracy of numeric integration methods ties directly to the size of the time step. If the time step is variable, the numeric integration methods will be volatile.
10. Velocity Verlet integration first computes the velocity at the midpoint of the time step, using the acceleration from last frame. This velocity is then applied to the integration of the position over the full time step. Then the acceleration is updated for the current frame, and this acceleration is applied to the computation of the velocity from the midpoint to the end of the time step.

Chapter 8: Cameras

1. Field of view is the angle that represents what is visible in the game world. A narrow field of view can be a source of motion sickness, especially in PC games.

2. The position of the basic follow camera can be computed using the following equation:

```
eye = target.position - target.forward * hDist + target.up * vDist
```

   Once the eye position is computed, the camera forward vector is simply the vector from the eye to the target, which is as follows:

```
forward = eye - target.position
```

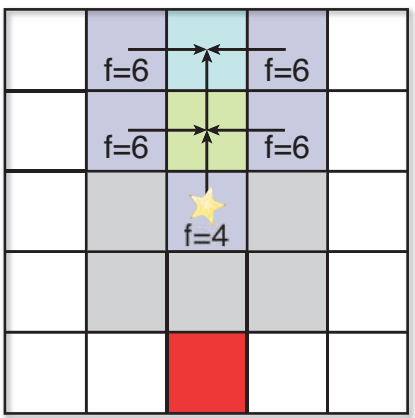
   Then the up vector of the camera can be computed by crossing the target up with the camera forward to get camera left, and then crossing camera forward with camera left to get the camera up.

3. A spring follow camera has both an ideal and actual camera position. The ideal position updates every frame based on the rigid basic follow camera, while the actual camera position lags behind a little bit. This creates a much more fluid camera experience than the rigidity of the basic follow camera.
4. The camera's position in an orbit camera should be stored as an offset from the target object, because it allows the rotations to more easily be applied.
5. The target position is stored as an offset from the camera, and as the character rotates and looks around, the target position must be rotated as well.
6. A Catmull-Rom spline is a type of spline curve that can be defined by a minimum of four control points: one before and one after the two points that are being interpolated between.
7. A spline camera can be very useful for cutscenes where the camera needs to follow a set path.
8. If a follow camera is being blocked by objects, one solution is to perform a ray cast from the target to the camera and then place the camera position at the first point of collision. Another solution is to reduce the alpha of objects that are between the camera and the target.
9. In an unprojection, a z-component of 0 corresponds to the 2D point on the near plane, whereas a value of 1 corresponds to the same point on the far plane.
10. Picking can be implemented by unprojecting the 2D point onto both the near plane and the far plane. Then, a ray cast between the two points can be performed to determine which object is intersected with.

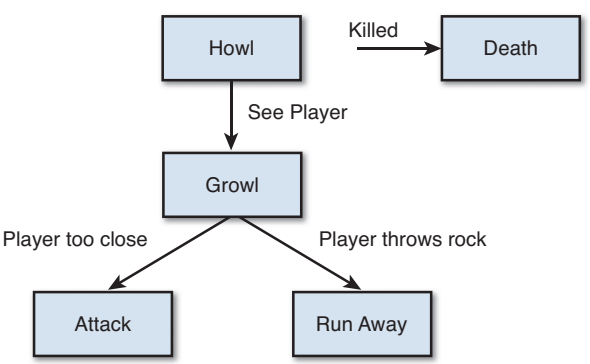
Chapter 9: Artificial Intelligence

1. Node 0: {1, 4}; Node 1: {0, 2, 4}; Node 2: {1, 3}; Node 3: {2, 4}; Node 4: {0, 1}.
2. A typical area requires fewer nodes when represented by a navigation mesh as opposed to path nodes. At the same time, you can get more coverage of the area with a navigation mesh.

3. A heuristic is considered admissible if its estimated cost is less than or equal to the actual cost.
4. Manhattan: 6. Euclidean: $2\sqrt{5}$.
5. The selected node has a star:



6. Dijkstra's algorithm.
7. When evaluating the cost of visiting a node, the A* algorithm takes into account both the actual cost from the start node and the estimated cost to the end node. Dijkstra's algorithm only looks at the actual cost from the start node. The A* algorithm is generally more efficient than Dijkstra's because it usually does not visit nearly as many nodes.
8. There are many potential solutions to this question. Here is one:



9. The state design pattern allows the behavior of one specific state to be encapsulated in its own separate class. This allows states to become reusable and/or interchangeable. It also means that the "controller" class can be far more clean than in the alternative enumeration implementation.

10. A strategy is the overall vision for an AI player. A common approach is to break down the strategy into several goals that, when prioritized, can then be achieved by the AI through discrete plans.

Chapter 10: User Interfaces

1. A menu stack provides a couple different benefits. First of all, it ensures that the user can easily return back to a previous menu by popping the current menu off of the stack. Furthermore, it allows multiple UI elements to be drawn on top of each other, such as when a dialog box pops up to confirm/reject a particular action.
2. Letter key codes are typically sequential, so the key code for C is one after the key code for B, which is one after the key code for A, and so on. This property is helpful because you can check whether a key code is within the range of A through Z and then simply subtract the key code for A from it. This gives the letter offset in the alphabet and can then be added to the character "A" to get the appropriate character.
3. Typically, there is a known 2D screen space position that a waypoint arrow should always be at. However, because the arrow is actually a 3D object, we need the correct world space position to draw it. The way to determine this is to take the 2D screen space position and unproject it, which will yield the appropriate 3D position.
4. In order to calculate the axis and angle of rotation for the waypoint arrow, we first must construct a vector from the 3D player to the waypoint. Once this vector is normalized, we can use the dot product between the new vector and the original facing vector to get the angle of rotation. Similarly, the cross product can be used to calculate the axis of rotation.
5. The aiming reticule is always at a 2D point on the screen. We take this 2D point and perform two unprojections: one at the near plane and one at the far plane. The code can then construct a ray cast between these two points, and check to see the first object this ray cast intersects with. If it's an enemy, the reticule should be red, and if it's a friendly, it should be green.
6. In order to take a 3D coordinate and convert it into a 2D one for the radar, we ignore the height component. In a y-up world, this means we create a 2D vector that uses the x-component and the z-component of the 3D vector.
7. Absolute coordinates are a specific pixel location on screen, whereas relative coordinates are relative either to a key point on the screen (such as the center or the corners) or to another UI element.
8. Hard-coding UI text makes it difficult for nonprogrammers to change the text in development. Furthermore, hard-coded text cannot be easily localized. This is why text should be stored in external text files.

9. The ASCII character set can only display English letters. The Unicode character set solves this problem in that it supports many different writing systems, including Arabic and Chinese. The most popular Unicode representation is UTF-8, which is a variable-width encoding for characters.
10. User experience refers to the reaction a user has as he or she is using an interface. A well-designed game UI should have a positive user experience.

Chapter 11: Scripting Languages and Data Formats

1. A scripting language might allow more rapid iteration of game elements, and also makes it more accessible for nonprogrammers to edit. But the disadvantage of a scripting language is that its performance is not going to be as good as a compiled language such as C++.
2. A scripting language might make sense for the camera, AI state machines, and the user interface.
3. UnrealScript is one custom scripting language that's popular for games. The advantage of a custom scripting language is that it can be tailored specifically for game use. In the case of UnrealScript, this means it supports state functionality. It also can have its performance tailored for game applications. The disadvantage, however, is that a custom language may have a lot more errors in its implementation than a general language.
4. A visual scripting system allows the use of basic flow charts to drive logic. It might be used for the logic in a particular level, such as when and where enemies spawn.
5. The statement `int xyz = myFunction();` would break down into the following tokens:

```
int
xyz
=
myFunction
(
)
;
```

6. The regular expression `[a-z][a-zA-Z]*` matches a single lowercase letter followed by zero or more lower- or uppercase letters.
7. An abstract syntax tree is a tree structure that stores the entire syntactical layout of a program. The AST is generated in syntax analysis and can then be traversed in a post-order manner either to generate or execute its code.

8. A binary file format is typically going to be both smaller and more efficient to load than a text-based one. But the problem is that it's much more difficult to determine the difference between two versions of a binary data file in comparison to a text data file.
9. For basic configuration settings, an INI file makes the most sense because it's very simple and would therefore be easy for the end user to edit.
10. The two main components of UI add-ons in *World of Warcraft* are the layout and the behavior. The layout is implemented in XML and the behavior is implemented in Lua.

Chapter 12: Networked Games

1. The Internet Protocol is the base protocol that needs to be followed in order to send any data over the Internet. The main difference between IPv4 and IPv6 is how the addresses are configured. In IPv4, addresses are a set of four bytes, for a total of about four billion possible combinations. This seemed like a lot when IPv4 was introduced, but in recent years addresses have been running out. IPv6 increases the number of address combinations to 2128, which means it will not run out of addresses any time soon.
2. ICMP, or Internet Control Messaging Protocol, is designed mostly for acquiring status information about a network and connections. One use for games, however, is the echo request/echo reply feature of ICMP. This allows games to "ping" and determine the amount of time it takes a packet to travel a round trip.
3. TCP is connection based, which means two computers have to perform a handshake before any data can be transmitted. Furthermore, TCP guarantees that all packets will be received—and received in the order they were sent.
4. A port can be thought of a virtual mailbox. Each computer has approximately 65,000 ports for TCP and 65,000 ports for UDP. Certain ports are traditionally used for certain types of data (such as 80 for HTTP servers), but there is not necessarily a strict use of a particular number. The importance of ports is that they allow multiple applications on the same computer to be using the network without interfering with each other.
5. UDP is generally more preferable for real-time games because the additional guarantees that TCP provides might cause unnecessary slowdown. A great deal of information is not mission critical and therefore should not be resent if it's lost. But with TCP, everything will be resent until an acknowledgement is received, which could substantially delay the data being processed if it's received out of order.
6. In a server/client model, all clients connect to one central server. This means that the server needs to be more powerful and have more bandwidth than the clients. In a peer-to-peer model, every computer is connected to every other computer, which means that the requirements are symmetrical.

7. The server will periodically send updates regarding what the other players in the game are doing. Client prediction tries to fill in the blocks of time in between so that the overall gameplay can be smooth. Without client prediction for movement, opposing players would teleport around the screen every time an update is received, which would not be very fun.
8. In a lockstep model, the game is broken down into turns that are 150ms–200ms in length. All input commands during that turn are queued up until the end of the turn. Each peer them simulates the result of all of said input commands. The end result is a tightly synchronized game where very little information needs to be sent.
9. One example of an information cheat would be adding a radar to a game where one does not exist. A radar could be created using the position information that's transmitted to all the clients. One way to solve this would be to have the server only send information regarding players the particular client should reasonably be able to see.
10. A man-in-the-middle attack is where a computer is placed in between the two computers that are communicating. This can be a big issue because it allows a nefarious person to intercept and read packets. The primary solution to this problem is to use an encryption scheme where the server is the only party that can read the contents of the packet.

This page intentionally left blank

APPENDIX B

USEFUL TOOLS FOR PROGRAMMERS

There are several tools that make programming less error prone and improve the overall ability of a programmer. This appendix covers some of the most valuable tools—the ones most professional developers will use on a daily basis. These tools are general enough that they aren't exclusively for game programmers, so any programmer can benefit from learning to use them.

Debugger

Novice programmers often try to debug problems in their code with text output statements, using something like `printf` in C or `cout` in C++. For a console application, this might make a lot of sense. If a user is required to type "yes" or "no" and instead enters "maybe," it's completely appropriate to have a text error message. But using print statements in real-time programs such as a game typically does not work too well.

Imagine you are trying to fix a problem with the position of a particular game object. It might be tempting to add a print statement in this object's `Update` function that outputs the position of the object every frame. But the problem with this approach is you will now have console output 60 times per second with the position of the object (assuming the game runs at 60 FPS). It would be difficult to extract meaningful information from that amount of spam—and worst of all, all that spam can slow down the frame rate.

That's not to say that print messages don't have any value in a game. If it's an error that only occurs once, such as a nonfatal inability to load a texture, it might be okay to output that as a text error. But an even better solution would be to actually show a temporary texture in-game, in hot pink or another neon color, so the problem is instantly visualized by anyone playing the game. The fact of the matter is that other team members likely are not going to pay attention to text console output, but they definitely will notice a hot pink texture.

So if print messages aren't a good way to debug a game (or most programs), what should be used? The answer is the debugger. Every single **IDE** (integrated development environment) has some sort of debugger built in. Some work better than others, but the basic principles are typically the same. This section looks at the debugger in Visual Studio 2010, but all of the concepts should transfer over to the IDE of your choosing (though, of course, the menus and key presses will be different). The screenshots specifically show the code from the tower defense game in Chapter 14, "Sample Game: Tower Defense for PC/Mac." In order to debug that game, you will need to put it into windowed mode, which can be done by setting `GlobalDefines.bFullScreen` to `false`.

Basic Breakpoint

A **breakpoint** on a line of code tells the debugger that whenever it reaches that particular line, it should pause the program. To set a breakpoint in Visual Studio, move the cursor to the line you'd like the breakpoint on and press F9 (alternatively, you can click the grey column on the left side of the editor window). When a breakpoint is hit, you can mouse over variables to see their current value. A breakpoint in the `Update` function of Enemy.cs is shown in Figure B.1.

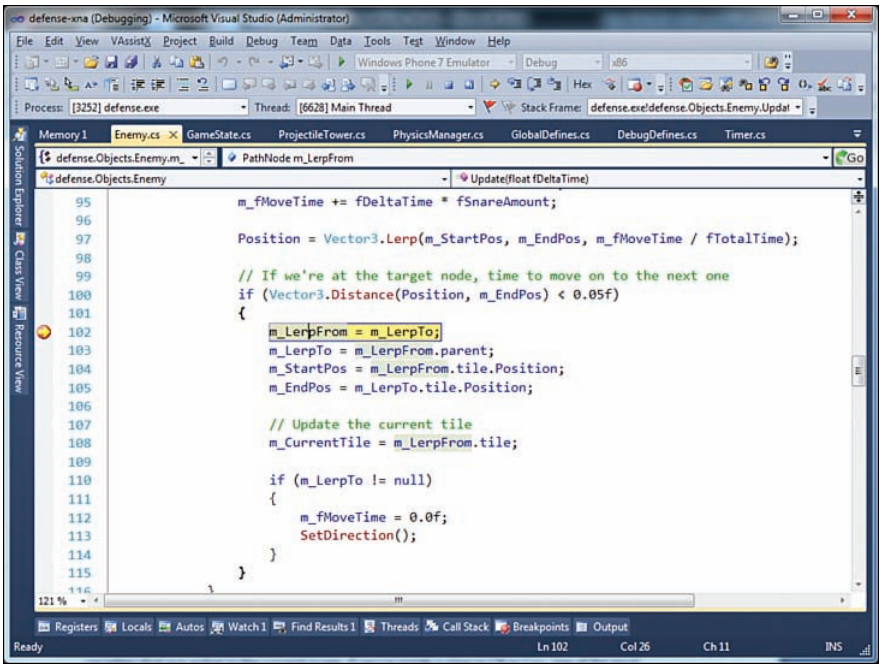


Figure B.1 A basic breakpoint in Visual Studio.

Once you hit a particular breakpoint, it's useful to be able to step through the code line by line, so that you can pinpoint exactly where the code goes wrong. There are two different ways to step through the code. If you use Step Over (by pressing F10), the code will continue to the next line *without* going into the code of any functions called on the current line. So, for example, suppose you had the following two lines of code, and the breakpoint was on the first line:

```
myFunction();
x = 5;
```

If you were to step over the first line, you would immediately jump to the second line, and only see the results of the `myFunction()` call. You wouldn't see any of the code that was executed inside of the function. If you do want to jump into a particular function and go through it line by line, you can use Step Into (by pressing F11). This will go into any functions on the line, in the order in which they are called. If you step into a function and realize you don't actually care about the line-by-line behavior of that particular function, you can jump out of it using Step Out (by pressing Shift+F11).

Watch

A **watch** allows you to track specific variables, without needing to always mouse over them to check their values. The easiest way to add a watch is to right-click a particular variable in the debugger and select Add Watch. When debugging, you will then have a watch window at the bottom of the IDE that shows the current value of the variable. If the variable is a struct or class, you may need to click the + symbol to expand it in order to see the member variables. As you step through lines of code, whenever a watch variable changes, the color of the variable will change to red. This is an easy way to eyeball what values are changing over time. A sample watch is shown in Figure B.2.

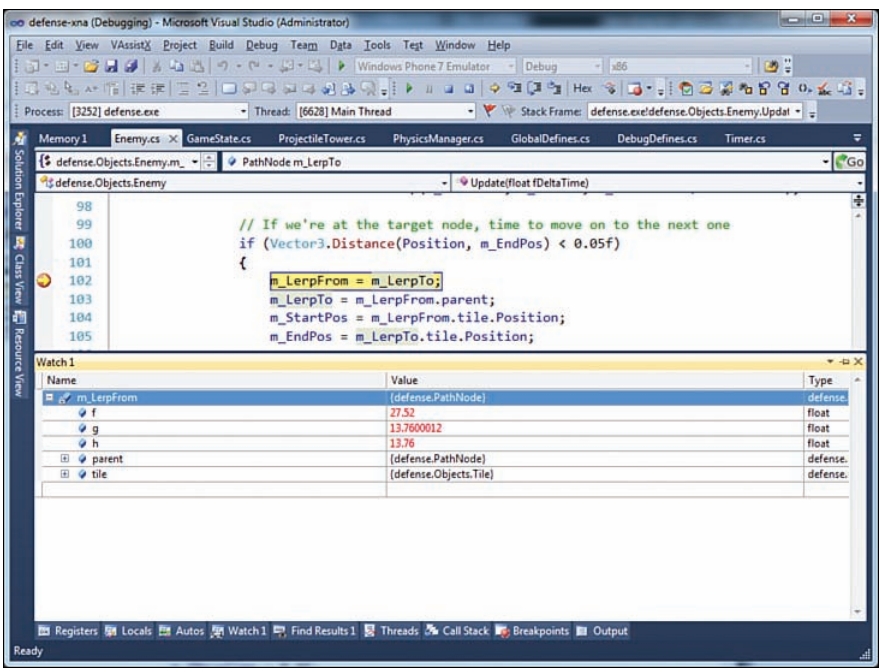


Figure B.2 Variable watch in Visual Studio.

There are also two automatic watch windows in Visual Studio: Locals and Autos. Locals refers to any variables that are active in the current scope. If you're inside a class in C# or C++, one of the local variables is always going to be the `this` pointer. The Autos watch list is a bit more restrictive, and only includes variables that are actively being processed by the lines of code relatively close to the current one.

One important thing to note is that all types of watches will work in a "Debug" build, but may not work in a "Release" build, depending on the language. That's because a Release build is optimized, and one of the optimizations is to remove any information that the debugger uses

to track variables. This is especially noticeable in C++, which has aggressive optimization routines in Release.

For AAA games, a common issue is that the Debug build is literally unplayable. If you have a game that's pushing a system to the brink, it's not uncommon to have a Debug build run at less than 5 FPS, which really doesn't allow for effective testing. One solution to this problem is to have an in-between build that has some of the debug functionality enabled, but also some of the optimizations. Another option is to run in Release, but turn off optimization for the particular file you are trying to debug. In Visual Studio, you can right-click the file in question in the Solution Explorer and turn off optimizations specifically for that file. But don't forget to change the settings back once you're done debugging!

Call Stack

The **call stack window** shows you which functions called the function that the debugger is currently in. This is especially helpful in the case of an unhandled exception or an outright crash. When you get a crash in Visual Studio, you will see a dialog box with a Break button. If you click the Break button, the debugger will pause right at the point of the crash. This allows you to inspect the values of variables as well as see what was on the call stack when the crash happened. The latter is especially important when you can't figure out why a particular function was called at a particular point.

What you definitely *should not* do when you encounter a crash is put print statements everywhere to try to pinpoint where the crash happen. This is an absolutely terrible (and often unreliable) way to find the source of a crash. You have a debugger designed to help with this problem, so you should most definitely use it.

Breakpoint Conditions

An issue with putting a breakpoint in a function that gets called single every frame is that the game will pause at that line over and over again. But suppose you only want to pause at the breakpoint when a particular condition is true, like for instance when a particular member variable is zero. This is where conditional breakpoints come into play. To add a condition to a breakpoint, right-click the red breakpoint circle and select Condition. You then will get a dialog box in which you can type the particular condition, as shown in Figure B.3.

Once the conditional breakpoint is set, you then will only stop at the line when the condition is met. This is a really handy way to get a break to only occur when you need it to. One other type of condition you can have is a **hit count condition**, which means that the breakpoint will only pause in the debugger once it has been hit the requested number of times. To use a hit count, you right-click a breakpoint and select Hit Count.

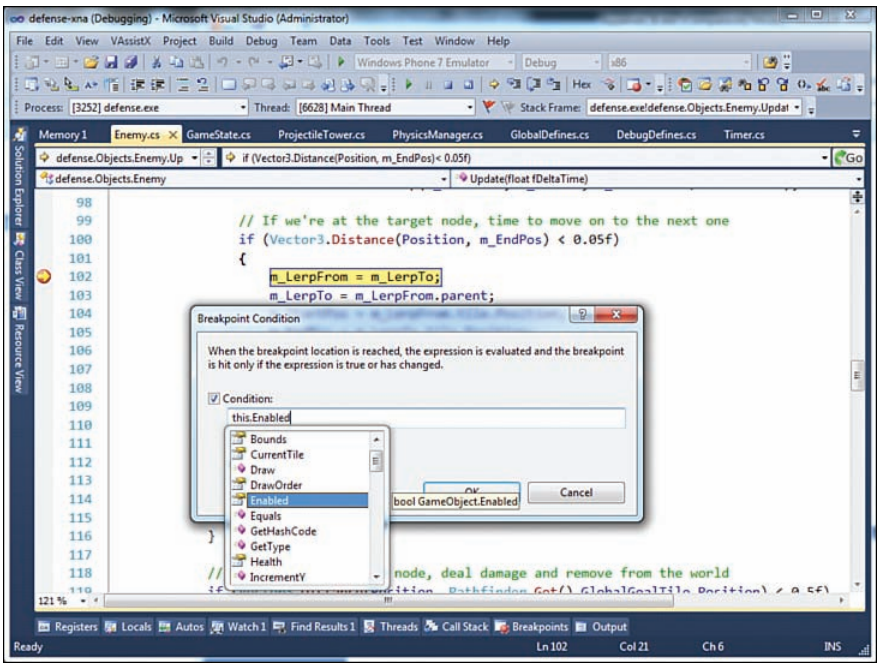


Figure B.3 Setting a conditional breakpoint in Visual Studio.

Data Breakpoint

Suppose in a C++ game you have a member variable called `m_Alive` that tracks whether the particular game object is alive or dead. While debugging, you notice that sometimes the `m_Alive` variable is changing even though the object in question never called the `Die` function to set it to `false`. This has to mean that somewhere and somehow, the memory location that stores `m_Alive` is being clobbered by bad values. This can be one of the most frustrating types of bugs to solve, because the culprit could be a bad pointer access literally anywhere else in the code.

One feature that's immensely useful to solve this problem in Visual Studio is the data breakpoint, a type of breakpoint that only works in C/C++. A **data breakpoint** tells the debugger to pause the program whenever a specific range of memory is modified. The size of the memory range typically is 4 bytes on 32-bit systems and 8 bytes on 64-bit ones, and the number of data breakpoints allowed depends on the system.

In our scenario with the `m_Alive` variable, in order to set a data breakpoint we would first put a regular breakpoint in the constructor of the game object. Once that is hit, we could then select from the menu Debug, New Breakpoint, New Data Breakpoint. For the address, we would type in `&m_Alive`, which is the address of `m_Alive` in memory. Then whenever that value changes in memory, the debugger will break, which will help us find the bad code that's modifying it.

Source Control

Imagine you are working by yourself on a simple spaceship combat game that has code spread out across ten files. There's a bug that if you pick up certain power-ups in a particular order, the ship stops being able to fire its weapons. You spend a couple hours debugging the problem (using a debugger, of course), and find the root cause in the code. You fix the bug and are happy that you solved the problem. You then move on to many other features and continue to change the files countless times.

A week later, suddenly, the bug appears again. You're perplexed. Was it that you never fixed the bug in the first place, or that one of the numerous other changes you've made since then broke the code again? Well, you have a problem remembering what you changed to fix the bug in the first place, because you have absolutely no history of when the files were changed and for what reason. Even if you kept your computer on and IDE open for the entire week, it's unlikely you can undo all the way back to the point where you fixed the bug. And anyways, you don't want to lose one week of all the other changes.

Source control provides a clear history of every single file and when it was changed. It shows all previous versions of the file that were submitted, and used properly it provides detailed descriptions of why the lines of codes were changed in each version. With source control, it would be relatively easy to go through the change history and find out the specific fix that was implemented, which will provide much insight as to why the game broke again.

Although extremely useful for even a solo developer, source control is almost mandatory for a team project. With source control, you can have a centralized location that has the latest version of the code. Team members can make sure they are always synchronized, and there is a very clear history of who changed what files and why. It also allows you to manage scenarios where multiple programmers change the same file, which is pretty much a nightmare without source control.

Numerous different types of source control systems are available, but once you become familiar with one system, it typically is not that difficult to use another. There are always going to be differences in the workflow, but the basic premise is the same. This section covers two of the most popular systems: SVN and Git. The discussion of Git is more detailed, simply because that's what's used for the code samples in Chapters 13 and 14. But that doesn't mean Git is the best solution in every scenario.

SVN

Subversion, or **SVN**, is a centralized source control system. In SVN, you first "check out" a copy of the code from the central server (called a **repository**). When you make any changes that you want to submit back to the server, you "commit" them. All operations, including checking out, committing, and updating to the latest version, must go through the central repository. The

advantage of this is that the central server will always have the latest version, but the disadvantage is that operations can be slower because they always have to travel over the network.

Because SVN requires a centralized server, you have to set one up to use this source control system. You have a few potential options for servers. If the project in question is **open source** (meaning anyone can freely access the code), several different websites will host the SVN server for you. Some popular services include Google Code (http://code.google.com) and SourceForge (http://sourceforge.net).

If you don't want to share your project's source code with the world, you'll need to have a private SVN server. On Windows, the easiest way to set up an SVN server is to use VisualSVN Server (http://www.visualsvn.com/server/). Alternatively, if you have a web host that provides shell access, you may be able to set up an SVN server through that, although it's definitely harder to do so.

Once you have an SVN server, you need a way to connect to it. Although a command-line SVN client is available, the easiest client to use for Windows is TortoiseSVN (http://tortoisesvn.net/). What's nice about TortoiseSVN is it integrates with the explorer, so you can right-click files and folders in order to perform operations on them.

The typical configuration of an SVN repository is to have three directories off of the root one: `trunk`, `branches`, and `tags`. The **trunk** is the current latest version, and is where most of the active development occurs. A **branch** is a separate copy of the code that might be used for working on a major new feature. The idea behind a branch is that the programmer tasked with the major feature can incrementally commit changes to the server without breaking the trunk for everyone. When the major new feature is tested sufficiently on the branch, it can then be merged back into the trunk. Finally, a **tag** is a copy of a specific version of the trunk. So, for example, if you release v1.0 of a program, you would want to make a v1.0 tag before continuing development. That way, the specific code for v1.0 is easily accessible, even after there has been many more changes to the trunk.

Git

Unlike SVN, **Git** is a *distributed* source control system. The distributed nature of it manifests in two main ways. First of all, you don't need to have a central server to use Git. When you install a Git client, you can create and use source control locally. This means that if you're working by yourself, operations can be faster than on a centralized source control system like SVN because you aren't ever connecting to a server over the network.

However, if a project has more than one developer, it's likely it still will need to have a central repository. This is where the other main difference between Git and SVN becomes apparent. Rather than just checking out the latest copy of the code from the central repository, in Git you **clone** the entire repository. This means you get not only the latest code, but a copy of the entire

version history of every file since the beginning of the repository. This is extremely useful for open source projects that tend to **fork**, or have slightly different versions branch off from each other. One such project that forks with a great deal of frequency is Linux, so it's not surprising that the main developer of Linux, Linus Torvalds, is also the main developer of Git.

Even if you don't plan on forking a project, a clone of the repository still has the advantage that any operations you perform will actually be on your local clone, not on the central repository. This means that actions such as committing files will be faster, and you don't necessarily have to be connected to a network to commit. But it also means that when you commit file changes, they will not automatically commit to the central server. To solve this, Git does provide functionality to "push" your commits from your local clone to the central repository and likewise "pull" changes from the central repository to the local one.

But if all you really want is just a central repository on a local network (as might happen in a professional environment), there may not really be much advantage to using Git. It has the additional overhead of needing to synchronize with the server, which if you're going to want to do anyway, you might as well use a system that's designed for that purpose.

Using Git

The source code for the games implemented in Chapters 13 and 14 is hosted on GitHub (http://github.com), which is the most popular site for hosting open source Git projects. Although you can use several different clients for Git, the easiest way to clone repositories from GitHub is to use the GitHub client. There are both Windows and Mac versions of the client, which both work roughly in the same way.

First, in order to use the GitHub client, you will need to create a GitHub account. This can just be done on the home page of their website. Once you have created an account, you can install the GitHub client for either Windows (http://windows.github.com) or Mac (http://mac.github.com). When you fire up the client for the first time, it'll ask you for the account user name and password you just created.

Once the client is set up, you're ready to use GitHub. Go to the web page for a GitHub repository you want to clone (for example, https://github.com/gamealgorithms/defense for the tower defense game in Chapter 14). Once you're on the page, on the right side you should see a button labeled either "Clone in Desktop" or "Clone in Mac." If you click that button, the Git clone procedure will kick off in the GitHub client. If this procedure worked properly, the project should be added to your list of repositories in the client.

Once a repository is cloned, you can right-click it in the client and select either Open in Explorer or Open in Finder, which will open the directory in which all the files are stored. You can then open the solution file and edit any files like you normally would.

When you're ready to commit a change to your local repository, you can double-click the repository name in the GitHub client to open a more detailed view. In this view, you will see a list of all the files that have been modified, and can also expand each entry to see what changes were specifically made to each file. Select all the files you want to commit and then type in a commit message. Your message should be as descriptive as possible, because if you type in a generic message, it makes it a lot harder to immediately determine what changed in a specific commit.

Once you click the commit button, all the selected files will then be committed to your local repository. However, if you then want to push those changes to the central repository (if you have permission to do so), you can use the sync button at the top of the window. This will automatically pull any updates first and then push your changes. Note that if you don't have permission to push your changes to the repository, you will get an error message.

The GitHub client does allow you to do other things, too, such as creating a separate branch. That, combined with the other features, is likely enough for most normal usage. But in order to perform the more complex operations Git is capable of, you will have to use another client. One such option is Atlassian SourceTree (http://www.atlassian.com/software/sourcetree/), or you can also use Git on the command line.

Diff and Merging Tools

A **diff tool** allows you to inspect the differences between two (or more) files. Many source control clients have this functionality integrated in some way, such as in the commit window in the GitHub client. The traditional diff tool was a program on UNIX that would spit out a text file that documented the differences between two other text files. Thankfully, more modern tools will actually display differences side by side in a graphical interface. When used in conjunction with source control, a diff tool can show you what was specifically changed in a file. One such open source diff tool is TortoiseMerge, which is included as part of the aforementioned TortoiseSVN.

A more complex use of such tools is to actually **merge** changes between multiple versions. Suppose you are editing GameState.cs, and at the same time another developer is also editing the file. If you commit your change to the central repository first, when the other developer tries to commit he'll find out his GameState.cs is out of date. The source control client will then require him to update the file. If you were editing different parts of the file, the client will usually be able to automatically merge the different versions.

But what happens if both you and the other developer edited the same exact line in GameState.cs? In this case, the source control tool has no way of knowing which lines it should use, and it will complain that there is a conflict. To resolve this conflict, you have to do a **three-way merge**. In a three-way merge, the typical setup has things split up into sections for "their" changes, "your" changes, and the final merged file. For each conflict area, you will

have to select which lines of code to use in the merged file. It might be theirs, yours, or some combination of the two.

Issue Tracking

Issue tracking involves documenting and then tracking the resolution of issues that pop up during development. An issue tracker is also called a bug database, though issues do not necessarily have to directly correspond to bugs. Although I wouldn't recommend issue tracking early on in development, at a certain point the majority of features of a particular game will be implemented. This causes the focus to shift from creating new code to fixing any bugs in the existing code. Once a game reaches this phase of development, issue tracking becomes almost mandatory.

The typical workflow with an issue tracker is that the QA team tests the game and generates a bug report for each unique bug they encounter. These bug reports are then forwarded to a producer or another lead developer, who then determines who would be the best person to fix each bug. This could be due to a variety of factors, including who wrote the code in the first place, as well as who has the biggest workload at that point in time. Each issue is also assigned a priority, with the highest priority being reserved for crashes and other showstoppers.

The convenient part of having an issue tracker is that each developer will have a list of tasks to go through. This gives each team member a clear direction as to what he or she should be working on, which is important. There's also some satisfaction in closing out bugs efficiently, or eventually getting down to zero bugs. Every time a bug is fixed, it's then forwarded to the QA team to verify the fix. Once the fix is verified, the issue can then be closed, though there's always the possibility of reopening a bug if it comes back.

Issue tracking also provides a way to get metrics such as how many total bugs there are, how many crash bugs there are, which developers are fixing the most bugs, and so on. So not only does issue tracking provide a clear list of tasks on an individual-by-individual basis, it also allows for large-scale statistics of how the project is progressing. If it's late in the project and there still are more bugs coming in than being fixed, it likely means the release date needs to be delayed.

Many open source hosting platforms also provide issue tracking, including GitHub. So if you encounter a bug in one of the code samples from Chapter 13 or 14, feel free to submit an issue there! But if you aren't using an open source host, you can use one of several other standalone programs, including Bugzilla (www.bugzilla.org) and Trac (http://trac.edgewall.org).

This page intentionally left blank

INDEX

Symbols

A

B

C

D

E

F

G

H

I

J–K

L

M

N

O

P

Q–R

S

V

W

X–Z