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.

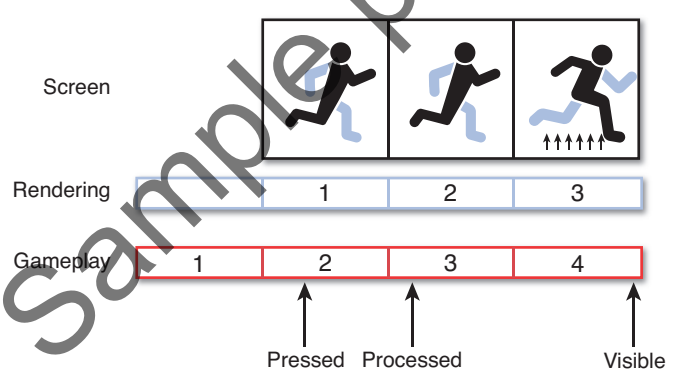

Figure 1.1 The jump is delayed a couple of frames due to input lag.

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.