A PLATFORM-AGNOSTIC APPROACH

Game Programming
ALGORITHMS
and
TECHNIQUES

Sanjay MADHAV

FREE SAMPLE CHAPTER
SHARE WITH OTHERS
Game Programming
Algorithms and
Techniques
Game Programming Algorithms and Techniques

A Platform-Agnostic Approach

Sanjay Madhav

Addison-Wesley
To my family for supporting me and to all my students for not driving me crazy (yet).
This page intentionally left blank
Contents

Preface .................................................. xv

1  Game Programming Overview ....................... 1
   Evolution of Video Game Programming ............... 2
   The Game Loop ........................................ 5
   Time and Games ...................................... 9
   Game Objects ......................................... 13
   Summary ............................................. 15
   Review Questions .................................... 16
   Additional References ............................... 16

2  2D Graphics .......................................... 19
   2D Rendering Foundations ........................... 20
   Sprites .............................................. 22
   Scrolling ........................................... 30
   Tile Maps ........................................... 35
   Summary ............................................ 39
   Review Questions ................................... 39
   Additional References ............................... 39

3  Linear Algebra for Games ......................... 41
   Vectors ............................................. 42
   Matrices ............................................ 58
   Summary ............................................ 62
   Review Questions ................................... 62
   Additional References ............................... 63
4  3D Graphics  .................................................. 65
   Basics ...................................................... 66
   Coordinate Spaces ...................................... 67
   Lighting and Shading .................................... 76
   Visibility .................................................. 85
   World Transform, Revisited ........................... 88
   Summary .................................................. 91
   Review Questions ........................................ 92
   Additional References ................................. 92

5  Input .......................................................... 93
   Input Devices .............................................. 94
   Event-Based Input Systems ............................ 99
   Mobile Input ............................................. 105
   Summary .................................................. 108
   Review Questions ....................................... 108
   Additional References ................................. 109

6  Sound .......................................................... 111
   Basic Sound ............................................... 112
   3D Sound .................................................. 115
   Digital Signal Processing .............................. 119
   Other Sound Topics ..................................... 122
   Summary .................................................. 124
   Review Questions ....................................... 125
   Additional References ................................. 125

7  Physics ........................................................ 127
   Planes, Rays, and Line Segments ..................... 128
   Collision Geometry ..................................... 130
   Collision Detection ...................................... 134
   Physics-Based Movement ............................. 148
Physics Middleware ........................................ 153
Summary .................................................. 154
Review Questions ........................................ 154
Additional References ................................. 155

8 Cameras .................................................. 157
Types of Cameras ......................................... 158
Perspective Projections ............................... 161
Camera Implementations ......................... 164
Camera Support Algorithms ..................... 175
Summary .................................................. 178
Review Questions ........................................ 178
Additional References ................................. 178

9 Artificial Intelligence ................................. 179
“Real” AI versus Game AI ......................... 180
Pathfinding ............................................... 180
State-Based Behaviors ............................. 192
Strategy and Planning ............................. 198
Summary .................................................. 200
Review Questions ........................................ 200
Additional References ................................. 202

10 User Interfaces ........................................ 203
Menu Systems .......................................... 204
HUD Elements .......................................... 207
Other UI Considerations ........................... 217
Summary .................................................. 221
Review Questions ........................................ 222
Additional References ................................. 222
**CONTENTS**

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chapter 3: Linear Algebra for Games</td>
<td>290</td>
</tr>
<tr>
<td>Chapter 4: 3D Graphics</td>
<td>291</td>
</tr>
<tr>
<td>Chapter 5: Input</td>
<td>292</td>
</tr>
<tr>
<td>Chapter 6: Sound</td>
<td>294</td>
</tr>
<tr>
<td>Chapter 7: Physics</td>
<td>295</td>
</tr>
<tr>
<td>Chapter 8: Cameras</td>
<td>295</td>
</tr>
<tr>
<td>Chapter 9: Artificial Intelligence</td>
<td>296</td>
</tr>
<tr>
<td>Chapter 10: User Interfaces</td>
<td>298</td>
</tr>
<tr>
<td>Chapter 11: Scripting Languages and Data Formats</td>
<td>299</td>
</tr>
<tr>
<td>Chapter 12: Networked Games</td>
<td>300</td>
</tr>
<tr>
<td>B Useful Tools for Programmers</td>
<td>303</td>
</tr>
<tr>
<td>Debugger</td>
<td>304</td>
</tr>
<tr>
<td>Source Control</td>
<td>309</td>
</tr>
<tr>
<td>Diff and Merging Tools</td>
<td>312</td>
</tr>
<tr>
<td>Issue Tracking</td>
<td>313</td>
</tr>
<tr>
<td>Index</td>
<td>315</td>
</tr>
</tbody>
</table>
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.
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.

![Dependency graph for Chapters 1–12.](image)

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:

```plaintext
function Update(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 P1.
Listing P.1 Sample Numbered Code Listing

```plaintext
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:

```plaintext
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
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.


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!*


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.


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.


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:

```plaintext
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

```plaintext
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,
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](image)

**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:

```c
// 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:

```cpp
// 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**

```cpp
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**

```c
// 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:

```cpp
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:

```cpp
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:

```cpp
// 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 DGameObject. 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:

```c
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

```c
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
CHAPTER 1  GAME PROGRAMMING OVERVIEW

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


Game Loops


West, Mick. “Programming Responsiveness” and “Measuring Responsiveness” (http://tinyurl.com/594f6r and http://tinyurl.com/Sqv5zt). 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


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
INDEX

Symbols

2D graphics
  characters, 54
  isometric tile maps, 38-39
  maps, 35-36
  rendering, 20-22
  sprites, 22-23
  animating, 25-28
  drawing, 23-25
  sheets, 28-29
  tile maps, 36-37

3D graphics
  cameras. See cameras
  coordinate systems, 56-57, 67
    converting, 298
  homogenous coordinates, 68
  model spaces, 67
  projection spaces, 74-76
  transforming 4D vectors by matrices, 69
  view/camera spaces, 72-73
  world spaces, 68-72

lighting/shading, 76-83, 85
  colors, 76
  Phong reflection models, 82-83
  vertex attributes, 77-79

object representations, 91
  overview of, 66
  polygons, 66
  sound, 115-119
  vectors, 42, 60-61
  visibility, 85-87
  world transforms, 88-90

4D vectors, 42, 69

5.1 surround sound systems, 118

A

AAA titles, 4
A* algorithm, 189-192
absolute coordinates, 218, 298
abstract syntax tree. See AST
accelerometers, 106-108, 293
accounts, GitHub, 311
adding
  breakpoint conditions, 307
  watches, 306-307

addition
  matrices, 58
  vectors, 43-44

addresses
  ping, 245
  protocols. See protocols
  admissible heuristics, 184-185
  advantages
    game state cheats, 256
    hosts, 253
  AI (artificial intelligence)
    pathfinding, 180-192
    planning/strategies, 198-200
  state-based behaviors, 192-198
  strategies, 298
  types of, 180

AIController class, 195

aiiming reticules, 211-212, 298

algebra
  matrices, 58
    addition/subtraction, 58
    inverse, 60
  multiplication, 59-60
  scalar multiplication, 58
  transforming 3D vectors, 60-61
  transposes, 60

solutions, 290-291

vectors, 42-43
  addition, 43-44
  coordinate systems, 56-57
  cross products, 51-53
  dot products, 48
  length, 45-46
  linear interpolation, 55-56
  reflection, 50-51
  rotating 2D characters, 54
  scalar multiplication, 47
  subtraction, 44-45

algorithms
  A+, 189-192
  Bresenham’s line-drawing, 66
  cameras, 175-178
  culling, 88
  Dijkstra’s, 192, 297
  greedy best-first, 185-189
  instantaneous collision detection, 141, 295
  occlusion, 88
  painter’s, 23, 85-86, 289, 292
pathfinding, 180-192
Rubine, 106, 293
ambient components, 82
ambient light, 80, 292
analog
filtering, 97, 293
input devices, 97-99
analysis
code
side scrollers, 262-267
tower defense games, 273-284
lexical, 230
syntax, 232-233
Angry Birds Space, 33
angular mechanics, 153
AnimatedSprite class, 26
animating sprites, 25-28
AnimFrameData structure, 26
AppDelegate class, 263
application debuggers, 304
breakpoints, 305-307
call stack windows, 307
data breakpoints, 308
watches, 306-307
applying
GitHub clients, 311-312
multiple transforms, 71
apps, mobile phones, 33
arbitrary convex polygons, 133
arbitrary points, planes, 295
arcade versions, 6
areas, nodes, 296
arrows, waypoint, 208-211, 298
artifacts, 66
Artificial Intelligence and Interactive Digital Entertainment (AIIDE) conference, 198
artificial intelligence. See AI
ASCII characters, 299
aspect ratios, 163
Assassins Creed: Revelations, 4
AST (abstract syntax tree), 232
Atari, 2-3
attacks, man-in-the-middle, 256-257, 301
attitudes, 108
Audacity, 112
audio
3D sound, 115
emitters/listeners, 116-118
falloff, 118
surround sound, 118-119
Doppler effects, 122-123
DSP, 119-122
obstruction, 123
sound
cues, 112-115
occlusion, 123
source data, 112
augmented reality games, 293
automating edges, 182
axis-aligned bounding boxes, 132, 135-136, 295
B
back face culling, 79
backgrounds, parallax scrolling, 32-34
Backus-Naur Form. See BNF
basic sounds, 112
sound cues, 112-115
source data, 112
basis vectors, 57
behaviors, state-based, 192-198
design patterns, 196-198
finite state machines, 193-194
state machines, implementing, 195-196
bidirectional reflectance distribution function (BRDF), 82
binary files, 235, 300
binary formats, 236
binary spatial partitioning. See BSP
binding, native, 227
BioShock, 163
BioShock: Infinite, 221
bit depth, 76
bitfield, 250
BNF (Backus-Naur Form), 232
bounding spheres, 131
branches, 310
BRDF (bidirectional reflectance distribution function), 82
breakpoints
conditions, 307
configuring, 304-305
data, 308
Bresenham’s line-drawing algorithm, 66
BSP (binary spatial partitioning), 88, 148
buffers
colors, 21-22, 289
do double buffering, 22
z-buffering, 86-87, 292
bugs, issue tracking, 313
Bugzilla, 313
builds, Release, 306
bullet through paper problem, 141
bullet time, 10
Burnout, 160
Bushnell, Nolan, 2
buttons, menus, 205-206
collisions, 276-277
GameplayScene, 263
GameState, 275
MainMenuLayer, 263
ObjectLayer, 266-267
Pathfinder, 278
Projectile, 265
ScrollingLayer, 264
Ship, 265
Singleton, 274
Timer, 278
clicks, mouse, 100
clients, 300
diff/merging tools, 312
Git, 310-312
predication, 252
server/client models, 250-253
closed sets, 186
cocos2d, 262
code. See also languages
analysis
side scrollers, 262-267
tower defense games, 273-284
debuggers, 304
breakpoints, 304-307
call stack windows, 307
data breakpoints, 308
watches, 306-307
duplication, 293
executing, 233-234
letter key, 298
scripting languages, 224
implementing, 229-234
Lua, 227
tradeoffs, 224-225
types of, 225-226
UnrealScript, 227-228
visual scripting systems, 229
source control, 310. See also source control
coefficient of restitution, 147
collisions
cameras, 175-176
detection, 134
axis-aligned bounding box intersection, 135-136
line segment versus plane intersection, 136-137
line segment versus triangle intersection, 138-139
optimizing, 147-148
responses, 146-147
sphere intersection, 134
sphere versus plane intersection, 140
swept sphere intersection, 141-145
geometry, 130-134

C

calculating
force, 151
frames, 27
length, 45-46
points, 177
Call of Duty, 217
call stack windows, 307
cameras, 158
algorithms, 175-178
cutsene, 161
Defense, 279
first-person, 159-160, 170
fixed, 158
follow, 160-165
implementing, 164-175
non-player controlled, 158
orbit, 168-170
perspective projections, 161-163
positioning, 296
spaces, 72-73
splines, 172-175, 296
spring follow, 165-168
capsules, 133
Carmack, John, 46-47
Cartesian coordinate spaces, 67
case studies, World of Warcraft, 239-241
cathode ray tubes. See CRTs
Catmull-Rom splines, 173, 296
CCD (continuous collision detection), 141
central processing units. See CPUs
ChangeAnim function, 27
debuggers, 304
classification, 224
implementing, 229-234
Lua, 227
tradeoffs, 224-225
types of, 225-226
UnrealScript, 227-228
visual scripting systems, 229
source control, 310. See also source control
coefficient of restitution, 147
collisions
cameras, 175-176
detection, 134
axis-aligned bounding box intersection, 135-136
line segment versus plane intersection, 136-137
line segment versus triangle intersection, 138-139
optimizing, 147-148
responses, 146-147
sphere intersection, 134
sphere versus plane intersection, 140
swept sphere intersection, 141-145
geometry, 130-134

C
calculating
force, 151
frames, 27
length, 45-46
points, 177
Call of Duty, 217
call stack windows, 307
cameras, 158
algorithms, 175-178
cutsene, 161
Defense, 279
first-person, 159-160, 170
fixed, 158
follow, 160-165
implementing, 164-175
non-player controlled, 158
orbit, 168-170
perspective projections, 161-163
positioning, 296
spaces, 72-73
splines, 172-175, 296
spring follow, 165-168
capsules, 133
Carmack, John, 46-47
Cartesian coordinate spaces, 67
case studies, World of Warcraft, 239-241
cathode ray tubes. See CRTs
Catmull-Rom splines, 173, 296
CCD (continuous collision detection), 141
central processing units. See CPUs
ChangeAnim function, 27
debuggers, 304
classification, 224
implementing, 229-234
Lua, 227
tradeoffs, 224-225
types of, 225-226
UnrealScript, 227-228
visual scripting systems, 229
source control, 310. See also source control
coefficient of restitution, 147
collisions
cameras, 175-176
detection, 134
axis-aligned bounding box intersection, 135-136
line segment versus plane intersection, 136-137
line segment versus triangle intersection, 138-139
optimizing, 147-148
responses, 146-147
sphere intersection, 134
sphere versus plane intersection, 140
swept sphere intersection, 141-145
geometry, 130-134
colors, 76
buffers, 21-22, 289
resolution, 20
column-major vectors, 61
Company of Heroes, 227, 255
components
ambient, 82
diffuse, 82
specular, 83
w-components, 68
compressors, 120
Computer Space, 2
conditions
breakpoints, adding, 307
hit count, 307
configuring
breakpoints, 304-305
conditions, 307
data, 308
_Defense, 273
source control, 309
Git, 310-312
SVN, 309
watches, 306-307
conjugate of quaternions, 90
connections
peer-to-peer, 253-254
server/client models, 250-253
TCP, 246. See also TCP
UDP, 249. See also UDP
consoles, 9, 288
ConstructPlan function, 199
continuous collision detection. See CCD
cycle
flow
game loops, 5
multithreaded, 7-8
objects in game loops, 13-15
phases of game loops, 5-7
source, 309
Git, 310-312
SVN, 309
converting
3D coordinates, 298
coordinate systems, 56
vectors, 46
convexes
hulls, 133
polygons, 121, 294
coordinate systems, 56-57
3D, converting, 298
absolute, 218, 298
coordinate spaces, 67, 291
model spaces, 67
projection spaces, 74-76
view/camera spaces, 72-73
world spaces, 68-72
homogenous coordinates, 68
textures, 77
UV, 78
CPUs (central processing units), 7
Crane, David, 3
crashes, issue tracking, 313
Crazy Taxi, 208
cross products, 48-53
CRTs (cathode ray tubes), 20-21
Crysis, 2
cues, sound, 112-115, 294
culling
algorithms, 88
back face, 79
cutscenes, 161, 296
cycles, development, 2

D
Dabney, Ted, 2
data breakpoints, 308
data formats, 235
binary, 236
INI, 236-237
JSON, 238
tradeoffs, 235-236
XML, 237
dBS (decibels), 118, 294
dead zones, 97
dependencies, 304
breakpoints, 304-308
call stack windows, 307
watches, 306-307
decibels, 118, 294
dedicated servers, 253
_Defense, 270
C# language, 270-272
code analysis, 273-284
exercises, 284-285
MonoGame, 273
XNA, 272
delta time, 10-12, 288
depth
bit, 76
buffers, 86
design, 196-198. See also configuring
Design Patterns, 196
detection, collision, 134
axis-aligned bounding box intersection,
135-136
line segments
versus plane intersection, 136-137
versus triangle intersection, 138-139
optimizing, 147-148
responses, 146-147
spheres
intersection, 134
versus plane intersection, 140
swept sphere intersection, 141-145
development, cycles, 2
devices
digital, 292
event-based input systems, 99-104
input, 94
analog, 97-99
digital, 95-97
mobile input, 105
accelerometers/gyroscopes, 106-108
touch screens, 105-106
Diablo III, 74
Diablo, 38
diamond problem, 14
diff tools, 312
diffuse component, 82
digital devices, 292
digital input devices, 95, 97
digital signal processing. See DSP
Dijkstra’s algorithm, 192, 297
dimensions
matrices
inverse, 60
multiplication, 59-60
scalar multiplication, 58
transforming 3D vectors, 60-61
transposes, 60
vectors, 42-43
addition, 43-44
coordinate systems, 56-57
cross products, 51-53
dot products, 48
length, 45-46
linear interpolation, 55-56
reflection, 50-51
rotating 2D characters, 54
scalar multiplication, 47
subtraction, 44-45
directional light, 81, 292
distance
Euclidean, 185
Manhattan, 184
documentation, issue tracking, 313
doppler effects, 122-123, 294
doppler shifts, 120
dot products, vectors, 48
double buffering, 22
doubly linked lists, 289
drawing
sprites, 23-25
vectors, 43
dropping frames, 12
DSP (digital signal processing), 119-122, 294
duplication of code, 293

echo packets, 245
drops, 181-182
effects
Doppler, 122-123, 294
DSP, 119-122, 294
electron guns, 289
elements, HUD, 207
aiming reticules, 211-212
radar, 212-217
waypoint arrows, 208-211
emitters, 116-118
encapsulation, 271
encryption, 257
Enemy class, 265
enumerations, keycode, 95
equality, vectors, 43
equations, parametric, 295
ers, operator precedence, 226
Euclidean distance, 185
Euler integration, 151-152
Euler rotations, 70, 292
event-based input systems, 99-104
events, 293
evolution of video game programming, 2
Atari era (1977-1985), 2-3
future of, 4-5
NES/SNES era (1985-1995), 3
Playstation/Playstation 2 era (1995-2005), 3
PS3/Wii/Xbox 360 era (2005-2013), 4
executing code, 233-234
exercises
Ship Attack, 267-268
tower defense games, 284-285
experience, UX, 221
expressions, regular, 231, 299
Extensible Markup Language. See XML

fall off, 118
Fallout, 38
field of view. See FOV
fields, vectors, 42
frames
calculating, 27
dropping, 12
limiting, 12
rates, 288
frames per second. See FPS
frameworks
cocos2d, 262
MonoGame, 273
XNA, 272
Fresnel acoustic diffraction, 123
Full Spectrum Warrior, 4
functions
ChangeAnim, 27
ConstructPlan, 199
Fire script, 101
getter/setter, 271
Initialize, 27
LoadLevel, 277
parametric, 129
SetState, 196
storing, 293
Update, 196
UpdateAnim, 27
UpdateInput, 101
Game class, 274
GameObject class, 276-277
GameplayScene class, 263
games
3D graphics, 66-76
AI
pathfinding, 180-192
planning/strategies, 198-200
state-based behaviors, 192-198
types of, 180
augmented reality, 293
cameras. See cameras
collision geometry, 130-134
data formats, 235-238
history of programming, 2-5
HUD elements, 207-217
linear algebra for
matrices, 58-61
vectors, 42-57
localization, 219-221
loops, 5, 288
multithreaded, 7-8
objects, 13-15
phases, 5-7
menus, 204-207
networked, 244. See also networked games
INI FILES

Havok Physics, 4
headers, 244-245
heads-up display. See HUD elements
head (vectors), 43
head-tracking devices, 94
height, aspect ratios, 163
heuristics, 184-185, 297
history
of video game programming, 2
Atari era (1977-1985), 2-3
future of, 4-5
NES/SNES era (1985-1995), 3
Playstation/Playstation 2 era (1995-2005), 3
PS3/Wii/Xbox 360 era (2005-2013), 4
source control, 309
Git, 310-312
SVN, 309
hit count conditions, 307
homogenous coordinates, 68
hosts, advantages, 253
HTTP (Hypertext Transfer Protocol), 257
HTTPS (Secure HTTP), 257
HUD (heads-up display) elements, 207
aiming reticules, 211-212
radar, 212-217
waypoint arrows, 208-211
ICMP (Internet Control Messaging Protocol), 245-246, 300
identifiers, tokenization, 230
if statements, 31
image files, 28
implementing
cameras, 158, 164-175
generating code, 233-234
goals, 4
GPUs (graphics processing units), 66
Grand Theft Auto III, 3
Grand Theft Auto IV, 4
graphical artifacts, 66
graphs, representing search spaces, 181-183
Grassmann products, 90
greedy best-first algorithms, 185-189
gui, 271
Grim Fandango, 227
Gouraud shading, 83, 292
GPS data, 108
scripting languages, 224-234
side scrollers, 260-268
sound, 112-115
state, 256, 275
third-person, 117, 294
time, 9-12
topologies, 250-254
tower defense, 270-275
GameState class, 275
Gears of War, 183, 253
Generating code, 233-234
outputs, 5
geometry, collision, 130
arbitrary convex polygons, 133
axis-aligned bounding boxes, 132
bounding spheres, 131
capsules, 133
lists, 134
oriented bounding boxes, 132
gestures, 105-106
getter functions, 271
ghosts in Pac Man, 194
Git, 310-312
GitHub, 311-313
God of War, 158
Gouraud shading, 83, 292
GPS data, 108
GPUs (graphics processing units), 66
Grand Theft Auto III, 3
Grand Theft Auto IV, 4
graphical artifacts, 66
graphics
2D. See 2D graphics
3D. See 3D graphics
Defense, 281
rendering, 7
graphics processing units. See GPUs
graphs, representing search spaces, 181-183
Grassmann products, 90
greedy best-first algorithms, 185-189
grids, matrices, 58
Grim Fandango, 227
Guitar controllers, 94
gyrosopes, 106-108, 293
Halo, 255
handedness of coordinate systems, 52
hard-coding UI text, 298
Havok, 153
Havok Physics, 4
headers, 244-245
heads-up display. See HUD elements
head (vectors), 43
head-tracking devices, 94
height, aspect ratios, 163
heuristics, 184-185, 297
history
of video game programming, 2
Atari era (1977-1985), 2-3
future of, 4-5
NES/SNES era (1985-1995), 3
Playstation/Playstation 2 era (1995-2005), 3
PS3/Wii/Xbox 360 era (2005-2013), 4
source control, 309
Git, 310-312
SVN, 309
hit count conditions, 307
homogenous coordinates, 68
hosts, advantages, 253
HTTP (Hypertext Transfer Protocol), 257
HTTPS (Secure HTTP), 257
HUD (heads-up display) elements, 207
aiming reticules, 211-212
radar, 212-217
waypoint arrows, 208-211
ICMP (Internet Control Messaging Protocol), 245-246, 300
identifiers, tokenization, 230
if statements, 31
image files, 28
implementing
cameras, 158, 164-175
generating code, 233-234
goals, 4
GPUs (graphics processing units), 66
Grand Theft Auto III, 3
Grand Theft Auto IV, 4
graphical artifacts, 66
graphs, representing search spaces, 181-183
Grassmann products, 90
greedy best-first algorithms, 185-189
grids, matrices, 58
Grim Fandango, 227
Guitar controllers, 94
gyrosopes, 106-108, 293
Halo, 255
handedness of coordinate systems, 52
hard-coding UI text, 298
Havok, 153
Initialize function, 27
input
  _Defense, 280
devices, 94
  analog, 97-99
digital, 95-97
event-based systems, 99-104
lag, 8, 288
menus, typing, 206-207
mobile, 105
  accelerometers/gyroscopes, 106-108
touch screens, 105-106
inputs, processing, 5
instantaneous collision detection algorithms, 141, 295
integration
  Euler, 151-152
methods, 153
numeric methods, 295
semi-implicit Euler, 151-152
velocity Verlet, 152-153, 295
interfaces, 293. See also UIs
Internet Control Messaging Protocol. See ICMP
Internet Protocol. See IP
intersections
  axis-aligned bounding boxes, 135-136
  line segment, 136-139
  planes, 136-140
  spheres, 134, 140-145, 295
  triangles, 138-139
inverse matrices, 60
inverse square roots, 46-47
IP (Internet Protocol), 244-245, 300
isometric tile maps, 38-39
isometric views, 290
issue tracking, 313
Iwatani, Toru, 194

J–K
JavaScript Object Notation. See JSON
Jetpack Joyride, 30, 33
Jobs, Steve, 261
joysticks, 6. See also input
JSON (JavaScript Object Notation), 113, 238
key codes, 95, 298
KeyState value, 96
keywords, tokenization, 230
Kinect, 94
Kismet, 229

L
lag, input, 8, 288
Lammers, Susan, 194
languages
  C, 2
  C#, 270-272
  Objective-C, 261
scripting, 224, 299
  implementing, 229-234
  Lua, 227
tradeoffs, 224-225
types of, 225-226
  UnrealScript, 227-228
  visual scripting systems, 229
XML, 237
Last of Us, The, 2
latency, 245
Legend of Zelda, The, 3, 35
length
  vectors, 45-46
  squared, 46
lerp. See linear interpolation
letter key codes, 298
lexical analysis, 230
LFE (low-frequency effects) channels, 118
libraries, XNA game, 272
lighting, 76, 79, 292
  colors, 76
  per pixel, 84
  Phong reflection models, 82-83
  shading, 83-85
  vertex attributes, 77-79
limiting frames, 12
linear algebra
  matrices, 58-61
  vectors, 42-43
    addition, 43-44
    coordinate systems, 56-57
    cross products, 51-53
    dot products, 48
    length, 45-46
    linear interpolation, 55-56
    reflection, 50-51
    rotating 2D characters, 54
    scalar multiplication, 47
  subtraction, 44-45
linear interpolation, 55-56
linear mechanics, 149
line segments, 129-130, 136-139
listeners, 116-118, 294
lists of collision geometries, 134
LoadLevel function, 277
localization, 219-221, 280
local lighting models, 82
lockstep models, 301
logic as function of delta time, 10-12
look-at matrices, 72
loops, game, 5, 288
    multithreaded, 7-8
        objects, 13-15
        phases, 5-7
Lord of the Rings: Conquest, 57, 117
loss, packets, 247
low-frequency effects. See LFE channels
low-pass filters, 120
Lua, 225, 227

M
machines, state
design patterns, 196-198
    finite, 193-194
    implementing, 195-196
macro strategies, 198
magnitude, 46. See also length
MainMenuLayer class, 263
Manhattan distance, 184
man-in-the-middle attacks, 256-257, 301
maps, 35-36
    textures, 77
    tiles, 35-39
marking regions, 121-122
mass, 149
math
    matrices, 58-61
    vectors, 43-53
matrices, 58
    3D vectors, transforming, 60-61
    4D vectors, transforming, 69
    addition/subtraction, 58
    inverse, 60
    look-at, 72
    multiplication, 59-60
    rotation, 70
    scalar multiplication, 58
    scale, 69
    transform, 69
    translation, 70
    transposes, 60
maximum transmission units. See MTUs
Max Payne, 10
mechanics
    angular, 153
    linear, 149
    memory, 2
menus, 204
    buttons, 205-206
    stacks, 204-205, 298
    typing, 206-207
merging tools, 312
messages, Objective-C, 261
methods. See also functions
    integration, 153
    numeric integration, 295
    partitioning, 148
metrics, 313
micro strategies, 198
middleware, 4, 153-154, 221, 288
mipmapming, 28
mobile input, 105
    accelerometers/gyroscopes, 106-108
    touch screens, 105-106
mobile phones, apps, 33
models
    local lighting, 82
    lockstep, 301
    peer-to-peer, 253-254
    Phong reflection, 82-83, 292
    server/client, 250-253, 300
    spaces, 67
monitors, CRTs, 20-21
MonoGame, 273
Moon Patrol, 33
Mountain Lion, 260
mouse clicks, 100, 293
movement, physics, 148
    angular mechanics, 153
    calculating force, 151
    Euler/semi-implicit Euler integration, 151-152
    integration methods, 153
    linear mechanics, 149
    variable time steps, 150
    velocity Verlet integration, 152-153
MTUs (maximum transmission units), 247
multicore consoles, 9
multicore CPUs, 7. See also CPUs
multiplayer support, 5
multiple resolution support, 218-219
multiple transforms, applying, 71
multiplication
    matrices, 58-60
    order of, 291
    vectors, 47
multithreaded game loops, 7-8
multi-touch, 105
N
Namco, 6
native binding, 227
navigating
accelerometers/gyroscopes, 106-108
HUD elements, 207
aiming reticules, 211-212
radar, 212-217
waypoint arrows, 208-211
menus, 204
buttons, 205-206
stacks, 204-205
typing, 206-207
touch screens, 105-106
near planes, 75
NES (Nintendo Entertainment System), 3
networked games
cheats, 255-257
protocols, 244
ICMP, 245-246
IP, 244-245
TCP, 246-249
UDP, 249-250
topologies, 250-254
NeXT, 261
nodes, 181
areas, 296
path, 182
scene, 262
non-player characters. See NPCs
non-player controlled cameras, 158
normalization, 46
normal to planes, 51
NPCs (non-player characters), 180
numbers
floating point, 46-47
matrices, 58
sequence, 250
numeric integration methods, 295
O
Objective-C language, 261
ObjectLayer class, 266-267
objects, 13
categories of, 289
game loops, 13-15
polygons, 66
representations, 91
sprites, 22-23
animating, 25-28
drawing, 23-25
sheets, 28-29

static, 13
types of, 13
obstruction, sound, 123
occlusion
algorithms, 88
sound, 123, 294
octets, 245
octrees, 148
Oculus Rift, 108
online multiplayer support, 5
OpenAL, 112
open sets, 186
operators
precedence errors, 226
tokenization, 230
optimizing collisions, 147-148
orbit cameras, 168-170
order
of multiplication, 291
winding, 78
oriented bounding boxes, 132
orthographic projections, 292
outputs, 5, 288
overdraw, 85

P
packets, 244
echo, 245
ICMP, 246
loss, 247
UDP, 249
Pac-Man, 3, 6-7, 194
painter’s algorithm, 23, 85-86, 289, 292
parallax scrolling, 32-34
parallelogram rule, 44
parametric equations, 295
parametric functions, 129
partitioning, 148
Pathfinder class, 278
pathfinding, AI, 180-192
path nodes, 182
patterns
design, 196-198
state design, 297
peer-to-peer models, 253-254
performance, TCP, 248
per pixel lighting, 84
perspective, projections, 74, 161-163
phases, game loops, 5-7
Phong
reflection models, 82-83, 292
shading, 84
physicss

projection spaces, 74-76
ray tracing, 87-128
reflection models, 82-83, 292
sphere intersection, 136
swept sphere intersection, 141-145
sphere versus plane intersection, 140
spatial data structures, 178-221

__Defense, 280
line segments, 129-130
middleware, 153-154
movement, 148-153
planes, 128
rays, 129-130

PhysX, 153
picking, 176-178, 209, 296
ping, 245
pitch systems, 120-123
Pitfall!, 3
pixels, 20-21
planes, 128
arbitrary points, 295
intersection, 136-137, 140
near, 75
planning AI, 198-200
Playstation/Playstation 2 era (1995-2005), 3
points
calculating, 177
light, 81
polling systems, 99, 293
polygons, 66, 291
arbitrary convex, 133
convex, 121, 294
Pong, 192
ports, 248, 300
positioning
cameras, 296
vectors, 42
precedence, operator errors, 226
prediction, clients, 252
Prince of Persia: The Sands of Time, 10
processing
CPUs. See CPUs
inputs, 5
processors, history of, 2
products
cross, 48-53
dot, 48
Grassmann, 90

Programmers at Work, 194
programming
3D graphics
coordinate spaces, 67-73
homogenous coordinates, 68
overview of, 66
polygons, 66
projection spaces, 74-76
transforming 4D vectors by matrices, 69
evolution of, 2-5
game loops, 5
multithreaded, 7-8
objects, 13-15
phases, 5-7
objects, 13
time, 9-12
Projectile class, 265
projections
__Defense, 279
orthographic, 292
perspective, 161-163
projection spaces, 74-76
protocols, 244
HTTP, 257
HTTPS, 257
ICMP, 245-246, 300
IP, 244-245, 300
TCP, 246-249, 300
UDP, 249-250, 300
PS3 era (2005-2013), 4
Python, 225

Q–R
quadtrees, 148
QuakeC, 225
quaternions, 89-90
radar, 212-217
radii squared, 134
RAM (random access memory), 2
rasterization, software, 66
rates, frames, 288
ratios, aspect, 163
rays, 129-130
real time, 10, 288
real-time strategy. See RTS
red, green, and blue. See RGB
reflection
Phong reflection models, 82-83, 292
vectors, 50-51
regions, marking, 121-122
regular expressions, 231, 299
relative coordinates, 218
Release builds, 306
rendering, 288
  2D graphics, 20
    color buffers, 21-22
    CRT monitor basics, 20-21
    isometric tile maps, 38-39
    maps, 35-36
    scrolling, 30-34
    sprites, 22-29
    tile maps, 36-37
    vertical sync, 21-22
  graphics, 7
reports, issue tracking, 313
repositories, 309
representations
  objects, 91
  search spaces, 181-183
Resident Evil, 158
resolution, 20, 218-219
responses, collisions, 146-147
reticules, aiming, 211-212, 298
reverb, 120, 294
RGB (red, green, and blue), 20, 76
right hand rule, 52
Rock Band, 180
rotations
  2D characters, 54
    Euler, 292
  matrices, 70
row-major vectors, 61
RTS (real-time strategy), 198
Rubine algorithm, 106, 293
Rubine, Dean, 106
rules
  parallelogram, 44
  right hand, 52
Runge-Kutta method, 153

S
sample games
  side scrollers, 260-268
  tower defense games, 270-285
sample problems
  2D characters, 54
  vector reflection, 50-51
  scalar multiplication
    matrices, 58
    vectors, 47
Scaleform, 221
scale matrices, 69
scene nodes, 262
screens. See also interfaces
  space, 74-76
  tearing, 21, 289
scripting languages, 239-241, 299

scrolling
  2D graphics, 30
    four-way, 34
    infinite, 32
    parallax, 32-34
    single-axis, 30-32
    single-direction, 289
ScrollingLayer class, 264
SDKs (software development kits), 3
SDL (Simple DirectMedia Layer), 95
searching greedy best-first algorithms, 185-189
search spaces, representing, 181-183
Secure HTTP. See HTTPS
segments, lines, 129-130, 136-139
semi-implicit Euler integration, 151-152
sending
  ICMP, 245-246
  IP, 244-245
sequences, 94
  numbers, 250
  regular expressions, 231
serialization, 236
servers, 300
  dedicated, 253
  repositories, 309
  server/client models, 250-253, 300
sets
  closed, 186
  open, 186
setState function, 196
setsStiles, 289
setter functions, 271
settings, _Defense, 273
shaders, 66
shading, 76, 83-85
  colors, 76
  Gouraud, 292
  lighting, 79
  Phong reflection models, 82-83
  vertex attributes, 77-79
sheets, sprites, 28-29
Ship Attack, 260
cocos2d, 262
  code analysis, 262-267
  exercises, 267-268
  Objective-C language, 261
Ship class, 265
side scrollers, 260
cocos2d, 262
  code analysis, 262-267
  exercises, 267-268
  Objective-C language, 261
signals, DSP, 119-122
Simple DirectMedia Layer. See SDL
Sims, The, 74
single-axis scrolling, 30-32
single-direction scrolling, 289
single models, 66
Singleton class, 274
Skyrim, 159
smartphones, 105
accelerometers/gyroscopes, 106-108
touch screens, 105-106
SNES era (1985-1995), 3
sockets, UDP, 250
software development kits. See SDKs
software rasterization, 66
solutions, algebra, 290-291
Sommefeldt, Rhys, 47
sound
3D, 115
  emitters/listeners, 116-118
  falloff, 118
  surround sound, 118-119
basic, 112
  sound cues, 112-115
  source data, 112
  cues, 294
  _Defense, 282
  Doppler effects, 122-123
DSP, 119-122
obstruction, 123
occlusion, 123, 294
source control, 309
diff/merging tools, 312
Git, 310-312
SVN, 309
source data
sound, 112, 294
switches between, 114
space
matrices, 58-61
vectors, 42-43
  addition, 43-44
  coordinate systems, 56-57
  cross products, 51-53
  dot products, 48
  length, 45-46
  linear interpolation, 55-56
  reflection, 50-51
  rotating 2D characters, 54
  scalar multiplication, 47
  subtraction, 44-45
specular component, 83
speed, updating, 10-12
spheres
  bounding, 131
  intersection, 134, 140
  intersections, 295
  swept, 141-145, 295
splines
cameras, 172-175, 296
Catmull-ROM, 296
spotlights, 81
spring follow cameras, 165-168
sprites, 22-23
  animating, 25-28
  drawing, 23-25
  sheets, 28-29, 289
stacks
call stack windows, 307
menus, 204-205, 298
Starcraft II, 256
StarCraft, 198, 200, 254
Star Wars: The Old Republic, 221
state
  behaviors, 192-198
  design patterns, 196-198
  finite state machines, 193-194
  implementing state machines, 195-196
  design patterns, 297
  games
    cheats, 256
    _Defense, 275
  machines
    design patterns, 196-198
    implementing, 195-196
  statements, if, 31
  static objects, 13
  storing functions, 293
  strategies, AI, 198-200, 298
  Street Fighter, 8, 95
subtraction
matrices, 58
vectors, 44-45
subversion. See SVN
Super Mario Bros. 2, 34
Super Mario Bros., 13, 150
Super Mario World, 3, 32
support
  camera algorithms, 175-178
  data format, 235-238
  localization, 219-221
  multiple resolution, 218-219
  scripting languages, 224-234
surfaces, planes, 128
surround sound, 118-119
SVN (subversion), 309
swaps, buffers, 22
swept spheres, 295
switches between source data, 114
  symbols, tokenization, 230
sync, vertical, 21-22
syntax
  - analysis, 232-233
  - abstract trees, 299
  - Objective-C, 261

T
  - tablets, 105
    - accelerometers/gyroscopes, 106-108
  - touch screens, 105-106
  - tags, 310
  - tails (vectors), 43
  - TCP (Transmission Control Protocol), 246-249, 300
  - Team Fortress 2, 255
  - tearing, screen, 289
  - text
    - localization, 219
    - tokenization, 230
    - UI, hard-coding, 298
  - text-based files, 235
  - TexturePacker, 29, 260-262
  - textures, 28
    - coordinates, 77
    - mapping, 77
  - third-person games, 117, 294
  - tile maps, 35-39
  - time, 9
    - delta, 288
    - game time/real time, 10
    - logic as function of delta, 10-12
    - real, 288
    - variable time steps, 150
  - timeouts, 246
  - Timer class, 278
  - titles, sets, 289
  - tokenization, 230
  - tokens, 230
  - Tony Hawk’s Project 8, 205
  - tools
    - diff, 312
    - merging, 312
  - topologies, 250
    - peer-to-peer models, 253-254
    - server/client models, 250-253
  - touch screens, 105-106
  - tower defense games, 270
    - C# language, 270-272
    - code analysis, 273-284
    - exercises, 284-285
    - MonoGame, 273
    - XNA, 272
  - tracking issues, 313
  - tradeoffs
    - data formats, 235-236
    - scripting languages, 224-225
  - transforms
    - 3D vectors by matrices, 60-61
    - 4D vectors by matrices, 69
    - matrices, 58-61, 69
    - multiple, 71
    - worlds, 88-91
  - translation matrices, 70
  - Transmission Control Protocol. See TCP
  - transposes, matrices, 60
    - abstract syntax, 299
    - octrees, 148
    - quadtrees, 148
  - trends in game programming, 4
  - triangles, 291
  - intersection, 138-139
  - planes, 128
  - polygons, 66
  - Tribes, 250
  - troubleshooting
    - debuggers, 304
      - adding breakpoint conditions, 307
      - breakpoints, 304-305
      - call stack windows, 307
      - data breakpoints, 308
      - watches, 306-307
    - issue tracking, 313
    - trunk, 310
  - types
    - of AI, 180
    - of cameras, 158
      - cutscene, 161
      - first-person, 159-170
      - fixed, 158
      - follow, 160-165
      - non-player controlled, 158
      - orbit, 168-170
      - splines, 172-175
      - spring follow, 165-168
    - of game objects, 13
    - of input devices, 94-99
    - of lighting, 81, 292
    - of objects, 289
    - of scripting languages, 225-226
    - of shading, 83-85
    - typing menus, 206-207

U
  - UDP (User Datagram Protocol), 249-250, 300
  - UIs (user interfaces)
    - _Defense, 282-284
    - HUD elements, 207-217
    - localization, 219-221
menus, 204-207
  middleware, 221
  multiple resolution support, 218-219
text, hard-coding, 298
Uncharted, 160
uniform scale, 70
unit quaternions, 90
unit vectors, 46
Unity, 4
unprojection, 296
Unreal, 4
Unreal Engines, 101
UnrealScript, 225-228, 299
UpdateAnim function, 27
Update function, 196
UpdateInput function, 101
updating
  servers, 301
  speed, 10-12
  worlds, 5
User Datagram Protocol. See UDP
user interfaces. See UIs
UV coordinates, 78
UX (user experience), 221, 299

V
  variable time steps, 150
VBLANK (vertical blank interval), 20-21, 289
vectors, 42-43
  2D characters, rotating, 54
  4D, transforming by matrices, 69
  addition, 43-44
  coordinate systems, 56-57
  cross products, 51-53
  dot products, 48
  length, 45-46
  linear interpolation, 55-56
  reflection, 50-51
  scalar multiplication, 47
  subtraction, 44-45
velocity, 252
  collision responses, 146
  Verlet integration, 152-153, 295
versions, IP, 245
vertical blank interval. See VBLANK
vertical sync, 21-22
vertices, 66
  attributes, 77-79
  normal, 78
video games, history of programming, 2
  Atari era (1977-1985), 2-3
  future of, 4-5
  NES/SNES era (1985-1995), 3
  Playstation/Playstation era (1995-2005), 3
  PS3/Wii/Xbox 360 era (2005-2013), 4
viewing call stack windows, 307
viewports, 163
views, 72-73
field of, 295
isometric, 38-39, 290
virtual controllers, 105
virtual reality headsets, 94
visibility, 85
  painter’s algorithm, 85-86
  world transforms, 91
z-buffering, 86-87
visual scripting systems, 229, 299
Visual Studio
  breakpoints
    conditions, 307
    configuring, 304-305
data, 308
call stack windows, 307
  watches, adding, 306-307

W
  watches, configuring, 306-307
  waypoint arrows, 208-211, 298
w-components, 68
whitepapers, 250
width, aspect ratios, 163
Wii era (2005-2013), 4
WiiMote, 94
winding order, 78
windows, call stack, 307
World of Warcraft, 227
case study, 239-241
TCP, 248
worlds
  spaces, 68-72
  transforms, 88-91
  updating, 5

X–Z
Xbox 360 era (2005-2013), 4
Xcode, 260
XCOM, 181
XML (Extensible Markup Language), 237
XNA game libraries, 272
z-buffering, 86-88, 292
z-fighting, 88
Zinman-Jeanes, Jacob, 260
zones, dead, 97