CREATING 3D GAMES

Game Programming in C++

Sanjay **MADHAV**

Game Programming in C++

Creating 3D Games

Sanjay Madhav

Addison-Wesley

Boston • Columbus • Indianapolis • New York • San Francisco
Amsterdam • Cape Town • Dubai • London • Madrid • Milan
Munich • Paris • Montreal • Toronto • Delhi • Mexico City • São
Paulo • Sydney • Hong Kong • Seoul • Singapore • Taipei • Tokyo

Library of Congress Control Number: 2017964125

ISBN-13: 978-0-13-459720-1
ISBN-10: 0-13-459720-6

1 18

Editor-in-Chief
Mark Taub

Executive Editor
Laura Lewin

Development Editor
Michael Thurston

Managing Editor
Sandra Schroeder

Tech Editors
Josh Glazer
Brian Overland
Matt Whiting

Senior Project Editor
Lori Lyons

Production Manager
Dhayanidhi Karunanidhi

Copy Editor
Kitty Wilson

Indexer
Lisa Stumpf

Proofreader
Larry Sulky

Editorial Assistant
Courtney Martin

Cover Designer
Chuti Prasertsith

Compositor
codemantra

To my family and friends: Thanks for the support.

Contents at a Glance

Contents

Preface

Today, video games are some of the most popular forms of entertainment. Newzoo's "Global Games Market Report" estimates over $100 billion in revenue for games in 2017. This staggering amount shows how popular this field truly is. Because of the size of this market, game programmers are in low supply and high demand.

Alongside this explosion of games, game technology has become increasingly democratized. A single developer can make award-winning and hit games by using one of many popular game engines and tools. For game designers, these tools are fantastic. So what value is there in learning how to program games in C++?

If you take a step back, you can see that many game engines and tools are, at their core, written in C++. This means that C++ is ultimately the technology behind every game created using one of these tools.

Furthermore, top-notch developers who release some of the most popular games today—including *Overwatch*, *Call of Duty*, and *Uncharted*—still predominantly use C++ because it provides a great combination of performance and usability. Thus, any developer who wants to eventually work for one of these companies needs a strong understanding of programming games—specifically in C++.

This book dives into many of the technologies and systems that real game developers use. The basis for much of the material in this book is video game programming courses taught at the University of Southern California over the course of almost a decade. The approach used in this book has successfully prepared many students to make it in the video games industry.

This book is also heavily focused on real working implementations of code integrated into actual game project demos. It is critical to understand how all the various systems that go into a game work together. For this reason, you should keep the source code handy while working through this book.

At this writing, all the code provided with this book works on both PC and macOS, using the Microsoft Visual Studio 2017 and Apple Xcode 9 development environments, respectively.

The source code for this book is available on GitHub, at **https://github.com/gameprogcpp/code**. For instructions on setting up the development environment for this book, see Chapter 1, "Game Programming Overview."

Who Should Read This Book?

This book is for you if you're a programmer who is comfortable with C++ and wants to learn how to program 3D video games. For readers rusty on C++, Appendix A, "Intermediate C++ Review," reviews several C++ concepts. However, if you have with little or no prior C++ experience, you should learn C++ before jumping into this book. (One option is *Programming Abstractions in C++* by Eric Roberts.) This book also expects you to be familiar with some common data structures, including dynamic arrays (vectors), trees, and graphs, and to have some recollection of high school-level algebra.

The topics covered in this book are applicable to readers in academic environments, hobbyists, and junior- and mid-level game programmers who want to expand their knowledge of game programming. The content in this book corresponds to a little more than a semester and a half of material in a university setting.

How This Book Is Organized

This book is intended to be read linearly from Chapter 1 through Chapter 14. However, in case you are not interested in some specific topics, Figure P.1 shows the dependencies between the chapters.

In the first handful of chapters, the games are in 2D as you learn core concepts. From Chapter 6 onward (with the exception of Chapter 8), the games are in 3D.

The chapters cover the following information:

- Chapter 1, “Game Programming Overview,” looks at the fundamental concepts of game programming and how to get an initial game up and running. It also introduces the Simple DirectMedia Layer (SDL) library.
- Chapter 2, “Game Objects and 2D Graphics,” discusses how programmers organize the objects in their games and explores additional 2D graphics concepts, such as flipbook animation.
- Chapter 3, “Vectors and Basic Physics,” covers mathematical vectors, which are critical tools for any game programmer. It also explores the basics of physics, for use with both motion and collisions.
- Chapter 4, “Artificial Intelligence,” looks at approaches to make game characters that are computer controlled, including concepts such as state machines and pathfinding.
- Chapter 5, “OpenGL,” explores how to create an OpenGL renderer, including implementing vertex and pixel shaders. It includes a discussion of matrices.

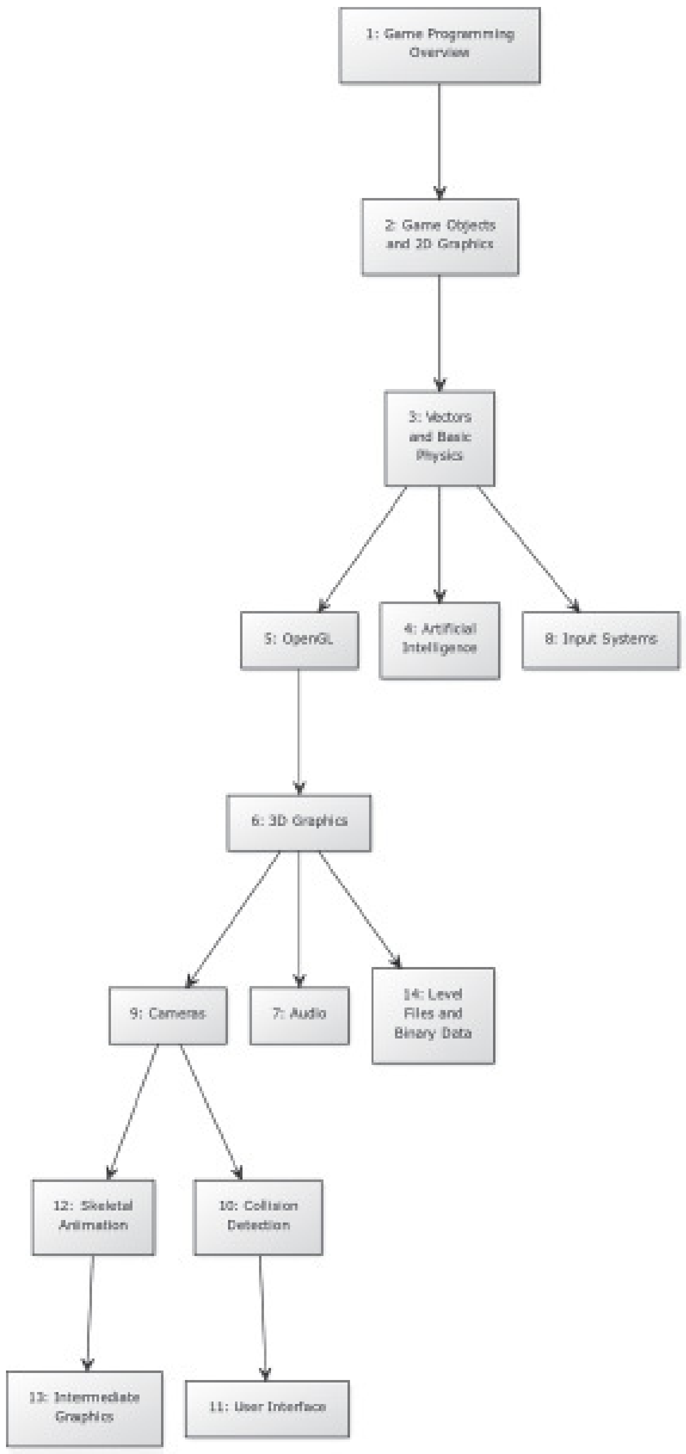
1: Game Programming Overview
2: Game Objects and 2D Graphics
3: Vectors and Basic Physics
5: OpenGL
4: Artificial Intelligence
8: Input Systems
6: 3D Graphics
9: Cameras
7: Audio
14: Level Files and Binary Data
12: Skeletal Animation
10: Collision Detection
13: Intermediate Graphics
11: User Interface

Figure P.1 Chapter dependencies

- Chapter 6, "3D Graphics," focuses on converting the code created so far to work for a 3D game, including how to represent the view, projection, and rotations.

- Chapter 7, "Audio," covers how to bootstrap an audio system using the excellent FMOD API. It includes coverage of 3D positional audio.
- Chapter 8, "Input Systems," discusses how to design a more robust input system for processing keyboard, mouse, and game controller events.
- Chapter 9, "Cameras," shows how to implement several different 3D cameras, including a first-person camera, a follow camera, and an orbit camera.
- Chapter 10, "Collision Detection," dives into methods of collision detection for games, including spheres, planes, line segments, and boxes.
- Chapter 11, "User Interfaces," looks at implementing both a menu system and heads-up display (HUD) elements such as a radar and aiming reticule.
- Chapter 12, "Skeletal Animation," covers how to animate characters in 3D.
- Chapter 13, "Intermediate Graphics," explores a handful of intermediate graphics topics, including how to implement deferred shading.
- Chapter 14, "Level Files and Binary Data," discusses how to load and save level files, as well as how to write binary file formats.
- Appendix A, "Intermediate C++ Review," reviews several intermediate C++ topics used throughout the book including memory allocation and collections.

Each chapter includes a corresponding game project (with source code available, as mentioned), recommended additional readings, and a couple of exercises. These exercises generally instruct you to add additional features to the code implemented in the chapter.

Conventions Used in This Book

New terms appear in **bold**. Code appears in a `monospaced font`. Small snippets of code sometimes appear as standalone paragraphs:

```
DoSomething();
```

Longer code segments appear in code listings, as in Listing P.1.

Listing P.1 Sample Code Listing

```
void DoSomething()
{
   // Do the thing
   ThisDoesSomething();
}
```

From time to time, some paragraphs appear as notes, tips, sidebars, and warnings. Here is an example of each.

note

Notes contain some useful information about implementation changes or other features that are worth noting.

tip

Tips provide hints on how to add certain additional features to your code.

warning

Warnings call out specific pitfalls that warrant caution.

SIDEBAR

Sidebars are lengthier discussions that are tangential to the main content of the chapter. This content is interesting but isn’t crucial to understanding the core topics of the chapter.

Acknowledgments

Although this is not my first book, writing this one has been an especially long process. I am thankful that Laura Lewin, the executive editor on this book, was especially patient throughout the two years this book was in progress. I would also like to thank the rest of the team at Pearson, including Michael Thurston, the development editor on this book.

I would also like to acknowledge the work put in by the technical editors on this book: Josh Glazer, Brian Overland, and Matt Whiting. The technical reviews were critical in making sure both that the content was correct and that it was accessible for the target audience.

I'd also like to thank all my colleagues at the USC Information Technology Program and especially those who helped shape the curriculum of the games courses I teach: Josh Glazer, Jason Gregory, Clark Kromenaker, Mike Sheehan, and Matt Whiting. Much of the inspiration for this book comes from that curriculum. I would also like to thank all my excellent TAs over the years, who are too numerous to name personally.

I would also like to thank the content creators on sites like https://opengameart.org and https://freesound.org for creating excellent game content released under Creative Commons licenses. These sites were critical to finding assets for the game projects in this book.

Finally, I'd like to thank my parents, as well my sister, Nita, and her family. Without their support, inspiration, and guidance, I never would have gotten here in the first place. I'd also like to thank my friends, like Kevin, who understood when I couldn't go see the latest movie, go to dinner, or really do anything social because I was "working on my book." Well, I guess I have time now.…

About The Author

Sanjay Madhav is a senior lecturer at the University of Southern California, where he teaches several programming and video game programming courses. He has taught at USC since 2008.

Prior to joining USC, Sanjay worked as a programmer for several video game developers, including Electronic Arts, Neversoft, and Pandemic Studios. His credited games include *Medal of Honor: Pacific Assault*, *Tony Hawk's Project 8*, *Lord of the Rings: Conquest*, and *The Saboteur*.

Sanjay is also the author of *Game Programming Algorithms and Techniques* and co-author of *Multiplayer Game Programming*. He has a B.S. and an M.S. in computer science and is pursuing a Ph.D. in computer science, all from USC.

Chapter 1
Game Programming Overview

This chapter first discusses how to set up a development environment and access the source code for this book. Next, it covers the core concepts behind any real-time game: the game loop, how a game updates over time, and the basics of game input and output. Throughout the chapter, you will see how to implement code for a version of the classic game *Pong*.

Setting Up a Development Environment

Although it’s possible to write the source code for any program with a text editor, professional developers typically use an **integrated development environment** (**IDE**). The advantage of an IDE is that it provides code completion and debugging in addition to text editing capabilities. The code for this book works on both Microsoft Windows and Apple macOS, and the choice of IDE depends on the choice of platform. For Windows, this book uses Microsoft Visual Studio, and for macOS, it uses Apple Xcode. The remainder of this section contains brief instructions on setup of these environments on their respective platforms.

Microsoft Windows

For Windows development, the most popular IDE by far is Microsoft Visual Studio. Visual Studio also tends to be the most popular IDE for C++ game developers, with most PC and console developers gravitating toward the IDE.

This book uses Microsoft Visual Studio Community 2017, which is available as a free download at https://www.visualstudio.com/downloads/. Installation of Visual Studio Community 2017 requires Microsoft Windows 7 or higher.

When you run the installer program for Visual Studio, it asks which “workloads” it should install. Make sure to minimally select the Game Development with C++ workload. Feel free to also select any other workloads or options desired.

warning

THERE ARE DIFFERENT VERSIONS OF VISUAL STUDIO: There are several other products in the Microsoft Visual Studio suite, including Visual Studio Code and Visual Studio for Mac. Neither of these products are the same thing as Visual Studio Community 2017, so be careful to install the correct version!

Apple macOS

On macOS, Apple provides the free Xcode IDE for development of programs for macOS, iOS, and other related platforms. The code for this book works in both Xcode 8 and 9. Note that Xcode 8 requires macOS 10.11 El Capitan or higher, while Xcode 9 requires macOS 10.12 Sierra or higher.

To install Xcode, simply go to the Apple App Store and search for Xcode. The first time Xcode runs, it asks if you want to enable debugging features. Make sure to select Yes.

Getting This Book's Source Code

Most professional developers utilize **source control** systems, which, among many other features, keep a history of the source code. With such a system, if code changes cause unexpected or undesired behavior, it's easy to return to a previously known working version of code. Furthermore, source control allows for much easier collaboration between multiple developers.

One popular source control system is Git, originally developed by Linus Torvalds of Linux fame. In Git, the term **repository** refers to a specific project hosted under source control. The GitHub website (https://github.com) provides for easy creation and management of Git repositories.

The source code for this book is available on GitHub at https://github.com/gameprogcpp/code. If you are unfamiliar with the Git system, you can simply click the green Clone or Download button and choose Download ZIP to download a compressed ZIP file that contains all the book's source code.

Alternatively, if you wish to use Git, you can clone the repository via the command line, as follows:

Click here to view code image

```
$ git clone https://github.com/gameprogcpp/code.git
```

This command works out of the box in the macOS terminal, but Windows users need to first install Git for Windows (see https://git-for-windows.github.io).

The source code contains a separate directory (or folder) for each chapter. For example, this chapter's source code is in the `Chapter01` directory. In this directory, there is a `Chapter01-Windows.sln` file for Microsoft Visual Studio and a `Chapter01-Mac.xcodeproj` file for Apple Xcode. Before moving forward, make sure that you can compile the code for this chapter.

Beyond the C++ Standard Library

The C++ Standard Library only supports text console input and output and does not have any graphics libraries built in. To implement graphics in a C++ program, you must use one of the many available external libraries.

Unfortunately, many libraries are **platform specific**, meaning they work on only one operating system or type of computer. For example, the Microsoft Windows application programming interface (API) can create windows and other UI elements supported by the Windows operating system. However, the Windows API doesn't work on Apple macOS—for obvious reasons. Likewise, macOS has its own set of libraries for these same features that do not work on Windows. As a game programmer, you can't always avoid platform-specific libraries. For instance, game developers working with the Sony PlayStation 4 console must use libraries provided by Sony.

Luckily, this book sticks to **cross-platform** libraries, meaning that the libraries work on many different platforms. All the source code for this book works on recent versions of both Windows and macOS. Although Linux support is untested, the game projects generally should also work on Linux.

One of the foundational libraries used in this book is Simple DirectMedia Layer (SDL; see https://www.libsdl.org). The SDL library is a cross-platform game development library written in C. It provides support for creating windows, creating basic 2D graphics, processing input, and outputting audio, among other features. SDL is a very lightweight library that works on many platforms, including Microsoft Windows, Apple macOS, Linux, iOS, and Android.

In this first chapter, the only external library needed is SDL. Subsequent chapters use other libraries and introduce them when needed.

The Game Loop and Game Class

One of the big differences between a game and any other program is that a game must update many times per second for as long as the program runs. A **game loop** is a loop that controls the overall flow for the entire game program. Like any other loop, a game loop has code it executes on every iteration, and it has a loop condition. For a game loop, you want to continue looping as long as the player hasn't quit the game program.

Each iteration of a game loop is a **frame**. If a game runs at 60 **frames per second** (**FPS**), this means the game loop completes 60 iterations every second. Many real-time games run at 30 or 60 FPS. By running this many iterations per second, the game gives the illusion of continuous motion even though it's only updating at periodic intervals. The term **frame rate** is interchangeable with FPS; a frame rate of 60 means the same thing as 60 FPS.

Anatomy of a Frame

At a high level, a game performs the following steps on each frame:

1. It processes any inputs.
2. It updates the game world.
3. It generates any outputs.

Each of these three steps has more depth than may be apparent at first glance. For instance, processing inputs (step 1) clearly implies detecting any inputs from devices such as a keyboard, mouse, or controller. But these might not be the only inputs for a game. Consider a game that supports an online multiplayer mode. In this case, the game receives data over the Internet as an input. In certain types of mobile games, another input might be what's visible to the camera, or perhaps GPS information. Ultimately, the inputs to a game depend on both the type of game and the platform it runs on.

Updating a game world (step 2) means going through every object in the game world and updating it as needed. This could be hundreds or even thousands of objects, including characters in the game world, parts of the user interface, and other objects that affect the game—even if they are not visible.

For step 3, generating any outputs, the most apparent output is the graphics. But there are other outputs, such as audio (including sound effects, music, and dialogue). As another example, most console games have **force feedback** effects, such as the controller shaking when something exciting happens in the game. And for an online multiplayer game, an additional output would be data sent to the other players over the Internet.

Consider how this style of game loop might apply to a simplified version of the classic Namco arcade game *Pac-Man*. For this simplified version of the game, assume that the game immediately begins with Pac-Man in a maze. The game program continues running until Pac-Man either completes the maze or dies. In this case, the “process inputs” phase of the game loop need only read in the joystick input.

The “update game world” phase of the loop updates Pac-Man based on this joystick input and then also updates the four ghosts, pellets, and the user interface. Part of this update code must determine whether Pac-Man runs into any ghosts. Pac-Man can also 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, based on what Pac-Man is doing, the UI may need to update what data it displays.

note

This style of game loop is **single-threaded**, meaning it does not take advantage of modern CPUs that can execute multiple threads simultaneously. Making a game loop that supports multiple threads is very complex, and not necessary for games that are smaller in scope. A good book to learn more about multi-threaded game loops is Jason Gregory’s, listed in the “Additional Reading” section at the end of this chapter.

The only outputs in the “generate outputs” phase of the classic *Pac-Man* game are the audio and video. Listing 1.1 provides pseudocode showing what the game loop for this simplified version of *Pac-Man* might look like.

Listing 1.1 *Pac-Man* Game Loop Pseudocode

Click here to view code image

```
void Game::RunLoop()
{
   while (!mShouldQuit)
   {
      // Process Inputs
      JoystickData j = GetJoystickData();

     // Update Game World
       UpdatePlayerPosition(j);

     for (Ghost& g : mGhost)
       {
          if (g.Collides(player))
          {
             // Handle Pac-Man colliding with a ghost
          }
          else
          {
             g.Update();
          }
       }

     // Handle Pac-Man eating pellets
       // ...

     // Generate Outputs
       RenderGraphics();
       RenderAudio();
   }
}
```

Implementing a Skeleton Game Class

You are now ready to use your basic knowledge of the game loop to create a `Game` class that contains code to initialize and shut down the game as well as run the game loop. If you are rusty in C++, you might want to first review the content in Appendix A, "Intermediate C++ Review," as the remainder of this book assumes familiarity with C++. In addition, it may be helpful to keep this chapter's completed source code handy while reading along, as doing so will help you understand how all the pieces fit together.

Listing 1.2 shows the declaration of the `Game` class in the `Game.h` header file. Because this declaration references an `SDL_Window` pointer, you need to also include the main SDL header file `SDL/SDL.h`. (If you wanted to avoid including this here, you could use a forward declaration.) Many of the member function names are self-explanatory; for example, the `Initialize` function initializes the `Game` class, the `Shutdown` function shuts down the game, and the `RunLoop` function runs the game loop. Finally, `ProcessInput`, `UpdateGame`, and `GenerateOutput` correspond to the three steps of the game loop.

Currently, the only member variables are a pointer to the window (which you'll create in the `Initialize` function) and a bool that signifies whether the game should continue running the game loop.

Listing 1.2 `Game` Declaration

Click here to view code image

```
class Game
{
public:
   Game();
   // Initialize the game
   bool Initialize();
   // Runs the game loop until the game is over
   void RunLoop();
   // Shutdown the game
   void Shutdown();
private:
   // Helper functions for the game loop
   void ProcessInput();
   void UpdateGame();
   void GenerateOutput();

  // Window created by SDL
   SDL_Window* mWindow;
   // Game should continue to run
   bool mIsRunning;
};
```

With this declaration in place, you can start implementing the member functions in `Game.cpp`. The constructor simply initializes `mWindow` to `nullptr` and `mIsRunning` to `true`.

Game::Initialize

The `Initialize` function returns `true` if initialization succeeds and `false` otherwise. You need to initialize the SDL library with the `SDL_Init` function. This function takes in a single parameter, a bitwise-OR of all subsystems to initialize. For now, you only need to initialize the video subsystem, which you do as follows:

Click here to view code image

```
int sdlResult = SDL_Init(SDL_INIT_VIDEO);
```

Note that `SDL_Init` returns an integer. If this integer is nonzero, it means the initialization failed. In this case, `Game::Initialize` should return `false` because without SDL, the game cannot continue:

Click here to view code image

```
if (sdlResult != 0)
{
   SDL_Log("Unable to initialize SDL: %s", SDL_GetError());
   return false;
}
```

Using the `SDL_Log` function is a simple way to output messages to the console in SDL. It uses the same syntax as the C `printf` function, so it supports outputting variables to `printf` specifiers such as `%s` for a C-style string and `%d` for an integer. The `SDL_GetError` function returns an error message as a C-style string, which is why it's passed in as the `%s` parameter in this code.

SDL contains several different subsystems that you can initialize with `SDL_Init`. Table 1.1 shows the most commonly used subsystems; for the full list, consult the SDL API reference at https://wiki.libsdl.org.

Table 1.1 SDL Subsystem Flags of Note

Flag	**Subsystem**
`SDL_INIT_AUDIO`	Audio device management, playback, and recording
`SDL_INIT_VIDEO`	Video subsystem for creating a window, interfacing with OpenGL, and 2D graphics

`SDL_INIT_HAPTIC`	Force feedback subsystem
`SDL_INIT_GAMECONTROLLER`	Subsystem for supporting controller input devices

If SDL initializes successfully, the next step is to create a window with the `SDL_CreateWindow` function. This is just like the window that any other Windows or macOS program uses. The `SDL_CreateWindow` function takes in several parameters: the title of the window, the x/y coordinates of the top-left corner, the width/height of the window, and optionally any window creation flags:

Click here to view code image

```
mWindow = SDL_CreateWindow(
   "Game Programming in C++ (Chapter 1)", // Window title
   100,   // Top left x-coordinate of window
   100,   // Top left y-coordinate of window
   1024,  // Width of window
   768,   // Height of window
   0      // Flags (0 for no flags set)
);
```

As with the `SDL_Init` call, you should verify that `SDL_CreateWindow` succeeded. In the event of failure, `mWindow` will be `nullptr`, so add this check:

Click here to view code image

```
if (!mWindow)
{
   SDL_Log("Failed to create window: %s", SDL_GetError());
   return false;
}
```

As with the initialization flags, there are several possible window creation flags, as shown in Table 1.2. As before, you can use a bitwise-OR to pass in multiple flags. Although many commercial games use full-screen mode, it's faster to debug code if the game runs in windowed mode, which is why this book shies away from full screen.

Table 1.2 Window Creation Flags of Note

Flag	**Result**
`SDL_WINDOW_FULLSCREEN`	Use full-screen mode
`SDL_WINDOW_FULLSCREEN_DESKTOP`	Use full-screen mode at the current desktop resolution (and ignore width/height parameters to `SDL_CreateWindow`)
`SDL_WINDOW_OPENGL`	Add support for the OpenGL graphics library
`SDL_WINDOW_RESIZABLE`	Allow the user to resize the window

If SDL initialization and window creation succeeds, `Game::Initialize` returns `true`.

Game::Shutdown

The `Shutdown` function does the opposite of `Initialize`. It first destroys the `SDL_Window` with `SDL_DestroyWindow` and then closes SDL with `SDL_Quit`:

```
void Game::Shutdown()
{
   SDL_DestroyWindow(mWindow);
   SDL_Quit();
}
```

Game::RunLoop

The `RunLoop` function keeps running iterations of the game loop until `mIsRunning` becomes `false`, at which point the function returns. Because you have the three helper functions for each phase of the game loop, `RunLoop` simply calls these helper functions inside the loop:

Click here to view code image

```
void Game::RunLoop()
{
   while (mIsRunning)
   {
      ProcessInput();
      UpdateGame();
      GenerateOutput();
   }
}
```

For now, you won't implement these three helper functions, which means that once in the loop, the game won't do anything just yet. You'll continue to build on this `Game` class and implement these helper functions throughout the remainder of the chapter.

Main Function

Although the `Game` class is a handy encapsulation of the game's behavior, the entry point of any C++ program is the `main` function. You must implement a `main` function (in `Main.cpp`) as shown in Listing 1.3.

Listing 1.3 `main` Implementation

Click here to view code image

```
int main(int argc, char** argv)
{
   Game game;
   bool success = game.Initialize();
   if (success)
   {
      game.RunLoop();
   }
   game.Shutdown();
   return 0;
}
```

This implementation of `main` first constructs an instance of the `Game` class. It then calls `Initialize`, which returns `true` if the game successfully initializes, and `false` otherwise. If the game initializes, you then enter the game loop with the call to `RunLoop`. Finally, once the loop ends, you call `Shutdown` on the game.

With this code in place, you can now run the game project. When you do, you see a blank window, as shown in Figure 1.1 (though on macOS, this window may appear black instead of white). Of course, there's a problem: The game never ends! Because no code changes the `mIsRunning` member variable, the game loop never ends, and the `RunLoop` function never returns. Naturally, the next step is to fix this problem by allowing the player to quit the game.

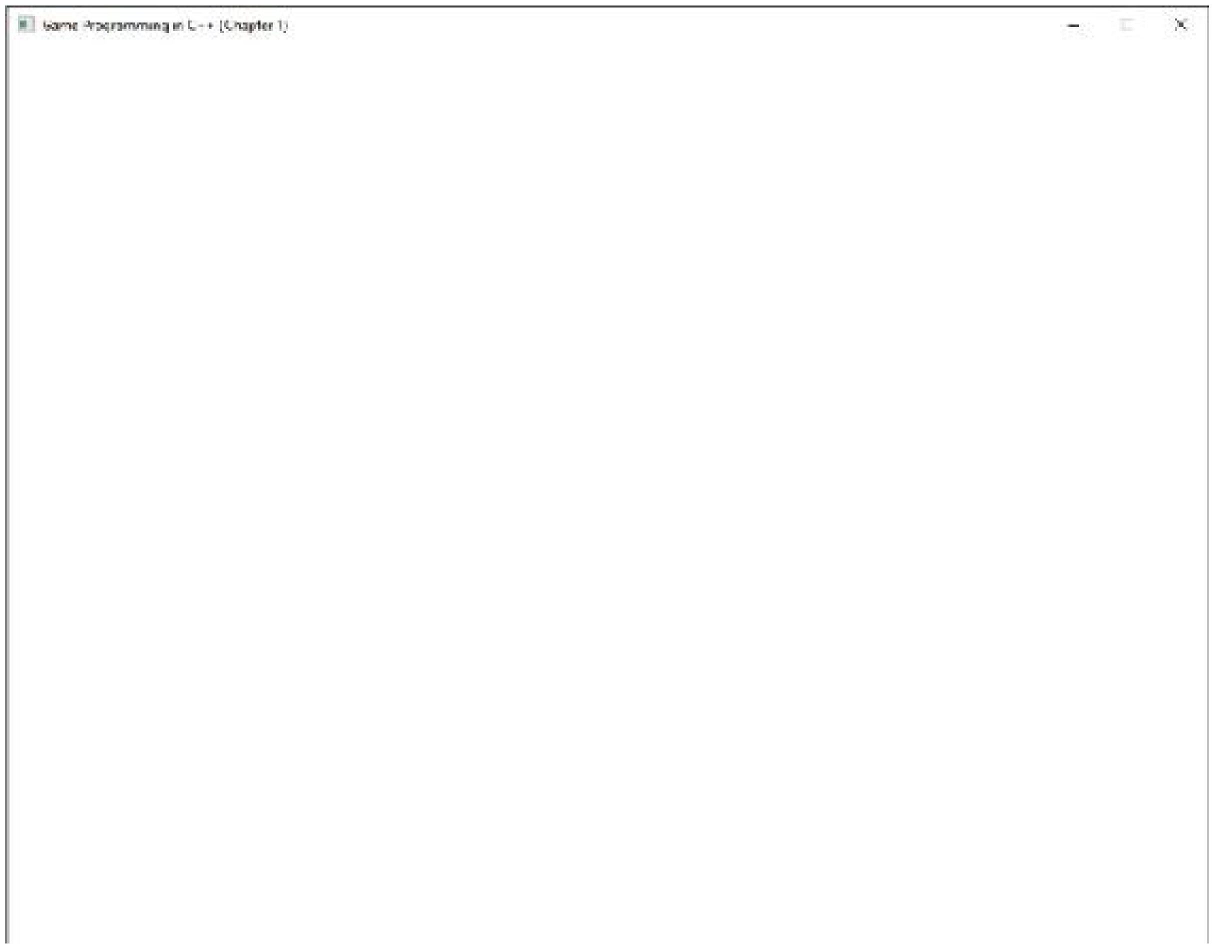

Figure 1.1 Creating a blank window

Basic Input Processing

In any desktop operating system, there are several actions that the user can perform on application windows. For example, the user can move a window, minimize or maximize a window, close a window (and program), and so on. A common way to represent these different actions is with events. When the user does something, the program receives events from the operating system and can choose to respond to these events.

SDL manages an internal queue of events that it receives from the operating system. This queue contains events for many different window actions, as well as events related to input devices. Every frame, the game must poll the queue for any events and choose either to ignore or process each event in the queue. For some events, such as moving the window around, ignoring the event means SDL will just automatically handle it. But for other events, ignoring the event means nothing will happen.

Because events are a type of input, it makes sense to implement event processing in `ProcessInput`. Because the event queue may contain multiple events on any given frame, you must loop over all events in the queue. The `SDL_PollEvent` function returns `true` if it finds an event in the queue. So, a very basic implementation of `ProcessInput` would keep calling `SDL_PollEvent` as long as it returns `true`:

Click here to view code image

```
void Game::ProcessInput()
{
   SDL_Event event;
   // While there are still events in the queue
   while (SDL_PollEvent(&event))
   {
   }
}
```

Note that the `SDL_PollEvent` function takes in an `SDL_Event` by pointer. This stores any information about the event just removed from the queue.

Although this version of `ProcessInput` makes the game window more responsive, the player still has no way to quit the game. This is because you simply remove all the events from the queue and don’t respond to them.

Given an `SDL_Event`, the `type` member variable contains the type of the event received. So, a common approach is to create a switch based on the type inside the `PollEvent` loop:

Click here to view code image

```
SDL_Event event;
while (SDL_PollEvent(&event))
{
   switch (event.type)
   {
      // Handle different event types here
   }
}
```

One useful event is `SDL_QUIT`, which the game receives when the user tries to close the window (either by clicking on the X or using a keyboard shortcut). You can update the code to set `mIsRunning` to `false` when it sees an `SDL_QUIT` event in the queue:

Click here to view code image

```
SDL_Event event;
while (SDL_PollEvent(&event))
{
   switch (event.type)
   {
      case SDL_QUIT:
         mIsRunning = false;
         break;
   }
}
```

Now when the game is running, clicking the X on the window causes the `while` loop inside `RunLoop` to terminate, which in turn shuts down the game and exits the program. But what if you want the game to quit when the user presses the Escape key? While you could check for a keyboard event corresponding to this, an easier approach is to grab the entire state of the keyboard with `SDL_GetKeyboardState`, which returns a pointer to an array that contains the current state of the keyboard:

Click here to view code image

```
const Uint8* state = SDL_GetKeyboardState(NULL);
```

Given this array, you can then query a specific key by indexing into this array with a corresponding `SDL_SCANCODE` value for the key. For example, the following sets `mIsRunning` to `false` if the user presses Escape:

Click here to view code image

```
if (state[SDL_SCANCODE_ESCAPE])
{
   mIsRunning = false;
}
```

Combining all this yields the current version of `ProcessInput`, shown in Listing 1.4. Now when running the game, the user can quit either by closing the window or pressing the Escape key.

Listing 1.4 `Game::ProcessInput` Implementation

Click here to view code image

```
void Game::ProcessInput()
{
   SDL_Event event;
   while (SDL_PollEvent(&event))
   {
      switch (event.type)
      {
         // If this is an SDL_QUIT event, end loop
         case SDL_QUIT:
            mIsRunning = false;
            break;
      }
   }


  // Get state of keyboard
   const Uint8* state = SDL_GetKeyboardState(NULL);
   // If escape is pressed, also end loop
   if (state[SDL_SCANCODE_ESCAPE])
   {
      mIsRunning = false;
   }
}
```

Basic 2D Graphics

Before you can implement the "generate outputs" phase of the game loop, you need some understanding of how 2D graphics work for games

Most displays in use today—whether televisions, computer monitors, tablets, or smartphones—use **raster graphics**, which means the display has a two-dimensional grid of picture elements (or **pixels**). These pixels can individually display different amounts of light as well as different colors. The intensity and color of these pixels combine to create a perception of a continuous image for the viewer. Zooming in on a part of a raster image makes each individual pixel discernable, as you can see in Figure 1.2.

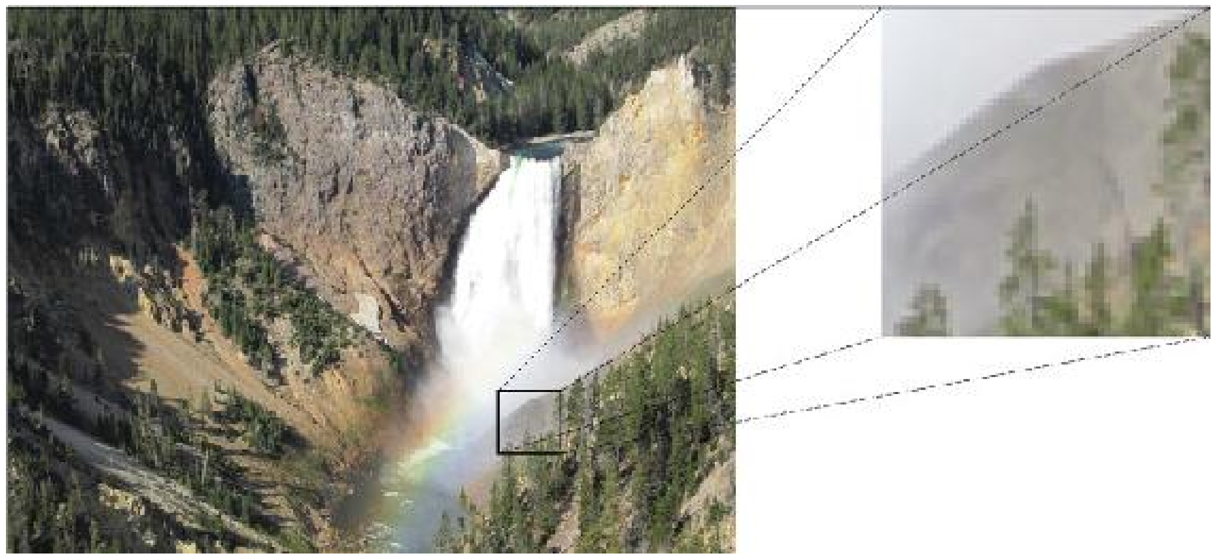

Figure 1.2 Zooming in on part of an image shows its distinct pixels

The **resolution** of a raster display refers to the width and height of the pixel grid. For example, a resolution of 1920×1080, commonly known as 1080p, means that there are 1080 rows of pixels, with each row containing 1920 pixels. Similarly, a resolution of 3840×2160, known as 4K, has 2160 rows with 3840 pixels per row.

Color displays mix colors additively to create a specific hue for each pixel. A common approach is to mix three colors together: red, green, and blue (abbreviated **RGB**). Different intensities of these RGB colors combine to create a range (or **gamut**) of colors. Although many modern displays also support color formats other than RGB, most video games output final colors in RGB. Whether or not RGB values convert to something else for display on the monitor is outside the purview of the game programmer.

However, many games *internally* use a different color representation for much of their graphics computations. For example, many games internally support transparency with an **alpha** value. The abbreviation **RGBA** references RGB colors with an additional alpha component. Adding an alpha component allows certain objects in a game, such as windows, to have some amount of transparency. But because few if any displays support transparency, the game ultimately needs to calculate a final RGB color and compute any perceived transparency itself.

The Color Buffer

For a display to show an RGB image, it must know the colors of each pixel. In computer graphics, the **color buffer** is a location in memory containing the color information for the entire screen. The display can use the color buffer for drawing the contents screen. Think of the color buffer as a two-dimensional array, where each (x, y) index corresponds to a pixel on the screen. In every frame during the "generate outputs" phase of the game loop, the game writes graphical output into the color buffer.

The memory usage of the color buffer depends on the number of bits that represent each pixel, called the **color depth**. For example, in the common 24-bit color depth, red, green, and blue each use 8 bits. This means there are 2^{24}, or 16,777,216, unique colors. If the game also wants to store an 8-bit alpha value, this results in a total of 32 bits for each pixel in the color buffer. A color buffer for a 1080p (1920×1080) target resolution with 32 bits per pixel uses 1920×1080×4 bytes, or approximately 7.9 MB.

note

Many game programmers also use the term **framebuffer** to reference the location in memory that contains the color data for a frame. However, a more precise definition of *framebuffer* is that it is the combination of the color buffer *and* other buffers (such as the depth buffer and stencil buffer). In the interest of clarity, this book references the specific buffers.

Some recent games use 16 bits per RGB component, which increases the number of unique colors. Of course, this doubles the memory usage of the color buffer, up to approximately 16 MB for 1080p. This may seem like an insignificant amount, given that most video cards have several gigabytes of video memory available. But when considering all the other memory usage of a cutting-edge game, 8 MB here and 8 MB there quickly adds up. Although most displays at this writing do not support 16 bits per color, some manufacturers now offer displays that support color depths higher than 8 bits per color.

Given an 8-bit value for a color, there are two ways to reference this value in code. One approach involves simply using an unsigned integer corresponding to the number of bits for each color (or **channel**). So, for a color depth with 8 bits per channel, each channel has a value between 0 and 255. The alternative approach is to normalize the integer over a decimal range from 0.0 to 1.0.

One advantage of using a decimal range is that a value yields roughly the same color, regardless of the underlying color depth. For example, the normalized RGB value (1.0, 0.0, 0.0) yields pure red whether the maximum value of red is 255 (8 bits per color) or 65,535 (16 bits per color). However, the unsigned integer RGB value (255, 0, 0) yields pure red only if there are 8 bits per color. With 16 bits per color, (255, 0, 0) is nearly black.

Converting between these two representations is straightforward. Given an unsigned integer value, divide it by the maximum unsigned integer value to get the normalized value. Conversely, given a normalized decimal value, multiply it by the maximum unsigned integer value to get an unsigned integer value. For now, you should use unsigned integers because the SDL library expects them.

Double Buffering

As mentioned earlier in this chapter, games update several times per second (at the common rates of 30 and 60 FPS). If a game updates the color buffer at the same rate, this gives the illusion of motion, much the way a flipbook appears to show an object in motion when you flip through the pages.

However, the **refresh rate**, or the frequency at which the display updates, may be different from the game's frame rate. For example, most NTSC TV displays have a refresh rate of 59.94 Hz, meaning they refresh very slightly less than 60 times per second. However, some newer computer monitors support a 144 Hz refresh rate, which is more than twice as fast.

Furthermore, no current display technology can instantaneously update the entire screen at once. There always is some update order—whether row by row, column by column, in a checkerboard, and so on. Whatever update pattern the display uses, it takes some fraction of a second for the whole screen to update.

Suppose a game writes to the color buffer, and the display reads from that same color buffer. Because the timing of the game's frame rate may not directly match the monitor’s refresh rate, it’s very like that the display will read from the color buffer while the game is writing to the buffer. This can be problematic.

For example, suppose the game writes the graphical data for frame A into the color buffer. The display then starts reading from the color buffer to show frame A on the screen. However, before the display finishes drawing frame A onto the screen, the game overwrites the color buffer with the graphical data for frame B. The display ends up showing part of frame A *and* part of frame B on the screen. Figure 1.3 illustrates this problem, known as **screen tearing**.

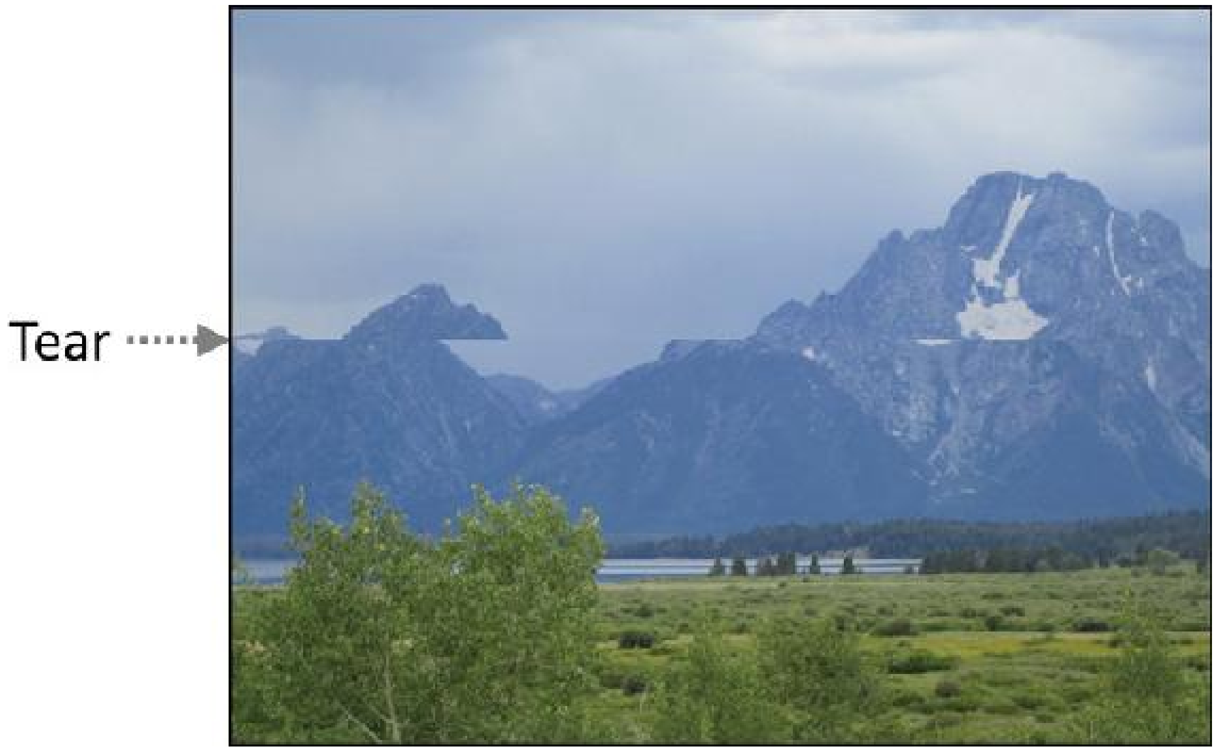

Figure 1.3 Simulation of screen tearing with a camera panning to the right

Eliminating screen tearing requires two changes. First, rather than having one color buffer that the game and display must share, you create two separate color buffers. Then the game and display alternate between the color buffers they use every frame. The idea is that with two separate buffers, the game can write to one (the **back buffer**) and, at the same time, the display can read from the other one (the **front buffer**). After the frame completes, the game and display swap their buffers. Due to the use of two color buffers, the name for this technique is **double buffering**.

As a more concrete example, consider the process shown in Figure 1.4. On frame A, the game writes its graphical output to buffer X, and the display draws buffer Y to the screen (which is empty). When this process completes, the game and display swap which buffers they use. Then on frame B, the game draws its graphical output to buffer Y, while the display shows buffer X on screen. On frame C, the game returns to buffer X, and the display returns to buffer Y. This swapping between the two buffers continues until the game program closes.

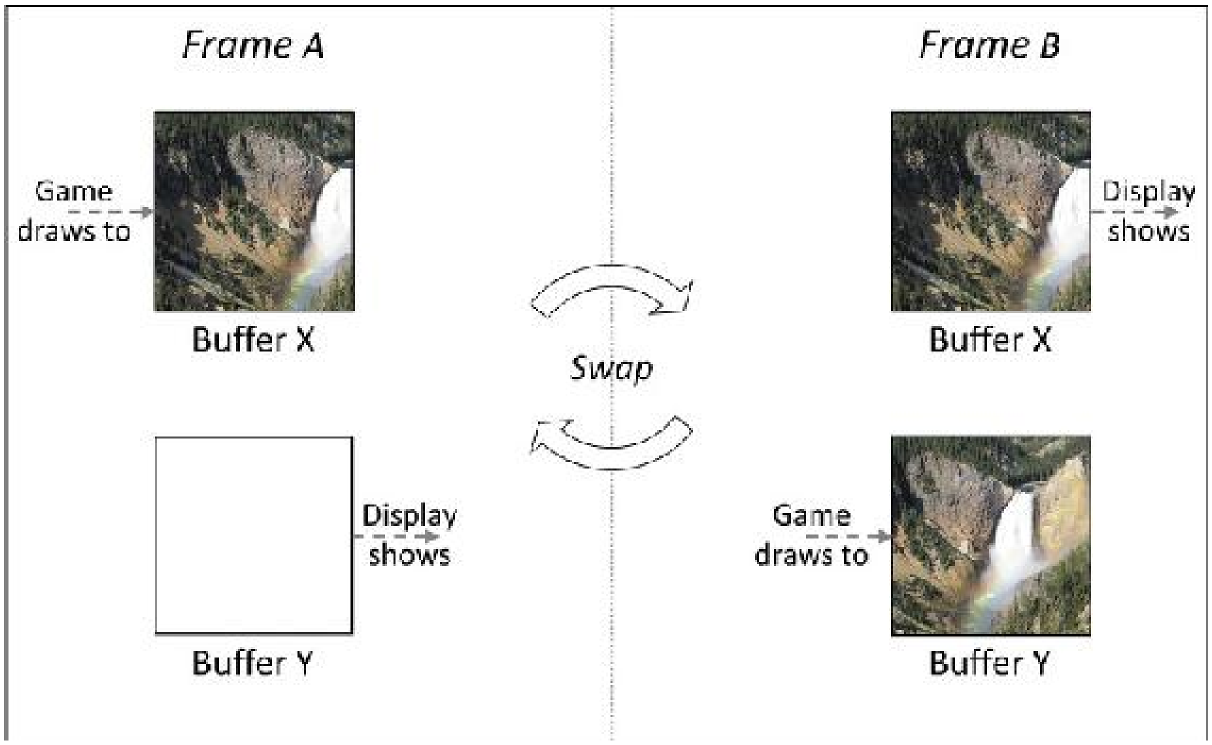

Figure 1.4 Double buffering involves swapping the buffers used by the game and display every frame

However, double buffering by itself does not eliminate screen tearing. Screen tearing still occurs if the display is drawing buffer X when the game wants to start writing to X. This usually happens only if the game is updating too quickly. The solution to this problem is to wait until the display finishes drawing its buffer before swapping. In other words, if the display is still drawing buffer X when the game wants to swap back to buffer X, the game must wait until the display finishes drawing buffer X. Developers call this approach **vertical**

synchronization, or **vsync**, named after the signal that monitors send when they are about to refresh the screen.

With vertical synchronization, the game might have to occasionally wait for a fraction of a second for the display to be ready. This means that the game loop may not be able to achieve its target frame rate of 30 or 60 FPS exactly. Some players argue that this causes unacceptable stuttering of the frame rate. Thus, the decision on whether to enable vsync varies depending on the game or player. A good idea is to offer vsync as an option in the engine so that you can choose between occasional screen tearing or occasional stuttering.

Recent advances in display technology seek to solve this dilemma with an **adaptive refresh rate** that varies based on the game. With this approach, rather than the display notifying the game when it refreshes, the game tells the display when to refresh. This way, the game and display are in sync. This provides the best of both worlds as it eliminates both screen tearing and frame rate stuttering. Unfortunately, at this writing, adaptive refresh technology is currently available only on certain high-end computer monitors.

Implementing Basic 2D Graphics

SDL has a simple set of functions for drawing 2D graphics. Because the focus of this chapter is 2D, you can stick with these functions. Starting in Chapter 5, “OpenGL,” you’ll switch to the OpenGL library for graphics, as it supports both 2D and 3D.

Initialization and Shutdown

To use SDL’s graphics code, you need to construct an `SDL_Renderer` via the `SDL_CreateRenderer` function. The term **renderer** generically refers to any system that draws graphics, whether 2D or 3D. Because you need to reference this `SDL_Renderer` object every time you draw something, first add an `mRenderer` member variable to `Game`:

```
SDL_Renderer* mRenderer;
```

Next, in `Game::Initialize`, after creating the window, create the renderer:

Click here to view code image

```
mRenderer = SDL_CreateRenderer(
   mWindow, // Window to create renderer for
   -1,      // Usually -1
   SDL_RENDERER_ACCELERATED | SDL_RENDERER_PRESENTVSYNC
);
```

The first parameter to `SDL_CreateRenderer` is the pointer to the window (which you saved in `mWindow`). The second parameter specifies which graphics driver to use; this might be relevant if the game has multiple windows. But with only a single window, the default is `-1`, which means to let SDL decide. As with the other SDL creation functions, the last parameter is for initialization flags. Here, you choose to use an accelerated renderer (meaning it takes advantage of graphics hardware) and enable vertical synchronization. These two flags are the only flags of note for `SDL_CreateRenderer`.

As with `SDL_CreateWindow`, the `SDL_CreateRenderer` function returns a `nullptr` if it fails to initialize the renderer. As with initializing SDL, `Game::Initialize` returns `false` if the renderer fails to initialize.

To shut down the renderer, simply add a call to `SDL_DestroyRenderer` in `Game::Shutdown`:

```
SDL_DestroyRenderer(mRenderer);
```

Basic Drawing Setup

At a high level, drawing in any graphics library for games usually involves the following steps:

1. Clear the back buffer to a color (the game's current buffer).
2. Draw the entire game scene.
3. Swap the front buffer and back buffer.

First, let's worry about the first and third steps. Because graphics are an output, it makes sense to put graphics drawing code in `Game::GenerateOutput`.

To clear the back buffer, you first need to specify a color with `SDL_SetRenderDrawColor`. This function takes in a pointer to the renderer, as well as the four RGBA components (from 0 to 255). For example, to set the color as blue with 100% opacity, use the following:

```
SDL_SetRenderDrawColor(
   mRenderer,
   0,    // R
   0,    // G
   255,  // B
   255   // A
);
```

Next, call `SDL_RenderClear` to clear the back buffer to the current draw color:

```
SDL_RenderClear(mRenderer);
```

The next step—skipped for now—is to draw the entire game scene.

Finally, to swap the front and back buffers, you call `SDL_RenderPresent`:

```
SDL_RenderPresent(mRenderer);
```

With this code in place, if you now run the game, you'll see a filled-in blue window, as shown in Figure 1.5.

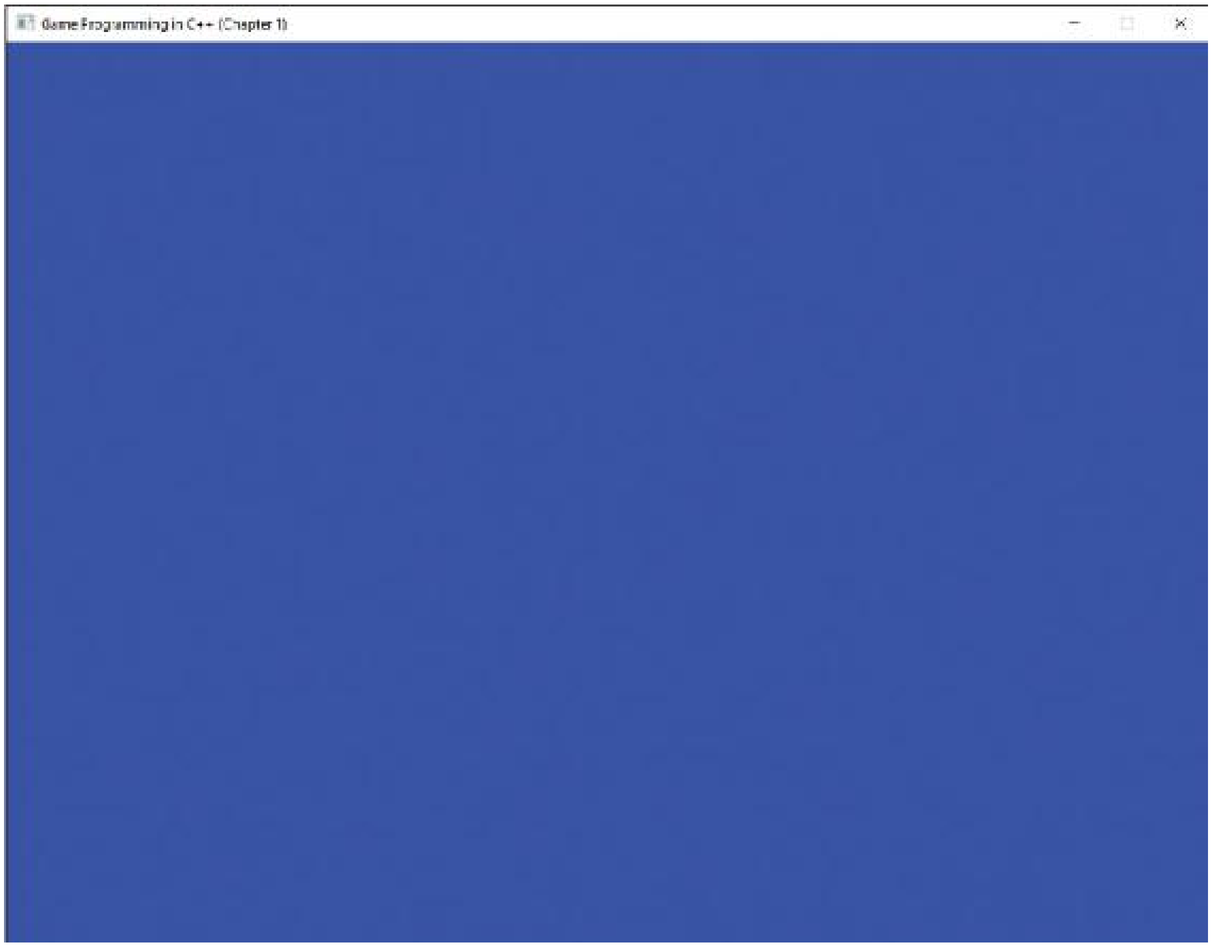

Figure 1.5 Game drawing a blue background

Drawing Walls, a Ball, and a Paddle

This chapter's game project is a version of the classic video game *Pong*, where a ball moves around the screen, and the player controls a paddle that can hit the ball. Making a version of *Pong* is a rite of passage for any aspiring game developer—analogous to making a "Hello World" program when first learning how to program. This section explores drawing rectangles to represent the objects in *Pong*. Because these are objects in the game world, you draw them in `GenerateOuput`—after clearing the back buffer but before swapping the front and back buffers.

For drawing filled rectangles, SDL has a `SDL_RenderFillRect` function. This function takes in an `SDL_Rect` that represents the bounds of the rectangle and draws a filled-in rectangle using the current draw color. Of course, if you keep the draw color the same as the background, you won't see any rectangles. You therefore need to change the draw color to white:

Click here to view code image

```
SDL_SetRenderDrawColor(mRenderer, 255, 255, 255, 255);
```

Next, to draw the rectangle, you need to specify dimensions via an `SDL_Rect` struct. The rectangle has four parameters: the x/y coordinates of the top-left corner of the rectangle onscreen, and the width/height of the rectangle. Keep in mind that in SDL rendering, as in many other 2D graphics libraries, the top-left corner of the screen is (0, 0), positive x is to the right, and positive y is down.

For example, if you want to draw a rectangle at the top of the screen, you can use the following declaration of an `SDL_Rect`:

```
SDL_Rect wall{
   0,         // Top left x
   0,         // Top left y
   1024,      // Width
   thickness // Height
};
```

Here, the x/y coordinates of the top-left corner are (0, 0), meaning the rectangle will be at the top left of the screen. You hard-code the width of the rectangle to 1024, corresponding to the width of the window. (It's generally frowned upon to assume a fixed window size, as is done here, and you'll remove this assumption in later chapters.) The `thickness` variable is `const int` set to `15`, which makes it easy to adjust the thickness of the wall.

Finally, you draw the rectangle with `SDL_RenderFillRect`, passing in `SDL_Rect` by pointer:

Click here to view code image

```
SDL_RenderFillRect(mRenderer, &wall);
```

The game then draws a wall in the top part of the screen. You can use similar code to draw the bottom wall and the right wall, only changing the parameters of the `SDL_Rect`. For example, the bottom wall could have the same rectangle as the top wall except that the top-left y coordinate could be `768 - thickness`.

Unfortunately, hard-coding the rectangles for the ball and paddle does not work because both objects will ultimately move in the `UpdateGame` stage of the loop. Although it makes some sense to represent both the ball and paddle as classes, this discussion doesn't happen until Chapter 2, "Game Objects and 2D Graphics." In the meantime, you can just use member variables to store the center positions of both objects and draw their rectangles based on these positions.

First, declare a simple `Vector2` struct that has both x and y components:

```
struct Vector2
{
   float x;
   float y;
};
```

For now, think of a vector (not a `std::vector`) as a simple container for coordinates. Chapter 3, "Vectors and Basic Physics," explores the topic of vectors in much greater detail.

Next, add two `Vector2`s as member variables to `Game`—one for the paddle position (`mPaddlePos`) and one for the ball's position (`mBallPos`). The game constructor then initializes these to sensible initial values: the ball position to the center of the screen and the paddle position to the center of the left side of the screen.

Armed with these member variables, you can then draw rectangles for the ball and paddle in `GenerateOutput`. However, keep in mind that the member variables represent the *center points* of the paddle and ball, while you define an `SDL_Rect` in terms of the *top-left point*. To convert from the center point to the top-left point, you simply subtract half the width/height from the x and y coordinates, respectively. For example, the following rectangle works for the ball:

Click here to view code image

```
SDL_Rect ball{
   static_cast<int>(mBallPos.x - thickness/2),
   static_cast<int>(mBallPos.y - thickness/2),
   thickness,
   thickness
};
```

The static casts here convert `mBallPos.x` and `mBallPos.y` from floats into integers (which `SDL_Rect` uses). In any event, you can make a similar calculation for drawing the paddle, except its width and height are different sizes.

With all these rectangles, the basic game drawing now works, as shown in Figure 1.6. The next step is to implement the `UpdateGame` phase of the loop, which moves the ball and paddle.

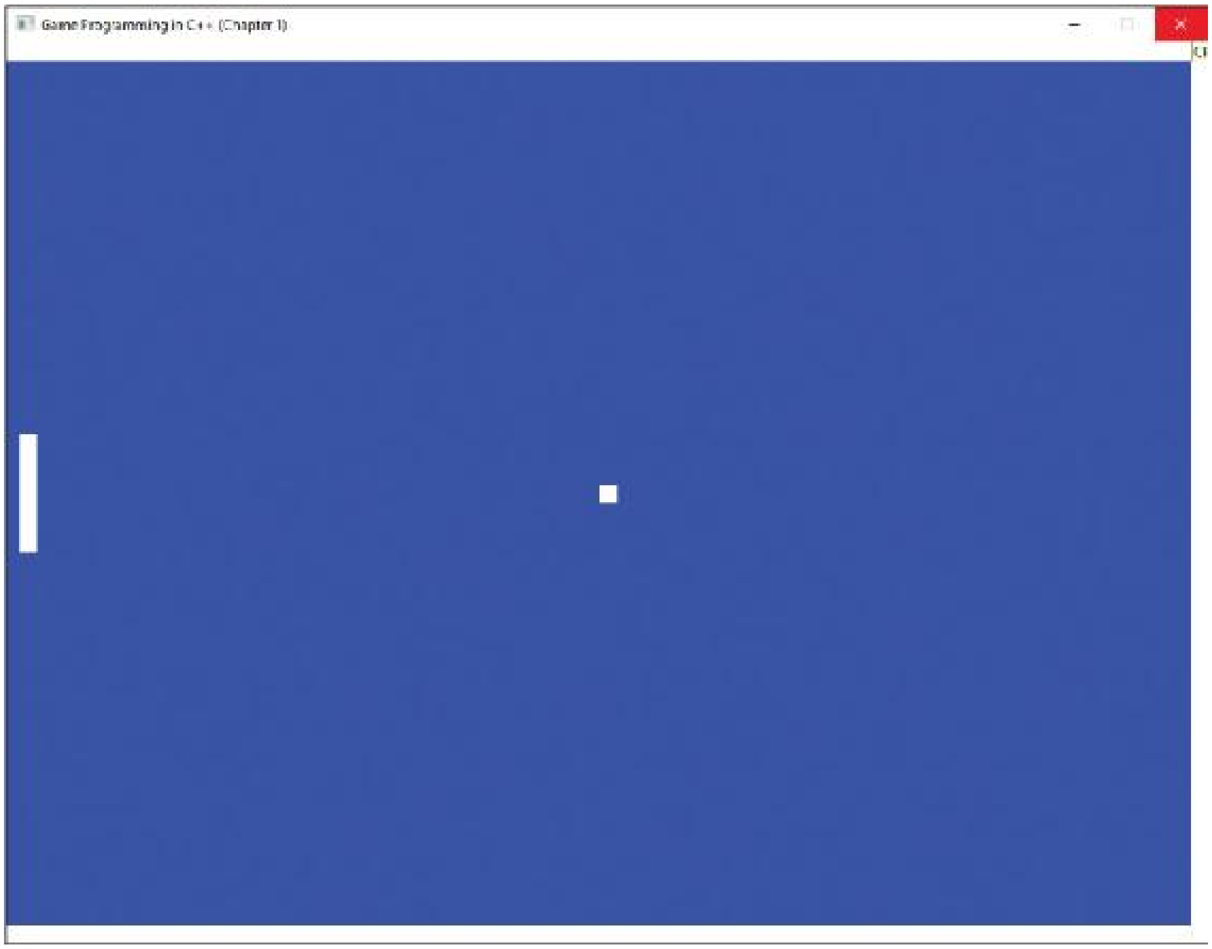

Figure 1.6 A game with walls, a paddle, and a ball drawing

Updating the Game

Most video games have some concept of time progression. For real-time games, you measure this progression of time in fractions of a second. For example, a game running at 30 FPS has roughly 33 milliseconds (ms) elapse from frame to frame. Remember that even though a game appears to feature continuous movement, it is merely an illusion. The game loop *actually* runs several times per second, and every iteration of the game loop updates the game in a discrete time step. So, in the 30 FPS example, each iteration of the game loop should simulate 33ms of time progression in the game. This section looks at how to consider this discrete progression of time when programming a game.

Real Time and Game Time

It is important to distinguish **real time**, the time elapsing in the real world, from **game time**, the time elapsing in the game's world. Although there often is a 1:1 correspondence between real time and game time, this isn't always the case. Take, for instance, a game in a paused state. Although a great deal of time might elapse in the real world, the game doesn't advance at all. It's not until the player unpauses the game that the game time resumes updating.

There are many other instances where real time and game time might diverge. For example, some games feature 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 each quarter takes only 7.5 minutes. And some games may even have time advance 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 ways real time and game time might diverge, it's clear that the "update game" phase of the game loop should account for elapsed game time.

Logic as a Function of Delta Time

Early game programmers assumed a specific processor speed and, therefore, a specific frame rate. The programmer might write the code assuming an 8 MHz processor, and if it worked properly for those processors, the code was just fine. When assuming a fixed frame rate, code that updates the position of an enemy might look something like this:

Click here to view code image

```
// Update x position by 5 pixels
enemy.mPosition.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, because the game loop now runs twice as fast, the enemy will now also *move* twice as fast. This could be the difference between a game that's challenging for players and one that's impossible. Imagine running this game on a modern processor that is thousands of times faster. The game would be over in a heartbeat!

To solve this issue, games use **delta time**: the amount of elapsed game time since the last frame. To convert the preceding code to using delta time, instead of thinking of movement as pixels per frame, you should think of it as pixels per second. So, if the ideal movement speed is 150 pixels per second, the following code is much more flexible:

Click here to view code image

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

Now the code will work well regardless of the frame rate. At 30 FPS, the delta time is ~0.033, so the enemy will move 5 pixels per frame, for a total of 150 pixels per second. At 60 FPS, the enemy will move only 2.5 pixels per frame but will still move a total of 150 pixels per second. The movement certainly will be smoother in the 60 FPS case, but the overall per-second speed remains the same.

Because this works across many frame rates, as a rule of thumb, everything in the game world should update as a function of delta time.

To help calculate delta time, SDL provides an `SDL_GetTicks` member function that returns the number of milliseconds elapsed since the `SDL_Init` function call. By saving the result of `SDL_GetTicks` from the previous frame in a member variable, you can use the current value to calculate delta time.

First, you declare an `mTicksCount` member variable (initializing it to zero in the constructor):

```
Uint32 mTicksCount;
```

Using `SDL_GetTicks`, you can then create a first implementation of `Game::UpdateGame`:

[Click here to view code image]

```
void Game::UpdateGame()
{
   // Delta time is the difference in ticks from last frame
   // (converted to seconds)
   float deltaTime = (SDL_GetTicks() - mTicksCount) / 1000.0f;
   // Update tick counts (for next frame)
   mTicksCount = SDL_GetTicks();


  // TODO: Update objects in game world as function of delta time!
   // ...
}
```

Consider what happens the very first time you call `UpdateGame`. Because `mTicksCount` starts at zero, you end up with some positive value of `SDL_GetTicks` (the milliseconds since initialization) and divide it by `1000.0f` to get a delta time in seconds. Next, you save the current value of `SDL_GetTicks` in `mTicksCount`. On the next frame, the `deltaTime` line calculates a new delta time based on the old value of `mTicksCount` and the new value. Thus, on every frame, you compute a delta time based on the ticks elapsed since the previous frame.

Although it may seem like a great idea to allow the game simulation to run at whatever frame rate the system allows, in practice there can be several issues with this. Most notably, any game that relies on physics (such as a platformer with jumping) will have differences in behavior based on the frame rate.

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, suppose that the target frame rate is 60 FPS. If a frame completes after only 15ms, frame limiting says to wait an additional ~1.6ms to meet the 16.6ms target time.

Conveniently, SDL also provides a method for frame limiting. For example, to ensure that at least 16ms elapses between frames, you can add the following code to the start of `UpdateGame`:

[Click here to view code image]

```
while (!SDL_TICKS_PASSED(SDL_GetTicks(), mTicksCount + 16))
   ;
```

You also must watch out for a delta time that's too high. Most notably, this happens when stepping through game code in the debugger. For example, if you pause at a breakpoint in the debugger for five seconds, you'll end up with a huge delta time, and everything will jump far forward in the simulation. To fix this problem, you can clamp the delta time to a maximum value (such as `0.05f`). This way, the game simulation will never jump too far forward on any one frame. This yields the version of `Game::UpdateGame` in Listing 1.5. While you aren't updating the position of the paddle or ball just yet, you are at least calculating the delta time value.

Listing 1.5 `Game::UpdateGame` Implementation

Click here to view code image

```
void Game::UpdateGame()
{
   // Wait until 16ms has elapsed since last frame
   while (!SDL_TICKS_PASSED(SDL_GetTicks(), mTicksCount + 16))
      ;

  // Delta time is the difference in ticks from last frame
   // (converted to seconds)
   float deltaTime = (SDL_GetTicks() - mTicksCount) / 1000.0f;


  // Clamp maximum delta time value
   if (deltaTime > 0.05f)
   {
      deltaTime = 0.05f;
   }

  // TODO: Update objects in game world as function of delta time!
}
```

Updating the Paddle's Position

In *Pong*, the player controls the position of the paddle based on input. Suppose you want the `W` key to move the paddle up and the `S` key to move the paddle down. Pressing neither key *or* both keys should mean the paddle doesn't move at all.

You can make this concrete by using a `mPaddleDir` integer member variable that's set to `0` if the paddle doesn't move, `-1` if if the paddle moves up (negative y), and `1` if the paddle moves down (positive y).

Because the player controls the position of the paddle via keyboard input, you need code in `ProcessInput` that updates `mPaddleDir` based on the input:

```
mPaddleDir = 0;
if (state[SDL_SCANCODE_W])
{
   mPaddleDir -= 1;
}
if (state[SDL_SCANCODE_S])
{
   mPaddleDir += 1;
}
```

Note how you add and subtract from `mPaddleDir`, which ensures that if the player presses both keys, `mPaddleDir` is zero.

Next, in `UpdateGame`, you can add code that updates the paddle based on delta time:

Click here to view code image

```
if (mPaddleDir != 0)
{
   mPaddlePos.y += mPaddleDir * 300.0f * deltaTime;
}
```

Here, you update the y position of the paddle based on the paddle direction, a speed of `300.0f` pixels/second, and delta time. If `mPaddleDir` is `-1`, the paddle will move up, and if it's `1`, it'll move down.

One problem is that this code allows the paddle to move off the screen. To fix this, you can add boundary conditions for the paddle's y position. If the position is too high or too low, move it back to a valid position:

Click here to view code image

```
if (mPaddleDir != 0)
{
   mPaddlePos.y += mPaddleDir * 300.0f * deltaTime;
   // Make sure paddle doesn't move off screen!
   if (mPaddlePos.y < (paddleH/2.0f + thickness))
```

```
    {
       mPaddlePos.y = paddleH/2.0f + thickness;
    }
    else if (mPaddlePos.y > (768.0f - paddleH/2.0f - thickness))
    {
       mPaddlePos.y = 768.0f - paddleH/2.0f - thickness;
    }
}
```

Here, the `paddleH` variable is a constant that describes the height of the paddle. With this code in place, the player can now move the paddle up and down, and the paddle can't move offscreen.

Updating the Ball's Position

Updating the position of the ball is a bit more complex than updating the position of the paddle. First, the ball travels in both the x and y directions, not just in one direction. Second, the ball needs to bounce off the walls and paddles, which changes the direction of travel. So you need to both represent the **velocity** (speed and direction) of the ball and perform **collision detection** to determine if the ball collides with a wall.

To represent the ball's velocity, add another `Vector2` member variable called `mBallVel`. Initialize `mBallVel` to `(-200.0f, 235.0f)`, which means the ball starts out moving −200 pixels/second in the x direction and 235 pixels/second in the y direction. (In other words, the ball moves diagonally down and to the left.)

To update the position of the ball in terms of the velocity, add the following two lines of code to `UpdateGame`:

Click here to view code image

```
mBallPos.x += mBallVel.x * deltaTime;
mBallPos.y += mBallVel.y * deltaTime;
```

This is like updating the paddle's position, except now you are updating the position of the ball in *both* the x *and* y directions.

Next, you need code that bounces the ball off walls. The code for determining whether the ball collides with a wall is like the code for checking whether the paddle is offscreen. For example, the ball collides with the top wall if its y position is less than or equal to the height of the ball.

The important question is: what to do when the ball collides with the wall? For example, suppose the ball moves upward and to the right before colliding against the top wall. In this case, you want the ball to now start moving downward and to the right. Similarly, if the ball hits the bottom wall, you want the ball to start moving upward. The insight is that bouncing off the top or bottom wall negates the y component of the velocity, as shown in Figure 1.7(a). Similarly, colliding with the paddle on the left or wall on the right should negate the x component of the velocity.

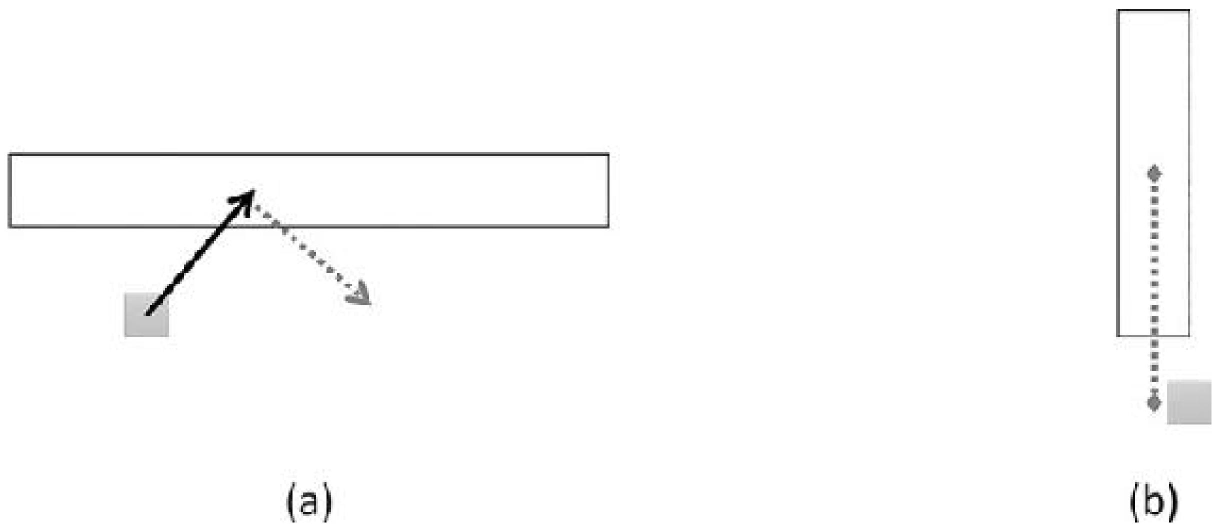

Figure 1.7 (a) The ball collides with the top wall so starts moving down. (b) The y difference between the ball and paddle is too large

For the case of the top wall, this yields code like the following:

```
if (mBallPos.y <= thickness)
{
   mBallVel.y *= -1;
}
```

However, there’s a key problem with this code. Suppose the ball collides with the top wall on frame A, so the code negates the y velocity to make the ball start moving downward. On frame B, the ball tries to move away from the wall, but it doesn’t move far enough. Because the ball still collides with the wall, the code negates the y velocity *again*, which means the ball starts moving upward. Then on every subsequent frame, the code keeps negating the ball’s y velocity, so it is forever stuck on the top wall.

To fix this issue of the ball getting stuck, you need an additional check. You want to only negate the y velocity if the ball collides with the top wall *and* the ball is moving toward the top wall (meaning the y velocity is negative):

Click here to view code image

```
if (mBallPos.y <= thickness && mBallVel.y < 0.0f)
{
   mBallVel.y *= -1;
}
```

This way, if the ball collides with the top wall but is moving away from the wall, you do not negate the y velocity.

The code for colliding against the bottom and right walls is very similar to the code for colliding against the top wall. Colliding against the paddle, however, is slightly more complex. First, you calculate the absolute value of the difference between the y position of the ball and the y position of the paddle. If this difference is greater than half the height of the paddle, the ball is too high or too low, as shown earlier in Figure 1.7(b). You also need to check that the ball's x-position lines up with the paddle, and the ball is not trying to move away from the paddle. Satisfying all these conditions means the ball collides with the paddle, and you should negate the x velocity:

Click here to view code image

```
if (
    // Our y-difference is small enough
    diff <= paddleH / 2.0f &&
    // Ball is at the correct x-position
    mBallPos.x <= 25.0f && mBallPos.x >= 20.0f &&
    // The ball is moving to the left
    mBallVel.x < 0.0f)
{
   mBallVel.x *= -1.0f;
}
```

With this code complete, the ball and paddle now both move onscreen, as in Figure 1.8. You have now completed your simple version of *Pong*!

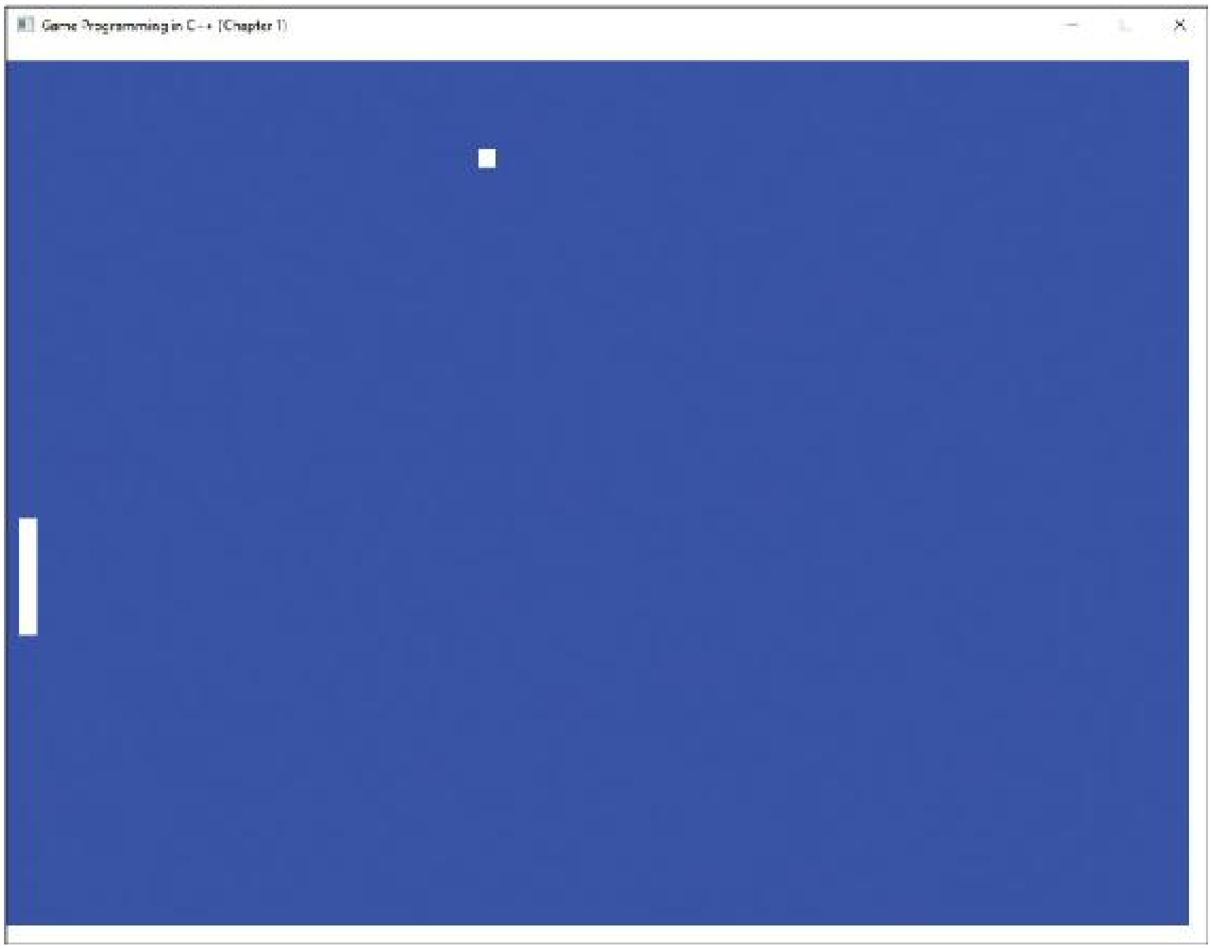

Figure 1.8 Final version of *Pong*

Game Project

This chapter’s game project implements the full *Pong* game code constructed throughout the chapter. To control the paddle, the player uses the `W` and `S` keys. The game ends when the ball moves offscreen. The code is available in the book’s GitHub repository in the `Chapter01` directory. Open `Chapter01-windows.sln` in Windows and `Chapter01-mac.xcodeproj` on Mac. (For instructions on how to access the GitHub repository, consult the instructions at the beginning of this chapter.)

Summary

Real-time games update many times per second via a loop called the game loop. Each iteration of this loop is a frame. For example, 60 frames per second means that there are 60 iterations of the game loop per second. The game loop has three main phases that it completes every frame: processing input, updating the game world, and generating output. Input involves not only input devices such as the keyboard and mouse but networking data, replay data, and so on. Outputs include graphics, audio, and force feedback controllers.

Most displays use raster graphics, where the display contains a grid of pixels. The size of the grid depends on the resolution of the display. The game maintains a color buffer that saves color data for every pixel. Most games use double buffering, where there are two color buffers, and the game and display alternate between using these buffers. This helps reduce the amount of screen tearing (that is, the screen showing parts of two frames at once). To eliminate screen tearing, you also must enable vertical synchronization, which means the buffers swap only when the display is ready.

For a game to work properly at variable frame rates, you need to write all game logic as a function of delta time—the time interval between frames. Thus, the “update game world” phase of the game loop should account for delta time. You can further add frame limiting to ensure that the frame rate does not go over some set cap.

In this chapter, you have combined all these different techniques to create a simple version of the classic video game *Pong*.

Additional Reading

Jason Gregory dedicates several pages to discussing the different formulations of a game loop, including how some games take better advantage of multi-core CPUs. There are also many excellent references online for the various libraries used; for example, the SDL API reference is handy.

Gregory, Jason. *Game Engine Architecture,* 2nd edition. Boca Raton: CRC Press, 2014.

SDL API Reference. https://wiki.libsdl.org/APIByCategory. Accessed June 15, 2016.

Exercises

Both of this chapter's exercises focus on modifying your version of *Pong*. The first exercise involves adding a second player, and the second exercise involves adding support for multiple balls.

Exercise 1.1

The original version of *Pong* supported two players. Remove the right wall onscreen and replace that wall with a second paddle for player 2. For this second paddle, use the `I` and `K` keys to move the paddle up and down. Supporting a second paddle requires duplicating all the functionality of the first paddle: a member variable for the paddle's position, the direction, code to process input for player 2, code that draws the paddle, and code that updates the paddle. Finally, make sure to update the ball collision code so that the ball correctly collides with both paddles.

Exercise 1.2

Many pinball games support "multiball," where multiple balls are in play at once. It turns out multiball is also fun for *Pong*! To support multiple balls, create a `Ball` struct that contains two `Vector2`s: one for the position and one for the velocity. Next, create a `std::vector<Ball>` member variable for `Game` to store these different balls. Then change the code in `Game::Initialize` to initialize the positions and velocities of several balls. In `Game::UpdateGame`, change the ball update code so that rather than using the individual `mBallVel` and `mBallPos` variables, the code loops over the `std::vector` for all the balls.

Chapter 2
Game Objects and 2D Graphics

Most games have many different characters and other objects, and an important decision is how to represent these objects. This chapter first covers different methods of object representation. Next, it continues the discussion of 2D graphics techniques by introducing sprites, sprite animations, and scrolling backgrounds. This chapter culminates with a side-scrolling demo that applies the covered techniques.

Game Objects

The *Pong* game created in Chapter 1 does not use separate classes to represent the wall, paddles, and ball. Instead, the `Game` class uses member variables to track the position and velocity of the different elements of the game. While this can work for a very simple game, it's not a scalable solution. The term **game object** refers to anything in the game world that updates, draws, or both updates *and* draws. There are several methods to represent game objects. Some games employ object hierarchies, others employ composition, and still others utilize more complex methods. Regardless of the implementation, a game needs some way to track and update these game objects.

Types of Game Objects

A common type of game object is one that's both updated every frame during the "update game world" phase of the loop and drawn every frame during the "generate outputs" phase. Any character, creature, or otherwise movable object falls under this umbrella. For example, in *Super Mario Bros.*, Mario, any enemies, and all the dynamic blocks are game objects that the game both updates and draws.

Developers sometimes use the term **static object** for game objects that draw but don't update. These objects are visible to the player but never need to update. An example of this is a building in the background of a level. In most games, a building doesn't move or attack the player but is visible onscreen.

A camera is an example of a game object that updates but doesn't draw to the screen. Another example is a **trigger**, which causes something to occur based on another object's intersection. For instance, a horror game might want to have zombies appear when the player approaches a door. In this case, the level designer would place a trigger object that can detect when the player is near and *trigger* the action to spawn the zombie. One way to implement a trigger is as an invisible box that updates every frame to check for intersection with the player.

Game Object Models

There are numerous **game object models**, or ways to represent game objects. This section discusses some types of game object models and the trade-offs between these approaches.

Game Objects as a Class Hierarchy

One game object model approach is to declare game objects in a standard object-oriented class hierarchy, which is sometimes called a **monolithic class hierarchy** because all game objects inherit from one base class.

To use this object model, you first need a base class:

Click here to view code image

```
class Actor
{
public:
   // Called every frame to update the Actor
   virtual void Update(float deltaTime);
   // Called every frame to draw the Actor
   virtual void Draw();
};
```

Then, different characters have different subclasses:

Click here to view code image

```
class PacMan : public Actor
{
public:
   void Update(float deltaTime) override;
   void Draw() override;
};
```

Similarly, you could declare other subclasses of `Actor`. For example, there may be a `Ghost` class that inherits from `Actor`, and then each individual ghost could have its own class that inherits from `Ghost`. Figure 2.1 illustrates this style of game object class hierarchy.

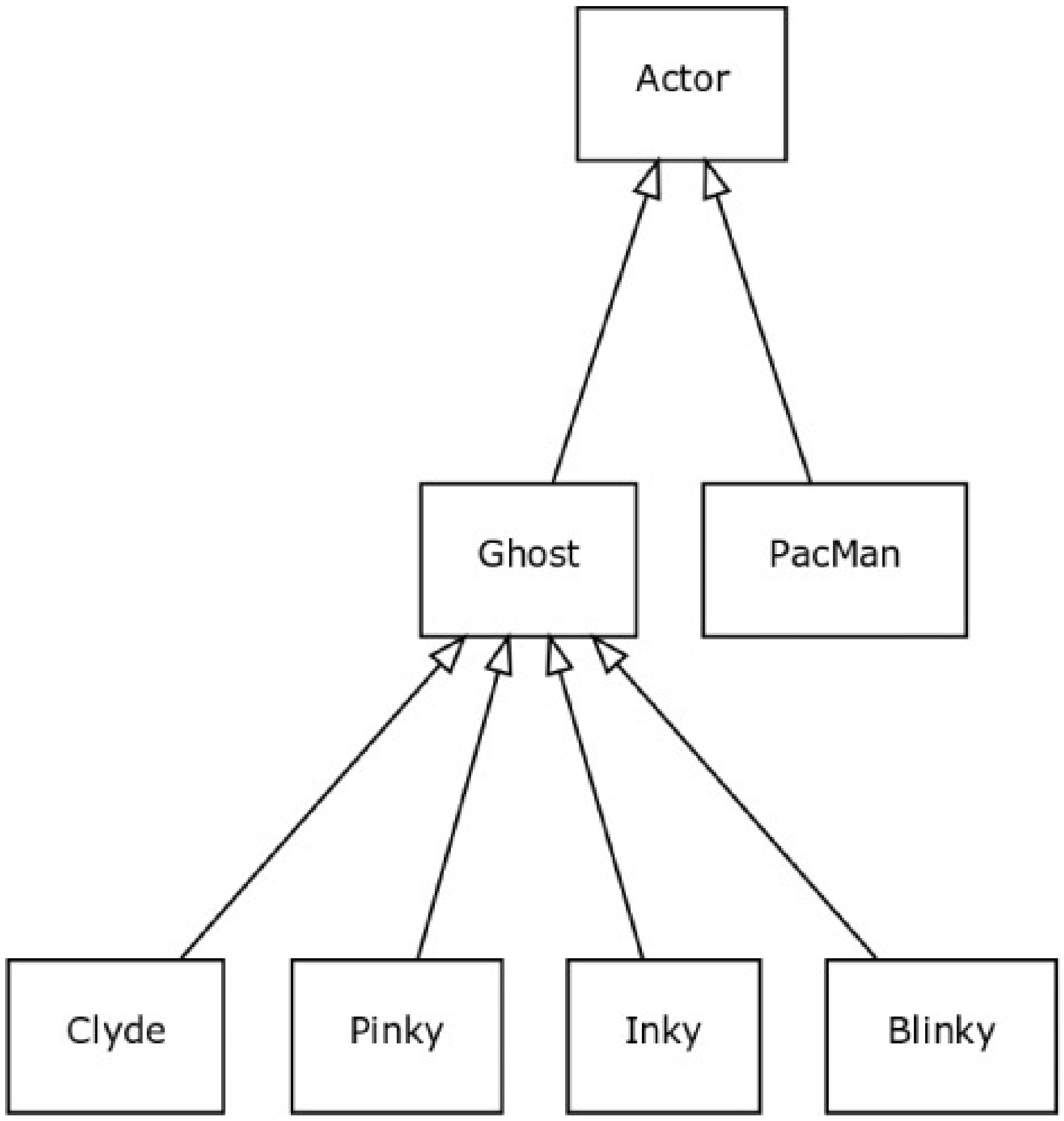

Figure 2.1 Partial class hierarchy for *Pac-Man*

A disadvantage of this approach is that it means that every game object must have all the properties and functions of the base game object (in this case, `Actor`). For example, this assumes that every `Actor` can update and draw. But as discussed, there may be objects that aren't visible, and thus calling `Draw` on these objects is a waste of time.

The problem becomes more apparent as the functionality of the game increases. Suppose many of the actors in the game—but not all of them—need to move. In the case of *Pac-Man*, the ghosts and Pac-Man need to move, but the pellets do not. One approach is to place the movement code inside `Actor`, but not every subclass will need this code. Alternatively, you could extend the hierarchy with a new `MovingActor` that exists between `Actor` and any subclasses that need movement. However, this adds more complexity to the class hierarchy.

Furthermore, having one big class hierarchy can cause difficulties when two sibling classes later need to have features shared between them. For instance, a game in the vein of *Grand Theft Auto* might have a base `Vehicle` class. From this class, it might make sense to create two subclasses: `LandVehicle` (for vehicles that traverse land) and `WaterVehicle` (for water-based vehicles like boats).

But what happens if one day a designer decides to add an amphibious vehicle? It may be tempting to create a new subclass called `AmphibiousVehicle` that inherits from both `LandVehicle` and `WaterVehicle`. However, this requires use of multiple inheritance and, furthermore, means that `AmphibiousVehicle` inherits from `Vehicle` along two different paths. This type of hierarchy, called **diamond inheritance**, can cause issues because the subclass might inherit multiple versions of a virtual function. For this reason, it's recommended that we avoid diamond hierarchies.

Game Objects with Components

Instead of using a monolithic hierarchy, many games instead use a **component-based** game object model. This model has become increasingly popular, especially because the Unity game engine uses it. In this approach, there is a game object class, but there are no subclasses of the game object. Instead, the game object class *has-a* collection of component objects that implement needed functionality.

For example, in the monolithic hierarchy we looked at earlier, `Pinky` is a subclass of `Ghost`, which is a subclass of `Actor`. However, in a component-based model, Pinky is a `GameObject` instance containing four components: `PinkyBehavior`, `CollisionComponent`, `TransformComponent`, and `DrawComponent`. Figure 2.2 shows this relationship.

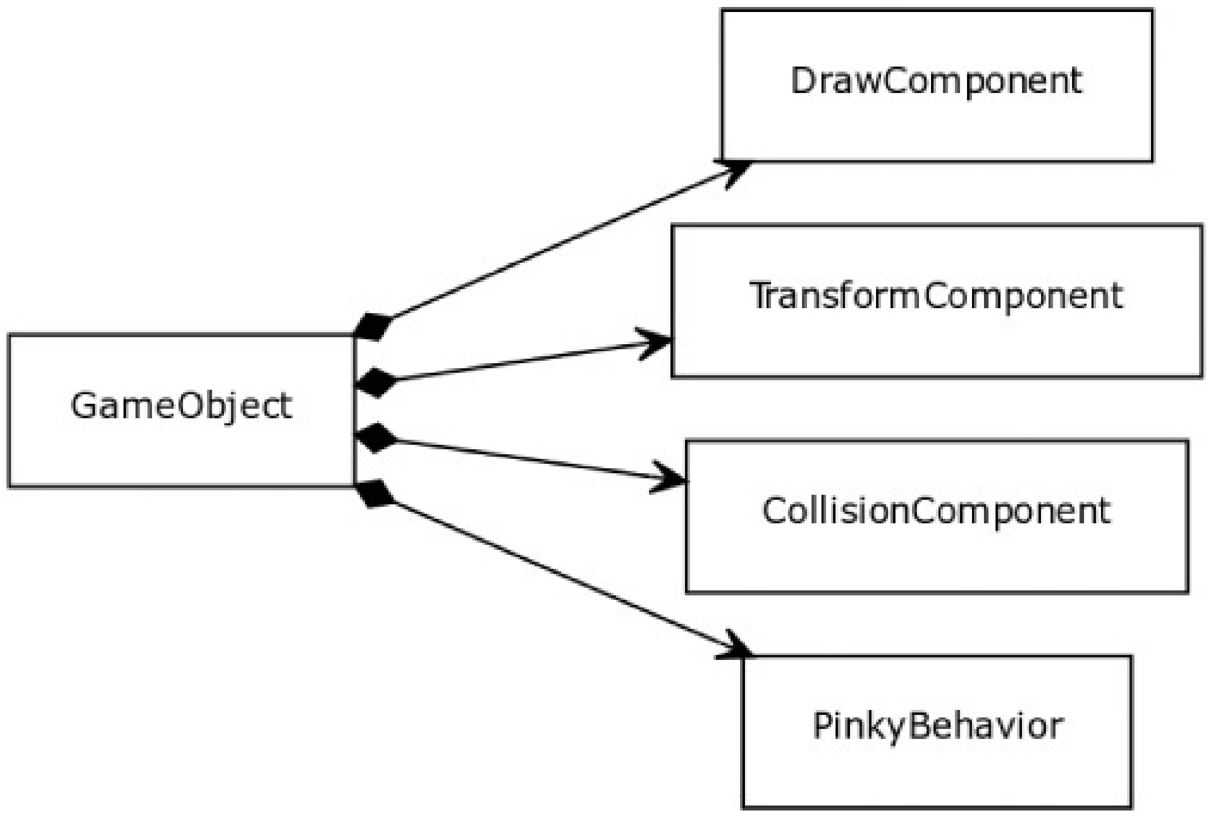

Figure 2.2 The components that make up the ghost Pinky

Each of these components has the specific properties and functionality needed for that component. For example, `DrawComponent` handles drawing the object to the screen, and `TransformComponent` stores the position and transformation of an object in the game world.

One way to implement a component object model is with a class hierarchy for components. This class hierarchy generally has a very shallow depth. Given a base `Component` class, `GameObject` then simply has a collection of components:

Click here to view code image

```
class GameObject
{
public:
   void AddComponent(Component* comp);
   void RemoveComponent(Component* comp);
private:
   std::unordered_set<Component*> mComponents;
};
```

Notice that `GameObject` contains only functions for adding and removing components. This makes systems that track different types of components necessary. For example, every `DrawComponent` might register with a `Renderer` object so that when it is time to draw the frame, the `Renderer` is aware of all the active `DrawComponents`.

One advantage of the component-based game object model is that it's easier to add functionality only to the specific game objects that require it. Any object that needs to draw can have a `DrawComponent`, but objects that don't simply don't have one.

However, a disadvantage of pure component systems is that dependencies between components in the same game object are not clear. For instance, it's likely that the `DrawComponent` needs to know about the `TransformComponent` in order know where the object should draw. This means that the `DrawComponent` likely needs to ask the owning `GameObject` about its `TransformComponent`. Depending on the implementation, the querying can become a noticeable performance bottleneck.

Game Objects as a Hierarchy with Components

The game object model used in this book is a hybrid of the monolithic hierarchy and the component object models. This is, in part, inspired by the game object model used in Unreal Engine 4. There is a base `Actor` class with a handful of virtual functions, but each actor also has a vector of components. Listing 2.1 shows the declaration of the `Actor` class, with some getter and setter functions omitted.

Listing 2.1 `Actor` Declaration

Click here to view code image

```
class Actor
{
public:
   // Used to track state of actor
   enum State
   {
      EActive,
      EPaused,
      EDead
   };
   // Constructor/destructor
   Actor(class Game* game);
   virtual ~Actor();

  // Update function called from Game (not overridable)
   void Update(float deltaTime);
   // Updates all the components attached to the actor (not
overridable)
   void UpdateComponents(float deltaTime);
   // Any actor-specific update code (overridable)
   virtual void UpdateActor(float deltaTime);

  // Getters/setters
   // ...


  // Add/remove components
   void AddComponent(class Component* component);
   void RemoveComponent(class Component* component);
private:
   // Actor's state
   State mState;
   // Transform
   Vector2 mPosition; // Center position of actor
   float mScale;      // Uniforms scale of actor (1.0f for 100%)
   float mRotation;   // Rotation angle (in radians)
   // Components held by this actor
   std::vector<class Component*> mComponents;
   class Game* mGame;
};
```

The `Actor` class has several notable features. The state `enum` tracks the status of the actor. For example, `Update` only updates the actor when in the `EActive` state. The `EDead` state notifies the game to remove the actor. The `Update` function calls `UpdateComponents` first and then `UpdateActor`. `UpdateComponents` loops over all the components and updates each in turn. The base implementation of `UpdateActor` is empty, but `Actor` subclasses can write custom behavior in an overridden `UpdateActor` function.

In addition, the `Actor` class needs access to the `Game` class for several reasons, including to create additional actors. One approach is to make the game object a **singleton**, a design pattern in which there is a single, globally accessible instance of a class. But the singleton pattern can cause issues if it turns out there actually need to be multiple instances of the class. Instead of using singletons, this book uses an approach called **dependency injection**. In this approach, the actor constructor receives a pointer to the `Game` class. Then an actor can use this pointer to create another actor (or access any other required Game functions).

As in the Chapter 1 game project, a Vector2 represents the position of an Actor. Actors also support a scale (which makes the actor bigger or smaller) and a rotation (to rotate the actor). Note that the rotation is in radians, not degrees.

Listing 2.2 contains the declaration of the Component class. The mUpdateOrder member variable is notable. It allows certain components to update before or after other components, which can be useful in many situations. For instance, a camera component tracking a player may want to update after the movement component moves the player. To maintain this ordering, the AddComponent function in Actor sorts the component vector whenever adding a new component. Finally, note that the Component class has a pointer to the owning actor. This is so that the component can access the transform data and any other information it deems necessary.

Listing 2.2 Component Declaration

Click here to view code image

```
class Component
{
public:
   // Constructor
   // (the lower the update order, the earlier the component updates)
   Component(class Actor* owner, int updateOrder = 100);
   // Destructor
   virtual ~Component();
   // Update this component by delta time
   virtual void Update(float deltaTime);
   int GetUpdateOrder() const { return mUpdateOrder; }
protected:
   // Owning actor
   class Actor* mOwner;
   // Update order of component
   int mUpdateOrder;
};
```

Currently, the implementations of Actor and Component do not account for player input devices, and this chapter’s game project simply uses special case code for input. Chapter 3, “Vector and Basic Physics,” revisits how to incorporate input into the hybrid game object model.

This hybrid approach does a better job of avoiding the deep class hierarchies in the monolithic object model, but certainly the depth of the hierarchy is greater than in a pure component-based model. The hybrid approach also generally avoids, though does not eliminate, the issues of communication overhead between components. This is because every actor has critical properties such as the transform data.

Other Approaches

There are many other approaches to game object models. Some use interface classes to declare the different possible functionalities, and each game object then implements the interfaces necessary to represent it. Other approaches extend the component model a step further and eliminate the containing game object entirely. Instead, these approaches use a component database that tracks objects with a numeric identifier. Still other approaches define objects by their properties. In these systems, adding a health property to an object means that it can take and receive damage.

With any game object model, each approach has advantages and disadvantages. However, this book sticks to the hybrid approach because it’s a good compromise and works relatively well for games of a certain complexity level.

Integrating Game Objects into the Game Loop

Integrating the hybrid game object model into the game loop requires some code, but it ultimately isn’t that complex. First, add two `std::vector` of `Actor` pointers to the `Game` class: one containing the active actors (`mActors`), and one containing pending actors (`mPendingActors`). You need the pending actors vector to handle the case where, while updating the actors (and thus looping over `mActors`), you decide to create a new actor. In this case, you cannot add an element to `mActors` because you’re iterating over it. Instead, you need to add to the `mPendingActors` vector and then move these actors into `mActors` after you’re done iterating.

Next, create two functions, `AddActor` and `RemoveActor`, which take in `Actor` pointers. The `AddActor` function adds the actor to either mPendingActors or mActors, depending on whether you are currently updating all mActors (denoted by the mUpdatingActors bool):

Click here to view code image

```
void Game::AddActor(Actor* actor)
{
   // If  updating actors, need to add to pending
   if (mUpdatingActors)
   {
      mPendingActors.emplace_back(actor);
   }
   else
   {
      mActors.emplace_back(actor);
   }
}
```

Similarly, RemoveActor removes the actor from whichever of the two vectors it is in.

You then need to change the UpdateGame function so that it updates all the actors, as shown in Listing 2.3. After computing delta time, as discussed in Chapter 1, you then loop over every actor in mActors and call Update on each. Next, you move any pending actors into the main mActors vector. Finally, you see if any actors are dead, in which case you delete them.

Listing 2.3 Game::UpdateGame Updating Actors

Click here to view code image

```
void Game::UpdateGame()
{
   // Compute delta time (as in Chapter 1)
   float deltaTime = /* ... */;


  // Update all actors
   mUpdatingActors = true;
   for (auto actor : mActors)
   {
      actor->Update(deltaTime);
   }
   mUpdatingActors = false;

  // Move any pending actors to mActors
   for (auto pending : mPendingActors)
   {
      mActors.emplace_back(pending);
   }
   mPendingActors.clear();

  // Add any dead actors to a temp vector
   std::vector<Actor*> deadActors;
   for (auto actor : mActors)
   {
      if (actor->GetState() == Actor::EDead)
      {
         deadActors.emplace_back(actor);
      }
   }

  // Delete dead actors (which removes them from mActors)
   for (auto actor : deadActors)
   {
      delete actor;
   }
}
```

Adding and removing actors from the game's `mActors` vector also adds some complexity to the code. This chapter's game project has the `Actor` object automatically add or remove itself from the game in its constructor and destructor, respectively. However, this means that code that loops over the `mActors` vector and deletes the actors (such as in `Game::Shutdown`) must be written carefully:

Click here to view code image

```
// Because ~Actor calls RemoveActor, use a different style loop
while (!mActors.empty())
{
   delete mActors.back();
}
```

Sprites

A **sprite** is a visual object in a 2D game, typically used to represent characters, backgrounds, and other dynamic objects. Most 2D games have dozens if not hundreds of sprites, and for mobile games, the sprite data accounts for much of the overall download size of the game. Because of the prevalence of sprites in 2D games, it is important to use them as efficiently as possible.

Each sprite has one or more image files associated with it. There are many different image file formats, and games use different formats based on platform and other constraints. For example, PNG is a compressed image format, so these files take up less space on disk. But hardware cannot natively draw PNG files, so they take longer to load. Some platforms recommend using graphics hardware–friendly formats such as PVR (for iOS) and DXT (for PC and Xbox). This book sticks with the PNG file format because image editing programs universally support PNGs.

Loading Image Files

For games that only need SDL's 2D graphics, the simplest way to load image files is to use the SDL Image library. The first step is to initialize SDL Image by using `IMG_Init`, which takes in a flag parameter for the requested file formats. To support PNG files, add the following call to `Game::Initialize`:

```
IMG_Init(IMG_INIT_PNG)
```

Table 2.1 lists the supported file formats. Note that SDL already supports the BMP file format without SDL Image, which is why there is no `IMG_INIT_BMP` flag in this table.

Table 2.1 SDL Image File Formats

Flag	**Format**
`IMG_INIT_JPG`	JPEG
`IMG_INIT_PNG`	PNG
`IMG_INIT_TIF`	TIFF

Once SDL Image is initialized, you can use `IMG_Load` to load an image file into an `SDL_Surface`:

Click here to view code image

```
// Loads an image from a file
// Returns a pointer to an SDL_Surface if successful, otherwise
```

```
nullptr
SDL_Surface* IMG_Load(
   const char* file // Image file name
);
```

Next, `SDL_CreateTextureFromSurface` converts an `SDL_Surface` into an `SDL_Texture` (which is what SDL requires for drawing):

Click here to view code image

```
// Converts an SDL_Surface to an SDL_Texture
// Returns a pointer to an SDL_Texture if successful, otherwise
nullptr
SDL_Texture* SDL_CreateTextureFromSurface(
   SDL_Renderer* renderer, // Renderer used
   SDL_Surface* surface    // Surface to convert
);
```

The following function encapsulates this image-loading process:

Click here to view code image

```
SDL_Texture* LoadTexture(const char* fileName)
{
   // Load from file
   SDL_Surface* surf = IMG_Load(fileName);
   if (!surf)
   {
      SDL_Log("Failed to load texture file %s", fileName);
      return nullptr;
   }
   // Create texture from surface
   SDL_Texture* text = SDL_CreateTextureFromSurface(mRenderer, surf);
   SDL_FreeSurface(surf);
   if (!text)
   {
      SDL_Log("Failed to convert surface to texture for %s",
fileName);
      return nullptr;
   }
   return text;
}
```

An interesting question is where to store the loaded textures. It’s very common for a game to use the same image file multiple times for multiple different actors. If there are 20 asteroids, and each asteroid uses the same image file, it doesn’t make sense to load the file from disk 20 times.

A simple approach is to create a map of filenames to `SDL_Texture` pointers in `Game`. You then create a `GetTexture` function in `Game` that takes in the name of a texture and returns its corresponding `SDL_Texture` pointer. This function first checks to see if the texture already exists in the map. If it does, you can simply return that texture pointer. Otherwise, you run the code that loads the texture from its file.

> **note**
>
> While a map of filenames to `SDL_Texture` pointers makes sense in a simple case, consider that a game has many different types of assets—textures, sound effects, 3D models, fonts, and so on. Therefore, writing a more robust system to generically handle all types of assets makes sense. But in the interest of simplicity, this book does not implement such an asset management system.

To help split up responsibilities, you also create a `LoadData` function in `Game`. This function is responsible for creating all the actors in the game world. For now these actors are hard-coded, but Chapter 14, "Level Files and Binary Data," adds support for loading the actors from a level file. You call the `LoadData` function at the end of `Game::Initialize`.

Drawing Sprites

Suppose a game has a basic 2D scene with a background image and a character. A simple way to draw this scene is by first drawing the background image and then the character. This is like how a painter would paint the scene, and hence this approach is known as the **painter's algorithm**. In the painter's algorithm, the game draws the sprites in back-to-front order. Figure 2.3 demonstrates the painter's algorithm, first drawing the background star field, then the moon, then any asteroids, and finally the ship.

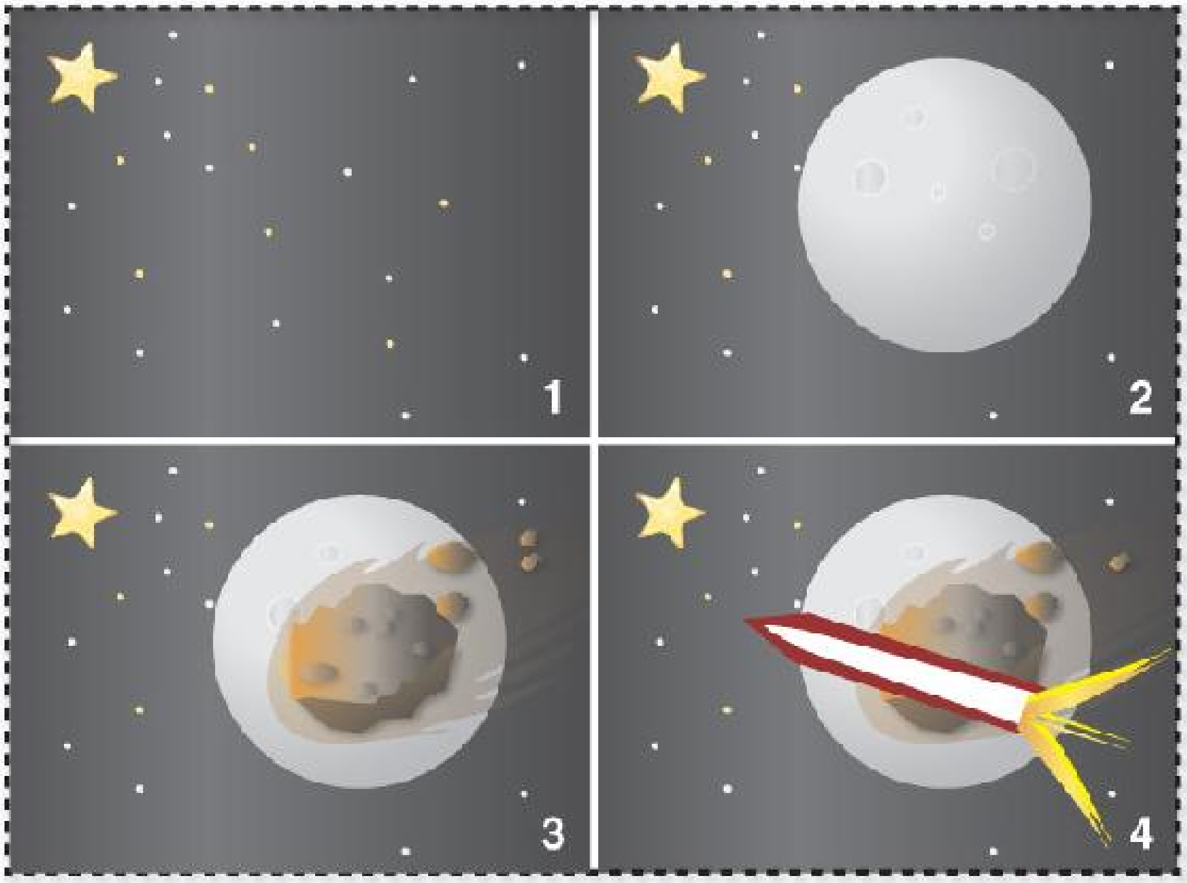

Figure 2.3 The painter's algorithm applied to a space scene

Because this book uses a game object model with components, it makes a great deal of sense to create a `SpriteComponent` class. Listing 2.4 provides the declaration of `SpriteComponent`.

Listing 2.4 `SpriteComponent` Declaration

Click here to view code image

```
class SpriteComponent : public Component
{
public:
   // (Lower draw order corresponds with further back)
   SpriteComponent(class Actor* owner, int drawOrder = 100);
   ~SpriteComponent();
   virtual void Draw(SDL_Renderer* renderer);
   virtual void SetTexture(SDL_Texture* texture);

   int GetDrawOrder() const { return mDrawOrder; }
   int GetTexHeight() const { return mTexHeight; }
   int GetTexWidth() const { return mTexWidth; }
protected:
   // Texture to draw
   SDL_Texture* mTexture;
   // Draw order used for painter's algorithm
   int mDrawOrder;
   // Width/height of texture
   int mTexWidth;
   int mTexHeight;
};
```

The game implements the painter's algorithm by drawing sprite components in the order specified by the `mDrawOrder` member variable. The `SpriteComponent` constructor adds itself to a vector of sprite components in the `Game` class via the `Game::AddSprite` function.

In `Game::AddSprite`, you need to ensure that `mSprites` stays sorted by draw order. Because every call to `AddSprite` preserves the sorted order, you can implement this as an insertion into an already-sorted vector:

Click here to view code image

```
void Game::AddSprite(SpriteComponent* sprite)
{
   // Find the insertion point in the sorted vector
   // (The first element with a higher draw order than me)
   int myDrawOrder = sprite->GetDrawOrder();
   auto iter = mSprites.begin();
   for ( ;
      iter != mSprites.end();
      ++iter)
   {
      if (myDrawOrder < (*iter)->GetDrawOrder())
      {
         break;
      }
   }

   // Inserts element before position of iterator
   mSprites.insert(iter, sprite);
}
```

Because this orders the sprite components by `mDrawOrder`, `Game::GenerateOutput` can just loop over the vector of sprite components and call `Draw` on each. You put this code in between the code that clears the back buffer and swaps the back buffer and front buffer, replacing the code in the Chapter 1 game that drew the wall, ball, and paddle rectangles.

As discussed in Chapter 6, "3D Graphics," 3D games can also use the painter's algorithm, though there are some drawbacks in doing so. But for 2D scenes, the painter's algorithm works very well.

The `SetTexture` function both sets the `mTexture` member variable and uses `SDL_QueryTexture` to get the width and height of the texture:

Click here to view code image

```
void SpriteComponent::SetTexture(SDL_Texture* texture)
{
   mTexture = texture;
   // Get width/height of texture
   SDL_QueryTexture(texture, nullptr, nullptr,
      &mTexWidth, &mTexHeight);
}
```

To draw textures, there are two different texture drawing functions in SDL. The simpler function is `SDL_RenderCopy`:

Click here to view code image

```
// Renders a texture to the rendering target
// Returns 0 on success, negative value on failure
int SDL_RenderCopy(
   SDL_Renderer* renderer,  // Render target to draw to
   SDL_Texture* texture,    // Texture to draw
   const SDL_Rect* srcrect, // Part of texture to draw (null if whole)
   const SDL_Rect* dstrect, // Rectangle to draw onto the target
);
```

However, for more advanced behavior (such as rotating sprites), you can use `SDL_RenderCopyEx`:

Click here to view code image

```
// Renders a texture to the rendering target
// Returns 0 on success, negative value on failure
int SDL_RenderCopyEx(
   SDL_Renderer* renderer,  // Render target to draw to
   SDL_Texture* texture,    // Texture to draw
   const SDL_Rect* srcrect, // Part of texture to draw (null if whole)
   const SDL_Rect* dstrect, // Rectangle to draw onto the target
   double angle,            // Rotation angle (in degrees, clockwise)
   const SDL_Point* center, // Point to rotate about (nullptr for
center)
   SDL_RenderFlip flip,     //  How to flip texture (usually
SDL_FLIP_NONE)
);
```

Because actors have a rotation, and you want your sprites to inherit this rotation, you must use `SDL_RenderCopyEx`. This introduces a few complexities to the `SpriteComponent::Draw` function. First, the `SDL_Rect` struct's x/y coordinates correspond to the top-left corner of the destination. However, the actor's position variable specifies the center position of the actor. So, as with the ball and paddle in Chapter 1, you must compute the coordinates for the top-left corner.

Second, SDL expects an angle in degrees, but `Actor` uses an angle in radians. Luckily, this book's custom math library in the `Math.h` header file includes a `Math::ToDegrees` function that can handle the conversion. Finally, in SDL a positive angle is clockwise, but this is the opposite of the unit circle (where positive angles are counterclockwise). Thus, negate the angle to maintain the unit circle behavior. Listing 2.5 shows the `SpriteComponent::Draw` function.

Listing 2.5 `SpriteComponent::Draw` Implementation

Click here to view code image

```
void SpriteComponent::Draw(SDL_Renderer* renderer)
{
   if (mTexture)
   {
      SDL_Rect r;
      // Scale the width/height by owner's scale
      r.w = static_cast<int>(mTexWidth * mOwner->GetScale());
      r.h = static_cast<int>(mTexHeight * mOwner->GetScale());
      // Center the rectangle around the position of the owner
      r.x = static_cast<int>(mOwner->GetPosition().x - r.w / 2);
      r.y = static_cast<int>(mOwner->GetPosition().y - r.h / 2);

     // Draw
      SDL_RenderCopyEx(renderer,
         mTexture, // Texture to draw
         nullptr,  // Source rectangle
         &r,       // Destination rectangle
         -Math::ToDegrees(mOwner->GetRotation()), // (Convert angle)
```

```
            nullptr, // Point of rotation
            SDL_FLIP_NONE); // Flip behavior
    }
}
```

This implementation of `Draw` assumes that the position of the actor corresponds to its position onscreen. This assumption holds only for games where the game world exactly corresponds to the screen. This doesn't work for a game like *Super Mario Bros.* that has a game world larger than one single screen. To handle such a case, the code needs a camera position. Chapter 9, "Cameras," discusses how to implement cameras in the context of a 3D game.

Animating Sprites

Most 2D games implement sprite animation using a technique like **flipbook animation**: a series of static 2D images played in rapid succession to create an illusion of motion. Figure 2.4 illustrates what such a series of images for different animations for a skeleton sprite might look like.

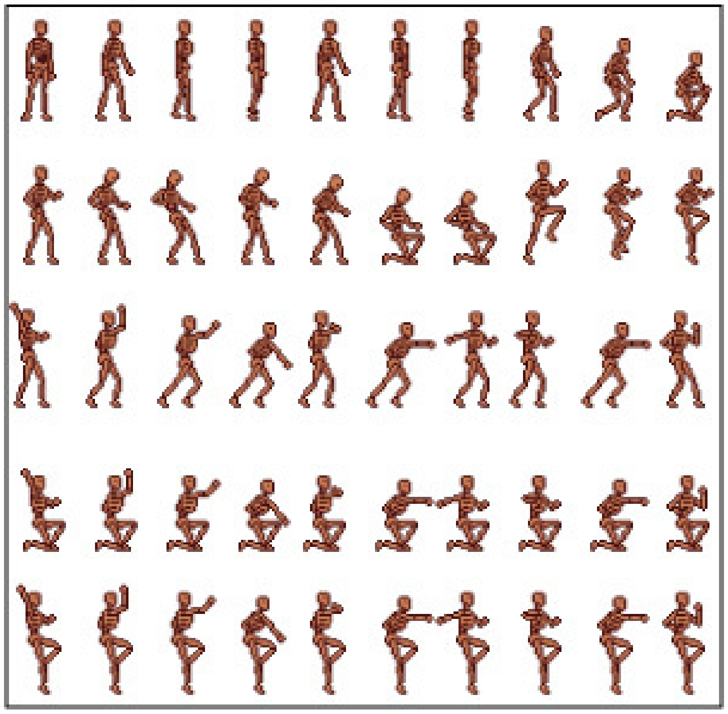

Figure 2.4 Series of images for a skeleton sprite

The frame rate of sprite animations can vary, but many games choose to use 24 FPS (the traditional frame rate used in film). This means that every second of an animation needs 24 individual images. Some genres, such as 2D fighting games, may use 60 FPS sprite animations, which dramatically increases the required number of images. Luckily, most sprite animations are significantly shorter than 1 second in duration.

The simplest way to represent an animated sprite is with a vector of the different images corresponding to each frame in an animation. The `AnimSpriteComponent` class, declared in Listing 2.6, uses this approach.

Listing 2.6 `AnimSpriteComponent` Declaration

Click here to view code image

```
class AnimSpriteComponent : public SpriteComponent
{
public:
   AnimSpriteComponent(class Actor* owner, int drawOrder = 100);
   // Update animation every frame (overriden from component)
   void Update(float deltaTime) override;
   // Set the textures used for animation
   void SetAnimTextures(const std::vector<SDL_Texture*>& textures);
   // Set/get the animation FPS
   float GetAnimFPS() const { return mAnimFPS; }
   void SetAnimFPS(float fps) { mAnimFPS = fps; }
private:
   // All textures in the animation
   std::vector<SDL_Texture*> mAnimTextures;
   // Current frame displayed
   float mCurrFrame;
   // Animation frame rate
   float mAnimFPS;
};
```

The `mAnimFPS` variable allows different animated sprites to run at different frame rates. It also allows the animation to dynamically speed up or slow down. For instance, as a character gains speed, you could increase the frame rate of the animation to further add to the illusion of speed. The `mCurrFrame` variable tracks the current frame displayed as a float, which allows you to also keep track of how long that frame has displayed.

The SetAnimTextures function simply sets the mAnimTextures member variable to the provided vector and resets mCurrFrame to zero. It also calls the SetTexture function (inherited from SpriteComponent) and passes in the first frame of the animation. Since this code uses the SetTexture function from SpriteComponent, it's unnecessary to override the inherited Draw function.

The Update function, shown in Listing 2.7, is where most of the heavy lifting of AnimSpriteComponent occurs. First, update mCurrFrame as a function of the animation FPS and delta time. Next, you make sure that mCurrFrame remains less than the number of textures (which means you need to wrap around back to the start of the animation if needed). Finally, cast mCurrFrame to an int, grab the correct texture from mAnimTextures, and call SetTexture.

Listing 2.7 AnimSpriteComponent::Update Implementation

Click here to view code image

```
void AnimSpriteComponent::Update(float deltaTime)
{
   SpriteComponent::Update(deltaTime);

  if (mAnimTextures.size() > 0)
   {
      // Update the current frame based on frame rate
      // and delta time
      mCurrFrame += mAnimFPS * deltaTime;


    // Wrap current frame if needed
      while (mCurrFrame >= mAnimTextures.size())
      {
         mCurrFrame -= mAnimTextures.size();
      }

    // Set the current texture
      SetTexture(mAnimTextures[static_cast<int>(mCurrFrame)]);
   }
}
```

One feature missing from `AnimSpriteComponent` is better support for switching between animations. Currently, the only way to switch an animation is to call `SetAnimTextures` repeatedly. It makes more sense to have a vector of all the different textures for all of a sprite's animations and then specify which images correspond to which animation. You'll explore this idea further in Exercise 2.2.

Scrolling Backgrounds

A trick often used in 2D games is having a background that scrolls by. This creates an impression of a larger world, and infinite scrolling games often use this technique. For now, we are focusing on scrolling backgrounds, as opposed to scrolling through an actual level. The easiest method is to split the background into screen-sized image segments, which are repositioned every frame to create the illusion of scrolling.

As with animated sprites, it makes sense to create a subclass of `SpriteComponent` for backgrounds. Listing 2.8 shows the declaration of `BGSpriteComponent`.

Listing 2.8 `BGSpriteComponent` Declaration

Click here to view code image

```
class BGSpriteComponent : public SpriteComponent
{
public:
   // Set draw order to default to lower (so it's in the background)
   BGSpriteComponent(class Actor* owner, int drawOrder = 10);
   // Update/draw overriden from parent
   void Update(float deltaTime) override;
   void Draw(SDL_Renderer* renderer) override;
   // Set the textures used for the background
   void SetBGTextures(const std::vector<SDL_Texture*>& textures);
   // Get/set screen size and scroll speed
   void SetScreenSize(const Vector2& size) { mScreenSize = size; }
   void SetScrollSpeed(float speed) { mScrollSpeed = speed; }
   float GetScrollSpeed() const { return mScrollSpeed; }
private:
   // Struct to encapsulate each BG image and its offset
   struct BGTexture
   {
      SDL_Texture* mTexture;
      Vector2 mOffset;
   };
   std::vector<BGTexture> mBGTextures;
   Vector2 mScreenSize;
   float mScrollSpeed;
};
```

The `BGTexture` struct associates each background texture with its corresponding offset. The offsets update every frame to create the scrolling effect. You need to initialize the offsets in `SetBGTextures`, positioning each background to the right of the previous one:

Click here to view code image

```
void BGSpriteComponent::SetBGTextures(const std::vector<SDL_Texture*>&
textures)
{
   int count = 0;
   for (auto tex : textures)
   {
      BGTexture temp;
      temp.mTexture = tex;
      // Each texture is screen width in offset
      temp.mOffset.x = count * mScreenSize.x;
      temp.mOffset.y = 0;
      mBGTextures.emplace_back(temp);
      count++;
   }
}
```

This code assumes that each background image has a width corresponding to the screen width, but it's certainly possible to modify the code to account for variable sizes. The `Update` code updates the offsets of each background, taking to account when one image moves all the way off the screen. This allows the images to infinitely repeat:

Click here to view code image

```
void BGSpriteComponent::Update(float deltaTime)
{
   SpriteComponent::Update(deltaTime);
   for (auto& bg : mBGTextures)
   {
      // Update the x offset
      bg.mOffset.x += mScrollSpeed * deltaTime;
      // If this is completely off the screen, reset offset to
      // the right of the last bg texture
      if (bg.mOffset.x < -mScreenSize.x)
```

```
        {
           bg.mOffset.x = (mBGTextures.size() - 1) * mScreenSize.x - 1;
        }
     }
  }
```

The `Draw` function simply draws each background texture using `SDL_RenderCopy`, making sure to adjust the position based on the owner's position and the offset of that background. This achieves the simple scrolling behavior.

Some games also implement **parallax scrolling**. In this approach, you use multiple layers for the background. Each layer scrolls at different speeds, which gives an illusion of depth. For example, a game might have a cloud layer and a ground layer. If the cloud layer scrolls more slowly than the ground layer, it gives the impression that the clouds are farther away than the ground. Traditional animation has used this technique for nearly a century, and it is effective. Typically, only three layers are necessary to create a believable parallax effect, as illustrated in Figure 2.5. Of course, more layers add more depth to the effect.

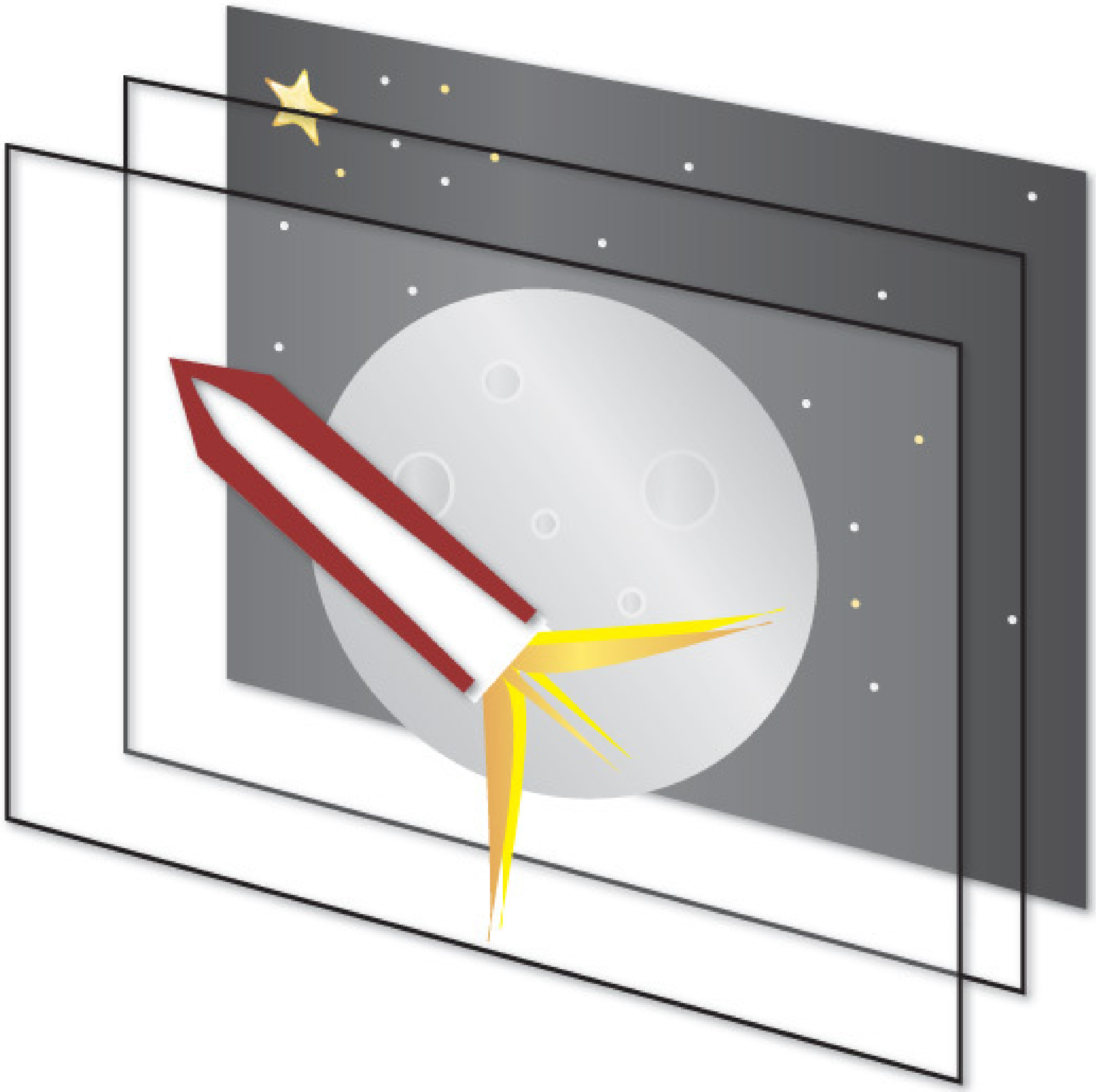

Figure 2.5 Space scene broken into three layers to facilitate parallax scrolling

To implement the parallax effect, attach multiple `BGSpriteComponent`s to a single actor and specify different draw order values. Then you can use a different scroll speed for each background to complete the effect.

Game Project

Unfortunately, you have not learned about enough new topics to make a game with noticeably more complex mechanics than the *Pong* clone created in Chapter 1, “Game Programming Overview.” And it wouldn’t be particularly interesting to just add sprites to the previous chapter’s game. So in lieu of a complete game, this chapter’s game project demonstrates the new techniques covered in this chapter. The code is available in the book’s GitHub repository, in the `Chapter02` directory. Open `Chapter02-windows.sln` on Windows and `Chapter02-mac.xcodeproj` on Mac. Figure 2.6 shows the game project in action. Jacob Zinman-Jeanes created the sprite images, which are licensed under the CC BY license.

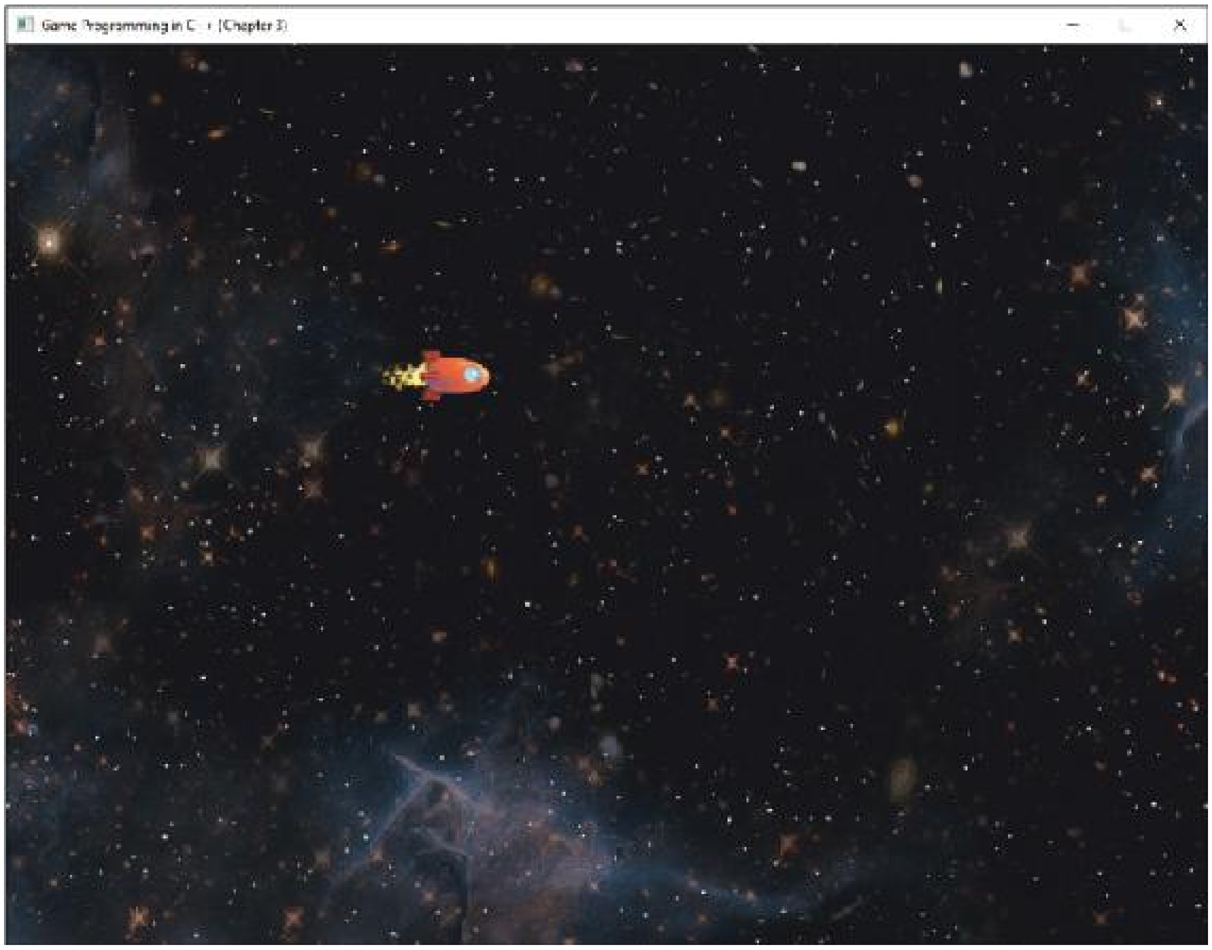

Figure 2.6 Side-scroller project in action

The code includes an implementation of the hybrid `Actor`/`Component` model, `SpriteComponent`, `AnimSpriteComponent`, and parallax scrolling. It also includes a subclass of `Actor` called `Ship`. The `Ship` class contains two speed variables to control the left/right speed and the up/down speed, respectively. Listing 2.9 shows the declaration of `Ship`.

Listing 2.9 `Ship` Declaration

Click here to view code image

```
class Ship : public Actor
{
public:
   Ship(class Game* game);
   void UpdateActor(float deltaTime) override;
   void ProcessKeyboard(const uint8_t* state);
   float GetRightSpeed() const { return mRightSpeed; }
   float GetDownSpeed() const { return mDownSpeed; }
private:
   float mRightSpeed;
   float mDownSpeed;
};
```

The `Ship` constructor initializes `mRightSpeed` and `mDownSpeed` to 0, and also creates an `AnimSpriteComponent` attached to the ship, with the associated textures:

Click here to view code image

```
AnimSpriteComponent* asc = new AnimSpriteComponent(this);
std::vector<SDL_Texture*> anims = {
   game->GetTexture("Assets/Ship01.png"),
   game->GetTexture("Assets/Ship02.png"),
   game->GetTexture("Assets/Ship03.png"),
   game->GetTexture("Assets/Ship04.png"),
};
asc->SetAnimTextures(anims);
```

The keyboard input directly affects the speed of the ship. The game uses the `W` and `S` keys to move the ship up and down and the `A` and `D` keys to move the ship left and right. The `ProcessKeyboard` function takes in these inputs and updates `mRightSpeed` and `mDownSpeed` as appropriate.

The `Ship::UpdateActor` function implements the ship's movement, using techniques similar to those shown in Chapter 1:

Click here to view code image