

C++ Game Development By Example

Learn to build games and graphics with SFML, OpenGL, and Vulkan using C++ programming



Siddharth Shekar

Packt

www.packt.com

C++ Game Development By Example

Learn to build games and graphics with SFML, OpenGL, and Vulkan using C++ programming

Siddharth Shekar

Packt

BIRMINGHAM - MUMBAI

C++ Game Development By Example

Copyright © 2019 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing or its dealers and distributors, will be held liable for any damages caused or alleged to have been caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

Commissioning Editor: Kunal Choudhari

Acquisition Editor: Trusha Shriyan

Content Development Editor: Keagan Carneiro

Technical Editor: Leena Patil

Copy Editor: Safis Editing

Language Support Editor: Storm Mann

Project Coordinator: Kinjal Bari

Proofreader: Safis Editing

Indexer: Pratik Shirodkar

Graphics: Alishon Mendonsa

Production Coordinator: Shraddha Falebhai

First published: May 2019

Production reference: 1020519

Published by Packt Publishing Ltd.

Livery Place

35 Livery Street

Birmingham

B3 2PB, UK.

ISBN 978-1-78953-530-3

www.packtpub.com

To my loving mother, Shanti Shekar, and my caring father, Shekar Rangarajan, for their sacrifices and for exemplifying the power of determination.

– Siddharth Shekar



mapt.io

Mapt is an online digital library that gives you full access to over 5,000 books and videos, as well as industry leading tools to help you plan your personal development and advance your career. For more information, please visit our website.

Why subscribe?

- Spend less time learning and more time coding with practical eBooks and Videos from over 4,000 industry professionals
- Improve your learning with Skill Plans built especially for you
- Get a free eBook or video every month
- Mapt is fully searchable
- Copy and paste, print, and bookmark content

Packt.com

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.packt.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at customercare@packtpub.com for more details.

At www.packt.com, you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on Packt books and eBooks.

Contributors

About the author

Siddharth Shekar is a game developer and teacher with over 6 years' industry experience and 12 years' experience in C++ and other programming languages. He is adept at graphics libraries such as OpenGL and Vulkan, and game engines such as Unity and Unreal.

He has published games on the iOS and Android app stores. He has also authored books including *Swift Game Development*, *Mastering Android Game Development with Unity*, and *Learning iOS 8 Game Development Using Swift*, all published by Packt Publishing.

He currently lives in Auckland, New Zealand, and is a lecturer in the games department at Media Design School. He teaches advanced computer graphics programming, PlayStation 4 native game development, and mentors final year production students.

About the reviewers

Simone Angeloni is a software engineer with over 14 years' experience in C/C++. His skill set includes cross-platform development, network communications, embedded systems, multithreading, databases, web applications, low-latency architectures, user interfaces, game development, and visual design. At present, he is the principal software engineer of the R&D department of MRMC, a subsidiary company of Nikon Corporation, where he develops robotic motion control solutions used for live broadcasts, film productions, and photography.

Andreas Oehlke is a professional full-stack software engineer. He holds a bachelor's degree in computer science and loves to experiment with software and hardware. His trademark has always been his enthusiasm and affinity for electronics and computers. His hobbies include game development, building embedded systems, sports, and making music. He currently works full-time as a senior software engineer for a German financial institution. Furthermore, he has worked as a consultant and game developer in San Francisco, CA. He is also the author of *Learning LibGDX Game Development*, published by Packt Publishing.

Packt is searching for authors like you

If you're interested in becoming an author for Packt, please visit authors.packtpub.com and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Table of Contents

Preface	1
<hr/>	
Section 1: Basic Concepts	
<hr/>	
Chapter 1: C++ Concepts	8
Program basics	9
Variables	14
Strings	17
Operators	18
Statements	22
Iteration	25
Jump statements	27
Switch statement	29
Functions	31
Scope of variables	34
Arrays	34
Pointers	39
Structs	43
Enums	44
Classes	46
Inheritance	48
Summary	52
Chapter 2: Mathematics and Graphics Concepts	53
3D coordinate system	54
Points	55
Vectors	57
Vector operations	58
Vector magnitude	61
Unit vector	62
Dot product	63
Cross product	64
Matrices	66
Matrix Addition and Subtraction	67
Matrix multiplication	67
Identity matrix	70
Matrix transpose	71
Matrix inverse	72
GLM OpenGL mathematics	72
OpenGL data types	74

Space transformations	75
Local/object space	75
World space	76
View space	77
Projection space	79
Screen space	82
Render pipeline	82
Vertex specification	84
Vertex shader	86
Vertex post-processing	87
Primitive assembly	88
Rasterization	88
Fragment shader	88
Per-sample operation	89
Framebuffer	90
Summary	90

Section 2: SFML 2D Game Development

Chapter 3: Setting Up Your Game	92
An overview of SFML	93
Downloading SFML and configuring Visual Studio	94
Creating a window	98
Drawing shapes	101
Adding sprites	107
Keyboard input	113
Handling player movement	114
Summary	117
Chapter 4: Creating Your Game	118
Starting afresh	119
Creating the hero class	121
Creating the enemy class	128
Adding enemies	130
Creating a rocket class	136
Adding rockets	138
Collision detection	141
Summary	143
Chapter 5: Finalizing Your Game	144
Finishing the Gameloop and adding scoring	145
Adding text	148
Adding audio	154
Adding player animations	157
Summary	162

Section 3: Modern OpenGL 3D Game Development

Chapter 6: Getting Started with OpenGL	164
What is OpenGL?	164
Creating our first OpenGL project	165
Creating a window and ClearScreen	168
Creating a Mesh class	172
Creating a Camera class	174
The ShaderLoader class	176
The Light Renderer class	180
Drawing the object	189
Summary	197
Chapter 7: Building on the Game Objects	198
Creating a MeshRenderer class	199
Creating the TextureLoader class	204
Adding Bullet Physics	213
Adding rigid bodies	217
Summary	224
Chapter 8: Enhancing Your Game with Collision, Loop, and Lighting	225
Adding a RigidBody name	226
Adding an enemy	226
Moving the enemy	229
Checking collision	232
Adding keyboard controls	234
Game loop and scoring	236
Text rendering	238
Adding lighting	253
Summary	262

Section 4: Rendering 3D Objects with Vulkan

Chapter 9: Getting Started with Vulkan	264
About Vulkan	264
Configuring Visual Studio	265
Vulkan validation layers and extensions	269
Vulkan instances	276
The Vulkan Context class	278
Creating the window surface	282
Picking a physical device and creating a logical device	283
Summary	300
Chapter 10: Preparing the Clear Screen	301
Creating the SwapChain	303

Creating the Renderpass	310
Using render targets and Framebuffers	314
Creating CommandBuffer	319
Begining and ending Renderpass	324
Creating the clear screen	325
Summary	331
Chapter 11: Creating Object Resources	332
Updating the Mesh class for Vulkan	333
Creating the ObjectBuffers class	335
Creating the Descriptor class	348
Creating the SPIR-V shader binary	354
Summary	358
Chapter 12: Drawing Vulkan Objects	359
Preparing the graphics pipeline class	360
Shader stage create info	365
Vertex input state create info	366
Input assembly create info	366
Rasterization state create info	367
MultiSample state create info	367
Depth and stencil create info	368
Color blend state create info	368
Dynamic state info	368
Viewport state create info	369
Graphics pipeline create info	370
Object renderer class	371
Changes to the VulkanContext class	375
Camera class	376
Drawing the object	378
Synchronizing the object	380
References	388
Summary	389
Appendix A: Other Books You May Enjoy	390
Leave a review - let other readers know what you think	392
Index	393

Preface

Computer graphics programming is considered to be one of the hardest subjects to cover, as it involves complex mathematics, programming, and graphics concepts that are intimidating to the average developer. Also, with alternative game engines available, such as Unity and Unreal, it is important to understand graphics programming, as it is a lot easier to make 2D or 3D games using these more sophisticated game engines. These engines also use some rendering APIs, such as OpenGL, Vulkan, Direct3D, and Metal, to draw objects in a scene, and the graphics engine in a game engine constitutes more than 50% of it. Therefore, it is imperative to have some knowledge about graphics programming and graphics APIs.

The objective of this book is to break down this complex subject into bite-sized chunks to make it easy to understand. So, we will start with the basic concepts that are required to understand the math, programming, and graphics basics.

In the next section of the book, we will create a 2D game, initially with **Simple and Fast Multimedia Library (SFML)**, which covers the basics that are required to create any game, and with which you can make any game with the utmost ease, without worrying about how a game object is drawn. We will be using SFML just to draw our game objects.

In the next part of the book, we will see how game objects get presented onscreen using OpenGL. OpenGL is a high-level Graphics API that enables us to get something rendered to a scene quickly. A simple sprite created in SFML goes through a lot of steps before actually getting drawn on the screen. We will see how a simple image gets loaded and gets displayed on the screen and what steps are required to do so. But that is just the start. We will see how to add 3D physics to the game and develop a physics-based game from the ground up. Finally, we will add some lighting to make the scene a little more interesting.

With that knowledge of OpenGL, we will dive further into graphics programming and see how Vulkan works. Vulkan is the successor to OpenGL and is a low-level, verbose graphics API. OpenGL is a high-level graphics API that hides a lot of inner workings. With Vulkan, you have complete access to the GPU, and with the Vulkan graphics API, we will learn how to render our game objects.

Who this book is for

This book is targeted at game developers keen to learn game development with C++ and graphics programming using OpenGL or the Vulkan graphics API. This book is also for those looking to update their existing knowledge of those subjects. Some prior knowledge of C++ programming is assumed.

What this book covers

Chapter 1, C++ Concepts, covers the basics of C++ programming, which are essential to understand and code the chapters in this book.

Chapter 2, Mathematics and Graphics Concepts, In this chapter we cover the basic topics of maths such as, vector calculations and knowledge on matrices. These are essential for graphics programming and basic physics programming. We then move on to basics of graphics programming, starting with how a bunch of vertices are sent to the graphics pipeline and how they are converted into shapes and rendered on the screen.

Chapter 3, Setting Up Your Game, introduces the SFML framework, its uses, and its limitations. It also covers creating a Visual Studio project and adding SFML to it, creating a basic window with the basic framework of the game to initialize, update, render, and close it. We will also learn how to draw different shapes and learn how to add a textured sprite to the scene and add keyboard input.

Chapter 4, Creating Your Game, covers the creation of the character class and adding functionality to a character to make them move and jump. We will also create the enemy class to populate enemies for the game. We will add a rockets class so the player can spawn rockets when they fire. Finally, we will add collision detection to detect collisions between two sprites.

Chapter 5, Finalizing Your Game, covers finishing the game and adding some polishing touches by adding scoring, text, and audio. We'll also add some animation to the player character to make the game more lively.

Chapter 6, Getting Started with OpenGL, looks at what OpenGL is, its advantages, and its shortcomings. We'll integrate OpenGL into the Visual Studio project and use GLFW to create a window. We'll create a basic game loop and render our first 3D object using Vertex and fragment shaders.

Chapter 7, *Building on the Game Objects*, covers adding textures to an object. We'll include the Bullet Physics library in the project to add physics to the scene. We will see how to integrate physics with our 3D OpenGL rendered object. Finally, we will create a basic level to test out the physics.

Chapter 8, *Enhancing Your Game with Collision, Loop, and Lighting*, covers adding a game-over condition and finishing the game loop. We will add some finishing touches to the game by adding some basic 3D lighting and text to display the score and game-over condition.

Chapter 9, *Getting Started with Vulkan*, looks at the need for Vulkan and how it is different from OpenGL. We'll look at the advantages and disadvantages of Vulkan, integrate it into the Visual Studio project, and add GLFW to create a window for the project. We will create an app and a Vulkan instance, and add validation layers to check whether Vulkan is running as required. We'll also get physical device properties and create a logical device.

Chapter 10, *Preparing the Clear Screen*, covers the creation of a window surface to which we can render the scene. We also need to create a swap chain so that we can ping-pong between the front buffer and back buffer, and create image views and frame buffers to which the views are attached. We will create the draw command buffer to record and submit graphics commands and create a renderpass clear screen.

Chapter 11, *Creating Object Resources*, covers creating the resources required to draw the geometry. This includes adding a mesh class that has all the geometry information, including vertex and index data. We'll create object buffers to store the vertex, index, and uniform buffers. We'll also create DescriptorSetLayout and Descriptor Sets, and finally, we'll create shaders and convert them to SPIR-V binary format.

Chapter 12, *Drawing Vulkan Objects*, covers creating the graphics pipeline, in which we set the vertices and enable viewports, multisampling, and depth and stencil testing. We'll also create an object class, which will help create the object buffers, descriptor set, and graphics pipeline for the object. We will create a camera class to view the world through, and then finally render the object. At the end of the chapter, we will also see how to synchronize information being sent.

To get the most out of this book

The book is designed to be read from the start, chapter by chapter. If you have prior knowledge of the contents of a chapter, then please feel free to skip ahead instead.

It is good to have some prior programming experience with C++, but if not, then there is a chapter on C++ programming, which covers the basics. No prior knowledge of graphics programming is assumed.

To run OpenGL and Vulkan projects, make sure your hardware supports the current version of the API. The book uses OpenGL 4.5 and Vulkan 1.1. Most GPU vendors support OpenGL and Vulkan, but for a full list of supported GPUs, please refer to the GPU manufacturer or to the wiki, at [https://en.wikipedia.org/wiki/Vulkan_\(API\)](https://en.wikipedia.org/wiki/Vulkan_(API)).

Download the example code files

You can download the example code files for this book from your account at www.packt.com. If you purchased this book elsewhere, you can visit www.packt.com/support and register to have the files emailed directly to you.

You can download the code files by following these steps:

1. Log in or register at www.packt.com.
2. Select the **SUPPORT** tab.
3. Click on **Code Downloads & Errata**.
4. Enter the name of the book in the **Search** box and follow the onscreen instructions.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR/7-Zip for Windows
- Zipeg/iZip/UnRarX for Mac
- 7-Zip/PeaZip for Linux

The code bundle for the book is also hosted on GitHub at <https://github.com/PacktPublishing/CPP-Game-Development-By-Example>. In case there's an update to the code, it will be updated on the existing GitHub repository.

We also have other code bundles from our rich catalog of books and videos available at <https://github.com/PacktPublishing/>. Check them out!

Download the color images

We also provide a PDF file that has color images of the screenshots/diagrams used in this book. You can download it here:

https://www.packtpub.com/sites/default/files/downloads/9781789535303_ColorImages.pdf.

Conventions used

There are a number of text conventions used throughout this book.

CodeInText: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. Here is an example: "Here, the printing of `Hello, World` is tasked to the `main` function."

A block of code is set as follows:

```
#include <iostream>
// Program prints out "Hello, World" to screen
int main()
{
    std::cout<< "Hello, World."<<std::endl;
    return 0;
}
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
int main() {
    //init game objects
    while (window.isOpen()) {
        // Handle Keyboard events
        // Update Game Objects in the scene
window.clear(sf::Color::Red);
        // Render Game Objects
window.display();
    }
    return 0;
}
```

Bold: Indicates a new term, an important word, or words that you see onscreen. For example, words in menus or dialog boxes appear in the text like this. Here is an example: "In **Input** and under **Linker**, type the following .lib files."



Warnings or important notes appear like this.



Tips and tricks appear like this.

Get in touch

Feedback from our readers is always welcome.

General feedback: If you have questions about any aspect of this book, mention the book title in the subject of your message and email us at customercare@packtpub.com.

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you would report this to us. Please visit www.packt.com/submit-errata, selecting your book, clicking on the Errata Submission Form link, and entering the details.

Piracy: If you come across any illegal copies of our works in any form on the Internet, we would be grateful if you would provide us with the location address or website name. Please contact us at copyright@packt.com with a link to the material.

If you are interested in becoming an author: If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, please visit authors.packtpub.com.

Reviews

Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential readers can then see and use your unbiased opinion to make purchase decisions, we at Packt can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about Packt, please visit packt.com.

1

Section 1: Basic Concepts

This section covers the basic concepts. We need to have a good understanding of math, programming, and computer graphics to get ready for the later sections in the book.

The following chapters are in this section:

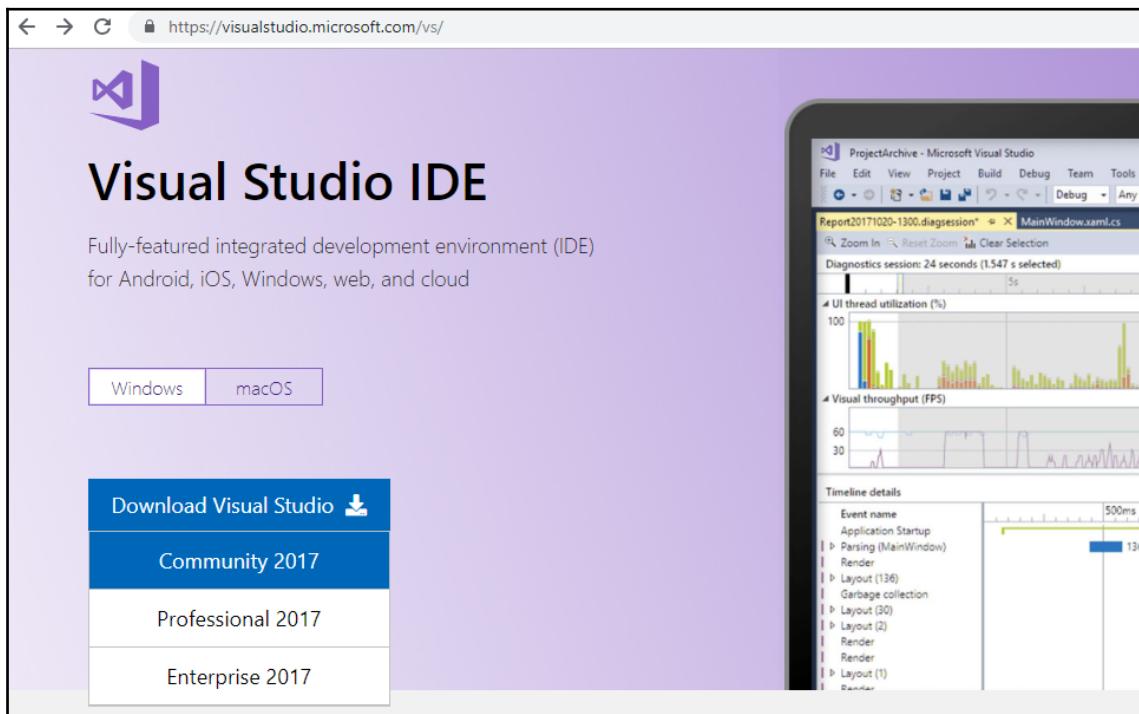
Chapter 1, C++ Concepts

Chapter 2, Mathematics and Graphics Concepts

1 C++ Concepts

In this chapter, we will explore the basics of writing a C++ program. Here, we will cover just enough to wrap our heads around what the capabilities of the C++ programming language are. This will be required to understand the code used in this book.

To run the examples, use Visual Studio 2017. You can download the Community version for free at <https://visualstudio.microsoft.com/vs/>:



The topics covered in this chapter are as follows:

- Program basics
- Variables
- Operators
- Statements
- Iteration
- Functions
- Arrays and pointers
- Struct and Enum
- Classes and inheritance

Program basics

C++ is a programming language, but what exactly is a program? A program is a set of instructions executed in sequence to give a desired output.

Let's look at our first program:

```
#include <iostream>
// Program prints out "Hello, World" to screen
int main()
{
    std::cout<< "Hello, World."<<std::endl;
    return 0;
}
```

We can look at this code line by line:

The hash (#) include is used when we want to include anything that is using valid C++ syntax. In this case, we are including a standard C++ library in our program. The file we want to include is then specified inside the angle brackets <>. Here, we are including a file called `iostream.h`. This file handles the input and output of data to the console/screen.

After the `include`, the double slash // is called a comment. Comments in code are not executed by the program. They are mainly to tell the person looking at the code what the code is currently doing. It is good practice to comment your code so that when you look at code you wrote a year ago, you will know what the code was doing then.

`main()` is a function. We will cover functions shortly, but a `main` function is the first function that is executed in a program, also called the entry point. A function is used to perform a certain task. Here, the printing of `Hello, World` is tasked to the main function. The contents that need to be executed must be enclosed in the curly brackets of the function. The `int` preceding the `main` keyword suggests that the function will return an integer. This is why we have returned `0` at the end of the `main` function, suggesting that the program executed and the program can terminate without errors.

When we want to print out something to the console/screen, we use the `std::cout` (console out) C++ command to send something to the screen. Whatever we want to send out should precede and end with the output operator, `<<`. `<<std::endl` is another C++ command, which specifies that it is the end of the line and nothing else should be printed on this line afterward. We have to use the prefix before the `std::` code to tell C++ that we are using the standard namespace with the namespace `std`. But why are namespaces necessary? We need namespaces because anyone can declare a variable name with `std`. How would the compiler differentiate between the two types of `std`? For this, we have namespaces to differentiate between the two.

Note that the two lines of code we have written in the `main` function have a semicolon (`;`) at the end of each line. The semicolon tells the compiler that this is the end of the instructions for that line of code so that the program can stop reading when it gets to the semicolon and go to the next line of instruction. Consequently, it is important to add a semicolon at the end of each line of instruction as it is mandatory.

The two lines of code we wrote before can be written in one line as follows:

```
std::cout<< "Hello, World."<<std::endl;return 0;
```

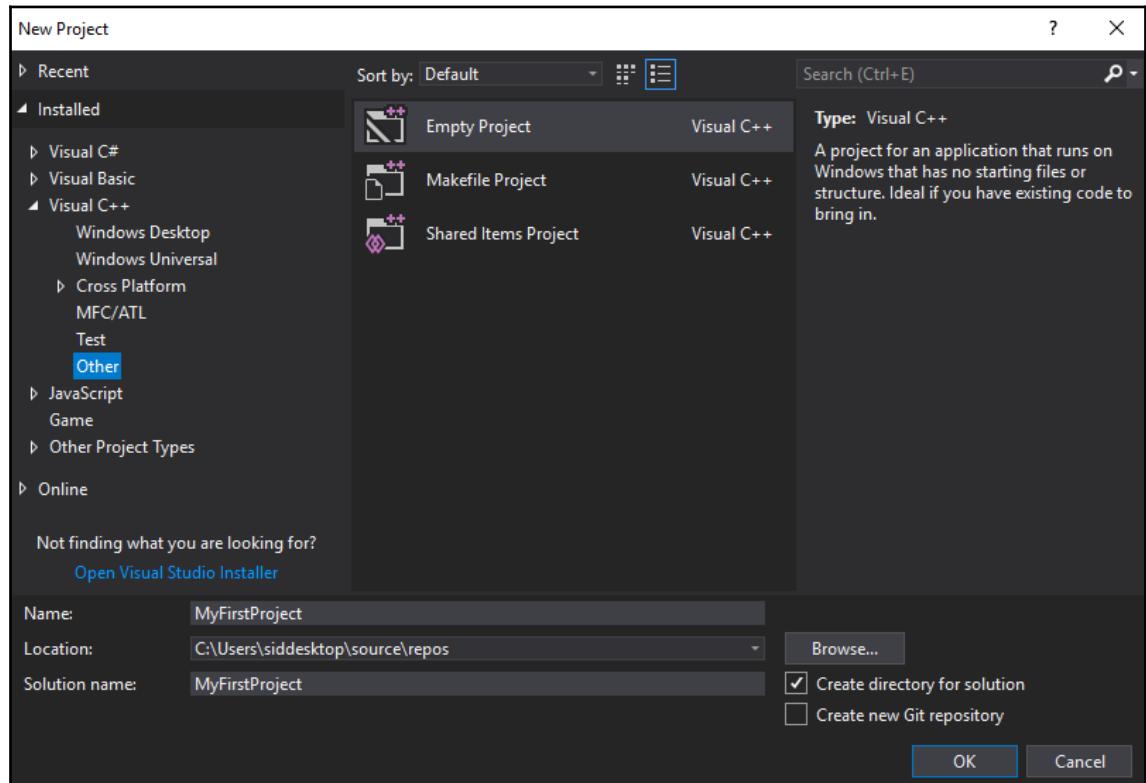
Even though it is written in a single line, for the compiler, there are two instructions with both instructions ending with a semicolon.

The first instruction is to print out `Hello, World` to the console and the second instruction is to terminate the program without any errors.

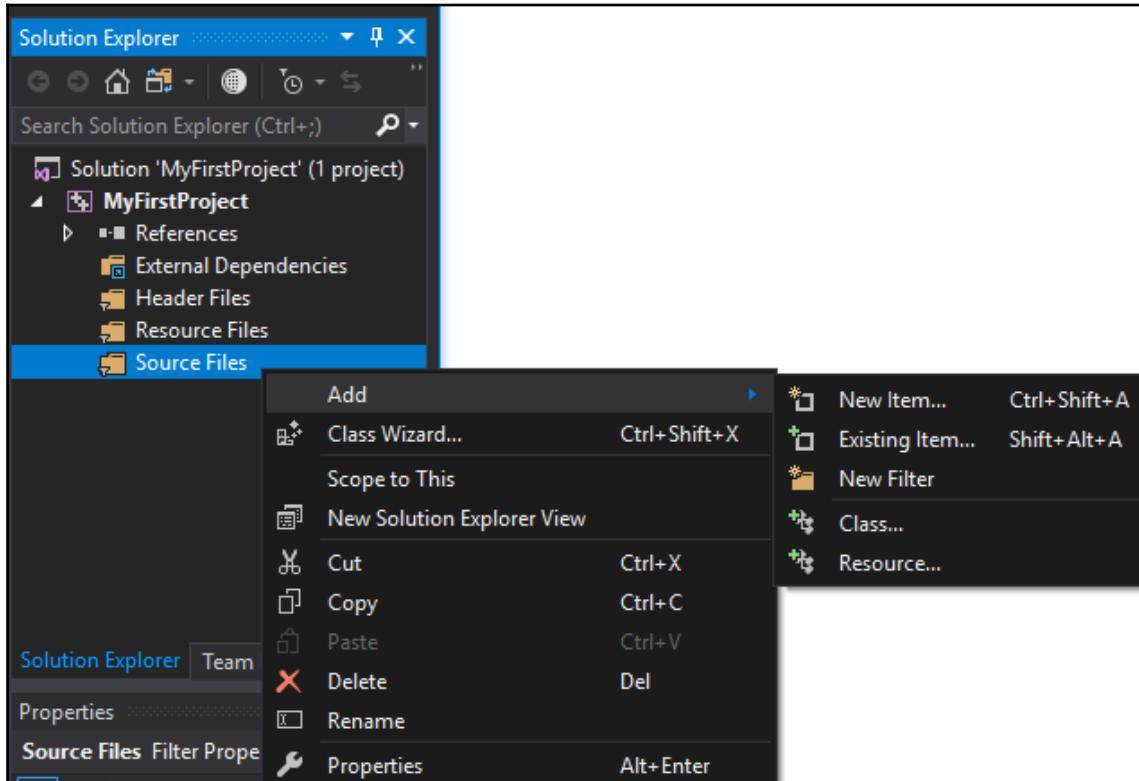
It is a very common mistake to forget semicolons and it happens to beginners as well as experienced programmers every now and then. So it's good to keep this in mind, in case you encounter your first compiler errors.

Let's run this code in Visual Studio using the following steps:

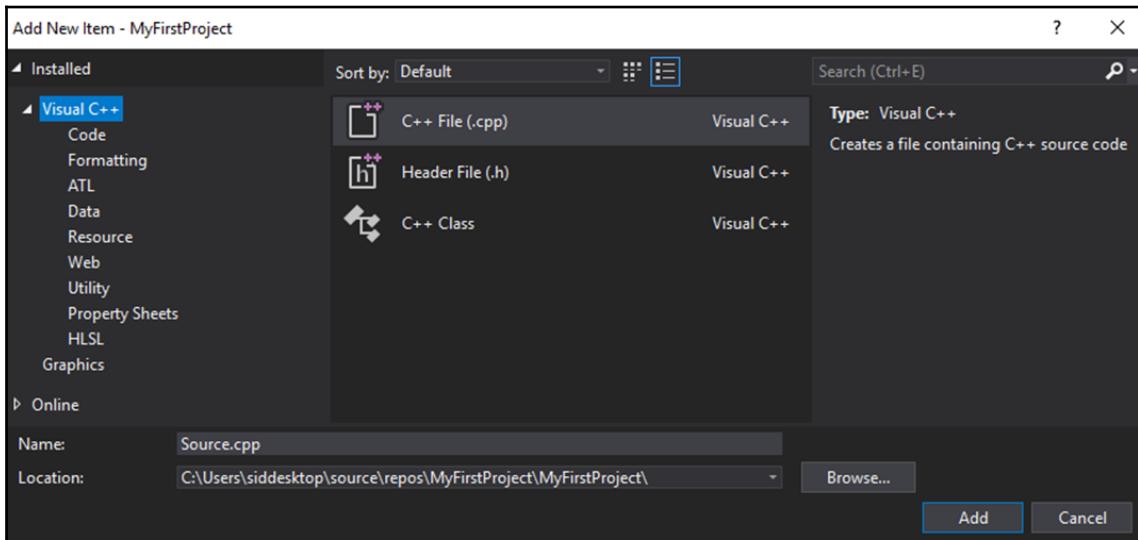
1. Open up Visual Studio and create a new project by going to **File | New | Project**.
2. On the left-hand side select **Visual C++** and then **Other**. For the **Project Type**, select **Empty Project**. Give this project a **Name**. Visual Studio automatically names the first project `MyFirstProject`. You can name it whatever you like.
3. Select the **Location** that you want the project to be saved in:



4. Once the project is created, in the **Solution Explorer**, right-click and select **Add | New Item**:



5. Create a new .cpp file called Source file:



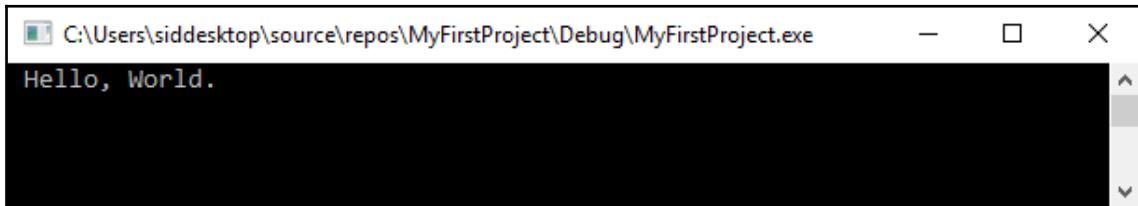
6. Copy the code at the start of the section into the Source.cpp file.
7. Now press the F5 key on the keyboard or press the **Local Window Debugger** button at the top of the window to run the application.
8. You will have seen a popup of the console after you ran the program. To make the console stay so that we can see what is happening, add the following highlighted lines to the code:

```
#include <iostream>
#include <conio.h>
// Program prints out "Hello, World" to screen
int main()
{
    std::cout << "Hello, World." << std::endl;
    _getch();
    return 0;
}
```

What `_getch()` does is it stalls the program and waits for a character input to the console without printing the character to the console. So, the program will wait for some input and then close the console.

To see what is printed to the console, we just add it for convenience. And to use this function, we need to include the `conio.h` header.

- When you run the project again, you will see the following output:



Now that we know how to run a basic program, let's look at the different data types that are included in C++.

Variables

A variable is used to store a value in it. Whatever value you store in a variable is stored in the memory location associated with that memory location. You assign a value to a variable with the following syntax.

We can first declare a variable type by specifying a type and then the variable name:

```
Type variable;
```

Here, `type` is the variable type and `variable` is the name of the variable.

Next, we can assign a value to a variable, which is done like so:

```
Variable = value;
```

Now that `value` is assigned to the variable.

Or, you can both declare the variable and assign a value to it in a single line, as follows:

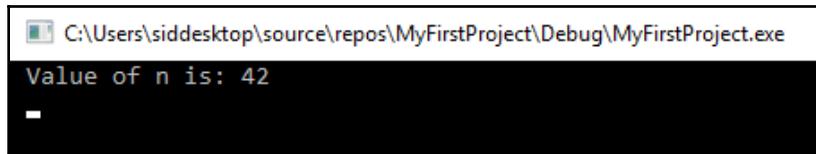
```
type variable = value;
```

Before you set a variable, you have to specify the variable type. You can then use the equal to (=) sign to assign the value to the variable.

Let's look at some example code:

```
#include <iostream>
#include <conio.h>
// Program prints out value of n to screen
int main()
{
    int n = 42;
    std::cout <<"Value of n is: "<< n << std::endl;
    _getch();
    return 0;
}
```

Replace the previous code with this code in `Source.cpp` and run the application. This is the output you should get:



In this program, we specify the data type as `int`. An `int` is a C++ data type that can store integers. So, it cannot store decimal values. We declare a variable called `n` and then we assign a value of 42 to it. Do not forget to add the semicolon at the end of the line.

In the next line, we print the value to the console. Note that, to print the value of `n`, we just pass in `n` in `cout` and don't have to add quotation marks.

On a 32-bit system, an `int` variable uses 4 bytes (which is equal to 32 bits) of memory. This basically means the `int` data type can hold values between 0 and $2^{32}-1$ (4,294,967,295). However, one bit is needed to describe the sign for the value (positive or negative), which leaves 31 bits remaining to express the actual value. Therefore, a signed `int` can hold values between -2^{31} (-2,147,483,648) and $2^{31}-1$ (2,147,483,647).

Let's look at some other data types:

Data type	Minimum	Maximum	Size (bytes)
bool	false	true	1
char	-128	127	1
short	-32768	327677	2
int	-2,147,483,648	2,147,483,647	4
long	-2,147,483,648	2,147,483,647	4
float	3.4 x 10-38	3.4 x 1038	4
double	1.7 x 10-308	1.7 x 10308	8

- **bool**: A bool can have only two values. It can either store `true` or `false`.
- **char**: This stores integers ranging between -128 and 127. `char` or character variables are used to store ASCII characters such as single characters—letters, for example.
- **short** and **long**: These are also integer types but these are able to store more information than just `int`. The size of `int` is system-dependent and `long` and `short` have fixed sizes irrespective of the system used.
- **float**: This is a floating point type. This means that it can store values with decimal spaces such as 3.14, 0.000875 and -9.875. It can store data with up to seven decimal places.
- **double**: This is a `float` with more precision. It can store decimal values up to 15 decimal places.

You also have unsigned data types of the same data type used to maximize the range of values it can store. Unsigned data types are used to store positive values. Consequently, all unsigned values start at 0.

So, `char` and `unsigned char` can store positive values from 0 to 255. Similar to `unsigned char` we have `unsigned short`, `int`, and `long`.

You can assign values to `bool`, `char`, and `float` as follows:

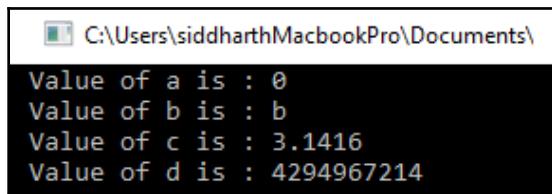
```
#include <iostream>
#include <conio.h>
// Program prints out value of bool, char and float to screen
int main()
{
    bool a = false;
    char b = 'b';
    float c = 3.1416f;
```

```
unsigned int d = -82;

std::cout << "Value of a is : " << a << std::endl;
std::cout << "Value of b is : " << b << std::endl;
std::cout << "Value of c is : " << c << std::endl;
std::cout << "Value of d is : " << d << std::endl;

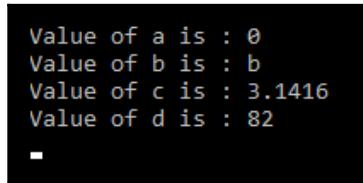
_getch();
return 0;
}
```

This is the output when you run the application:



```
C:\Users\siddharthMacbookPro\Documents\
Value of a is : 0
Value of b is : b
Value of c is : 3.1416
Value of d is : 4294967214
```

Everything is printing fine except d, which was assigned -82. What happened here? Well that's because d can store only unsigned values, so if we assign it -82, it gives a garbage value. Change it to just 82 without the negative sign and it will print the correct value:



```
Value of a is : 0
Value of b is : b
Value of c is : 3.1416
Value of d is : 82
```

Unlike int, bool stores a binary value where false is 0 and true is 1. So, when you print out the values of true and false, the output will be 1 and 0.

char store characters specified with single quotation marks, and values with decimals are printed just how you stored the values in the floats. The f is added at the end of the value when assigning a float, to tell the system that it is a float and not a double.

Strings

Variables that are non-numerical are either a single character or a series of characters called strings. In C++, a series of characters can be stored in a special variable called a string. A string is provided through a standard `string` class.

To declare and use string objects we have to include the string header file. After `#include <conio.h>`, also add `#include <string>` at the top of the file.

A string variable is declared in the same way as other variable types, except before the string type you have to use the std namespace.

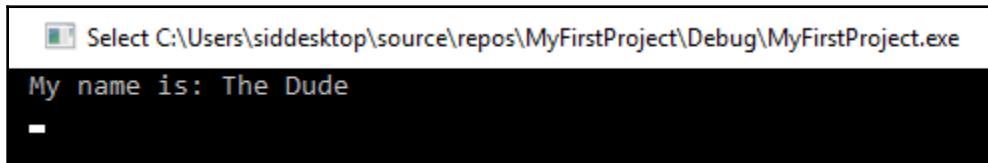
If you don't like adding the `std::` namespace prefix, you can also add the line using the `std` namespace. After the `#include`. This way, you won't have to add the `std::` prefix, as the program will understand. However, it can be printed out just like other variables:

```
#include <iostream>
#include <conio.h>
#include <string>

// Program prints out values to screen

int main()
{
    std::string name = "The Dude";
    std::cout << "My name is: " << name << std::endl;
    _getch();
    return 0;
}
```

Here is the output:



Operators

An operator is a symbol that performs a certain operation on a variable or expression. So far, we have used the `=` sign, which calls an assignment operator that assigns a value or expression from the right-hand side of the equals sign to a variable on the left-hand side.

The simplest form of other kinds of operators are arithmetic operators such as `+`, `-`, `*`, `/`, and `%`. These operators operate on a variable such as `int` and `float`. Let's look at some of the use cases of these operators:

```
#include <iostream>
#include <conio.h>
// Program prints out value of a + b and x + y to screen
int main()
{
    int a = 8;
    int b = 12;
    std::cout << "Value of a + b is : " << a + b << std::endl;
    float x = 7.345f;
    float y = 12.8354;
    std::cout << "Value of x + y is : " << x + y << std::endl;

    _getch();
    return 0;
}
```

The output of this is as follows:



```
C:\Users\sid\Desktop\source\repos\MyFirstProject\Debug\MyFirstProject.exe
Value of a + b is : 20
Value of x + y is : 20.1804
```

Let's look at examples for other operations as well:

```
#include <iostream>
#include <conio.h>

// Program prints out values to screen

int main()
{
    int a = 36;
    int b = 5;

    std::cout << "Value of a + b is : " << a + b << std::endl;
    std::cout << "Value of a - b is : " << a - b << std::endl;
    std::cout << "Value of a * b is : " << a * b << std::endl;
    std::cout << "Value of a / b is : " << a / b << std::endl;
    std::cout << "Value of a % b is : " << a % b << std::endl;
```

```
    getch();
    return 0;
}
```

The output is as shown as follows:

```
C:\Users\siddesktop\source\repos\MyFirstProject\Debug\MyFirstProject.exe
Value of a + b is : 41
Value of a - b is : 31
Value of a * b is : 180
Value of a / b is : 7
Value of a % b is : 1
```

+, -, *, and / are self-explanatory. However, there is one more arithmetic operator: %, which is called the module operator. It returns the remainder of a division .

How many times is 5 contained in 36? 7 times with a remainder of 1. That's why the result is 1.

Apart from the arithmetic operator, we also have an increment/decrement operator.

In programming, we increment variables often. You can do `a=a+1;` to increment and `a=a-1;` to decrement a variable value. Alternatively, you can even do `a+=1;` and `a-=1;` to increment and decrement, but in C++ programming there is an even shorter way of doing that, which is by using the ++ and -- signs to increment and decrement the value of a variable by 1.

Let's look at an example of how to use it to increment and decrement a value by 1:

```
#include <iostream>
#include <conio.h>

// Program prints out values to screen

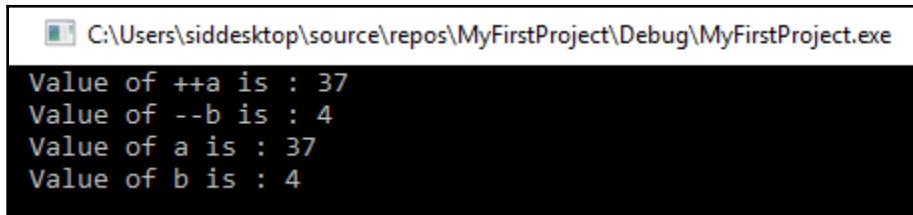
int main()
{
    int a = 36;
    int b = 5;

    std::cout << "Value of ++a is : " << ++a << std::endl;
    std::cout << "Value of --b is : " << --b << std::endl;
```

```
    std::cout << "Value of a is : " << a << std::endl;
    std::cout << "Value of b is : " << b << std::endl;

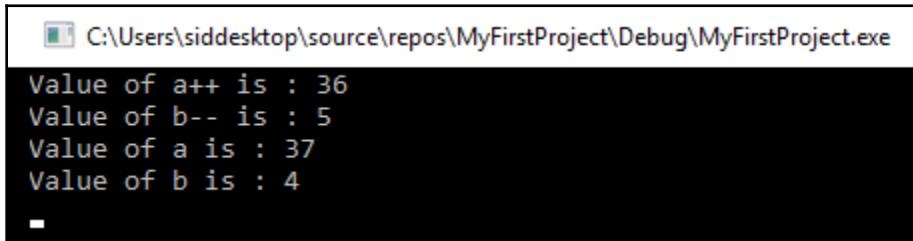
    _getch();
    return 0;
}
```

The output of this is as follows:



```
Value of ++a is : 37
Value of --b is : 4
Value of a is : 37
Value of b is : 4
```

Consequently, the `++` or `--` operator increments the value permanently. If the `++` is to the left of the variable, it is called a pre-increment operator. If it is put afterward, it is called a post-increment operator. There is a slight difference between the two. If we put the `++` on the other side, we get the following output:



```
Value of a++ is : 36
Value of b-- is : 5
Value of a is : 37
Value of b is : 4
-
```

In this case, `a` and `b` are incremented and decremented in the next line. So, when you print the values, it prints out the correct result.

It doesn't make a difference here, as it is a simple example, but overall it does make a difference and it is good to understand this difference. In this book, we will mostly be using post-increment operators.

In fact, this is how C++ got its name; it is an increment of C.

Apart from arithmetic, increment, and decrement operators, you also have logical and comparison operators, as shown:

Logical operators:

Operator	Operation
!	NOT
&&	AND
	OR

Comparison operators:

Operator	Comparison
==	Equal to
!=	Not equal to
<	Less than
>	Greater than
<=	Less than equal to
>=	Greater than equal to

We will cover these operators in the next section.

Statements

A program may not always be linear. Depending on your requirements, you might have to branch out or bifurcate, repeat a set of code, or take a decision. For this, there are conditional statements and loops.

In a conditional statement, you will check whether a condition is true. If it is, you will go ahead and execute the statement.

The first of the conditional statements is the `if` statement. The syntax for this looks as follows:

```
If (condition) statement;
```

Let's look at how to use this in code in the following code. Let's use one of the comparison operators here:

```
#include <iostream>
#include <conio.h>

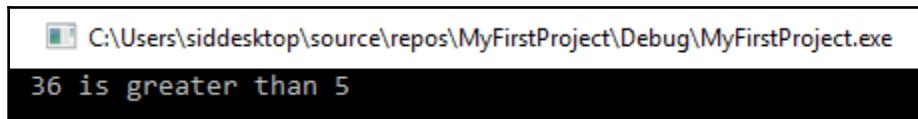
// Program prints out values to screen

int main()
{
    int a = 36;
    int b = 5;

    if (a > b)
        std::cout << a << " is greater than " << b << std::endl;

    getch();
    return 0;
}
```

The output is as follows:



We check the condition if `a` is greater than `b`, and if the condition is true, then we print out the statement.

But what if the opposite was true? For this, we have the `if...else` statement, which is a statement that basically executes the alternate statement. The syntax looks like this:

```
if (condition) statement1;
else statement2;
```

Let's look at it in code:

```
#include <iostream>
#include <conio.h>

// Program prints out values to screen

int main()
{

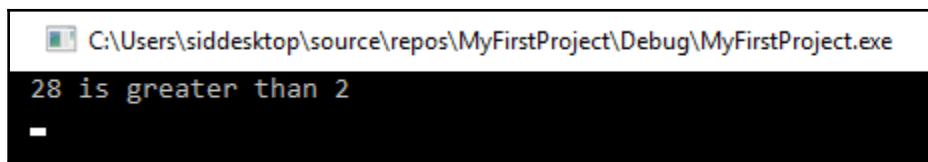
    int a = 2;
```

```
int b = 28;

if (a > b)
    std::cout << a << " is greater than " << b << std::endl;
else
    std::cout << b << " is greater than " << a << std::endl;

_getch();
return 0;
}
```

Here, the values of `a` and `b` are changed so that `b` is greater than `a`:



One thing to note is that after the `if` and `else` conditions, C++ will execute a single line of statement. If there are multiple statements after the `if` or `else`, then the statements need to be in curly brackets, as shown:

```
if (a > b)
{
    std::cout << a << " is greater than " << b << std::endl;
}
else
{
    std::cout << b << " is greater than " << a << std::endl;
}
```

You can also have `if` statements after using `else if`:

```
#include <iostream>
#include <conio.h>

// Program prints out values to screen

int main()
{

    int a = 28;
    int b = 28;

    if (a > b)
```

```
{  
    std::cout << a << " is greater than " << b << std::endl;  
}  
else if (a == b)  
{  
    std::cout << a << " is equal to " << b << std::endl;  
}  
else  
{  
    std::cout << b << " is greater than " << a << std::endl;  
}  
  
_getch();  
return 0;  
}
```

The output is as follows:



```
C:\Users\siddesktop\source\repos\MyFirstProject\Debug\MyFirstProject.exe  
28 is equal to 28  
-
```

Iteration

Iteration is the process of calling the same statement repeatedly. C++ has three iteration statements: the `while`, `do...while`, and `for` statements. Iteration is also commonly referred to as loops.

The `while` loop syntax looks like the following:

```
while (condition) statement;
```

Let's look at it in action:

```
#include <iostream>  
#include <conio.h>  
// Program prints out values to screen  
int main()  
{  
    int a = 10;  
    int n = 0;  
    while (n < a) {  
        std::cout << "value of n is: " << n << std::endl;  
        n++;
```

```
    }
    getch();
    return 0;
}
```

Here is the output of this code:

```
C:\Users\siddesktop\source\repos\MyFirstProject\Debug\MyFirstProject.exe
value of n is: 0
value of n is: 1
value of n is: 2
value of n is: 3
value of n is: 4
value of n is: 5
value of n is: 6
value of n is: 7
value of n is: 8
value of n is: 9
```

Here, the value of `n` is printed to the console until the condition is met.

The `do while` statement is almost the same as a `while` statement except, in this case, the statement is executed first and then the condition is tested. The syntax is as follows:

```
do statement
while (condition);
```

You can give it a go yourself and check the result.

The loop that is most commonly used in programming is the `for` loop. The syntax for this looks as follows:

```
for (initialization; continuing condition; update) statement;
```

The `for` loop is very self-contained. In `while` loops, we have to initialize `n` outside the `while` loop, but in the `for` loop, the initialization is done in the declaration of the `for` loop itself.

Here is the same example as the `while` loop but with the `for` loop:

```
#include <iostream>
#include <conio.h>
// Program prints out values to screen
int main()
```

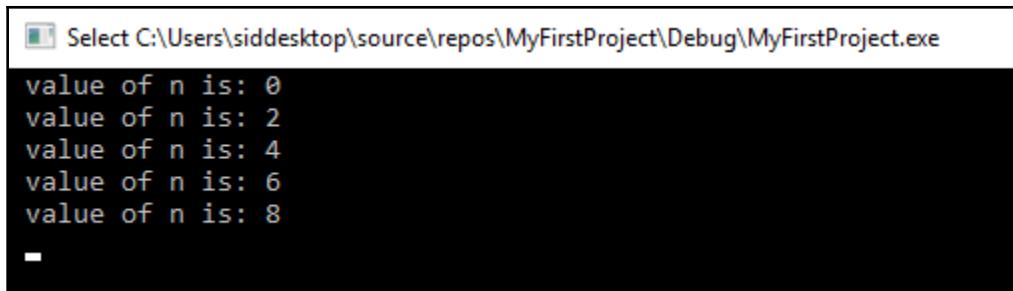
```
{  
    for (int n = 0; n < 10; n++)  
        std::cout << "value of n is: " << n << std::endl;  
    _getch();  
    return 0;  
}
```

The output is the same as the `while` loop but look how compact the code is compared to the `while` loop. Also the `n` is scoped locally to the `for` loop body.

We can also increment `n` by 2 instead of 1, as shown:

```
#include <iostream>  
#include <conio.h>  
  
// Program prints out values to screen  
int main()  
{  
    for (int n = 0; n < 10; n+=2)  
        std::cout << "value of n is: " << n << std::endl;  
    _getch();  
    return 0;  
}
```

Here is the output of this code:



```
value of n is: 0  
value of n is: 2  
value of n is: 4  
value of n is: 6  
value of n is: 8  
-
```

Jump statements

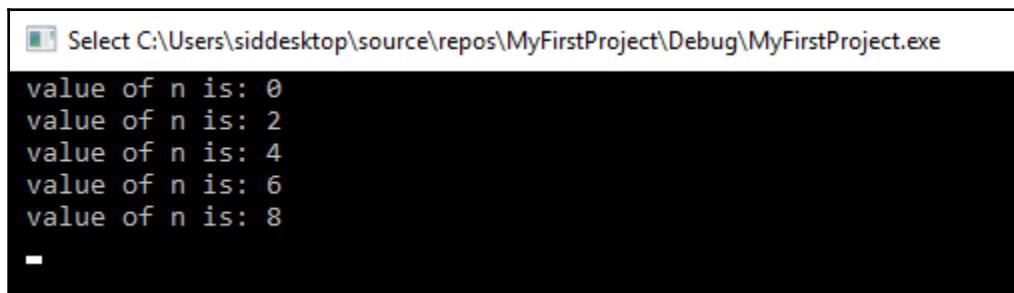
As well as condition and iteration statements, you also have `break` and `continue` statements.

The `break` statement is used to break out of an iteration. We can leave a loop and force it to quit if a certain condition is met.

Let's look at the `break` statement in use:

```
#include <iostream>
#include <conio.h>
// Program prints out values to screen
int main()
{
    for (int n = 0; n < 10; n++)
    {
        if (n == 5) {
            std::cout << "break" << std::endl;
            break;
        }
        std::cout << "value of n is: " << n << std::endl;
    }
    getch();
    return 0;
}
```

The output of this is as follows:



```
value of n is: 0
value of n is: 2
value of n is: 4
value of n is: 6
value of n is: 8
```

The `continue` statement will skip the current iteration and continue the execution of the statement until the end of the loop. In the `break` code, replace the `break` with `continue` to see the difference:

```
#include <iostream>
#include <conio.h>

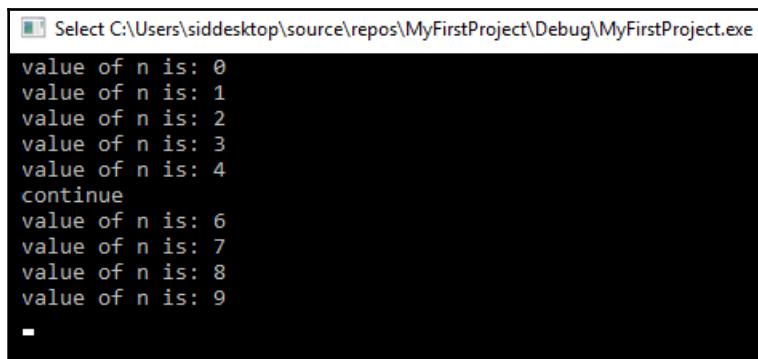
// Program prints out values to screen

int main()
{

    for (int n = 0; n < 10; n++)
    {
        if (n == 5) {
```

```
        std::cout << "continue" << std::endl;
        continue;
    }
    std::cout << "value of n is: " << n << std::endl;
}
_getch();
return 0;
}
```

Here is the output when `break` is replaced with `continue`:



```
value of n is: 0
value of n is: 1
value of n is: 2
value of n is: 3
value of n is: 4
continue
value of n is: 6
value of n is: 7
value of n is: 8
value of n is: 9
```

Switch statement

The last of the statements is the `switch` statement. A `switch` statement checks for several cases of values and if a value matches the expression, then it executes the corresponding statement and breaks out of the `switch` statement. If it doesn't find any of the values, then it will output a default statement.

The syntax for it looks as follows:

```
switch( expression ){

    case constant1:  statement1; break;
    case constant2:  statement2; break;
    .
    .
    .
    default: default statement; break;

}
```

This looks very familiar to the `else if` statements, but this is more sophisticated. Here is an example:

```
#include <iostream>
#include <conio.h>

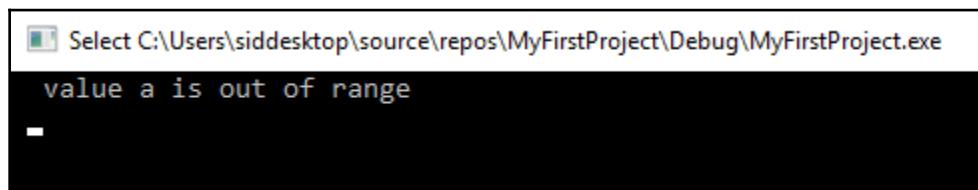
// Program prints out values to screen

int main()
{
    int a = 28;

    switch (a)
    {
        case 1: std::cout << " value of a is " << a << std::endl; break;
        case 2: std::cout << " value of a is " << a << std::endl; break;
        case 3: std::cout << " value of a is " << a << std::endl; break;
        case 4: std::cout << " value of a is " << a << std::endl; break;
        case 5: std::cout << " value of a is " << a << std::endl; break;
        default: std::cout << " value a is out of range " << std::endl; break;
    }

    _getch();
    return 0;
}
```

The output is as follows:



Change the value of `a` to equal 2 and you will see that it prints out the statement to when case 2 is correct.

Also note that it is important to add the `break` statement. If you forget to add it, then the program will not break out of the statement.

Functions

So far, we have written all of our code in the main function. This is fine if you are doing a single task, but once you start doing more with a program, the code will become bigger and over a period of time everything will be in the main function, which will look very confusing.

With functions, you can break your code up into smaller, manageable chunks. This will make you able to structure your program better.

A function has the following syntax:

```
type function name (parameter1, parameter2) {statements;}
```

Going from left to right, the `type` here is the return type. After performing a statement, a function is capable of returning a value. This value could be of any type, so we specify a type here. A function has only one variable at a time.

The function name is the name of the function itself.

Then, inside brackets, you will pass in parameters. These parameters are variables of a certain type that are passed into the function to perform a certain function.

Here as an example: two parameters are passed in but you can pass as many parameters you want. You can pass in more than one parameter per function and each parameter is separated by a comma.

Let's look at this example:

```
#include <iostream>
#include <conio.h>

// Program prints out values to screen

void add(int a, int b)
{
    int c = a + b;

    std::cout << "Sum of " << a << " and " << b << " is " << c <<
    std::endl;
}
int main()
{
    int x = 28;
    int y = 12;
```

```
    add(x, y);  
  
    getch();  
    return 0;  
}
```

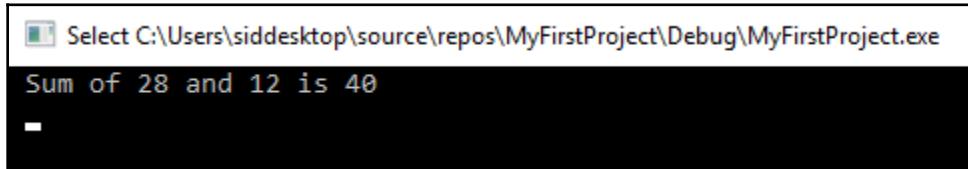
Here, we create a new function called `add`. For now, make sure the functions are added before the `main` function, otherwise `main` will not know that the function exists.

The `add` function doesn't return anything so we use the `void` keyword at the start of the function. Not all functions have to return a value. Next, we name the function `add` and then pass in two parameters, which are `a` and `b` of type `int`.

In the function, we create a new variable called `c` of type `int`, add the values of the arguments passed in, and assign it to `c`. The new `add` function finally prints out the value of `c`.

Furthermore, in the `main` function, we create two variables called `x` and `y` of type `int`, call the `add` function, and pass in `x` and `y` as arguments.

When we call the function, we pass the value of `x` to `a` and the value of `y` to `b`, which is added and stored in `c` to get the following output:



When you create new functions, make sure they are written above the `main` function, otherwise it will not be able to see the functions and the compiler will throw errors.

Now let's write one more function. This time, we will make sure the function returns a value. Create a new function called `multiply`, as follows:

```
int multiply(int a, int b) {  
  
    return a * b;  
}
```

In the `main` function, after we've called the `add` function, add the following lines:

```
add(x, y);

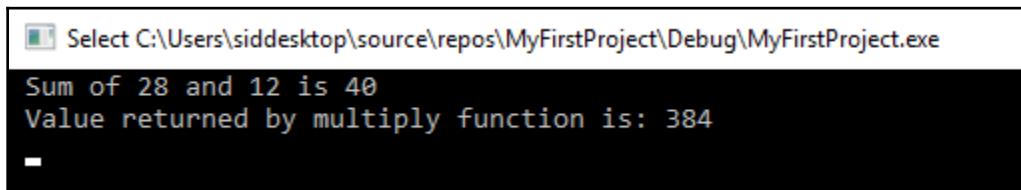
int c = multiply(12, 32);

std::cout << "Value returned by multiply function is: " << c <<
std::endl;
```

In the `multiply` function, we have a return type of `int`, so the function will expect a return value at the end of the function, which we return using the `return` keyword. The returned value is variable `a` multiplied by variable `b`.

In the `main` function, we create a new variable called `c`; call the `multiply` function and pass in `12` and `32`. After being multiplied, the return value will be assigned to the value of `c`. After this, we print out the value of `c` in the `main` function.

The output of this is as follows:



```
Select C:\Users\sid\Desktop\source\repos\MyFirstProject\Debug\MyFirstProject.exe
Sum of 28 and 12 is 40
Value returned by multiply function is: 384
-
```

We can have a function with the same name but we can pass in different variables or different numbers of them. This is called **function overloading**.

Create a new function called `multiply`, but this time, pass in floats and set the return value to a float as well:

```
float multiply(float a, float b) {
    return a * b;
}
```

This is called function overloading, where the function name is the same but it takes different types of arguments.

In the `main` function, after we've printed the value of `c`, add the following code:

```
float d = multiply(8.352f, -12.365f);
std::cout << "Value returned by multiply function is: " << d << std::endl;
```

So, what is this `f` after the float value? Well, `f` just converts the doubles to floats. If we don't add the `f`, then the value will be treated as a double by the compiler.

When you run the program, you'll get the value of `d` printed out:

```
C:\Users\siddesktop\source\repos\MyFirstProject\Debug\MyFirstProject.exe
Sum of 28 and 12 is 40
Value returned by multiply function is: 384
Value returned by multiply function is: -103.272
```

Scope of variables

You may have noticed that we have two variables called `c` in the program right now. There is a `c` in the `main` function as well as a `c` in the `add` function. How is it that they are both named `c` but have different values?

In C++, there is the concept of a local variable. This means that the definition of a variable is confined to the local block of code it is defined in. Consequently, the `c` variable in the `add` function is treated differently to the `c` variable in the `main` function.

There are also global variables, which need to be declared outside of the function or block of code. Any piece of code written between curly brackets is considered to be a block of code. Consequently, for a variable to be considered a global variable, it needs to be in the body of the program or it needs to be declared outside a block of code of a function.

Arrays

So far, we have only looked at single variables, but what if we want a bunch of variables grouped together? Like the ages of all the students in a class, for example. You can keep creating separate variables: `a`, `b`, `c`, `d`, and so on, and to access each you would have to call each of them, which is cumbersome, as you won't know what kind of data they hold.

To organize data better, we can use arrays. Arrays use continuous memory space to store values in a series and you can access each element with an index number.

The syntax for arrays is as follows:

```
type name [size] = { value0, value1, ...., valuesize-1};
```

So, we can store the age of five students as follows:

```
int age[5] = {12, 6, 18, 7, 9};
```

When creating an array with a set number of values, you don't have to specify a size but it is a good idea to do so. To access each value, we use the index from 0-4 as the first element with a value of 12 at the 0th index and the last element, 9, in the fourth index.

Let's see how to use this in code:

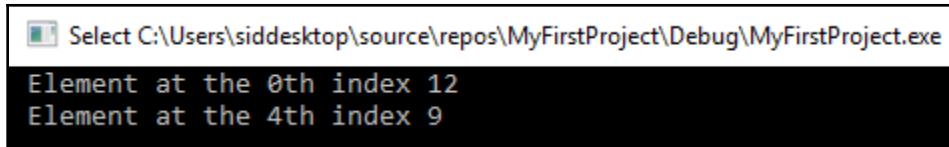
```
#include <iostream>
#include <conio.h>

// Program prints out values to screen
int main()
{
    int age[5] = { 12, 6, 18, 7, 9 };

    std::cout << "Element at the 0th index " << age[0] << std::endl;
    std::cout << "Element at the 4th index " << age[4] << std::endl;

    _getch();
    return 0;
}
```

The output is as follows:



To access each element in the array you can use the `for` loop:

```
#include <iostream>
#include <conio.h>

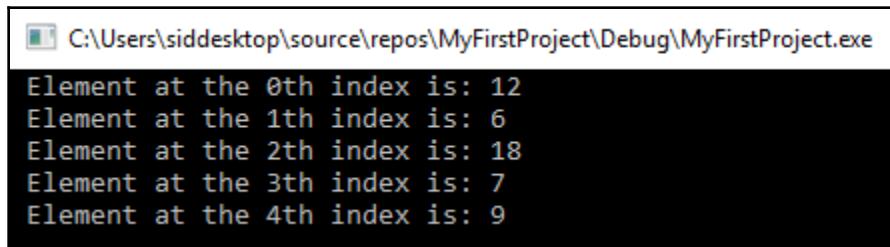
// Program prints out values to screen
int main()
{
    int age[5] = { 12, 6, 18, 7, 9 };

    for (int i = 0; i < 5; i++) {

        std::cout << "Element at the " << i << "th index is: " <<
        age[i] << std::endl;
    }
}
```

```
    }
    getch();
    return 0;
}
```

The output of this is as follows:



```
C:\Users\siddesktop\source\repos\MyFirstProject\Debug\MyFirstProject.exe
Element at the 0th index is: 12
Element at the 1th index is: 6
Element at the 2th index is: 18
Element at the 3th index is: 7
Element at the 4th index is: 9
```

Instead of calling `age[0]`, and so on, we use the `i` index from the `for` loop itself and pass it into the `age` array to print out the index and the value stored at that index.

The `age` array is a single-dimension array. In graphics programming, we have seen that we use a two-dimensional array, which is mostly a 4×4 matrix. Let's look at an example of a two-dimensional 4×4 array. A two-dimensional array is defined as follows:

```
int matrix[4][4] = {
{2, 8, 10, -5},
{15, 21, 22, 32},
{3, 0, 19, 5},
{5, 7, -23, 18}
};
```

To access each element, you will use a nested `for` loop.

Let's look at this in the following code:

```
#include <iostream>
#include <conio.h>

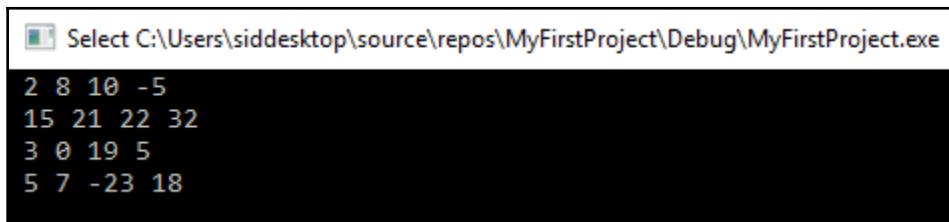
// Program prints out values to screen
int main()
{

    int matrix[4][4] = {
        {2, 8, 10, -5},
        {15, 21, 22, 32},
        {3, 0, 19, 5},
        {5, 7, -23, 18}
    };
}
```

```
    for (int x = 0; x < 4; x++) {
        for (int y = 0; y < 4; y++) {
            std::cout << matrix[x][y] << " ";
        }
        std::cout << "" << std::endl;
    }

    _getch();
    return 0;
}
```

The output is as follows:



```
2 8 10 -5
15 21 22 32
3 0 19 5
5 7 -23 18
```

As a test, create two matrices and attempt to carry out matrix multiplication.

You can even pass arrays as parameters to functions, shown as follows.

Here, the `matrixPrinter` function doesn't return anything but prints out the values stored in each element of the 4x4 matrix:

```
#include <iostream>
#include <conio.h>

void matrixPrinter(int a[4][4]) {

    for (int x = 0; x < 4; x++) {
        for (int y = 0; y < 4; y++) {
            std::cout << a[x][y] << " ";
        }
        std::cout << "" << std::endl;
    }
}

// Program prints out values to screen
int main()
{

    int matrix[4][4] = {
```

```
        {2, 8, 10, -5},  
        {15, 21, 22, 32},  
        {3, 0, 19, 5},  
        {5, 7, -23, 18}  
    };  
  
    matrixPrinter(matrix);  
  
    _getch();  
    return 0;  
}
```

We can even use an array of `char` to create a string of words. Unlike `int` and `float` arrays, the characters in an array don't have to be in curly brackets and they don't need to be separated by a comma.

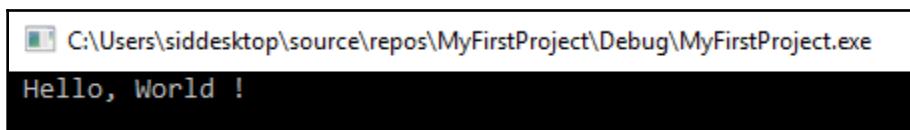
To create a character array, you will define it as follows:

```
char name[] = "Hello, World !";
```

You can print out the values just by calling out the name of the array, as follows:

```
#include <iostream>  
#include <conio.h>  
  
// Program prints out values to screen  
int main()  
{  
  
    char name[] = "Hello, World !";  
  
    std::cout << name << std::endl;  
  
    _getch();  
    return 0;  
}
```

The output of this is as follows:



Pointers

Whenever we declare new variables so that we can store values in them, we actually send a memory allocation request to the operating system. The OS will try to reserve a block of continuous memory for our application if there is enough free memory left.

When we want to access the value stored in that memory space, we call the variable name.

We don't have to worry about the memory location where we have stored the value. However, what if we want to get the address of the location where the variable is stored?

The address that locates the variable within the memory is called a reference to the variable. To access this, we use an address-of & operator. To get the address location, we place the operator before the variable.

Pointers are variables, and like any other variables they are used to store a value; however, this specific variable type allows the storage of the address—the reference—of another variable.

In C/C++, every variable can also be declared as a pointer that holds a reference to a value of a certain data type by preceding its variable name with an asterisk (*). This means, for example, that an int pointer holds a reference to a memory address where a value of an int may be stored.

A pointer can be used with any built-in or custom data type. If we access the value of a pointer variable, we will simply get the memory address it references. So, in order to access the actual value a pointer variable references, we have to use the so-called de-referencing operator (*).

If we have a variable called age and assign a value to it, to get the reference address location, we use &age to store this address in a variable. To store the reference address, we can't just use a regular variable, we have to use a pointer variable and use the deference operator before it to access the address, as follows:

```
int age = 10;
int *location = &age;
```

Here, the pointer location will store the address of where the age variable's value is stored.

If we print the value of location, we will get the reference address where age is stored:

```
#include <iostream>
#include <conio.h>
// Program prints out values to screen
int main()
```

```
{  
    int age = 10;  
    int *location = &age;  
    std::cout << location << std::endl;  
    _getch();  
    return 0;  
}
```

This is the output:



```
C:\Users\siddesktop\source\repos\MyFirstProject\Debug\MyFirstProject.exe  
00AFFA00
```

This value might be different for you, as the location will be different from machine to machine.

To get the location of where the `location` variable itself is stored, we can print out `&location` to get this value as well.

This is the memory location of the variable on my system memory:



```
C:\Users\siddesktop\source\repos\MyFirstProject\Debug\MyFirstProject.exe  
00AFFA00
```

Let's look at another example, as follows:

```
#include <iostream>  
#include <conio.h>  
  
// Program prints out values to screen  
int main()  
{  
    int age = 18;  
    int *pointer;  
  
    pointer = &age;  
  
    *pointer = 12;  
  
    std::cout << age << std::endl;
```

```
    _getch();
    return 0;
}
```

Here, we create two `int` variables; one is a regular `int` and the other is a pointer type.

We first set the `age` variable equal to 18, then we set the address of `age` and assign it to the pointer variable called `pointer`.

The `int` pointer is now pointing to the same address where the `age` variable stores its `int` value.

Next, use the de-reference operator on the `pointer` variable to give us access to the `int` values stored at the referenced address and change the current value to 12.

Now, when we print out the value of the `age` variable, we will see that the previous statement has indeed changed the value of the `age` variable. A null pointer is a pointer that is not pointing to anything, set as follows:

```
int *p = nullptr;
```

Pointers are very much associated with arrays. As arrays are nothing but a continuous sequence of memory, we can use pointers with them.

Consider our arrays example from the arrays section as follows:

```
int age[5] = { 12, 6, 18 , 7, 9 };
```

Instead of using the index, we can use pointers to point to the values in the array.

Consider the following code:

```
#include <iostream>
#include <conio.h>

// Program prints out values to screen
int main()
{
    int *p = nullptr;
    int age[5] = { 12, 6, 18 , 7, 9 };
    p = age;
    std::cout << *p << std::endl;

    p++;

    std::cout << *p << std::endl;
```

```
    std::cout << *(p + 3) << std::endl;
    std::cout << *p << std::endl;

    _getch();
    return 0;
}
```

In the main function, we create a pointer called `pointer`, as well as an array with five elements. We assign the array to the pointer. This causes the pointer to get the location of the address of the first element of the array. So, when we print the value pointed to by the pointer, we get the value of the first element of the array.

With `pointer`, we can also increment and decrement as a regular `int`. However, unlike a regular `int` increment, which increments the value of the variable when you increment a pointer, it will point to the next memory location. So, when we increment `p` it is now pointing to the next memory location of the array. Incrementing and decrementing a pointer precisely means moving the referenced address by a certain number of bytes. The number of bytes depends on the data type that is used for the `pointer` variable .

Here, the pointer is type `int`, so when we move the pointer by one, it moves 4 bytes and points to the next integer. When we print the value that `p` is pointing to now, it prints the second element's value.

We can also get the value of other elements in the array by getting the pointer's current location and by adding to it the n^{th} number you want to get from the current location using `* (p + n)`, where n is the n^{th} element from `p`. So, when we do `* (p + 3)`, we will get the third element from where `p` is pointing to currently. Since `p` was incremented to the second element, the third element from the second element is the fifth element, and so the value of the fifth element is printed out.

However, this doesn't change the location to which `p` is pointing, which is still the second position.

Here is the output:

```
12
6
9
6
-
```

Structs

Structures or structs are used to group data together. A struct can have different data elements in it, called members, integers, floats, chars, and so on. You can create many objects of a similar struct and store values in the struct for data management.

The syntax of a struct is shown as follows:

```
struct name{  
    type1 name1;  
    type2 name2;  
    .  
    .  
} ;
```

An object of a struct can be created as follows:

```
struct_name      object_name;
```

An object is an instance of a struct to which we can assign properties to the data types we created when creating the struct. An example of this is as follows.

In a situation in which you want to maintain a database of students' ages and the height of a section, your struct definition will look like this:

```
struct student {  
    int age;  
    float height;  
};
```

Now you can create an array of objects and store the values for each student:

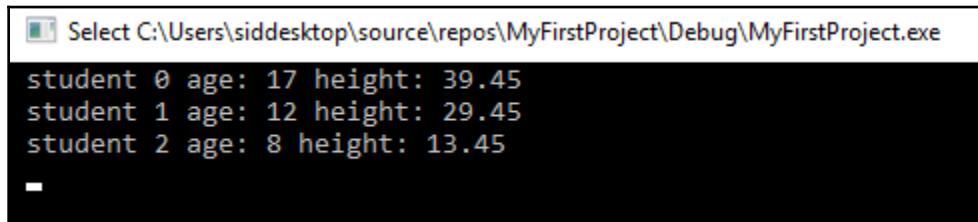
```
int main()  
{  
    student section[3];  
  
    section[0].age = 17;  
    section[0].height = 39.45f;  
  
    section[1].age = 12;  
    section[1].height = 29.45f;  
  
    section[2].age = 8;
```

```
section[2].height = 13.45f;

for (int i = 0; i < 3; i++) {
    std::cout << "student " << i << " age: " << section[i].age << "
height: " << section[i].height << std::endl;
}

_getch();
return 0;
}
```

Here is the output of this:



```
Select C:\Users\siddesktop\source\repos\MyFirstProject\Debug\MyFirstProject.exe
student 0 age: 17 height: 39.45
student 1 age: 12 height: 29.45
student 2 age: 8 height: 13.45
-
```

Enums

Enums are used for enumerating items in a list. When comparing items, it is easier to compare names rather than just numbers. For example, the days in a week are Monday to Sunday. In a program, we will assign Monday to 0, Tuesday to 1, and Sunday to 7, for example. To check whether today is Friday, you will have to count to and arrive at 5. However, wouldn't it be easier to just check if `Today == Friday`?

For this, we have enumerations, declared as follows:

```
enum name{
value1,
value2,
.
.
.
};
```

So, in our example, it would be something like this:

```
#include <iostream>
#include <conio.h>
```

```
// Program prints out values to screen

enum Weekdays {
    Monday = 0,
    Tuesday,
    Wednesday,
    Thursday,
    Friday,
    Saturday,
    Sunday,
};

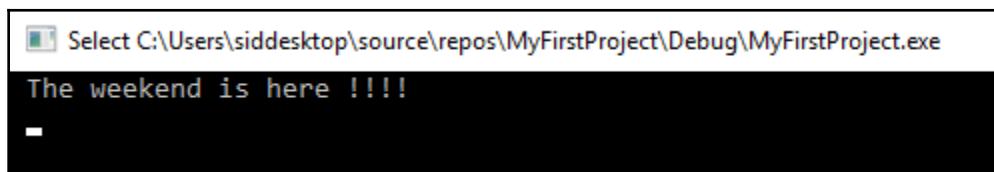
int main()
{
    Weekdays today;

    today = Friday;

    if (today == Friday) {
        std::cout << "The weekend is here !!!!"
        << std::endl;
    }

    _getch();
    return 0;
}
```

The output of this is as follows:



Also note, here, `Monday = 0`. If we don't use initializers, the first item's value is set to 0. Each following item that does not use an initializer will use the value of the preceding item plus 1 for its value.

Classes

In C++, structs and classes are identical. You can do absolutely the same thing with both of them. The only difference is the default access specifier: public for structs and private for classes.

The declaration of a class looks like the following code:

```
class name{  
    access specifier:  
    name();  
    ~name();  
  
    member1;  
    member2;  
};
```

A class starts with `class` keyword, followed by the name of the class.

In a class, we first specify the access specifiers. There are three access specifiers: `public`, `private`, and `protected`:

- `public`: All members are accessible from anywhere
- `private`: Members are accessible from within the class itself only
- `protected`: Members are accessed by other classes that inherit from the class

By default, all members are `private`.

`name()` and `~name()` are called the constructor and destructor of a class. They have the same name as the name of the class itself.

The constructor is a special function that gets called when you create a new object of the class. The destructor is called when the object is destroyed.

We can customize a constructor to set values before using the member variables. This is called constructor overloading.

Notice that, although the constructor and destructor are functions, no return is provided. This is because they are not there for returning values.

Let's look at an example of a class where we create a class called `shape`. This has two member variables for sides `a` and `b`, and a member function, which calculates and prints the area:

```
class shape {
    int a, b;
public:

    shape(int _length, int _width) {
        a = _length;
        b = _width;

        std::cout << "length is: " << a << " width is: " << b <<
        std::endl;
    }

    void area() {
        std::cout << "Area is: " << a * b << std::endl;
    }
};
```

We use the class by creating objects of the class.

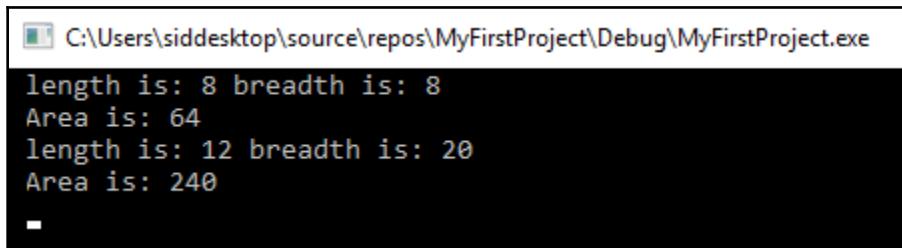
Here, we create two objects, called `square` and `rectangle`. We set the values by calling the custom constructor, which sets the value of `a` and `b`. Then, we call the `area` function of the object by using the dot operator by pressing the (.) button on the keyboard after typing the name of the object:

```
int main()
{
    shape square(8, 8);
    square.area();

    shape rectangle(12, 20);
    rectangle.area();

    getch();
    return 0;
}
```

The output is as follows:



```
C:\Users\siddesktop\source\repos\MyFirstProject\Debug\MyFirstProject.exe
length is: 8 breadth is: 8
Area is: 64
length is: 12 breadth is: 20
Area is: 240
```

Inheritance

One of the key features of C++ is inheritance, with which we can create classes that are derived from other classes so that the derived or the child class automatically includes some of its parent's member variables and functions.

For example, we looked at the `shape` class. From this, we can have a separate class called `circle` and another class called `triangle`, which has same properties like area as other shapes, such as area.

The syntax for an inherited class is as follows:

```
class inheritedClassName: accessSpecifier parenClassName{
};
```

`accessSpecifier` could be `public`, `private`, or `protected` depending upon the minimum access level you want to provide to the parent member variables and functions.

Let's look at an example of inheritance. Consider the same `shape` class, which will be the parent class:

```
class shape {
protected:
    float a, b;
public:
    void setValues(float _length, float _width)
    {
        a = _length;
        b = _width;
```

```
        std::cout << "length is: " << a << " width is: " << b <<
        std::endl;
    }
void area() {
    std::cout << "Area is: " << a * b << std::endl;
}
};
```

Since we want the `triangle` class to access `a` and `b` of the parent class, we have to set the access specifier to `protected`, as shown previously, otherwise, it will be set to `private` by default. In addition to this, we also change the data type to `floats` for more precision. After doing this, we create a `setValues` function instead of the constructor to set the values for `a` and `b`. We then create a child class of `shape` and call it `triangle`:

```
class triangle : public shape {

public:
    void area() {

        std::cout << "Area of a Triangle is: " << 0.5f * a * b <<
        std::endl;
    }
};
```

Due to inheritance from the `shape` class, we don't have to add `a` and `b` member variables, and we don't need to add the `setValues` member function either, as this is inherited from the `shape` class. We just add a new function called `area`, which calculates the area of a triangle.

In the main function, we create an object of the `triangle` class, set the values, and print the area, shown as follows:

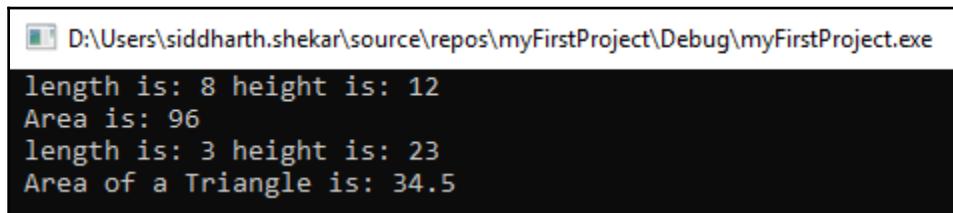
```
int main()
{
    shape rectangle;
    rectangle.setValues(8.0f, 12.0f);
    rectangle.area();

    triangle tri;
    tri.setValues(3.0f, 23.0f);
    tri.area();

    _getch();
}
```

```
    return 0;  
}
```

Here is the output of this:



```
D:\Users\siddharth.shekar\source\repos\myFirstProject\Debug\myFirstProject.exe  
length is: 8 height is: 12  
Area is: 96  
length is: 3 height is: 23  
Area of a Triangle is: 34.5
```

To calculate the area of circle, we modify the shape class and add a new overloaded setValues function, as follows:

```
#include <iostream>  
#include <conio.h>  
  
class shape {  
  
protected:  
  
    float a, b;  
  
public:  
  
    void setValues(float _length, float _width)  
    {  
        a = _length;  
        b = _width;  
  
        std::cout << "length is: " << a << " height is: " << b <<  
        std::endl;  
    }  
  
    void setValues(float _a)  
    {  
        a = _a;  
    }  
    void area() {  
  
        std::cout << "Area is: " << a * b << std::endl;  
    }  
};
```

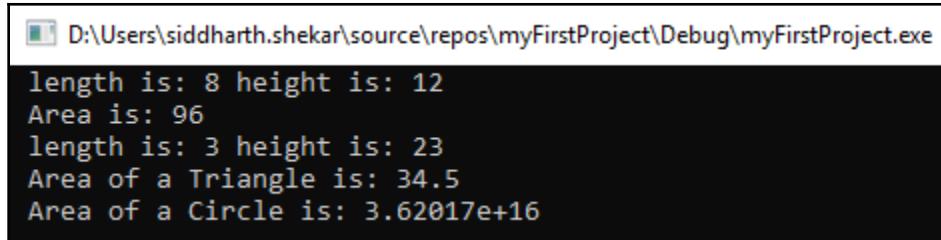
We will then add a new inherited class, called `circle`:

```
class circle : public shape {  
  
public:  
    void area() {  
  
        std::cout << "Area of a Circle is: " << 3.14f * a * a <<  
        std::endl;  
    }  
  
};
```

In the main function, we create a new `circle` object, set the radius, and print the area:

```
int main()  
{  
  
    shape rectangle;  
    rectangle.setValues(8.0f, 12.0f);  
    rectangle.area();  
  
    triangle tri;  
    tri.setValues(3.0f, 23.0f);  
    tri.area();  
  
    circle c;  
    c.setValues(5.0f);  
    c.area();  
  
  
    _getch();  
    return 0;  
}
```

Here is the output of this:



```
D:\Users\siddharth.shekhar\source\repos\myFirstProject\Debug\myFirstProject.exe  
length is: 8 height is: 12  
Area is: 96  
length is: 3 height is: 23  
Area of a Triangle is: 34.5  
Area of a Circle is: 3.62017e+16
```

Summary

In this chapter, we covered the basics of programming—from what variables are and how to store values in them, to looking at operators and statements, to how to decide when each is required. After that, we looked at iterators and functions, which can be used to make our job simpler and automate the code as much as possible. Arrays and pointers help us to group and store data of a similar type, and with `struct` and `enum` we can create custom data types. Finally, we looked at classes and inheritance, which is the crux of using C++ and makes it convenient to define our data types with custom properties.

In the next chapter, we will look at the foundation of graphics programming and explore how 3D and 2D objects are displayed on the screen.

2

Mathematics and Graphics Concepts

Before we begin rendering objects , it is essential that you are familiar with the math that will be used for the projects in later chapters. Mathematics plays a crucial role in games development, and graphics programming generally uses vectors and matrices extensively. In this chapter, you will see where these math concepts come in handy. We'll first go over some key mathematical concepts, and then apply them to working with space transformations and a render pipeline. There are dedicated books which cover all the math related topics required for game development. Since we will be covering Graphics Programming with C++, other mathematics topics are out of the scope of this book.

In the following chapters we will be using OpenGL or Vulkan graphics APIs for rendering our objects and will use the GLM math library for doing the maths. In this chapter we will explore the process of creating a 3D object in a virtual world using matrix and vector transforms. Then we will look at how we transform a 3D point into a 2D location using space transforms, and how the graphics pipeline helps us to achieve this.

We will cover the following topics:

- Learn basics about the 3D coordinate system.
- The basics of vector and matrix math and transforms
- Getting familiar with GLM C++ Math Library
- How to implement Space transformations in 3D graphics
- Understanding the flow of Graphics pipeline

3D coordinate system

When we want to specify a location, we have to specify a coordinate system before we specify a point. A 3D coordinate system has three axes: the X axis, Y axis, and Z axis. The three axes start from the origin where the three axes intersect.

The positive X axis starts from the origin and starts moving endlessly in a direction and the negative X axis moves in the opposite direction. The positive Y axis starts from the origin as the X axis and starts going in the upward direction at 90 degrees to the X axis and the negative Y moves in the opposite direction. This describes a 2D XY plane which forms the basis for a 2D coordinate system.

The positive Z axis also starts from the origin as the X and Y axis and is perpendicular in to X and Y axis. The positive Z axis can go in either direction to the XY plane forming a 3D coordinate system.

Assuming the positive X axis is to the right and positive Y axis is to the up direction then the Z axis can either go into or comes out of the screen as the Z axis is perpendicular to both the X and Y axis.

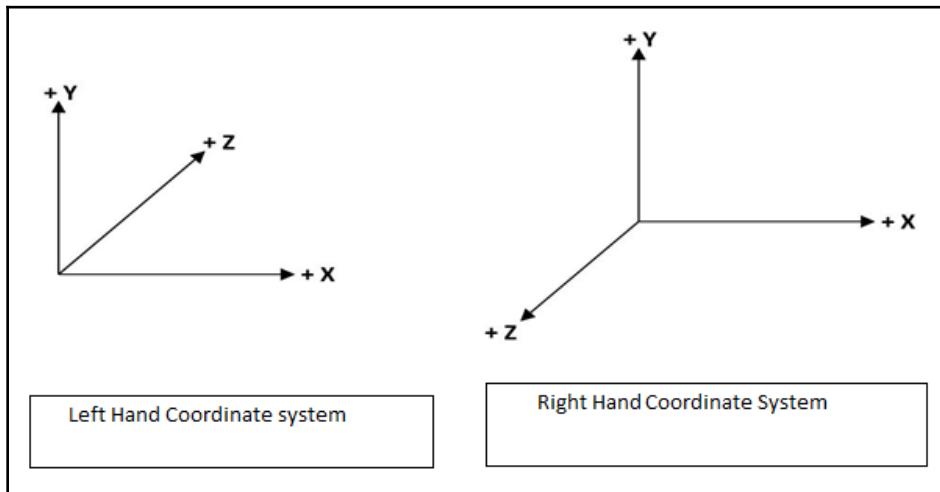
If the positive Z axis moves into the screen, it is called a **Left Hand Coordinate System**. If the positive Z axis comes out of the screen, then it is called **Right Hand Coordinate System**.

Open out your right arm with the palm facing towards you and make a fist. Extend the thumb to the right, and extend the index finger upwards. Now extend the middle finger which faces towards you. This is the right hand coordinate system.

The thumb represents the direction of the positive X axis, the index finger represents the direction of the positive Y axis and the middle finger is the direction of the positive Z axis. OpenGL, Vulkan or any graphics framework that use them also use this coordinate system.

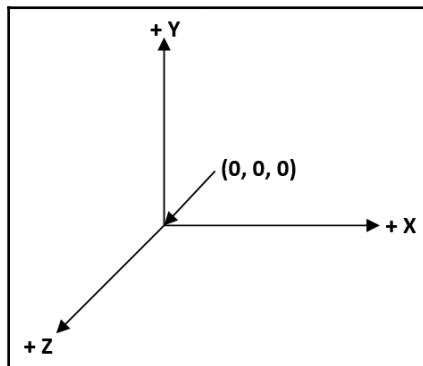
For the left-hand coordinate system, extend your left arm out with the palm of the hand facing away from you and make a fist. Next extend the thumb and index finger in the right and upwards direction. Now extend the middle finger away from you. In this case, the thumb also represents the direction of the X axis and the index finger is pointing in the direction of the positive Y axis which faces in the right and upwards direction. The Z axis (which is the middle finger) is now facing away from you. This is the **Left Hand Coordinate System**. Direct3D of DirectX uses this coordinate system.

In this book, since we are going to be covering OpenGL and Vulkan, we will be using the **Right Hand Coordinate System**:



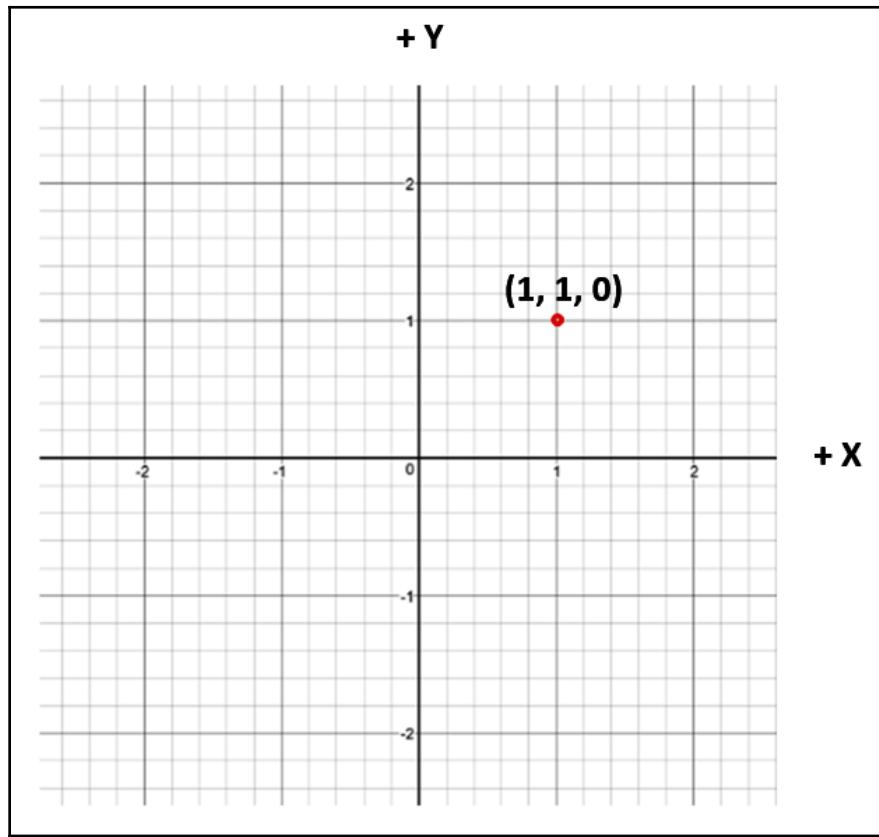
Points

After we have defined the coordinate system, we can now specify what a point is. A 3D point is a location in 3D space specified by distance in the X, Y and Z axis from the origin of the coordinate system. It is specified as (X, Y, Z) where X, Y and Z are the distance from the origin. But what is this origin you speak of? Origin is also a point where the 3 axes meet. The origin is at (0, 0, 0), and the location of the origin is specified in the coordinate system, as shown as follows:



For specifying points within a coordinate system, imagine that in each direction, the axis is made of a smaller unit. This unit could be 1 millimeter, 1 centimeter or 1 kilometer, for example, depending upon how much data you have.

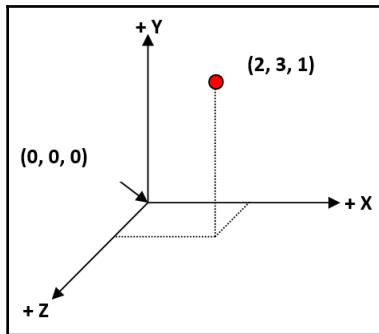
If we just look at the X and Y axis, this would look something like the picture as follows:



If we look at the X axis, the value 1 and 2 specify the distance along the axis of that point from the origin which is at value 0. So, point 1 in the X axis is at $(1,0,0)$ along the X axis. Similarly, point 1 along the Y -axis is at $(0,1,0)$.

In addition, the location of the red dot will be at $(1,1,0)$ - that is, 1 unit along the X axis and 1 along the Y axis. Since Z is 0 we specify the value of it as 0.

Similarly, points in 3D space are represented as follows:



Vectors

A vector is a quantity that has a magnitude and a direction. Examples of quantities that have a magnitude and direction are displacement, velocity, acceleration and force. With displacement, you can specify the direction as well as the net distance moved by the object.

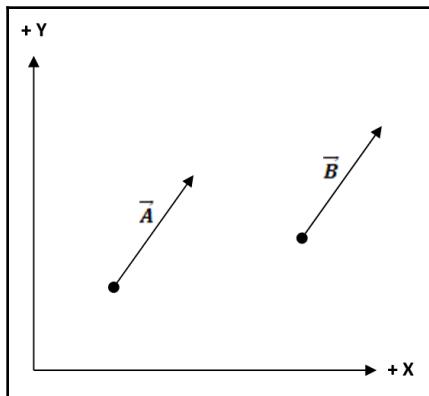
The difference between speed and velocity is that speed only specifies the speed with which any object is moving but doesn't establish the direction the object is moving in. However, velocity specifies the magnitude, which includes the speed and the direction. Similar to velocity, we have acceleration. A form of acceleration is gravity, and we know that this always acts downwards and is always approximately 9.81 m/s^2 . Well atleast on Earth. It is $1/6^{\text{th}}$ of this on the moon.

An example of force is weight. Weight also acts downwards and is calculated as mass multiplied acceleration.

Vectors are graphically represented by a pointed line segment with the length of the line denoting the magnitude of the vector and the pointed arrow showing the direction of the vector. We can move around a vector as doing this doesn't change the magnitude or the direction of it.

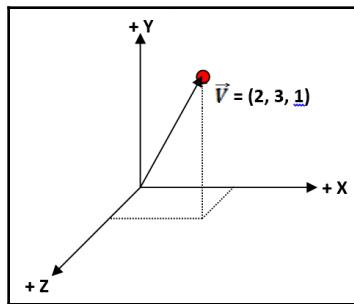
Two vectors are said to be equal if they both have the same magnitude and direction even if they are in different locations. Vectors are denoted by arrow marks above the letter

The as follows vectors \vec{A} and \vec{B} are starting in different locations. Since the direction and magnitude of the arrows are the same, they are equal:



In a 3D coordinate system, a vector is specified by the coordinates with respect to the coordinate system. In the example as follows, the vector \vec{V} is equal to $(2, 3, 1)$ and is also denoted as is denoted as:

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



Vector operations

Just like scalar information, vectors can also be added, subtracted and multiplied with each other. Suppose you have two vectors: and \vec{A} and \vec{B} , with $\vec{A} = (a_x, a_y, a_z)$ and $\vec{B} = (b_x, b_y, b_z)$. In this case, let us see how you will add and subtract these vectors with each other.

When adding the vectors, we add the components individually to give a new vector:

$$\vec{C} = \vec{A} + \vec{B}$$

$$\vec{C} = ((ax + bx), (ay + by), (az + bz))$$

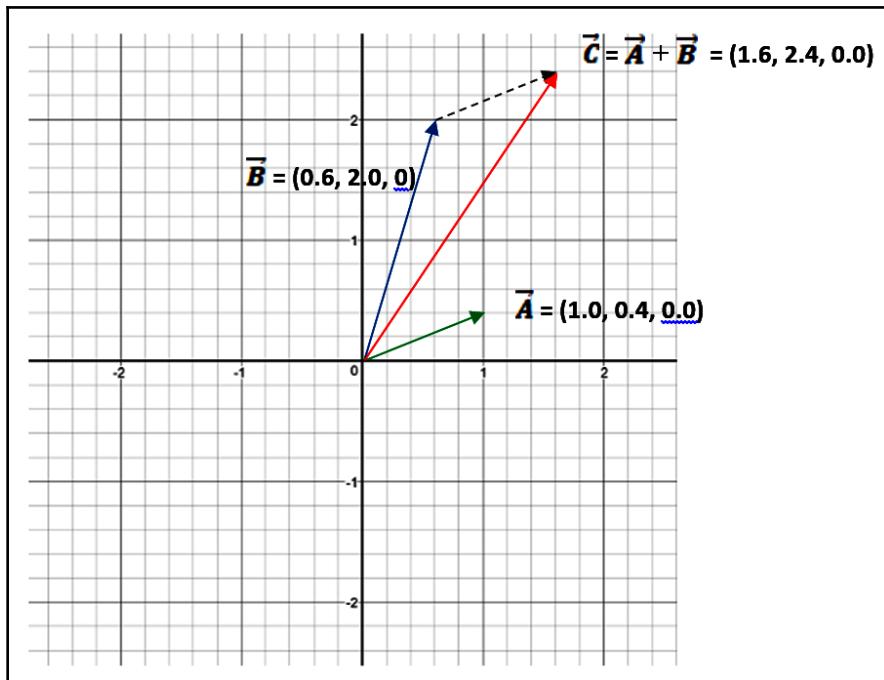
Let's now visualize the addition of two vectors in a graph as shown in the following image.

The Z value is kept as 0.0 for convenience:

Here, $\vec{A} = (1.0, 0.4, 0.0)$ and $\vec{B} = (0.6, 2.0, 0.0)$

So the resultant vector $\vec{C} = \vec{A} + \vec{B} = (1.0 + 0.6, 0.4 + 2.0, 0.0 + 0.0)$

$$= (1.6, 2.4, 0.0)$$



Vectors are also commutative meaning $\vec{A} + \vec{B}$ will give the same result as $\vec{B} + \vec{A}$. But if we add \vec{B} to \vec{A} then in the diagram above the dotted line will go from the tip of the blue arrow to the tip of the red arrow.

Furthermore, in vector subtraction, we subtract the individual components of the vectors to give a new vector:

$$\vec{C} = \vec{A} - \vec{B}$$

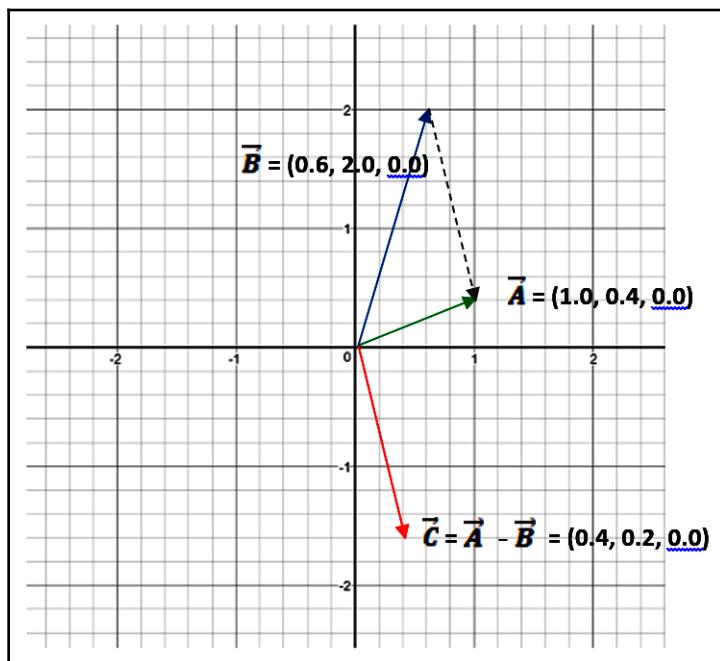
$$\vec{C} = ((ax - bx), (ay - by), (az - bz))$$

Now let's visualize the subtraction of 2 vectors in a graph as shown in the following image:

Here $\vec{A} = (1.0, 0.4, 0.0)$ and $\vec{B} = (0.6, 2.0, 0.0)$

So the resultant vector $\vec{C} = \vec{A} - \vec{B} = (1.0 - 0.6, 0.4 - 2.0, 0.0 - 0.0)$

$$= (0.4, -1.6, 0.0)$$



If vectors A and B are equal, the result will be a zero vector with all three components being zero.

If $\vec{A} = \vec{B}$. This means $a_x = b_x$, $a_y = b_y$, $a_z = b_z$, then

$$\vec{C} = \vec{A} - \vec{B} = (0, 0, 0)$$

We can multiply a scalar with a vector. The result is again a vector with each component of the vector multiplied by the scalar.

For example, if \vec{A} is multiplied by a single value of s we will have the following result:

$$\vec{C} = s \times \vec{A}$$

$$\vec{C} = s \times (a_x, a_y, a_z)$$

$$\vec{C} = (s \times a_x, s \times a_y, s \times a_z)$$

If $\vec{C} = (3, -5, 7)$ and $s = 0.5$ then $\vec{C} = s\vec{A}$ is

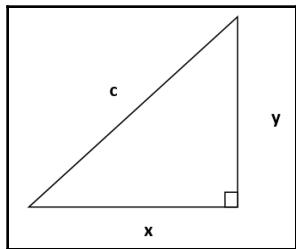
$$= (3 \times 0.5, -5 \times 0.5, 7 \times 0.5)$$

$$= (1.5, -2.5, 3.5)$$

Vector magnitude

The magnitude of the vector is equal to the length of the vector itself. But how do we calculate it mathematically? .

The magnitude of a vector is given by the Pythagorean theorem, which specifies that in a right triangle , the square of length of a diagonal is equal to the sum of the squares of the adjacent sides. So when we look at the right triangle as follows, $c^2 = x^2 + y^2$.



This can be extended to three dimensions with $c^2 = x^2 + y^2 + z^2$.

Magnitudes of vectors are indicated by double vertical bars, so the magnitude of a vector \vec{A} is denoted by $\|A\|$. The magnitude is always greater than or equal to zero.

So, if vector $A = (X, Y, Z)$ then the magnitude is given by the following equation:

$$\|A\| = \sqrt{x^2 + y^2 + z^2}$$

If $\vec{A} = (3, -5, 7)$, Then:

$$\|A\| = \sqrt{x^2 + y^2 + z^2}$$

$$= \sqrt{3^2 + (-5)^2 + 7^2}$$

$$= 9.110$$

The vector \vec{A} is therefore 9.11 units long

Unit vector

In some cases, we don't care about the magnitude of the vector, we just want to know the direction of the vector. To find this out, we want the length of the vector in the X, Y and Z direction to be equal to 1. Such vectors are called unit vectors or normalized vectors.

In unit vectors, the X, Y, and Z components of the vector are divided by the magnitude to create a vector of unit length. They are denoted by a hat on top of the vector name instead of an arrow. So, a unit vector of A will be denoted as \hat{a} .

$$\hat{a} = \vec{A} / \|A\|$$

$$\hat{a} = (x / \|A\|, y / \|A\|, z / \|A\|)$$

When a vector is converted into a unit vector, it is said to be normalized. Meaning the value is always between 0.0 and 1.0. The original value has been rescaled to be in this range. Let's normalize the vector $\vec{A} = (3, -5, 7)$

First we have to calculate the magnitude of \vec{A} , which we have already done before i.e 9.11

So, unit vector $\hat{a} = \vec{A} / \|A\|$

$$\begin{aligned} &= (3, -5, 7) / 9.11 \\ &= (3 / 9.11, -5 / 9.11, 7 / 9.11) \\ &= (0.33, -0.51, 0.77) \end{aligned}$$

Dot product

A dot product is a type of vector multiplication in which the resultant vector is a scalar. It is also referred to as a scalar product for the same reason. Scalar product of two vectors is the sum of the products of the corresponding components.

If you have two vectors $A = (a_x, a_y, a_z)$ and $B = (b_x, b_y, b_z)$, then is given as:

$$\vec{A} \cdot \vec{B} = a_x \times b_x + a_y \times b_y + a_z \times b_z \quad \text{--- (1)}$$

The dot product of two vectors is also equal to the cosine of the angle between the vectors multiplied by the magnitudes of both vectors. Notice the dot product is represented as a dot between the vector.

$$\vec{A} \cdot \vec{B} = \|A\| \times \|B\| \times \cos(\theta)$$

θ is always between 0 and π . By putting an equation of 1 and 2 together, we can figure out the angle between 2 vectors:

$$\|A\| * \|B\| * \cos\theta = a_x * b_x + a_y * b_y + a_z * b_z$$

$$\cos\theta = (a_x * b_x + a_y * b_y + a_z * b_z) / (\|A\| * \|B\|)$$

Consequently, $\theta = \cos^{-1}((a_x * b_x + a_y * b_y + a_z * b_z) / (\|A\| * \|B\|))$

This form has some unique geometric properties:

If $\vec{A} \cdot \vec{B} = 0$ then \vec{A} is perpendicular to \vec{B} as $\cos 90^\circ = 0$.

$\vec{A} \cdot \vec{B} = \|A\| * \|B\|$ then the 2 vectors are parallel to each other as $\cos 0^\circ = 1$.

If $\vec{A} \cdot \vec{B} > 0$ then the angle between the vectors is less than 90 degrees.

If $\vec{A} \cdot \vec{B} < 0$ then the angle between the vectors is greater than 90 degrees.

Let us see an example of a dot product:

If $\vec{A} = (3, -5, 7)$ and $\vec{B} = (2, 4, 1)$

Then $\|A\| = 9.110$ and

$$\|B\| = \sqrt{2^2 + 4^2 + 1^2} = \sqrt{21}$$

Next, we calculate $\vec{A} \cdot \vec{B} = a_x * b_x + a_y * b_y + a_z * b_z$

$$\vec{A} \cdot \vec{B} = 3 * 2 + (-5) * 4 + 7 * 1$$

$$= 6 - 20 + 7$$

$$= -7$$

$$\cos \theta = -7 / (9.110 * 4.582) = -7 / 41.74$$

$$\theta = \cos^{-1}(-7/41.74) = \cos^{-1}(0.26770) = 99.65^\circ \text{ (approx)}$$

Cross product

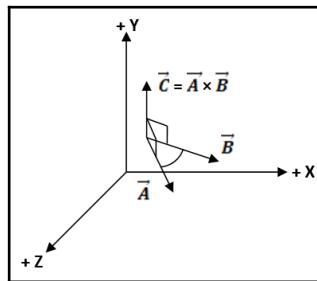
Cross product is the other vector multiplication form in which the resultant product of the multiplication is another vector. Taking the cross product between \vec{A} and \vec{B} vectors will result in a third vector that it is perpendicular to both vectors \vec{A} and \vec{B} .

If you have two vectors, $\vec{A} = (a_x, a_y, a_z)$ and $\vec{B} = (b_x, b_y, b_z)$, then $\vec{A} \times \vec{B}$. is given as follows:

$$\vec{C} = \vec{A} \times \vec{B} = (a_y b_z - a_z b_y, a_z b_x - a_x b_z, a_x b_y - a_y b_x)$$

The following is a matrix and graphical implementation of the cross product between vectors:

$$\vec{C} = \begin{bmatrix} ay bz - az by \\ az bx - ax bz \\ ax by - ay bx \end{bmatrix}$$



The direction of the resultant normal vector will obey the right-hand rule, where curling the fingers in the right hand from \vec{A} to \vec{B} will cause the thumb to point in the direction of the resultant normal vector \vec{C} .

Also note that the order in which you multiply the vectors is important as if you multiply the other way around then the resultant vector will be a point in the opposite direction.

The cross product will become very useful when we want to find the normal of the face of a polygon. For example, finding the normal of the face of a triangle.

Let us find the cross product of vectors $\vec{A} = (3, -5, 7)$ and $\vec{B} = (2, 4, 1)$:

$$\begin{aligned} C = A \times B &= (ay bz - az by, az bx - ax bz, ax by - ay bx) \\ &= (-5 * 1 - 7 * 4, 7 * 2 - 3 * 1, 3 * 4 - (-5) * 2) \\ &= (-5 - 28, 14 - 3, 12 + 10) \\ &= (-33, 11, 22) \end{aligned}$$

Matrices

In computer graphics, matrices are used to calculate object transforms like translation which is movement, scaling in X, Y and Z axis, and rotation around the X, Y and Z axis. We will also be changing the position of objects from one coordinate system to the other, which are called space transforms. We will see how matrices work and how they help in simplifying the mathematics.

Matrices are represented as having rows and columns. A matrix with m number of rows and n number of columns is said to be a matrix of size $m \times n$. Each element of a matrix is represented as indices ij where i specifies the row number and j represents the column number.

So, a matrix M of size 3×2 is represented as follows:

$$M = \begin{bmatrix} m_{11} & m_{12} \\ m_{21} & m_{22} \\ m_{31} & m_{32} \end{bmatrix}$$

Here, matrix M has three rows and two columns and each element is represented as m_{11} , m_{12} and so on until m_{32} , which is the size of the matrix.

In 3D graphics programming, we will be mostly dealing with a 4×4 matrix. So, let us look at another matrix of size 4×4 .

$$M = \begin{bmatrix} m_{11} & m_{12} & m_{13} & m_{14} \\ m_{21} & m_{22} & m_{23} & m_{24} \\ m_{31} & m_{32} & m_{33} & m_{34} \\ m_{41} & m_{42} & m_{43} & m_{44} \end{bmatrix}$$

Matrix A with numbers in it is:

$$A = \begin{bmatrix} 3 & 1 & 0.5 & 0 \\ \sqrt{2} & -2 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 2 & 2.5 & 5 & 1 \end{bmatrix}$$

Here, the element $A_{11} = 3$, $A_{32} = 1$ and $A_{44} = 1$ and the dimension of the matrix is 4×4 .

We can also have a single-dimension matrix like vector B as follows which is called the row vector or a column vector as shown as follows ad vector C:

$$B = [b_1 \ b_2 \ b_3 \ b_4] \quad C = \begin{bmatrix} c_1 \\ c_2 \\ c_3 \\ c_4 \end{bmatrix}$$

- Two matrices are equal if the number of rows and columns are the same and if the corresponding elements are of the same value.
- Two matrices can be added if they have the same number of rows and columns. We add each element of the corresponding location on both matrices to get a third matrix of the same dimension as the added matrices.

Matrix Addition and Subtraction

Consider the following two A and B matrices. Both of these are of the size as 3x3, shown as follows:

$$A = \begin{bmatrix} A & B & C \\ D & E & F \\ G & H & I \end{bmatrix} \quad B = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix}$$

Then $C = A + B$ is given as:

$$C = \begin{bmatrix} A + a & B + b & C + c \\ D + d & E + e & F + f \\ G + g & H + h & I + i \end{bmatrix}$$

Matrix subtraction works in the same way when each element of the matrix is subtracted with the corresponding element of the other matrix.

Matrix multiplication

We now see how a scalar value can be multiplied to a matrix by multiplying each element of the matrix by the same scalar value. This will give us a new matrix of the same dimension as the original matrix.

Again, consider a Matrix A multiplied by some scalars, then $s \times A$ is given as follows.

$$C = s \times \begin{bmatrix} A & B & C \\ D & E & F \\ G & H & I \end{bmatrix} = \begin{bmatrix} s \times A & s \times B & s \times C \\ s \times D & s \times E & s \times F \\ s \times G & s \times H & s \times I \end{bmatrix}$$

Two A and B matrices can be multiplied, provided that the number of columns of A is equal to the number of rows of B. So, if matrix A has the dimension $a \times b$ and B has the dimension $X \times Y$ then for A to be multiplied with B, b should be equal to X.

The resultant size of the matrix will be $a \times Y$. Two matrices are multiplied as follows:

$$A = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \\ a_{31} & a_{32} \end{bmatrix} \quad B = \begin{bmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \end{bmatrix}$$

Here, the size of A is 3×2 and the size of B is 2×3 , therefore the resultant matrix C will be of size 3×3 .

$$C = A \times B = \begin{bmatrix} a_{11} \times b_{11} + a_{12} \times b_{21} & a_{11} \times b_{12} + a_{12} \times b_{22} & a_{11} \times b_{13} + a_{12} \times b_{23} \\ a_{21} \times b_{11} + a_{22} \times b_{21} & a_{21} \times b_{12} + a_{22} \times b_{22} & a_{21} \times b_{13} + a_{22} \times b_{23} \\ a_{31} \times b_{11} + a_{32} \times b_{21} & a_{31} \times b_{12} + a_{32} \times b_{22} & a_{31} \times b_{13} + a_{32} \times b_{23} \end{bmatrix}$$

However, keep in mind that matrix multiplication is not commutative, meaning that $A \times B \neq B \times A$. In fact, in some cases it is not even possible to multiply the other way around just like the case above. Here, we can't even multiply $B \times A$ as the number of columns of B is not equal to the number of rows of A. In other words, the internal dimensions of the matrices should match so the dimensions should be in the form of $[a \times t]$ and $[t \times b]$.

You can also multiply a vector matrix with a regular matrix as follows:

$$A = \begin{bmatrix} A & B & C \\ D & E & F \\ G & H & I \end{bmatrix} \quad V = \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

The result will be a one dimensional vector of size 3×1 as shown as follows:

$$C = A \times V = \begin{bmatrix} A \times x + B \times y + C \times z \\ D \times x + E \times y + F \times z \\ G \times x + H \times y + I \times z \end{bmatrix}$$

Note that when multiplying the matrix with the vector matrix, the vector is on the right of the matrix. This is done so that the matrix of size 3×3 is able to multiply the vector matrix of size 3×1 .

When we have a matrix with just one column, this is called column major matrix. So, Matrix C is a column major matrix just like matrix V, which is also a column major matrix.

If the same vector V was expressed with just a row it would be called a row major matrix and would look like as follows:

$$V = [x \ y \ z]$$

So how would we multiply a matrix A of size 3×3 with a row major matrix V of size 1×3 , since the internal dimensions don't match?

The simple solution here is that instead of multiplying matrix $A \times V$, we multiply $V \times A$. This way, the internal dimensions of the vector matrix and the regular matrix will match 1×3 and 3×3 and the resultant matrix will also be a row major matrix.

However, throughout the book we will be using the column major matrix.

But if in the we are going to be using 4×4 matrices, how would we multiply a 4×4 matrix using the coordinates of x, y, and z?

So, when multiplying a 4×4 matrix with point X, Y, and Z, we add one more row to the column major matrix and set the value of it to 1. The new point will be (X, Y, Z, 1), which is called a homogeneous point. This makes it easy to multiply a 4×4 matrix with a 4×1 vector.

$$A = \begin{bmatrix} A & B & C & D \\ E & F & G & H \\ I & J & K & L \\ M & N & O & P \end{bmatrix} \quad P = \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

$$C = A \times P = \begin{bmatrix} Ax + By + Cz + D * 1 \\ Ex + Fy + Gz + H * 1 \\ Ix + Jy + Kz + L * 1 \\ Mx + Ny + Oz + P * 1 \end{bmatrix}$$

Matrix multiplication can be extrapolated to multiplication of a 4×4 matrix with another 4×4 matrix. Let's look at how to do this:

$$\text{Let } A = \begin{bmatrix} A & B & C & D \\ E & F & G & H \\ I & J & K & L \\ M & N & O & P \end{bmatrix} \quad B = \begin{bmatrix} a & b & c & d \\ e & f & g & h \\ i & j & k & l \\ m & n & o & p \end{bmatrix}$$

$$A \times B =$$

$$\begin{bmatrix} Aa + Be + Ci + Dm & Ab + Bf + Cj + Dn & Ac + Bg + Ck + Do & Ad + Bh + Cl + Dp \\ Ea + Fe + Gi + Hm & Eb + Ff + Gj + Hn & Ec + Fg + Gk + Ho & Ed + Fh + Gl + Hp \\ Ia + Je + Ki + Lm & Ib + Jf + Kj + Ln & Ic + Jg + Kk + Lo & Id + Jh + Kl + Lp \\ Ma + Ne + Oi + Pm & Mb + Nf + Oj + Pn & Mc + Ng + Ok + Po & Md + Nh + Ol + Pp \end{bmatrix}$$

Identity matrix

An Identity is a special kind of matrix in which the number of rows is equal to the number of columns called a square matrix. In an identity matrix the elements in the diagonal of the matrix are all 1 and the rest of the elements are 0.

Here is an example of a 4×4 identity matrix:

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

Identity matrix acts in a similar way to how we get a result when we multiply any number with 1 and get the same number. Similarly, when we multiply any matrix with an identity matrix, we get the same matrix.

So, $A \times I = A$ where A is a 4×4 matrix and I is an identity matrix of the same size. Let's see an example of this:

$$\text{Let } A = \begin{bmatrix} A & B & C & D \\ E & F & G & H \\ I & J & K & L \\ M & N & O & P \end{bmatrix} \quad I = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$A \times I =$$

$$\begin{bmatrix} A \times 1 + B \times 0 + C \times 0 + D \times 0 & A \times 0 + B \times 1 + C \times 0 + D \times 0 & A \times 0 + B \times 0 + C \times 1 + D \times 0 & A \times 0 + B \times 0 + C \times 0 + D \times 1 \\ E \times 1 + F \times 0 + G \times 0 + H \times 0 & E \times 0 + F \times 1 + G \times 0 + H \times 0 & E \times 0 + F \times 0 + G \times 1 + H \times 0 & E \times 0 + F \times 0 + G \times 0 + H \times 1 \\ I \times 1 + J \times 0 + K \times 0 + L \times 0 & I \times 0 + J \times 1 + K \times 0 + L \times 0 & I \times 0 + J \times 0 + K \times 1 + L \times 0 & I \times 0 + J \times 0 + K \times 0 + L \times 1 \\ M \times 1 + N \times 0 + O \times 0 + P \times 0 & M \times 0 + N \times 1 + O \times 0 + P \times 0 & M \times 0 + N \times 0 + O \times 1 + P \times 0 & M \times 0 + N \times 0 + O \times 0 + P \times 1 \end{bmatrix}$$

$$A \times I = \begin{bmatrix} A & B & C & D \\ E & F & G & H \\ I & J & K & L \\ M & N & O & P \end{bmatrix}$$

Matrix transpose

A transpose of a matrix occurs when the rows and columns are interchanged with each other. So, the transpose of a $m \times n$ matrix is $n \times m$. The transpose of any matrix M is written as M^T . The transpose of a matrix is as follows:

$$A = \begin{bmatrix} A & B & C & D \\ E & F & G & H \\ I & J & K & L \\ M & N & O & P \end{bmatrix} \quad A^T = \begin{bmatrix} A & E & I & M \\ B & F & J & N \\ C & G & K & O \\ D & H & L & P \end{bmatrix}$$

Observe how the elements in the diagonal of the matrix remain in the same place but all the elements around the diagonal have been swapped.

In Matrices, this diagonal of the matrix which runs from the top left to the bottom right is called the Main diagonal.

Obviously if you transpose a transposed matrix, you get the original matrix, so $(A^T)^T = A$

Matrix inverse

An inverse of any matrix is such that any matrix, when multiplied by its inverse, will result in an identity matrix. For a matrix M, the inverse of the matrix is denoted as M^{-1} .

Inverse is very useful in graphics programming when we want to undo the multiplication of a matrix.

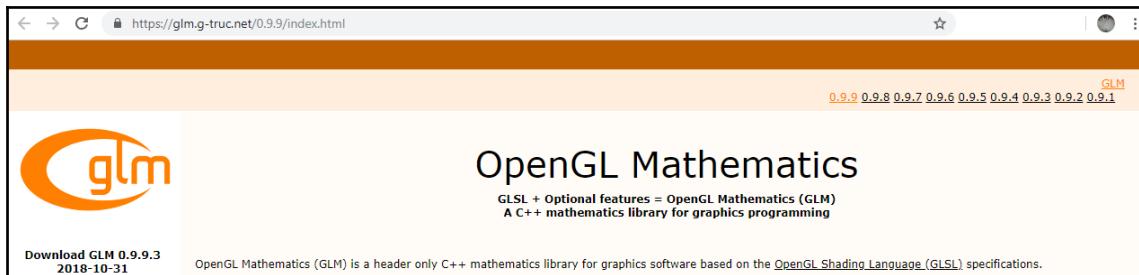
For example, Matrix M is equal to $A \times B \times C \times D$ where A, B, C and D are also matrices. And now we want to know what $A \times B \times C$ is instead of multiplying the three matrices, which is a two-step operation as you will first multiply A with B and then multiply the resulting matrix with C. You can multiply M with D^{-1} and that will yield the same result:

$$M = A \times B \times C \times D$$

$$A \times B \times C = M \times D^{-1}$$

GLM OpenGL mathematics

To carry out the mathematics operation in OpenGL and Vulkan projects, we will be using a header-only C++ mathematics library called GLM. This was initially developed to be used with OpenGL, it is now used with Vulkan as well:



The latest version of GLM can be downloaded from <https://glm.g-truc.net/0.9.9/index.html>.

Apart from being able to create points and perform vector addition and subtraction, GLM can also define matrices, carry out matrix transforms, generate random numbers, and generate noise. Shown following are a few examples on how these functions are carried out.

- To define 2D and 3D points, we will need to include `#include <glm/glm.hpp>`, which uses the `glm` namespace. To define a 2D point in space, we will do the following:

```
glm::vec2 p1 = glm::vec2(2.0f, 10.0f);
```

Where the 2 arguments passed in are the `x` and `y` position.

- To define a 3D point, we will specify the following:

```
glm::vec3 p2 = glm::vec3(10.0f, 5.0f, 2.0f);
```

- A 4×4 matrix is created using `glm`, in the as following code. A 4×4 matrix is of the `mat4` type and is created like this:

```
glm::mat4 matrix = glm::mat4(1.0f);
```

Here the `1.0f` parameter passed in shows that the matrix is initialized as a identity matrix.

- For translation and rotation, of this matrix you will need to include GLM extensions, as shown in the following code:

```
#include <glm/ext.hpp>
glm::mat4 translation = glm::translate(glm::mat4(1.0f),
glm::vec3(3.0f, 4.0f, 8.0f));
```

- To now translate the object to $(3.0, 4.0, 8.0)$ from its current position, as follows:

```
glm::mat4 scale = glm::scale(glm::mat4(1.0f),
glm::vec3(2.0f, 2.0f, 2.0f));
```

- We can also scale the value to double its size in the x , y , and z directions:

```
glm::mat4 rxMatrix = glm::rotate(glm::mat4(), glm::radians(45.0f),
glm::vec3(1.0, 0.0, 0.0));
glm::mat4 ryMatrix = glm::rotate(glm::mat4(), glm::radians(25.0f),
glm::vec3(0.0, 1.0, 0.0));
glm::mat4 rzMatrix = glm::rotate(glm::mat4(), glm::radians(10.0f),
glm::vec3(0.0, 0.0, 1.0));
```

This above code block rotates the object by 45 degrees along the x axis, 25.0f degrees along the y axis, and 10.0f degrees along the z axis.

Note that we use `glm::radians()`. This `glm` function converts degrees into radians. More GLM functions will be introduced as we progress throughout this chapter.

OpenGL data types

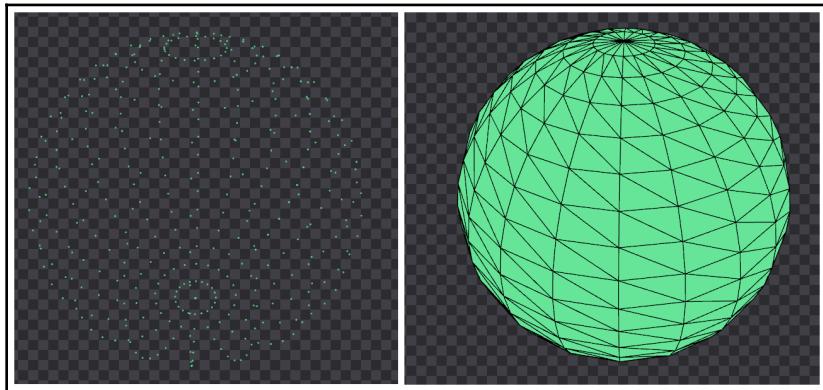
OpenGL also has its own data types, which are portable across platforms.

OpenGL data types are prefixed with GL, followed by the data type. Consequently, a GL equivalent to an `int` variable will be `GLint`, and so on. The following table shows a list of the GL data types (the list can be viewed at https://www.khronos.org/opengl/wiki/OpenGL_Type):

C Type	Bitdepth	Description	Common Enum
<code>GLboolean</code>	1+	A boolean value, either <code>GL_TRUE</code> or <code>GL_FALSE</code>	
<code>GLbyte</code>	8	Signed, 2's complement binary integer	<code>GL_BYTE</code>
<code>GLubyte</code>	8	Unsigned binary integer	<code>GL_UNSIGNED_BYTE</code>
<code>GLshort</code>	16	Signed, 2's complement binary integer	<code>GL_SHORT</code>
<code>GLushort</code>	16	Unsigned binary integer	<code>GL_UNSIGNED_SHORT</code>
<code>GLint</code>	32	Signed, 2's complement binary integer	<code>GL_INT</code>
<code>GLuint</code>	32	Unsigned binary integer	<code>GL_UNSIGNED_INT</code>
<code>GLfixed</code>	32	Signed, 2's complement 16.16 integer	<code>GL_FIXED</code>
<code>GLint64</code>	64	Signed, 2's complement binary integer	
<code>GLuint64</code>	64	Unsigned binary integer	
<code>GLsizei</code>	32	A non-negative binary integer, for sizes.	
<code>GLenum</code>	32	An OpenGL enumerator value	
<code>GLintptr</code>	<code>ptrbits</code> ¹	Signed, 2's complement binary integer	
<code>GLsizeiptr</code>	<code>ptrbits</code> ¹	Non-negative binary integer size, for memory offsets and ranges	
<code>GLsync</code>	<code>ptrbits</code> ¹	Sync Object handle	
<code>GLbitfield</code>	32	A bitfield value	
<code>GLhalf</code>	16	An IEEE-754 floating-point value	<code>GL_HALF_FLOAT</code>
<code>GLfloat</code>	32	An IEEE-754 floating-point value	<code>GL_FLOAT</code>
<code>GLclampf</code>	32	An IEEE-754 floating-point value, clamped to the range [0,1]	
<code>GLdouble</code>	64	An IEEE-754 floating-point value	<code>GL_DOUBLE</code>
<code>GLclampd</code>	64	An IEEE-754 floating-point value, clamped to the range [0,1]	

Space transformations

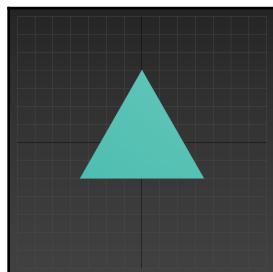
The major job of 3D graphics is to simulate a 3D world and project this world into a 2D location, which is the viewport window. 3D or 2D objects that we want to render are nothing but a collection of vertices. These vertices are then made into a collection of triangles to form the shape of the object sphere:



The screenshot on the left shows the vertices that were passed in and on the right, the vertices were used to create triangles. Each triangle forms a small piece of the surface for the final shape of the object.

Local/object space

When setting the vertices for any object, we start at the origin of a coordinate system. These vertices or points are placed and then connected, to create the shape of the object, like a triangle as shown follows. This coordinate system around which a model is created is called the object space, model space, or local space:



World space

Now that we have specified the shape of the model, we want to place it in a scene along with a couple of other shapes, such as a sphere and a cube. The cube and sphere shapes are also created using their own model space. When placing these objects in a 3D scene, we will be doing so with respect to another coordinate system around which the 3D objects will be placed. This new coordinate system is called the world coordinate system, or world space.

For moving the object from the object space to the world space is done through matrix transforms. The local position of the object is multiplied by the world space matrix.

Consequently, each vertex is multiplied by the world space matrix to transform its scale, rotation, and position from the local space to the world space.

The world space matrix is a product of scale, rotation, and translation matrices, as shown in the following formula:

$$\text{World Matrix} = W = T \times R \times S$$

S, R, and T are the scale, rotation, and translation of the local space relative to the world space.

- The scale matrix for a 3D space is a 4×4 matrix with its diagonal representing the scale in the x , y , and z directions, as follows:

$$S = \begin{pmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & S_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

- The rotation matrix can take three forms depending on which axis you are rotating the object. The R_x , R_y , and R_z are matrices for rotation along each axis , as shown in the following matrix:

$$R(\theta)x = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(\theta) & \sin(\theta) & 0 \\ 0 & -\sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$\text{R}(\theta)y = \begin{pmatrix} \cos(\theta) & 0 & -\sin(\theta) & 0 \\ 0 & 1 & 0 & 0 \\ \sin(\theta) & 0 & \cos(\theta) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$\text{R}(\theta)z = \begin{pmatrix} \cos(\theta) & -\sin(\theta) & 0 & 0 \\ \sin(\theta) & \cos(\theta) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

- The translation matrix is an identity matrix with the last column representing the translation in the x , y , and z directions:

$$T = \begin{pmatrix} 1 & 0 & 0 & Tx \\ 0 & 1 & 0 & Ty \\ 0 & 0 & 1 & Tz \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

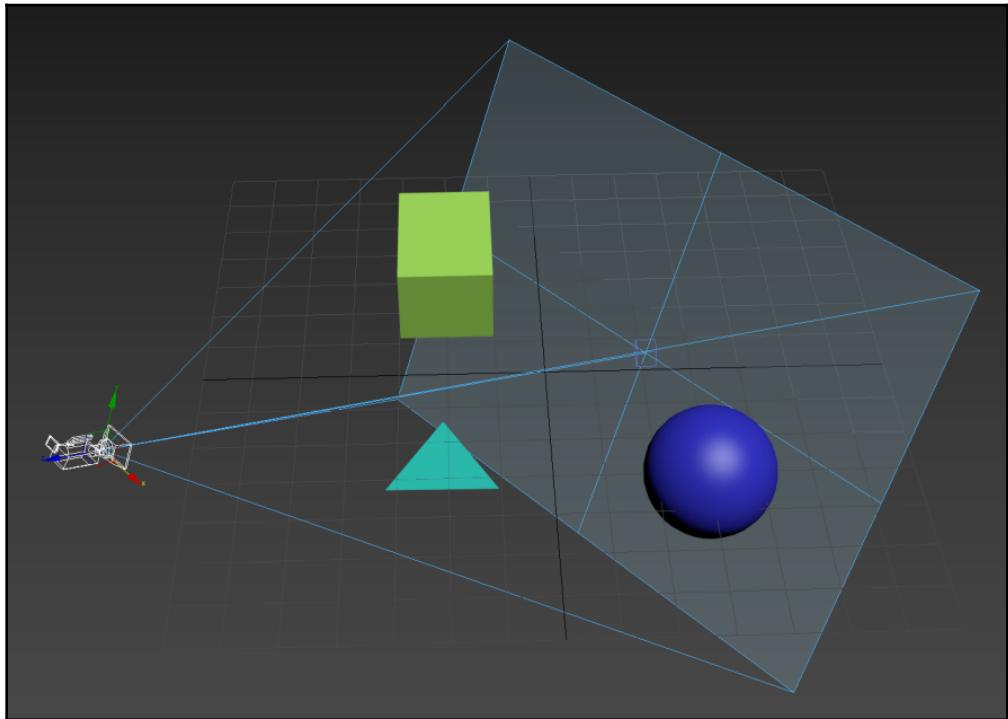
Now we can get the World position by multiplying the local position of the objects with the world matrix, as follows:

$$\text{Position}_{\text{World}} = \text{Matrix}_{\text{World}} \times \text{Position}_{\text{local}}$$

View space

For us to view the whole scene, we will need a camera. This camera will also decide which objects will be visible to us and which objects won't be rendered to the screen.

Consequently, we place a virtual camera in the scene at a certain world location, as shown in the following diagram:



The objects in the scene are then transformed from the world space to a new coordinate system present at the location of the camera. This new coordinate system at the location of the camera is called the view space, camera space, or eye space. The x axis is red, the y axis is green, and the positive z axis is blue.

To transform the points from the world space to the camera space, we have to translate them using the negative of the virtual camera location and rotate them using the negative of the camera orientation.

However, there is an easier way to create the view matrix using GLM. We give three variables to define the camera position, camera target position, and camera up vector as shown follows:

```
glm::vec3cameraPos = glm::vec3(0.0f, 0.0f, 200.0f);  
glm::vec3cameraFront = glm::vec3(0.0f, 0.0f, 0.0f);  
glm::vec3cameraUp = glm::vec3(0.0f, 1.0f, 0.0f);
```

We can use these variables to create a view matrix by calling the `lookAt` function and passing the camera position, the look at position, and up vector, as follows:

```
glm::mat4 viewMatrix = glm::lookAt(cameraPos, cameraPos +
cameraFront, cameraUp);
```

Once we have the view matrix, the local positions can be multiplied by the world and the view matrix to get the position in the view space, as follows:

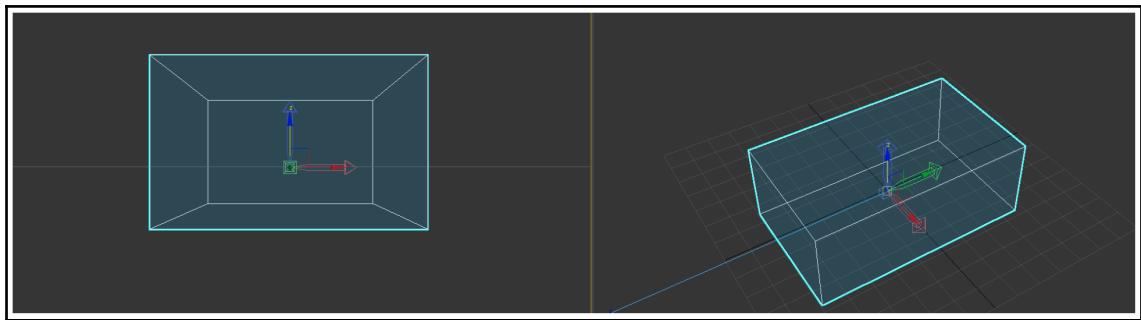
$$\text{Position}_{\text{view}} = \text{View}_{\text{matrix}} \times \text{World}_{\text{matrix}} \times \text{Position}_{\text{local}}$$

Projection space

The next task is to project the 3D objects viewed by the camera onto the 2D plane. The projection needs to be done in such a way that the furthest object appears smaller and the objects that are closer appear bigger. Basically, when viewing an object, the points needs to converge at a vanishing point.

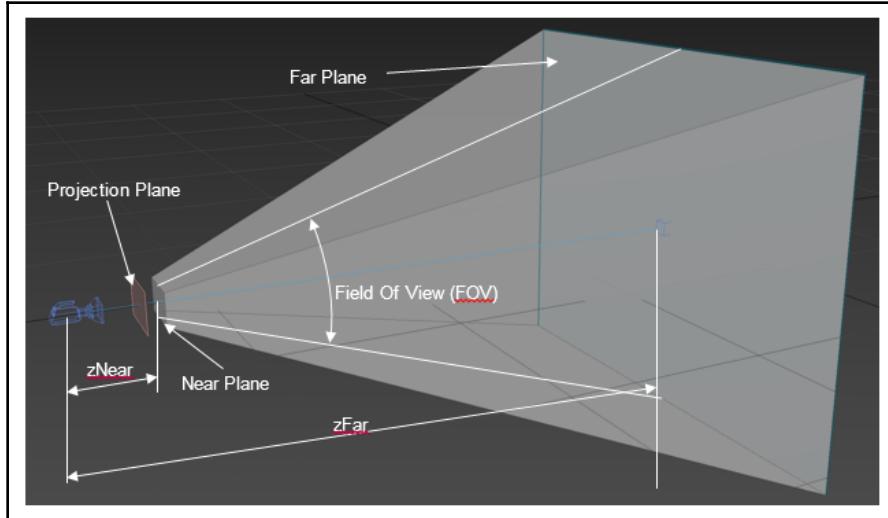
In the following screenshot, the image on the right shows that a cube is be rendered. Note how the lines on the longer sides are actually parallel.

However, when the same box is viewed from the camera, the same side lines converge, and when these lines are extended, they will converge at a point behind the box:



We will now introduce one more matrix, called the projection matrix, which allows objects to be rendered with perspective projection. The vertices of the objects will be transformed using what is called a projection matrix to perform the perspective projection transformation.

In the projection matrix, we define a projection volume called the frustum. All the objects inside the frustum will be projected onto the 2D display. The objects outside the projection plane will not be rendered:



The projection matrix is created as follows:

$$\text{Matrix}_{\text{projection}} = \begin{pmatrix} A & 0 & 0 & 0 \\ 0 & q & 0 & 0 \\ 0 & 0 & B & C \\ 0 & 0 & -1 & 0 \end{pmatrix}$$

$$q = 1/\tan(FieldOfView/2)$$

$$A = q/\text{Aspect Ratio}$$

$$B = (zNear + zFar)/(zNear - zFar)$$

$$C = 2 * (zNear * zFar)/(zNear - zFar)$$



The aspect ratio is the width of the projection plane divided by the height of the projection plane. *zNear* is the distance from the camera to the near plane. *zFar* is the distance from the camera to the far plane. The **field of view (FOV)** is the angle between the top and bottom planes of the view frustum.

In GLM, there is a function we can use to create the perspective projection matrix, as follows:

```
GLfloat FOV = 45.0f;
GLfloat width = 1280.0f;
GLfloat height = 720.0f;
GLfloat nearPlane = 0.1f;
GLfloat farPlane = 1000.0f;

glm::mat4 projectionMatrix = glm::perspective(FOV, width /height,
nearPlane, farPlane);
```

Note that the *nearPlane* always needs to be greater than *0.0f* to create the start of the frustum in form of the camera.

The *glm::perspective* function takes four parameters.

- The first is the FOV,
- The second is the aspect ratio,
- The third is the distance to the near plane,
- The fourth is the distance to the far plane.

So, after obtaining the projection matrix, we can finally perform a perspective projection transform on our view-transformed points to project the vertices onto the screen:

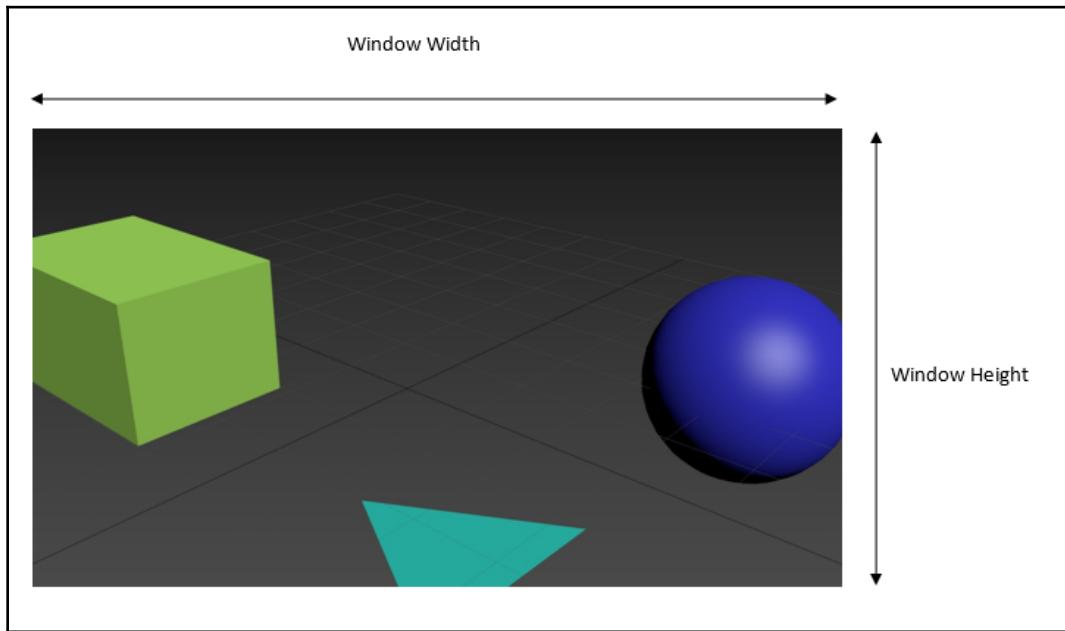
$$\text{Position}_{\text{final}} = \text{Projection}_{\text{matrix}} \times \text{View}_{\text{matrix}} \times \text{World}_{\text{matrix}} \times \text{Position}_{\text{local}}$$

Now that we understand this in theory, let's see how this is actually implemented.

Screen space

After multiplying the local position by the model, view, and projection matrix, OpenGL will transform the scene into screen space.

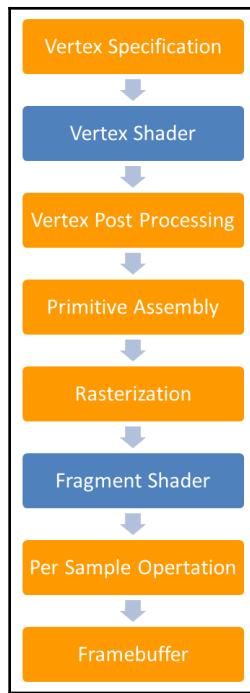
If the screen size of your application is of the resolution 1280 x 720, then it will project the scene onto the screen like the following. This is what is viewed by the camera in the scene in the view-space heading:



For this example, the width of the window will be 1,280 pixels and the height of the window will be 720 pixels.

Render pipeline

As I said earlier, we have to convert 3D objects made of vertices and textures and represent them in a 2D screen as pixels on the screen. This is done through what is called a render pipeline:



A pipeline is simply a series of steps that are carried out one after the other to achieve a certain objective. The stages highlighted in orange or lightly shaded boxes are fixed, meaning that you cannot modify how the data in the stage is processed. The stages in blue or darker boxes are programmable stages, meaning that you can write special programs to modify the output. Shown here is the basic pipeline which includes the minimum stages to render objects. There are other stages like Geometry, Tessellation and Compute which are all optional stages which are not discussed in the book as the book covers the introduction to graphics programming only.

Also the graphics pipeline itself is common for both OpenGL and Vulkan. Obviously the implementation is different as we will see in the following chapters

The render pipeline is used to render 2D or 3D objects onto the gamer's TV or monitor. Let's look at each of the stages in the graphics pipeline in detail.

Vertex specification

When we want to render an object to the screen, we set information regarding that object first. The very basic information that we will need to set are the points or vertices that will make up the geometry. We will be creating the object by creating an array of vertices. These will be used to create a number of triangles that will make up the geometry we want to render. These vertices need to be sent to the graphics pipeline.

To send information to the pipeline in OpenGL for example, we use **vertex array objects (VAO)** and **vertex buffer objects (VBO)**. VAOs are used to define what data each vertex has and VBOs have the actual vertex data.

Vertex data can have a series of attributes. A vertex could have property attributes, such as position, color, normal, and so on. Obviously, one of the main attributes that any vertex needs to have is the position information. Apart from the position, we will also see other types of information that can be passed in for example, the color of each vertex. We will see a couple more attributes in future chapters when we cover rendering primitives using OpenGL.

For example, say that we have an array of three points. Let's see how we would create the VAO and the VBO:

```
float vertices[] = {  
    -0.5f, -0.5f, 0.0f,  
    0.5f, -0.5f, 0.0f,  
    0.0f, 0.5f, 0.0f  
};
```

1. We generate a vertex array object of the `Clint` type. OpenGL just return a handle for future reference to the actual object as follows:

```
unsigned int VAO;  
glGenVertexArrays(1, &VAO);
```

2. Then, we generate a vertex buffer object, as follows:

```
unsigned int VBO;  
glGenBuffers(1, &VBO);
```

3. Next, we specify the type of buffer object. Here, it is a `GL_ARRAY_BUFFER` type, meaning that this is an array buffer:

```
glBindBuffer(GL_ARRAY_BUFFER, VBO);
```

4. We then bind the data to the buffer, as follows:

```
glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices,  
GL_STATIC_DRAW);
```

The first parameter is a type of buffer data, which is the `GL_ARRAY_BUFFER` type. The second parameter is the size of the data type that is passed in. The `sizeof()` is a C++ keyword that gets the size of the data in bytes.

The next parameter is the data itself and the last parameter is used to specify whether this data will change. The `GL_STATIC_DRAW` means that the data will not be modified once the values are stored.

5. We then specify the vertex attributes, shown as follows:

- The first parameter is the index of the attribute. In this case we just have one attribute that has a position located at the 0th index.
- The second is the size of the attribute. Each vertex is represented by three floats for x , y , and z , so here the value specified is 3.
- The third parameter is the type of data that is passed in, which is of the `GL_FLOAT` type.
- The fourth parameter is a Boolean asking whether the value should be normalized or used as is. We specify `GL_FALSE`, as we don't want the data to be normalized.
- The fifth parameter is called the stride; this specifies the offset between the attributes. Here, the value for the next position is the size of three floats, that is, x , y , and z .
- The final parameter specifies the starting offset of the first component, which is 0 here. We are typecasting the data to a more generic data type (`void*`) called a void pointer.

```
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 3 *  
sizeof(float), (void*)0);
```

6. Finally, we enable the attribute at the 0th index, as follows:

```
 glEnableVertexAttribArray(0);
```

This is a basic example. We will see how this changes when we add additional attributes, such as color, and so on.

Vertex shader

The vertex shader stage performs operations on a per-vertex basis. Depending upon the number of vertices you have passed into the vertex shader, the vertex shader will be called that many times. If you are passing three vertices to form a triangle, then the vertex shader will be called thrice.

Depending upon the attributes that are passed into the shader, the vertex shader modifies the value that is passed in and outputs the final value of that attribute. For example, when you pass in a position attribute, then you can manipulate its value and send out the final position of that vertex at the end of the vertex shader.

The following is an example code for a basic vertex shader for the single attribute passed in the earlier stage:

```
#version 430 core
layout (location = 0) in vec3 position;

void main()
{
    gl_Position = vec4(position.x, position.y, position.z, 1.0);
}
```



The shaders are programs written in a language similar to C. The shader will always begin with the version of the **GL Shader Language (GLSL)**. There are other shader languages, including the **High-Level Shading Language (HLSL)** used by Direct 3D and CG. CG is also used in Unity

Now, we will declare all the input attributes that we want to use. For this, we use the `layout` keyword and specify the location or the index of the attribute we want to use in brackets. Since we passed in one attribute for the vertex position and specified the index for it as 0 while specifying the attribute pointer, we set the location as 0. We then use the `in` keyword to say that we are receiving the information, and we will store each position's value in a variable type called `vec3` with a name `position`.

The variable type `vec3` are vectors that are GLSL-intrinsic data types that can store data passed into the shader. Here, since we are passing in position information that has an *x*, *y*, and *z* component, it's convenient to use a `vec3`. We also have a `vec4`, which has an additional *w* component that can be used to store color information, for example.

Each shader needs to have a main function in which the major function relevant to the shader will be performed. Here, we are not doing anything too complicated: we just get the `vec3`, convert it into a `vec4`, and then set the value to `gl_Position`. We have to convert the `vec3` to `vec4` as the `gl_Position` is a `vec4` and it is also GLSL's intrinsic variable, which is used to store and output value from the vertex shader.

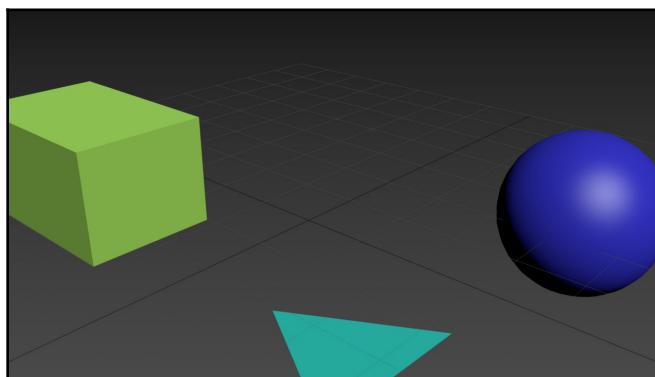
Since this is a basic example of a vertex shader we are not multiplying each point with the `ModelViewProjection` matrix yet to transform the point to the projection space. We will expand on this example in the future chapters.

Vertex post-processing

At this stage, clipping occurs. Objects that are not in the visible cone of the camera are not rendered to the screen. These primitives, which are not displayed, are said to be clipped.

Suppose only part of a sphere is visible. The primitive is broken into smaller primitives and only the primitives that are visible will be displayed. The vertex positions of the primitives are transformed from the clip space to the window space.

For example, here, only part of the sphere, triangle, and cuboid are visible, as the rest is not visible to the camera and is clipped:



Primitive assembly

The primitive assembly stage gets all the primitives that are not clipped from the previous stage and creates a sequence of primitives.

Face culling is also performed at this stage. Face culling is the process in which primitives that are in front of the view but are facing backwards will be culled, as they won't be visible. For example, when you are looking at a cube, you only see the front face of the cube and not the back of the cube, so there is no point in rendering the back of the cube when it is not visible. This is called back-face culling.

Rasterization

The GPU needs to convert the geometry described as vectors into pixels. We call this rasterization. The primitives that pass the previous stage of clipping and culling will be processed further in order to be rasterized. The rasterization process creates a series of fragments from the primitives. The process of converting a primitive into a rasterized image is done by the GPU. In the process of rasterization (vector to pixel), we always lose information, hence the name fragment of primitives. The fragment shader is used to calculate the final pixel's color value, which will be outputted to the screen.

Fragment shader

Fragments from the rasterization stage are then processed using the fragment shader. Just like the vertex shader stage, the fragment shader is also a program that can be written to perform modifications to each fragment.

The fragment shader will be called for each fragment from the previous stage.

Here is an example of a basic fragment shader:

```
#version 430 core
out vec4 Color;

void main()
{
    Color = vec4(0.0f, 0.0f, 1.0f, 1.0f);
}
```

Just like the vertex shader, you will first specify the GLSL version to use.

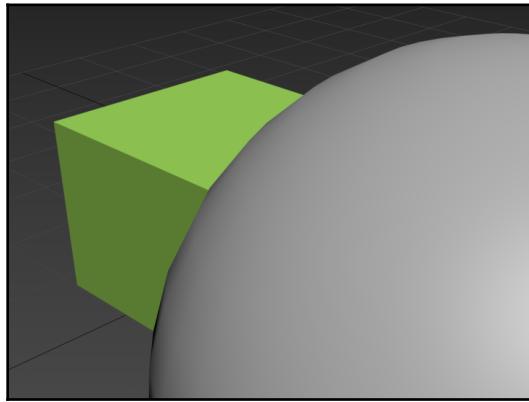
We use the `out` keyword to send the output value from the fragment shader. Here, we want to send out a variable of the `vec4` type called `Color`. The main function is where all the magic happens. For each fragment that gets processed, we set the value of `Color` to blue. So, when the primitive gets rendered to the viewport, it will be completely blue.

And this is how the sphere gets its blue color.

Per-sample operation

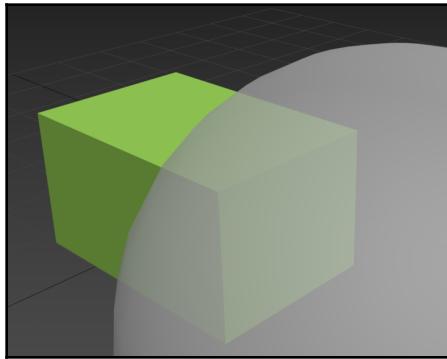
In the same way that the vertex post-processing stage clipped a primitive, the per-sample operation also removes fragments that are not to be shown. Whether a fragment needs to be displayed on the screen depends on certain tests that can be enabled by the user.

One of the more commonly used tests is the depth test. When enabled, this will check if a fragment is behind another fragment. If it is, then the current fragment will be discarded. For example, here, only part of the cuboid is visible, as it is behind the grey sphere:



There are other tests as well, such as a scissor and stencil test, which will only show a portion of the screen or object based on certain conditions that we specify.

Color blending is also done at this stage. Here, based on certain blending equations, colors can be blended. For example, here, the sphere is transparent, so we see the color of the cuboid blending with the color of the sphere:



Framebuffer

Finally, when a per-sample operation is completed for all the fragments in a frame, the final image is then rendered to the framebuffer, which is then presented on the screen.

A framebuffer is a collection of images that are drawn per frame. Each of these images is attached to the framebuffer. The framebuffer has attachments, like the color image which is shown on the screen. There are also other attachments, such as depth or the image/textures; it just stores the depth information of each pixel, which the end user never sees but that is sometimes used for graphical purposes by games.

In OpenGL, the framebuffer is created automatically at the start. There are also user-created framebuffers that can be used to render the scene first, apply post-processing to it, and then hand it back to the system to be displayed on the screen.

Summary

In this chapter, we covered some of the basics of mathematics that we will be using in this book. We covered the basics about coordinate systems, points, vectors and matrices. We then learned how to apply these concepts in the OpenGL mathematics and space transform. We then looked at GLM, which is a math library we will be using to make our mathematical calculations easier. We also covered space transforms and understood the flow of the graphics pipeline.

2

Section 2: SFML 2D Game Development

We will create a basic side-scrolling 2D action game in which we will cover basic game concepts, including creating a game loop, rendering a 2D game scene using SFML, 2D sprite creation, 2D sprite animations, UI text and buttons, physics, and collision detection.

The following chapters are in this section:

Chapter 3, Setting Up Your Game

Chapter 4, Creating Your Game

Chapter 5, Finalizing Your Game

3

Setting Up Your Game

Let's start with the basics of how a game is made and what basic graphical components the game requires. Since this book is going to be covering graphics with C++, we will mostly look at what is graphically required from the graphics engine in a game. We will also cover the sound system in order to make the game more interesting.

In order to show a basic graphics engine, we will use **Simple and Fast Media Library (SFML)**, as this includes most of the functionality that is needed to get a game up and running. The reason for choosing SFML is that it is very basic and easy to understand, unlike other engines and frameworks.

In this chapter we will create a window for our game and add animations to it. We will also learn how to create and control our player's movement.

The topics covered in this chapter:

- An overview of SFML
- Downloading SFML and configuring Visual Studio
- Creating a window
- Drawing shapes
- Adding sprites
- Keyboard input
- Handing player movement

An overview of SFML

Games and video games (unlike other entertainment media) actually involve loading various resources such as images, videos, sound, font types, and more. SFML provides functions for loading all of these features into games.

SFML is cross-platform compatible, meaning that it allows you to develop and run a game on different platforms. It also supports various languages other than C++. Additionally, it is open source, so you can take a look at the source code and add a feature to it (if it is not included).

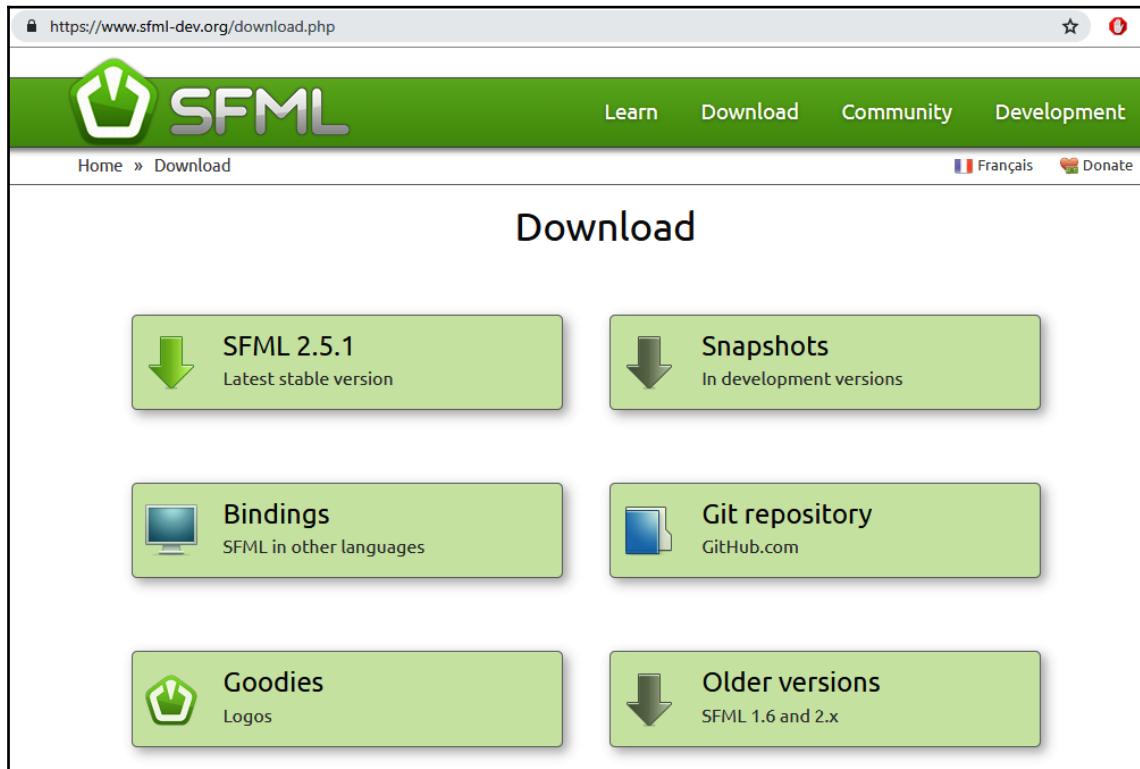
SFML is broken down into five modules, which can be defined as follows:

- **System:** This module directly interacts with a system such as Windows, which is essentially the **operating system (OS)** that it will use. Since SFML is cross-platform compatible and each OS is different in how it handles data, this module takes care of interacting with the OS.
- **Window:** When rendering anything to the screen, the first thing we need is a viewport or a window. Once we have access to this, we can start sending our rendered scene to it. The window module takes care of how a window is created, the input is handled, and more.
- **Graphics:** After we have access to a window, we can use the graphics module to begin rendering our scene to the window. In SFML, the graphics module is primarily rendered using OpenGL and deals with two-dimensional scene rendering only. Therefore, it can't be used to make three-dimensional games.
- **Audio:** The audio module is responsible for playing audio, audio streams, and recording audio.
- **Networking:** SFML also includes a networking library for sending and receiving data, which can be used for developing multiplayer games.

Downloading SFML and configuring Visual Studio

Now that we are familiar with the basics, let's begin:

1. Navigate to the SFML download page: <https://www.sfml-dev.org/download.php>:

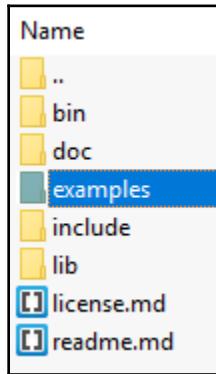


2. Select SFML 2.5.1, Alternatively, you can clone the repository and build the latest version using CMake.
3. Download either the 32bit or 64bit version (depending on your OS) for 2017 Visual Studio.



Although we are going to be developing the game for Windows, you can download SFML for Linux or macOS on the same web page.

In the downloaded ZIP file, you will see the following directories:

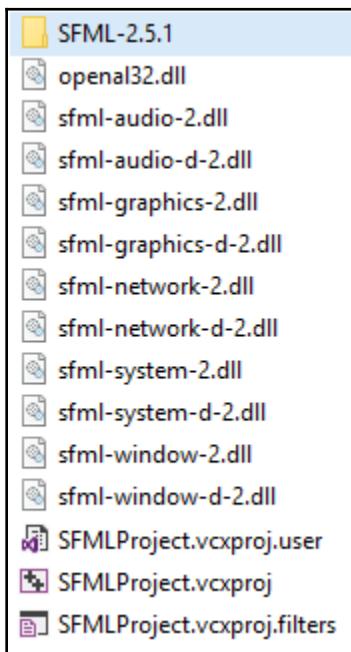


The directories are defined as follows:

- **bin:** This contains all the **dynamic link libraries (DLLs)** that are required for running all SFML modules. This has the .dll file for both the debug and the release version. The debug version has a -d suffix at the end of the file, while files that don't have this suffix are the release version .dll files.
- **doc:** This has the documentation for SFML provided in HTML format.
- **examples:** This includes examples for implementing the modules and features of SFML. It includes how to open a window, include OpenGL, carry out networking, and how to create a basic pong-like game.
- **include:** This has all the header files for the modules. The graphics module has classes for creating sprites, loading textures, creating shapes, and more.
- **lib:** This includes all the library files that we will need in order to run SFML.

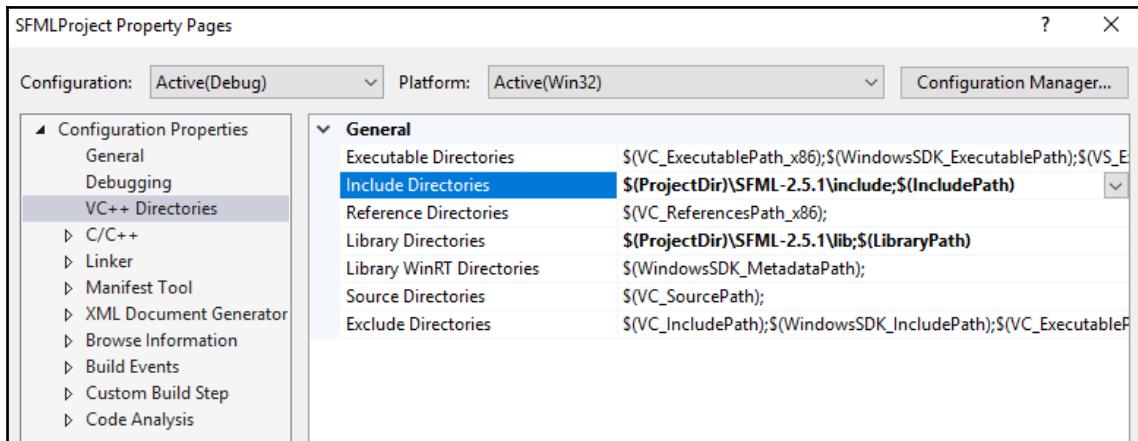
There is also a `readme.md` and a `license.md` file. The license file indicates that the SFML can be used for commercial purposes. Therefore, it can be altered and redistributed, provided that you don't claim that you created it.

4. To setup a Visual Studio project, create a new project called `SFMLProject`. In this Visual Studio project root directory, where `SFMLProject.vcxproj` is located, extract the `SFML-2.5.1` folder into it.
5. Then, in the root directory, move all the `.dll` files from the `bin` folder into the root directory. Your project root directory should be similar to the following screenshot:



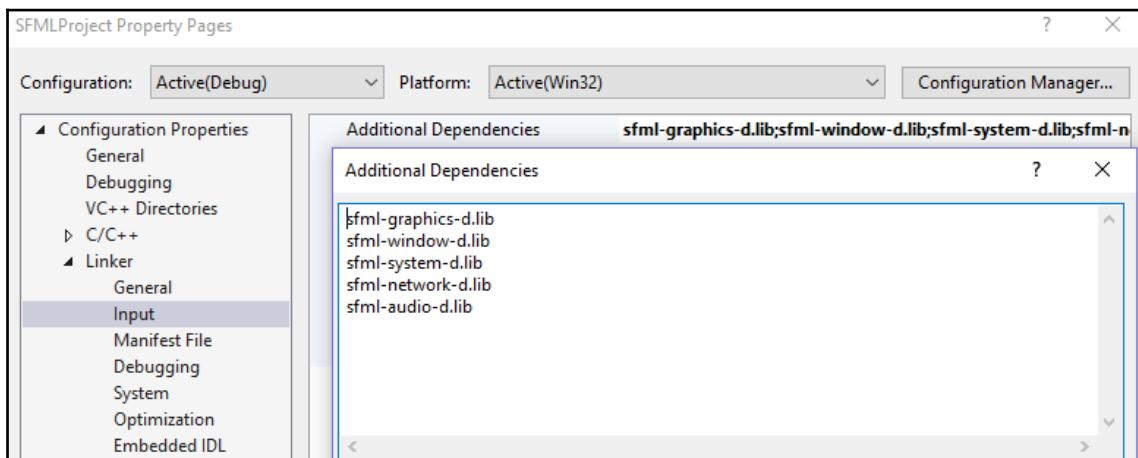
6. In the Visual Studio Project, create a new `source.cpp` file.
7. Next, open **Project Properties** by right-clicking on the project in the solution explorer.

8. Make sure that the **Win32** configuration is selected. Under **Configuration Properties**, select **VC++ Directories**. Add `$(ProjectDir)\SFML-2.5.1\include` in the **Include Directories**, and then add `$(ProjectDir)\SFML-2.5.1\lib` in **Library Directories**, as shown in the following screenshot:



The `$(ProjectDir)` keyword always makes sure that files are searched with reference to the project directory, which is where the `.vcxproj` file is. This makes the project portable and able to run on any Windows system.

9. Next, we have to set what libraries we want to use; select **Input** in the **Linker** drop-down menu and type in the following `.lib` files:





Although we won't be using `sfml-network-d.lib` in this book, it is better to include it so that if you do want to make a multiplayer game later, then you will already be setup for it.

All the setup is now complete, so we can finally start typing some code.

Creating a window

Before we draw anything, the first thing we need is a window for us to display something on the screen. So let's begin creating a window:

1. At the top of the `source.cpp` file, include the `Graphics.hpp` file to gain access to the SFML graphics library:

```
#include "SFML-2.5.1\include\SFML\Graphics.hpp"
```

2. Next, add the main function that will be the application's main entry point:

```
int main() {  
    return 0;  
}
```

3. To create the window, we have to specify the size for the window that we want to create. SFML has a `Vector2f` data type, which takes an `x` and a `y` value and uses them to define the size of the window that we will be using.

Between `include` and `main`, add the following line of code. We create a variable called `viewSize` and set the `x` and `y` values to 1024 and 768, respectively:

```
sf::Vector2f viewSize(1024, 768);
```

The assets for the game are created for the resolution so I am using this view size; we also need to specify a `viewMode` class.



The `viewMode` is a SFML class that sets the width and the height of the window. It also gets the bits that are required to represent a color value in a pixel. The `viewMode` also obtains the different resolutions that your monitor supports, so that we can let the user set the resolution of the game to glorious 4K resolution if they desire.

4. In order to set the view mode, add the following code after setting the `viewSize` variable:

```
sf::videoMode vm(viewSize.x, viewSize.y);
```

5. Now, we can finally create a window. The window is created using the `RenderWindow` class. The `RenderWindow` constructor takes three parameters: a `viewMode` parameter, a window name parameter, and a `Style` parameter.

We have already created a `viewMode` parameter and we can pass in a window name here using a string. The third parameter is `Style`. `Style` is an `enum` value; we can add a of values, called a bitmask, to create the window style that we want:

- `sf::style::Titlebar`: This adds a title bar at the top of the window.
- `sf::style::Fullscreen`: This creates a fullscreen mode window.
- `sf::style::Default`: This is the default style that combines the ability to resize a window, close it, and add a title bar.

6. We will create a default-style window. First, create the window using the following command and add it after the creation of the `viewMode` parameter:

```
sf::RenderWindow window(vm, "Hello SFMLGame !!!",  
sf::Style::Default);
```

7. In the `main()` class, we will create a `while` loop, which handles the main game loop for our game. This will check whether or not the window is open so that we can add some keyboard events by updating and rendering the objects in the scene. The `while` loop will run as long as the window is open and in the `main` function, add the following code:

```
int main() {  
    //Initialize Game Objects  
    while (window.isOpen()) {  
  
        // Handle Keyboard Events  
        // Update Game Objects in the scene  
        // Render Game Objects  
  
    }  
  
    return 0;  
}
```

You can now run the application. You have your not-so-interesting game with a window that has a white background. Hey, at least you have a window now! To display something in the video, we have to clear the window and display whatever we draw in every frame. This is done using the `clear` and `display` functions.

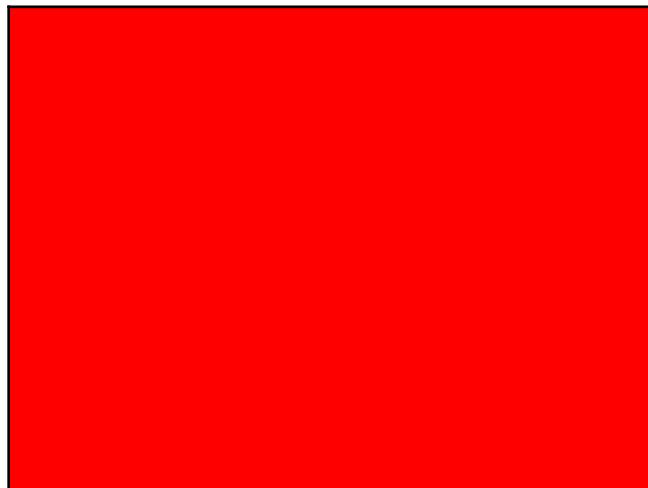
8. We have to call `window.clear()` before we render the scene, and call `window.display()` afterward to display the scene objects.

In the while loop, add the `clear` and `display` functions. Game objects will be drawn between the `clear` function and the `display` function:

```
int main() {
    //init game objects
    while (window.isOpen()) {
        // Handle Keyboard events
        // Update Game Objects in the scene
        window.clear(sf::Color::Red);
        // Render Game Objects
        window.display();
    }
    return 0;
}
```

The `clear` function takes in a clear color. Here, we pass in the color red as a value into the function. This function fills the whole window with the solid color value.

Hence, our window now is filled with the color red:



Drawing shapes

SFML provides the functionality to draw basic shapes such as a rectangle, circle, and triangle, for convenience. The shape can be set to a certain size and has functions, such as `fillColor`, `Position`, and `Origin`, so that we can set the color, the position of the shape in the viewport, and the origin around which the shape can rotate. Let's take a look at an example of a rectangular shape.

1. Before the `while` loop, add the following code to setup the rectangle:

```
sf::RectangleShape rect(sf::Vector2f(500.0f, 300.0f));
rect.setFillColor(sf::Color::Yellow);
rect.setPosition(viewSize.x / 2, viewSize.y / 2);
rect.setOrigin(sf::Vector2f(rect.getSize().x / 2,
rect.getSize().y / 2));
```

Here, we create a `Rectangle` parameter of type `RectangleShape` and name it `rect`. The constructor of `RectangleShape` takes a size of the rectangle. We create the size of 500 by 300. We then set the color of the rectangle to yellow. After this, we set the position of the rectangle to the center of the viewport and set the origin to the center of the rectangle.

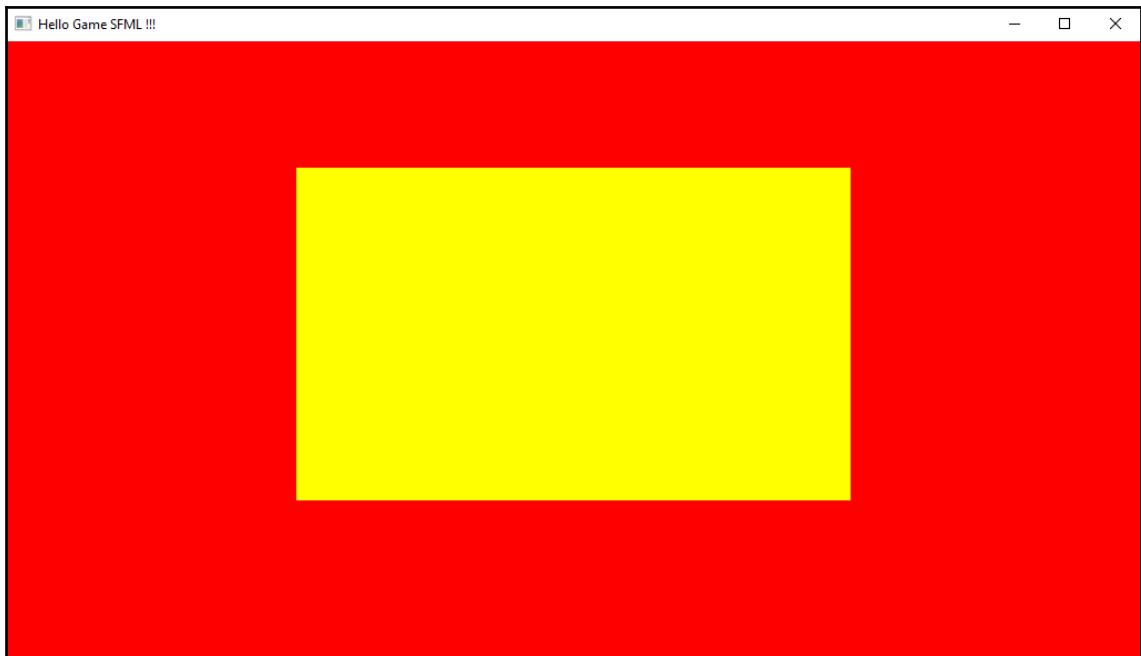
2. To draw the rectangle, we have to call the `window.draw()` function and pass the rectangle in it. Make sure you call this function between the `clear` and `display` functions in the `while` loop. You can now add the code as follows:

```
#include "SFML-2.5.1\include\SFML\Graphics.hpp"

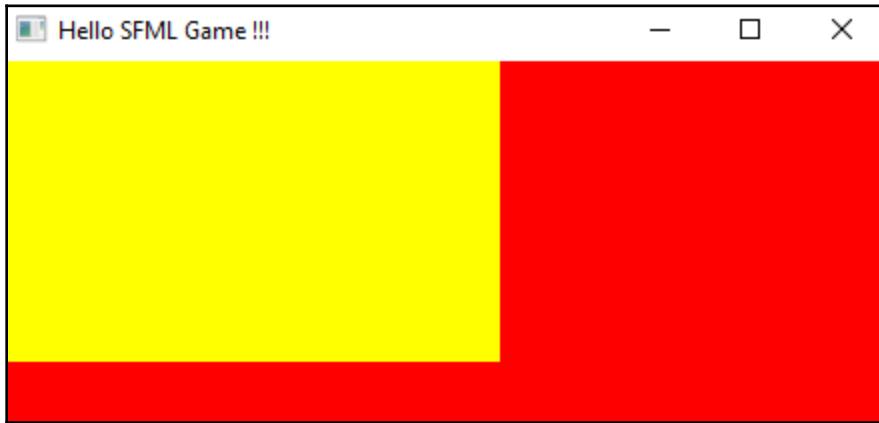
sf::Vector2f viewSize(1024, 768);
sf::VideoMode vm(viewSize.x, viewSize.y);
sf::RenderWindow window(vm, "Hello Game SFML !!!",
sf::Style::Default);
int main() {
//init game objects
sf::RectangleShape rect(sf::Vector2f(500.0f, 300.0f));
rect.setFillColor(sf::Color::Yellow);
rect.setPosition(viewSize.x / 2, viewSize.y / 2);
rect.setOrigin(sf::Vector2f(rect.getSize().x / 2, rect.getSize().y
/
2));
while (window.isOpen()) {
// Handle Keyboard events
// Update Game Objects in the scene
```

```
        window.clear(sf::Color::Red);
        // Render Game Objects
        window.draw(rect);
        window.display();
    }
    return 0;
}
```

3. Now run the project; you will see a yellow rectangle in a red viewport, as follows:



4. If we set the position to `(0, 0)`, you will see where the origin is for the two-dimensional rectangle in SFML—it is in the top-left corner of the viewport:



5. Move it back to the center of the viewport by undoing the previous action, and then set the rectangle to the center of the viewport again, as follows:

```
rect.setPosition(viewSize.x / 2, viewSize.y / 2);
```

6. We can now add a few more shapes like a circle and a triangle. A circle is created using the `CircleShape` class and a triangle is created using the `ConvexShape` class. Before the main loop, we will create a circle by using `CircleShape` and `Triangle` with `ConvexShape`, as follows:

```
sf::CircleShape circle(100);
circle.setFillColor(sf::Color::Green);
circle.setPosition(viewSize.x / 2, viewSize.y / 2);
circle.setOrigin(sf::Vector2f(circle.getRadius(),
circle.getRadius()));
sf::ConvexShape triangle;
triangle.setPointCount(3);
triangle.setPoint(0, sf::Vector2f(-100, 0));
triangle.setPoint(1, sf::Vector2f(0, -100));
triangle.setPoint(2, sf::Vector2f(100, 0));
triangle.setFillColor(sf::Color(128, 0, 128, 255));
triangle.setPosition(viewSize.x / 2, viewSize.y / 2);
```

The `CircleShape` class takes only one parameter (which is the radius of the circle) in comparison to the rectangle, which takes two parameters. We set the color of the circle to green using the `setFillColor` function, and then set its position and origin.

To create the triangle, we use the `ConvexShape` class. To create a shape, we first specify the `setPointCount`, which takes one parameter, and with it we will specify how many points will make up the shape. Next, using the `setPoint` function, we set the location of the points. This takes two parameters: the first is the index of the point and the second is the location of the point.

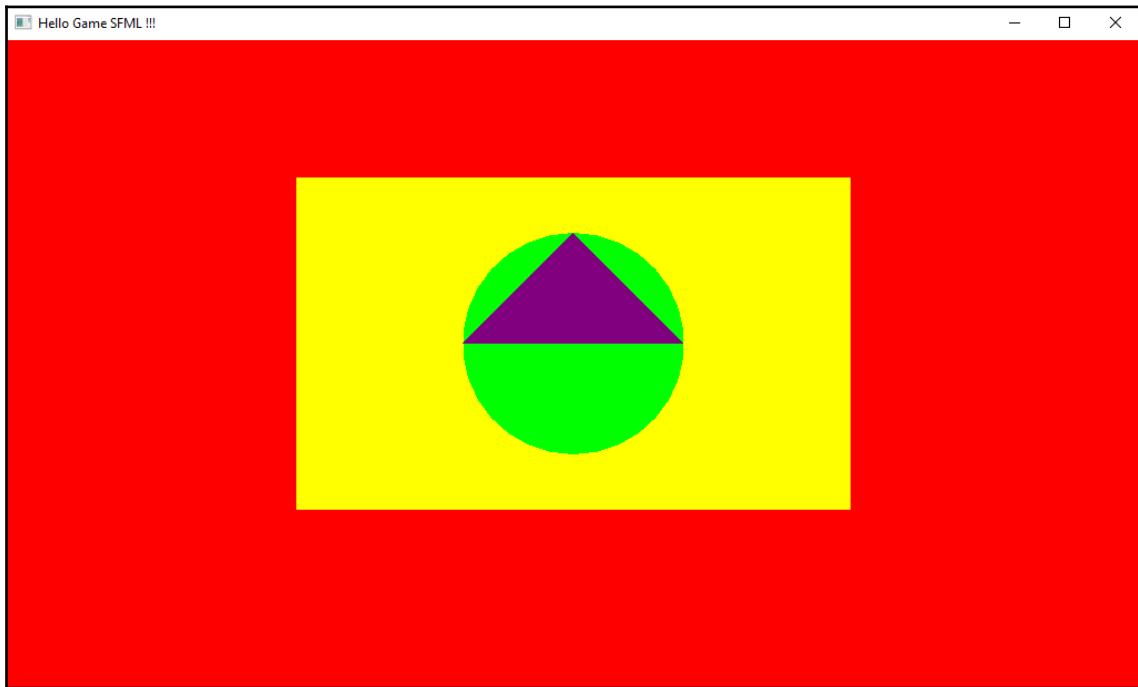
In order to create the triangle, we use three points: first, with an index of 0 and a location of `(-100, 0)`; second, with an index of 1 and a location of `(0, -100)`; and third, with an index of 2 and a location of `(100, 0)`.

We set the color of the triangle; we do this by setting the values of the red, green, blue, and alpha values. Colors in SFML seem to be 8-bit integer values. So, each color range is between 0 and 255, where 0 is black, while 255 is the maximum color range. So, when we set it to `triangle.setFillColor(sf::Color(128, 0, 128, 255))`, red is at half of its maximum range, there is no green, blue is also half of its maximum range, and alpha is 255, making the triangle fully opaque. We then set the position of the triangle to the center of the screen.

7. Next, we draw the circle and triangle. Call the `draw` function for the circle and triangle after drawing the rectangle, as follows:

```
while (window.isOpen()) {  
    // Handle Keyboard events  
    // Update Game Objects in the scene  
    window.clear(sf::Color::Red);  
    // Render Game Objects  
    window.draw(rect);  
    window.draw(circle);  
    window.draw(triangle);  
    window.display();  
}
```

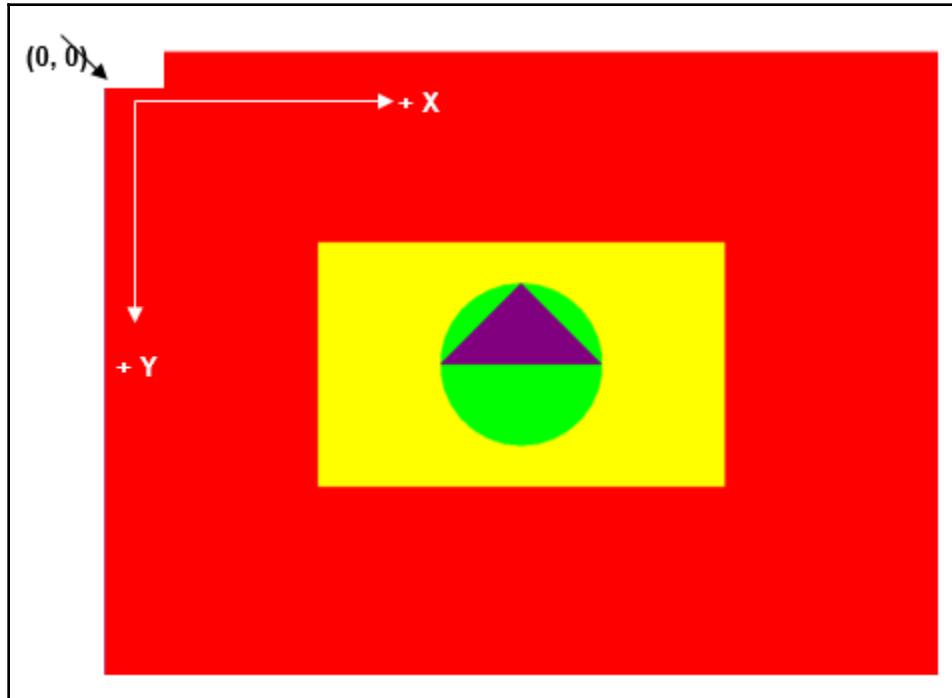
8. The output of the preceding code is as follows:



There is one thing that is very interesting to notice here is that, when creating the triangle, the second point was created with a negative y value of 100:

```
triangle.setPoint(0, sf::Vector2f(-100, 0));  
triangle.setPoint(1, sf::Vector2f(0, -100));  
triangle.setPoint(2, sf::Vector2f(100, 0));
```

However, the triangle is pointing upward. This means the $+y$ axis is downward. You will find that this is mostly the case in two-dimensional frameworks. Furthermore, the origin for the scene is in the top-left corner, so the coordinate system is as follows:

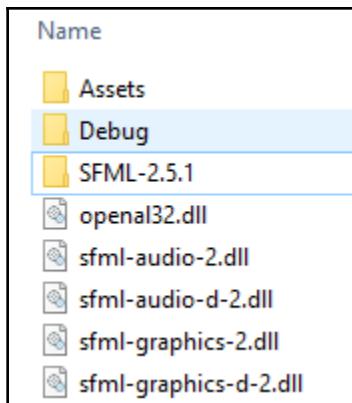


It is also important to note that the drawing order matters. Drawing happens from back to front. So, the first shape drawn will be behind the shapes that are drawn later in the same location. Objects drawn later simply draw over the earlier objects, in the same way that an artist would draw in real life when painting on a canvas. So, make sure that you draw the bigger objects first and then draw the smaller ones later. If you draw the smaller objects before the bigger ones, then the smaller objects will be behind the bigger objects and they won't appear to be drawing. Make sure this doesn't happen, as you won't get any errors and everything in the code will be correct, but it still won't look like it is supposed to.

Adding sprites

A sprite is a rectangle with a picture applied to it. You may be wondering, *so why not just use a picture?* Of course, a picture can be loaded, but then we won't be able to move or rotate it. Therefore, we apply a picture or texture to a rectangle that is able to move and rotate, making it look as if the picture is doing so.

1. Since we will be loading images in our game project, in the root directory of the project, first create a folder called `Assets`.
2. In this folder, create another folder called `graphics`, and then copy and paste the `sky.png` file into the `graphics` folder:



To create sprites, we use the `Sprite` class from SFML. The `Sprite` class takes in a texture and the picture is then loaded using the `Texture` class. While drawing, you will call `window.draw.(sprite)` to draw the sprite. Let's take a look at how to do this:

3. Declare a `Texture` class called `skyTexture` and a `Sprite` class called `skySprite` globally. This should be done after the creation of the `RenderWindow` class:

```
sf::Texture skyTexture;  
sf::Sprite skySprite;
```

4. Create a new function called `init` in the `source.cpp` file to appear right before the `main` function. Since we don't want the `main` function to be cluttered, we will add the code to initialize `skyTexture` and `skySprite` in it.

In the `init` function, add the following code:

```
void init() {  
  
    // Load sky Texture  
    skyTexture.loadFromFile("Assets/graphics/sky.png");  
    // Set and Attach a Texture to Sprite  
    skySprite.setTexture(skyTexture);  
  
}
```

First, we load the `skyTexture` function by calling the `loadFromFile` function. We pass in the path and filename of the file that we want to load. Here, we want to load the `sky.png` file from the `Assets` folder.

Next, we use the `setTexture` function of `sprite` and pass the `skyTexture` function into it.

5. To do this, create a new function called `draw()` above the `main` and `init` functions. We call the `draw(skySprite)` in it in order to draw the sprite, as follows:

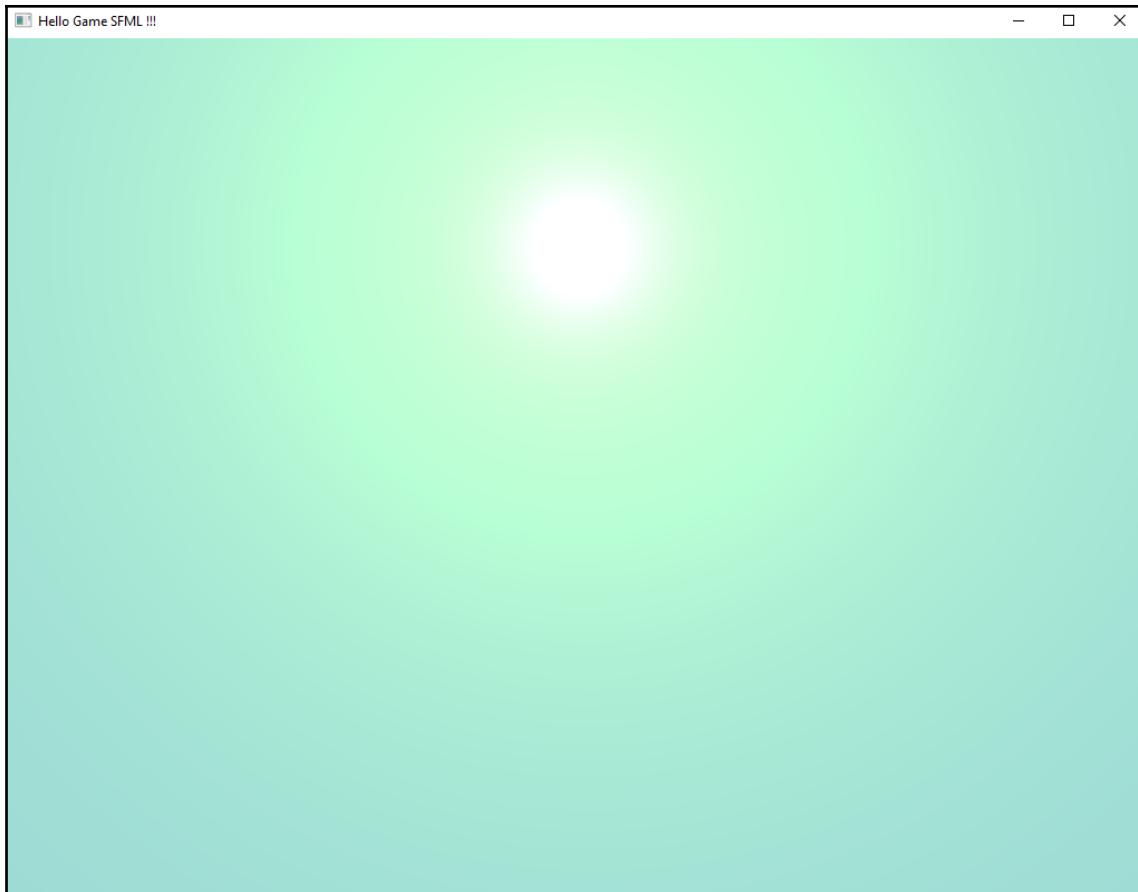
```
void draw() {  
    window.draw(skySprite);  
  
}
```

6. Now we have to call `init()` at the beginning of the `main` function, and `draw()` in the `while` loop that we added to the `main` function. You can remove all the code regarding the creation and drawing of the shapes in the `main` function. Your `main` function should appear as follows:

```
#include "SFML-2.5.1\include\SFML\Graphics.hpp"  
  
sf::Vector2f viewSize(1024, 768);  
sf::VideoMode vm(viewSize.x, viewSize.y);  
sf::RenderWindow window(vm, "Hello Game SFML !!!",  
sf::Style::Default);  
  
sf::Texture skyTexture;  
sf::Sprite skySprite;
```

```
void init() {  
  
    skyTexture.loadFromFile("Assets/graphics/sky.png");  
    skySprite.setTexture(skyTexture);  
  
}  
  
void draw() {  
    window.draw(skySprite);  
}  
  
int main() {  
    init();  
  
    while (window.isOpen()) {  
        window.clear(sf::Color::Red);  
        draw();  
  
        window.display();  
    }  
  
    return 0;  
}
```

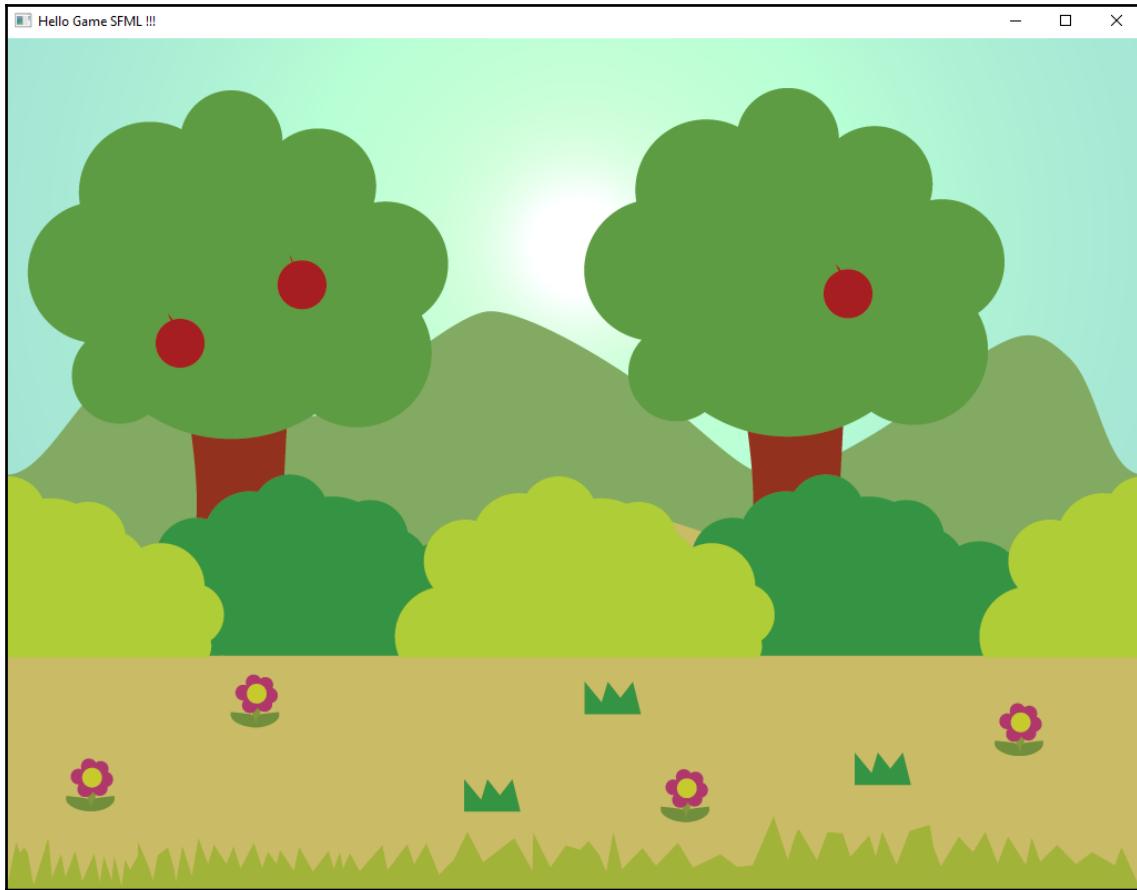
The output is as follows:



Praise the sun! Lo and behold, we have the sky texture loaded and have drawn it as a sprite in the window.

7. I have included a background texture picture as well, called `bg.png`, which is available in the `Assets` folder for this chapter's project. Try and load the texture and draw the texture in the same way.
8. I name the variables of the background texture and sprite as `bgTexture` and `bgSprite`, and draw the `bgSprite` variable in the scene. Don't forget to add the `bg.png` file in the `Assets/graphics` directory.

Your scene should now appear as follows:



9. Next, add another sprite called `heroSprite` and load the picture with `heroTexture`. Set the origin of the sprite to its center and place it in the middle of the scene. The `hero.png` file image is provided, so make sure you place it in the `Assets/graphics` folder.

So, declare `heroSprite` and `heroTexture` as follows:

```
sf::Texture heroTexture;  
sf::Sprite heroSprite;
```

In the `init` function initialize the following values:
`heroTexture.loadFromFile("Assets/graphics/hero.png");`
`heroSprite.setTexture(heroTexture);`

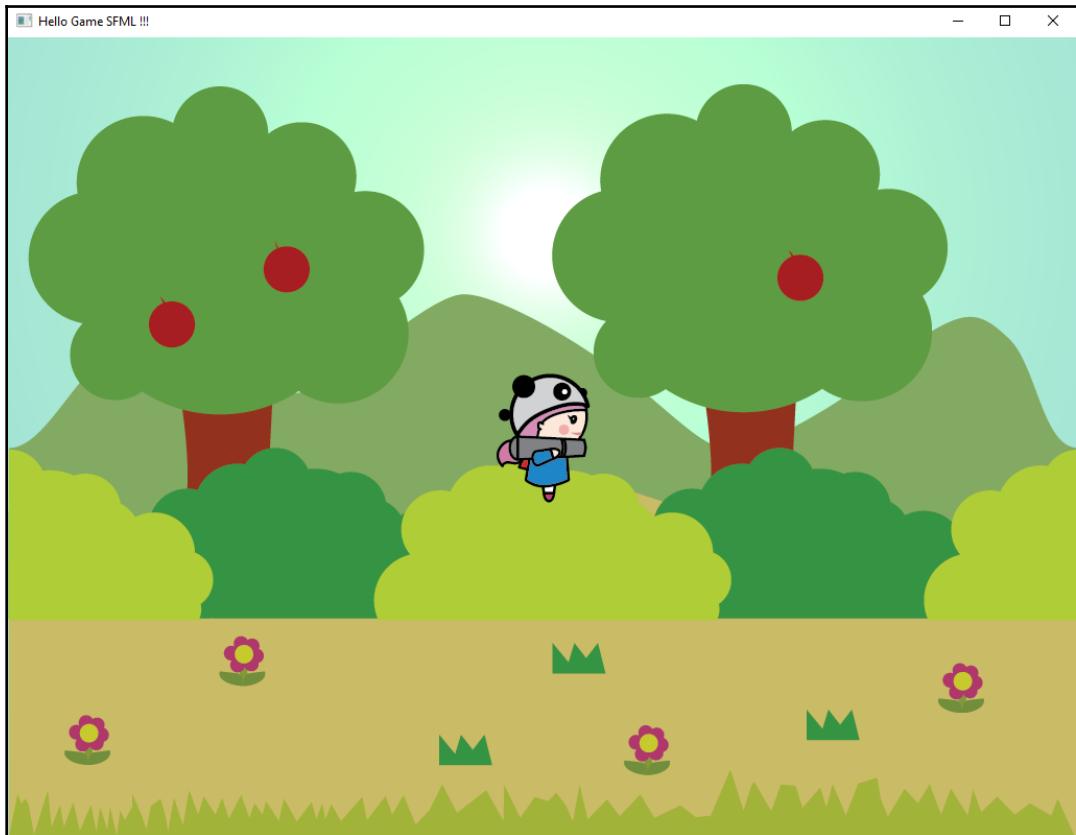
```
    heroSprite.setPosition(sf::Vector2f(viewSize.x/2,  
viewSize.y/2));  
    heroSprite.setOrigin(heroTexture.getSize().x / 2,  
heroTexture.getSize().y / 2);
```

10. To set the origin, we take the textures and the height, and divide them by 2 to set the origin for the sprite.

Using the `draw` function, draw the `heroSprite` sprite, as follows:

```
void draw() {  
    window.draw(skySprite);  
    window.draw(bgSprite);  
    window.draw(heroSprite);  
}
```

11. Our hero will now appear in the scene as follows:



Keyboard input

It is great that we are able to add shapes, sprites, and textures; however, computer games, by nature, are interactive. We will need to get the players, keyboard input in order to let them access the contents of the game. But how do we know which button on the keyboard the player is pressing? Well, that is handled through the polling of events. Polling just checks the status of the keys regularly, and events are to check whether any event was triggered, such as the closing of the viewport.

To check events, SFML has the `sf::Event` class, which we can use to poll events. We can use the `pollEvent` function of window to check for events that may be occurring, such as a player pressing a button.

Create a new function called `updateInput()`. Here, we will create a new object of `sf::Event` called `event`. We will create a `while` loop, call `window.pollEvent`, and then pass in the `event` variable to check for events.

So far, we have been using *Shift + F5* or the stop button in Visual Studio in order to stop the application. One of the basic things we can do is to check whether the *Esc* key is pressed. If it is pressed, we want to close the window.

Therefore, add the following code:

```
void updateInput() {  
  
    sf::Event event;  
  
    while (window.pollEvent(event)) {  
  
        if (event.key.code == sf::Keyboard::Escape ||  
            event.type == sf::Event::Closed)  
            window.close();  
    }  
}
```

In the `while` loop, we check if the event key code is that of the *Esc* key code, or if the type of event itself is of `Event::closed`. Then, we call the `window.close()` function to close the window. When we close the window it shuts down the application. Call the `updateInput()` function in the main `while` loop before the `window.clear()` function.

Now when you press *Esc* while the application is running, it will close. SFML doesn't limit inputs just to keyboards; it also has functionality for mouse, joystick, and touch input.

Handing player movement

Now that we have access to the player's keyboard, we can learn how to move game objects. Let's move the player character to the right when the right arrow key is pressed on the keyboard. We will stop moving the hero when the right arrow key is released:

1. Create a global `Vector2f` called `playerPosition`, right after `heroSprite`.
2. Create a Boolean data type called `playerMoving` and set it to `false`.
3. In the `updateInput` function, we will check if the right key is pressed or released. If it is pressed, we set `playerMoving` to `true`. If the button is released, then we set `playerMoving` to `false`.

The `updateInput` function should be as follows:

```
void updateInput() {  
  
    sf::Event event;  
  
    while (window.pollEvent(event)) {  
  
        if (event.type == sf::Event::KeyPressed) {  
  
            if (event.key.code == sf::Keyboard::Right) {  
  
                playerMoving = true;  
            }  
        }  
        if (event.type == sf::Event::KeyReleased) {  
  
            if (event.key.code == sf::Keyboard::Right) {  
                playerMoving = false;  
            }  
        }  
        if (event.key.code == sf::Keyboard::Escape || event.type  
        == sf::Event::Closed)  
            window.close();  
    }  
}
```

4. In order to update the objects in the scene, we will create a function called `update`, which will take a float called `dt`. This stands for delta time and refers to the time that has passed between the previous update and the current update call. In the `update` function, we will check whether the player is moving. If the player is moving, then we will move the position of the player in the `+x` direction and multiply this by `dt`.

The reason we multiply by delta time is because if we don't, then the update will not be time-dependent, but processor-dependent instead. If you don't multiply the position by `dt`, then the update will happen faster on a faster PC and will be slower on a slower PC. So, make sure that any movement is always multiplied by `dt`.

The `update` function should look like the following code block; make sure that this function appears before the `main` function:

```
void update(float dt) {  
  
    if (playerMoving) {  
        heroSprite.move(50.0f * dt, 0);  
    }  
}
```

5. At the beginning of the `main` function, create an object of type `sf::Clock` called `Clock`. The `Clock` class takes care of getting the system clock and has functionality to get the delta time in seconds, milliseconds, or microseconds.
6. In the while loop, after calling `updateInput()`, create a variable called `dt` of type `sf::Time`, and then set the `dt` variable by calling `clock.restart()`.
7. Now call the `update` function and pass in `dt.asSeconds()`, which will give the delta time as 60 frames per seconds, or approximately .0167 seconds.

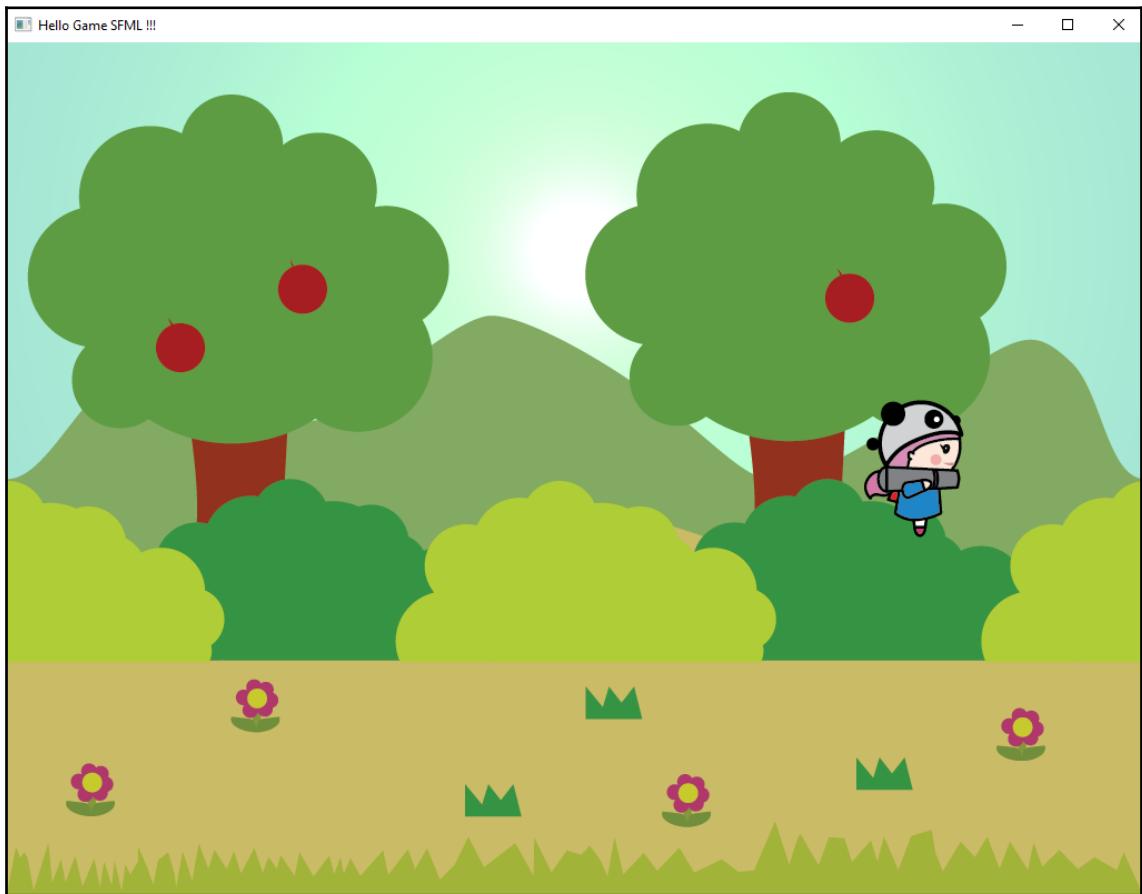
The `main` function should appear as follows:

```
int main() {  
    sf::Clock clock;  
    init();  
    while (window.isOpen()) {  
        // Update input  
        updateInput();  
  
        // Update Game  
        sf::Time dt = clock.restart();  
        update(dt.asSeconds());
```

```
    window.clear(sf::Color::Red);
    //Draw Game
    draw();

    window.display();
}
return 0;
}
```

8. Now when you run the project and press the right-arrow button on the keyboard, the player will start moving right, and will stop when you release the right-arrow key:



Summary

In this chapter, we looked at how to setup SFML to begin creating a game. We covered the five basic modules that make up SFML, as well as creating shapes using SFML and adding the background and player sprite to the scene. We also added keyboard input and used this to make the player character move within the scene.

In the next chapter, we will create the basic skeleton of the game. We will also move the player character to a separate class and add some basic physics to the character to allow them to jump in the game.

4

Creating Your Game

In this chapter, we will make our project more flexible by adding game objects as classes instead of adding them in the `source.cpp` file. For this, we will use classes to make the main character and the enemy. We will create a new rocket class that the player will be able to shoot at the enemy. We will then spawn enemies at regular intervals along with new rockets when we press a button. We will finally check for a collision between the rocket and the enemy, and accordingly, remove the enemy from the scene.

In this chapter, the following topics will be covered:

- Starting afresh
- Creating the hero class
- Creating the enemy class
- Adding enemies
- Creating the rocket class
- Adding rockets
- Collision detection

Starting afresh

Since we are going to create a new class for the main character, we will remove the code pertaining to the player character in the main file. Let's see how to do so:

Now, remove all player-related code from the `main.cpp` file. After doing so the file should appear as follows:

```
#include "SFML-2.5.1\include\SFML\Graphics.hpp"

sf::Vector2f viewSize(1024, 768);
sf::VideoMode vm(viewSize.x, viewSize.y);
sf::RenderWindow window(vm, "Hello SFML Game !!!", sf::Style::Default);

sf::Vector2f playerPosition;
bool playerMoving = false;

sf::Texture skyTexture;
sf::Sprite skySprite;

sf::Texture bgTexture;
sf::Sprite bgSprite;

void init() {

    skyTexture.loadFromFile("Assets/graphics/sky.png");
    skySprite.setTexture(skyTexture);

    bgTexture.loadFromFile("Assets/graphics/bg.png");
    bgSprite.setTexture(bgTexture);

}

void updateInput() {

    sf::Event event;

    // while there are pending events...
    while (window.pollEvent(event)) {

        if (event.key.code == sf::Keyboard::Escape || event.type ==
sf::Event::Closed)
```

```
        window.close();  
  
    }  
  
}  
  
void update(float dt) {  
  
}  
  
void draw() {  
  
    window.draw(skySprite);  
    window.draw(bgSprite);  
  
}  
  
  
int main() {  
  
    sf::Clock clock;  
    window.setFramerateLimit(60);  
  
    init();  
  
    while (window.isOpen()) {  
  
        updateInput();  
  
        sf::Time dt = clock.restart();  
        update(dt.asSeconds());  
  
        window.clear(sf::Color::Red);  
  
        draw();  
  
        window.display();  
  
    }  
  
    return 0;  
}
```

Creating the hero class

We now move on to create a new class. To do so:

1. Select the project in the solution explorer, and then right-click and select **Add | Class**. In this class name, specify the name as `Hero`. You will see the `.h` and `.cpp` file sections automatically populated as `Hero.h` and `Hero.cpp`. Click on **Ok**.
2. In the `Hero.h` file, add the SFML graphics header and create the `Hero` class:

```
#include "SFML-2.5.0\include\SFML\Graphics.hpp"

class Hero{
```

```
};
```

3. In the `Hero` class, we will create methods and variables that will be required by the class. We will also create some public properties that will be accessible outside the class, as follows:

```
public:
    Hero();
    ~Hero();

    void init(std::string textureName, sf::Vector2f position, float mass);
    void update(float dt);
    void jump(float velocity);
    sf::Sprite getSprite();
```

Here, we have the constructor and destructor, which will be called when an object is created and destroyed. We will add an initialization function in which to pass in a texture name, spawn the player, and specify a mass. We are specifying a mass here because we will be creating some very basic physics so that when we hit the jump button, the player will jump up and land back down on their feet.

Additionally, the `update`, `jump`, and `getSprite` functions will update the player position, make the player jump, and get the sprite of the player used for drawing the player character.

4. Apart from these `public` variables, we will also need some `private` variables that can only be accessed within the class. Add these in the `Hero` class, as follows:

```
private:
    sf::Texture m_texture;
```

```
sf::Sprite m_sprite;
sf::Vector2f m_position;

int jumpCount = 0;
float m_mass;
float m_velocity;
const float m_gravity = 9.80f;
bool m_grounded;
```

In the `private` section, we create variables for the texture, sprite, and position so that we can set these values locally. We have the `int` variable called `jumpCount` so that we can check the number of times the player character has jumped. This is needed because the player can sometimes double-jump, which is something we don't want.

We will also need the `float` variables to store the player's mass, the velocity of the player when they jump, and the gravitational force when they fall back to the ground, which is a constant. The `const` keyword tells the program that it is a constant and under no circumstance should the value be made changeable.

Lastly, we add a `bool` value to check whether the player is on the ground. Only when the player is on the ground can they start jumping.

5. Next, in the `Hero.cpp` file, we will implement the functions that we added in the `.h` file. At the top of the `.cpp` file, include the `Hero.h` file, and then add the constructor and destructor:

```
#include "Hero.h"
Hero::Hero() {

}
```



`::` represents the scope resolution operator. Functions that have the same name can be defined in two different classes. In order to access the methods of a particular class, the scope resolution operator is used.

6. So, here, the `Hero` function is scoped to the `Hero` class:

```
Hero::~Hero() {

}
```

7. Next, we will set up the `init` function, as follows:

```
void Hero::init(std::string textureName, sf::Vector2f position,
float mass){

    m_position = position;
    m_mass = mass;

    m_grounded = false;
    // Load a Texture
    m_texture.loadFromFile(textureName.c_str());

    // Create Sprite and Attach a Texture
    m_sprite.setTexture(m_texture);
    m_sprite.setPosition(m_position);
    m_sprite.setOrigin(m_texture.getSize().x / 2,
    m_texture.getSize().y / 2);

}
```

We set the position and mass to the local variable and set the grounded state to `false`. Then, we set the texture by calling `loadFromFile` and passing in the string of the texture name to it. `c_str()` returns a pointer to an array that contains a null-terminated sequence of characters (that is, a C string) representing the current value of the string object (http://www.cplusplus.com/reference/string/string/c_str/). We then set the sprite texture, position, and the origin of the sprite itself.

8. Now, we add the `update` function where we implement the logic for updating the player position. The player's character cannot move left or right; instead, it can only move in the up direction, which is the `y` direction. When an initial velocity is applied, the player will jump up and then start falling down due to gravity. Add the `update` function to update the position of the hero, as follows:

```
void Hero::update(float dt){

    m_force -= m_mass * m_gravity * dt;

    m_position.y -= m_force * dt;

    m_sprite.setPosition(m_position);

    if (m_position.y >= 768 * 0.75f) {

        m_position.y = 768 * 0.75f;
```

```
m_force = 0;  
m_grounded = true;  
jumpCount = 0;  
}  
  
}
```

When the velocity is applied to the character, the player will initially go up because of the force, but will then start coming down due to gravity. The resultant velocity acts in the downward direction and is calculated by the following formula:

$$\text{Velocity} = \text{Acceleration} \times \text{Time}$$

We multiply the acceleration by the mass so that the player falls faster. To calculate the distance moved vertically, we use the following formula:

$$\text{Distance} = \text{Velocity} \times \text{Time}$$

Then, we calculate the distance moved between the previous and the current frame. We then set the position of the sprite based on the position that we calculated.

We also have a condition to check whether the player is at one-fourth of the distance from the bottom of the screen, so we multiply by 768, which is the height of the window, and then multiply it by .75f, at which point, the player is considered on the ground. If that condition is satisfied, we set the position of the player, set the resultant velocity to 0, set the grounded Boolean to `true`, and, finally, reset the jump counter to 0.

9. Next, whenever we want to make the player jump, we will call the `jump` function, which takes an initial velocity. We now add the `jump` function, as follows:

```
void Hero::jump(float velocity) {  
  
    if (jumpCount < 2) {  
        jumpCount++;  
  
        m_velocity = VELOCITY;  
        m_grounded = false;  
    }  
  
}
```

Here, we first check whether `jumpCount` is less than 2 as we only want the player to only jump twice. If `jumpCount` is less than 2, then increment the `jumpCount` value by 1, set the initial velocity, and set the `grounded` Boolean to `false`.

10. Finally, we now add the `getSprite` function, which simply gets the current sprite, as follows:

```
sf::Sprite Hero::getSprite() {  
    return m_sprite;  
}
```

Congrats! We now have our `Hero` class ready. Let's now use it in the `source.cpp` file. To do so:

1. Include `Hero.h` at the top of the `main.cpp` file:

```
#include "SFML-2.5.1\include\SFML\Graphics.hpp"  
#include "Hero.h"
```

2. Next, add an instance of the `Hero` class as follows:

```
sf::Texture skyTexture;  
sf::Sprite skySprite;  
  
sf::Texture bgTexture;  
sf::Sprite bgSprite;  
Hero hero;
```

3. In the `init` function, initialize the `Hero` class:

```
// Load bg Texture  
  
bgTexture.loadFromFile("Assets/graphics/bg.png");  
  
// Create Sprite and Attach a Texture  
bgSprite.setTexture(bgTexture);  
hero.init("Assets/graphics/hero.png", sf::Vector2f(viewSize.x *  
0.25f, viewSize.y * 0.5f), 200);
```

Here, we set the texture picture; to do so, set the position to be at .25 (or 25%) from the left of the screen and center it along the y axis. We also set the mass to be 200, as our character is quite chubby.

4. Next, we want the player to jump when we press the up arrow key. Therefore, in the `updateInput` function, while polling for window events, we add the following code:

```
while (window.pollEvent(event)) {  
    if (event.type == sf::Event::KeyPressed) {  
        if (event.key.code == sf::Keyboard::Up) {  
            hero.jump(750.0f);  
        }  
    }  
    if (event.key.code == sf::Keyboard::Escape || event.type ==  
sf::Event::Closed)  
        window.close();  
  
}
```

Here, we check whether a key was pressed by the player. If a key is pressed and the button is the up arrow on the keyboard, then we call the `hero.jump` function and pass in an initial velocity value of 750.

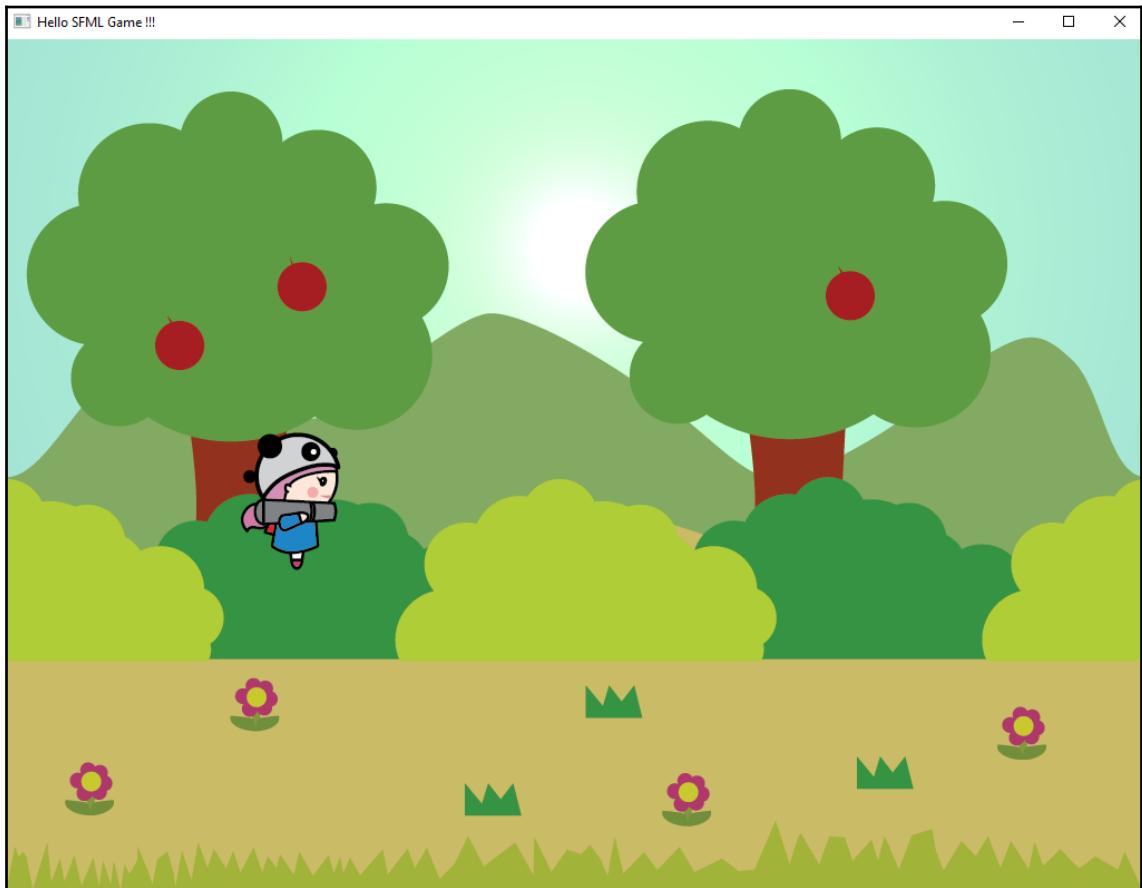
5. Next, in the `update` function, we call the `hero.update` function, as follows:

```
void update(float dt) {  
    hero.update(dt);  
}
```

6. Finally, in the `draw` function, we draw the hero sprite:

```
void draw() {  
  
    window.draw(skySprite);  
    window.draw(bgSprite);  
    window.draw(hero.getSprite());  
  
}
```

7. You can now run the game; when the player is on the ground, press the up arrow button on the keyboard to see the player jump. Furthermore, when the player is in the air, press the jump button again and you will see the player jump again in midair:



Creating the enemy class

The player character looks very lonely now. She is ready to cause some mayhem, but there is nothing to shoot right now. So, let's add some enemies to solve this problem.

1. The enemies will be created using an enemy class; let's create a new class and call it `Enemy`.
2. Just like the `Hero` class, the `Enemy` class will have a `.h` file and a `.cpp` file. In the `Enemy.h` file, add the following code:

```
#pragma once
#include "SFML-2.5.1\include\SFML\Graphics.hpp"

class Enemy
{
public:
    Enemy();
    ~Enemy();

    void init(std::string textureName, sf::Vector2f position, float
    _speed);
    void update(float dt);
    sf::Sprite getSprite();

private:
    sf::Texture m_texture;
    sf::Sprite m_sprite;
    sf::Vector2f m_position;
    float m_speed;

};
```

Here, the `Enemy` class, just like the `Hero` class, also has a constructor and destructor. Additionally, it has an `init` function that takes in the texture and position; however, instead of mass, it takes in a float variable that will be used to set the initial velocity of the enemy. The enemy won't be affected by gravity and will only spawn from the right of the screen and move toward the left of the screen. There are also `update` and `getSprite` functions; since the enemy won't be jumping, there won't be a `jump` function. Lastly in the private section, we create local variables for the texture, sprite, position, and speed.

3. In the `Enemy.cpp` file, we add the constructor, destructor, `init`, `update`, and `getSprite` functions:

```
#include "Enemy.h"

Enemy::Enemy() {}

Enemy::~Enemy() {}

void Enemy::init(std::string textureName, sf::Vector2f position,
float _speed) {

    m_speed = _speed;
    m_position = position;

    // Load a Texture
    m_texture.loadFromFile(textureName.c_str());

    // Create Sprite and Attach a Texture
    m_sprite.setTexture(m_texture);
    m_sprite.setPosition(m_position);
    m_sprite.setOrigin(m_texture.getSize().x / 2,
    m_texture.getSize().y / 2);

}
```

Don't forget to include `Enemy.h` at the top of the main function; we then add the constructor and destructor. In the `init` function, we set the local speed and position values. Next, we load `Texture` from the file and set the texture, position, and origin of the enemy sprite.

4. In the `update` and `getSprite` functions, we update the position and get the enemy sprite:

```
void Enemy::update(float dt) {

    m_sprite.move(m_speed * dt, 0);

}

sf::Sprite Enemy::getSprite() {

    return m_sprite;
}
```

5. We now have our `Enemy` class ready. Let's now see how we can use it in the game.

Adding enemies

In the `main.cpp` class, include the `Enemy` header file. Since we want more than one enemy instance, we need to add a vector called `enemies` and then add all the newly-created enemies to it.

In this context, vector has absolutely nothing to do with math, but rather with a list of objects. In fact, it is like an array in which we can store multiple objects. Vectors are used instead of an array because vectors are dynamic in nature, so it makes it easier to add and remove objects from the list (unlike an array, which, by comparison, is a static list).

1. Hence, we need to include `<vector>` in the `main.cpp` file; so, add it as follows:

```
#include "SFML-2.5.1\include\SFML\Graphics.hpp"
#include <vector>

#include "Hero.h"
#include "Enemy.h"
```

2. Next, add a new variable called `enemies` of the `vector` type, which will store the `Enemy` data type in it:

```
sf::Texture bgTexture;
sf::Sprite bgSprite;

Hero hero;

std::vector<Enemy*> enemies;
```

3. In order to create a vector of a certain object type, you use the `vector` keyword and, inside the arrow brackets, specify the data type that the vector will hold, and then specify a name for the vector you have created. Hence, we create a new function called `spawnEnemy()` and add a prototype for it at the top of the main function.



When any function is written below, the main function is not aware that such a function exists. Therefore, a prototype is created and placed above the main function. This means that the function can now be implemented below the main function – essentially, the prototype tells the main function that there is a function that will be implemented below it, and so to keep a lookout for it.

```
sf::Vector2f viewSize(1024, 768);
sf::VideoMode vm(viewSize.x, viewSize.y);
sf::RenderWindow window(vm, "Hello SFML Game !!!",
sf::Style::Default);

void spawnEnemy();
```

Now, we want the enemy to spawn from the right of the screen, but we also want the enemy to spawn at either the same height as the player, slightly higher than the player, or much higher than the player, so that the player will have to use a single jump or a double jump to attack the enemy.

4. To do this, we add some randomness to the game to make it less predictable. For this, we add the following line of code underneath the `init` function:

```
hero.init("Assets/graphics/hero.png", sf::Vector2f(viewSize.x *
0.25f, viewSize.y * 0.5f), 200);

srand((int)time(0));
```

`srand` is a pseudo-random number that is initialized by passing a seed value. In this case, we are passing in the current time as a seed value.

For each seed value, a series of numbers will be generated. If the seed value is always the same, then the same series of numbers will be generated. That is why we pass in the time value – so that the sequence of numbers that are generated will be different each time. We can get the next random number in the series by calling the `rand` function.

5. Next, we add the `spawnEnemy` function as follows:

```
void spawnEnemy() {
    int randLoc = rand() % 3;

    sf::Vector2f enemyPos;

    float speed;
```

```
switch (randLoc) {  
  
    case 0: enemyPos = sf::Vector2f(viewSize.x, viewSize.y * 0.75f);  
    speed = -400; break;  
  
    case 1: enemyPos = sf::Vector2f(viewSize.x, viewSize.y * 0.60f);  
    speed = -550; break;  
  
    case 2: enemyPos = sf::Vector2f(viewSize.x, viewSize.y * 0.40f);  
    speed = -650; break;  
  
    default: printf("incorrect y value \n"); return;  
}  
  
Enemy* enemy = new Enemy();  
enemy->init("Assets(graphics/enemy.png", enemyPos, speed);  
  
enemies.push_back(enemy);  
}
```

Here, we first get a random number – this will create a new random number from 0 to 2 because of the modulo operator while getting the random location. So, the value of `randLoc` will either be 0, 1, or 2 each time the function is called.

A new `enemyPos` variable is created that will be assigned depending upon the `randLoc` value. We will also be setting the speed of the enemy depending on the `randLoc` value; so, for this, we create a new float called `speed`, which we will assign later. We then create a `switch` statement that takes in the `randLoc` value—this enables the random location to spawn the enemy.

Depending upon the scenario, we can set the `enemyPosition` variable and the speed of the enemy.

- When `randLoc` is 0, the enemy spawns from the bottom and moves with a speed of -400;
- when `randLoc` is 1, the enemy spawns from the middle of the screen and moves at a speed of -500;
- when the value of `randLoc` is 2, then the enemy spawns from the top of the screen and moves at the faster speed of -650.
- If `randLoc` is none of these values, then a message is printed out saying that value of `y` is incorrect, and instead of breaking, we return to make sure that the enemy doesn't spawn in a random location.

To print out the message to the console, we can use the `printf` function, which takes a string value. At the end of the string, we specify `\n`, this is a keyword to tell compiler that it is the end of the line and whatever is written after needs to be put in a new line, similar to when calling `std::cout`.

6. Once we know the position and speed, we can then create the enemy object itself and initialize it. Notice that the enemy is created as a pointer; otherwise, the reference to the texture is lost and the texture won't display when the enemy is spawned. Additionally, when we create the enemy as a raw pointer, with a new keyword memory is allocated by the system which we will have to delete later.
7. After the enemy is created, we add it to the `enemies` vector by calling the `push` function of vectors.
8. Now we want the enemy to spawn automatically at regular intervals. For this, we create two new variables to keep track of the current time and to spawn a new enemy every `1.125` seconds.
9. Next, create two new variables of the `float` type, called `currentTime` and `prevTime`:

```
Hero hero;

std::vector<Enemy*> enemies;

float currentTime;
float prevTime = 0.0f;
```

10. Then, in the `update` function, after updating the `hero` function, add the following lines of code in order to create a new enemy:

```
hero.update(dt);
currentTime += dt;
// Spawn Enemies
if (currentTime >= prevTime + 1.125f))) {
spawnEnemy();
prevTime = currentTime;
}
```

Here, we first increment the `currentTime` variable. This variable will begin increasing as soon as the game starts so that we can track how long it has been since we started the game. Next, we check whether the current time is greater than or equal to the previous time plus the 1.125 seconds, as that is when we want the new enemy to spawn. If it is `true`, then we call the `spawnEnemy` function, which will create a new enemy. We also set the previous time as equal to the current time so that we know when the last enemy was spawned. Good! So, now that we have enemies spawning in the game, we can update the enemies and also draw them.

11. Now in the `update` function, we also create a `for` loop to update the enemies and delete the enemies once they go beyond the left of the screen. To do this, we add the following code to the `update` function:

```
// Update Enemies

for (int i = 0; i < enemies.size(); i++) {

    Enemy *enemy = enemies[i];

    enemy->update(dt);

    if (enemy->getSprite().getPosition().x < 0) {

        enemies.erase(enemies.begin() + i);
        delete(enemy);

    }
}
```

This is where the use of vectors becomes really useful. Vectors have the functionality to add, delete, and insert elements in the vector. In the example here, we get the reference of the enemy at the location index of `i` in the vector. If that enemy goes offscreen and needs to be deleted, then we can just use the `erase` function and pass the location index from the beginning of the vector to remove the enemy at that index. When we reset the game, we also delete the local reference of the enemy we created. This will also free the memory space allocated when we created the new enemy.

12. In the `draw` function, we go through each of the enemies in a `for...each` loop and draw them:

```
window.draw(skySprite);
window.draw(bgSprite);
```

```
window.draw(hero.getSprite());  
  
for (Enemy *enemy : enemies) {  
  
    window.draw(enemy->getSprite());  
}
```

We use the `for...each` loop to go through all the enemies, since the `getSprite` function needs to be called on all of them. Interestingly, we didn't use `for...each` when we had to update the enemies, because with the `for` loop, we can simply use the index of the enemy if we have to delete it.

13. Finally, add the `Enemy.png` file in the `Assets/graphics` folder. Now when you run the game, you will see enemies spawning at different heights and moving toward the left of the screen:



Creating a rocket class

The game has enemies in it now, but the player still can't shoot at them. Let's create some rockets so that these can be launched from the player's bazooka. To do so:

1. In the project, create a new class called `Rocket`. As you can see from the following code block, the `Rocket.h` class is very similar to the `Enemy.h` class:

```
#pragma once

#include "SFML-2.5.1\include\SFML\Graphics.hpp"

class Rocket
{
public:
    Rocket();
    ~Rocket();

    void init(std::string textureName, sf::Vector2f position, float
    _speed);
    void update(float dt);
    sf::Sprite getSprite();

private:
    sf::Texture m_texture;
    sf::Sprite m_sprite;
    sf::Vector2f m_position;
    float m_speed;

};
```

The `public` section contains the `init`, `update`, and `getSprite` functions. `init` takes in the name of the texture to load, the position to set, and the speed at which the object is initialized. The `private` section has local variables for the `texture`, `sprite`, `position`, and `speed`.

2. In the Rocket.cpp file, we add the constructor and destructor, as follows:

```
#include "Rocket.h"

Rocket::Rocket() {
}

Rocket::~Rocket() {
```

In the init function, we set the speed and position variables. Then, we set the texture variable and initialize the sprite with the texture variable.

3. Next, we set the position variable and origin of the sprite, as follows:

```
void Rocket::init(std::string textureName, sf::Vector2f position,
float _speed) {

    m_speed = _speed;
    m_position = position;

    // Load a Texture
    m_texture.loadFromFile(textureName.c_str());

    // Create Sprite and Attach a Texture
    m_sprite.setTexture(m_texture);
    m_sprite.setPosition(m_position);
    m_sprite.setOrigin(m_texture.getSize().x / 2,
m_texture.getSize().y / 2);

}
```

4. In the update function, the object is moved according to the speed variable:

```
void Rocket::update(float dt) {
    \
    m_sprite.move(m_speed * dt, 0);

}
```

5. The getSprite function returns the current sprite as follows:

```
sf::Sprite Rocket::getSprite() {

    return m_sprite;
}
```

Adding rockets

Now that we have created the rockets, lets see how to add them:

1. In the `main.cpp` file, we include the `Rocket.h` class, as follows:

```
#include "Hero.h"
#include "Enemy.h"
#include "Rocket.h"
```

2. We then create a new vector of `Rocket` called `rockets`, which takes in `Rocket`:

```
std::vector<Enemy*> enemies;
std::vector<Rocket*> rockets;
```

3. In the update function, after we have updated all the enemies, we update all the rockets. We also delete the rockets that go beyond the right of the screen:

```
// Update Enemies

for (int i = 0; i < enemies.size(); i++) {

    Enemy* enemy = enemies[i];

    enemy->update(dt);

    if (enemy->getSprite().getPosition().x < 0) {

        enemies.erase(enemies.begin() + i);
        delete(enemy);

    }
}

// Update rockets

for (int i = 0; i < rockets.size(); i++) {

    Rocket* rocket = rockets[i];

    rocket->update(dt);

    if (rocket->getSprite().getPosition().x > viewSize.x) {
        rockets.erase(rockets.begin() + i);
        delete(rocket);
    }
}
```

- Finally, we draw all the rockets with the `draw` function by going through each rocket in the scene:

```
for (Enemy *enemy : enemies) {  
  
    window.draw(enemy->getSprite());  
}  
  
for (Rocket *rocket : rockets) {  
    window.draw(rocket->getSprite());  
}
```

- Now, we can actually shoot the rockets. In the `main.cpp`, class, create a new function called `shoot()` and add a prototype for it at the top of the main function:

```
void spawnEnemy();  
void shoot();
```

- In the `shoot` function, we will add the functionality to shoot the rockets. We will spawn new rockets and push them back to the `rockets` vector. You can add the `shoot` function as follows:

```
void shoot() {  
  
    Rocket* rocket = new Rocket();  
  
    rocket->init("Assets/graphics/rocket.png",  
                  hero.getSprite().getPosition(),  
                  400.0f);  
  
    rockets.push_back(rocket);  
}
```

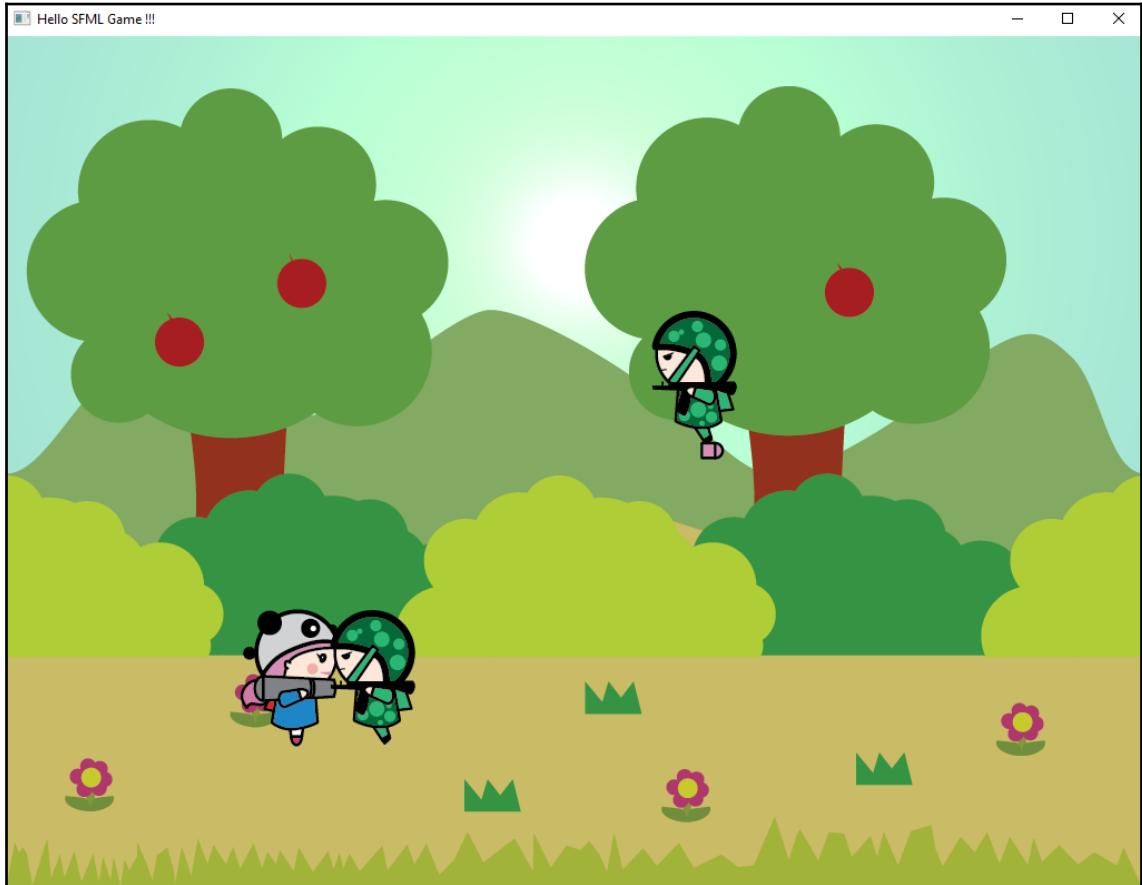
When this function is called, it creates a new `Rocket` and initializes it with the `Rocket.png` file, sets the position of it as equal to the position of the `hero` sprite, and then sets the velocity to `400.0f`. The rocket is then added to the `rockets` vector.

- Now in the `updateInput` function, add the following code so that when the down arrow key is pressed the `shoot` function is called:

```
if (event.type == sf::Event::KeyPressed) {  
  
    if (event.key.code == sf::Keyboard::Up) {
```

```
        hero.jump(750.0f);  
    }  
  
    if (event.key.code == sf::Keyboard::Down) {  
  
        shoot();  
    }  
}
```

8. Don't forget to place the `rocket.png` file in the `assets` folder. When you run the game now and press the down arrow key, a rocket is fired:



Collision detection

For the final section for this chapter, let's add some collision detection so that the rocket actually kills an enemy when they both come into contact with each other.

1. Create a new function called `checkCollision`, and then create a prototype for it at the top of the main function:

```
void spawnEnemy();
void shoot();

bool checkCollision(sf::Sprite sprite1, sf::Sprite sprite2);
```

2. This function takes two sprites so that we can check the intersection of one with the other. Add the following code for the function in the same place that we added the `shoot` function:

```
void shoot() {

    Rocket* rocket = new Rocket();

    rocket->init("Assets(graphics/rocket.png",
hero.getSprite().getPosition(), 400.0f);

    rockets.push_back(rocket);

}

bool checkCollision(sf::Sprite sprite1, sf::Sprite sprite2) {

    sf::FloatRect shape1 = sprite1.getGlobalBounds();
    sf::FloatRect shape2 = sprite2.getGlobalBounds();

    if (shape1.intersects(shape2)) {

        return true;

    }
    else {

        return false;

    }

}
```

Inside this `checkCollision` function, we create two local variables of the `FloatRect` type. We then assign the `GlobalBounds` of the sprites to each `FloatRect` variable named `shape1` and `shape2`. The `GlobalBounds` gets the rectangular region the object is spanning of the sprite from where it is currently.

The `FloatRect` type is simply a rectangle; we can use the `intersects` function to check if this rectangle intersects with another rectangle. If the first rectangle intersects with the other rectangle, then we return `true` to say that there is an intersection or collision between the sprites. If there is no intersection, then we return `false`.

3. In the `update` function, after updating the `enemy` and `rocket` classes, we check the collision between each rocket and each enemy in a nested `for` loop. You can add the collision check as follows:

```
// Update rockets

for (int i = 0; i < rockets.size(); i++) {

    Rocket* rocket = rockets[i];

    rocket->update(dt);
    if (rocket->getSprite().getPosition().x > viewSize.x) {

        rockets.erase(rockets.begin() + i);
        delete(rocket);

    }
}

// Check collision between Rocket and Enemies

for (int i = 0; i < rockets.size(); i++) {
    for (int j = 0; j < enemies.size(); j++) {
        Rocket* rocket = rockets[i];
        Enemy* enemy = enemies[j];

        if (checkCollision(rocket->getSprite(), enemy->getSprite())) {

            rockets.erase(rockets.begin() + i);
            enemies.erase(enemies.begin() + j);

            delete(rocket);
            delete(enemy);

        }
    }
}
```

```
    printf(" rocket intersects enemy \n");
}

}

Here, we create a double for loop, call the checkCollision function, and then
pass each rocket and enemy into it to check the intersection between them.
```

4. If there is an intersection, we remove the rocket and enemy from the vector and delete them from the scene. With this we are done with Collision Detection.

Summary

In this chapter, we created a separate `Hero` class so that all the code pertaining to the `Hero` class was in one single file. In this `Hero` class, we managed the jumping and the shooting of the rockets in the class. Next, we created the `Enemy` class – because for every hero, there needs to be a villain in the story! We learned how to add enemies to a vector so that it is easier to loop between the enemies in order to update their position. We also created a `Rocket` class and managed the rockets using a vector. Finally, we learned how to check for collisions between the enemies and the rockets. This creates the foundation of the gameplay loop.

In the next chapter, we will finish the game, adding sound and text to it in order to give audio feedback to the player and use text to show the current score.

5

Finalizing Your Game

In the previous chapter, we saw how to create the game; in this chapter, we will finish the **Gameloop** so that you can play the game. The objective of the game is to make sure that none of the enemies are able to make it to the left of the screen. If they do, it is game over.

We will add a scoring system so that the player knows how much they have scored in a round. For each enemy that is shot down, the player will get one point. We will also add text in the game to display the title of the game, the player's score, and a small tutorial that shows you how to play the game.

At the end of the chapter, we will embellish the game. We will add audio to use as background music, and sound effects for when the player shoots the rocket and when the player's rockets hit the enemy. We will also add some animation to the player so that the character looks more lively.

We will cover the following topics in this chapter:

- Finishing the Gameloop and adding scoring
- Adding text
- Adding audio
- Adding player animations

So let's begin!

Finishing the Gameloop and adding scoring

The following steps will show you how to finish the Gameloop and add scoring into the game code:

1. Add two new variables in the `source.cpp` file: one of the `int` type, called `score`, and one of the `bool` type, called `gameover`. Initialize the `score` to `0` and `gameover` to `true`:

```
std::vector<Enemy*> enemies;
std::vector<Rocket*> rockets;

float currentTime;
float prevTime = 0.0f;

int score = 0;
bool gameover = true;
```

2. Create a new function, called `reset()`, which we will use to reset the variables. So create a prototype for the `reset` function at the top:

```
bool checkCollision(sf::Sprite sprite1, sf::Sprite sprite2);
void reset();
```

At the bottom of the `source.cpp` file, after where we created the `checkCollision` function, add the `reset` function itself so that when the game resets, all the values are also reset. To do this, use the following code:

```
void reset() {

    score = 0;
    currentTime = 0.0f;
    prevTime = 0.0;

    for (Enemy *enemy : enemies) {
        delete(enemy);
    }
    for (Rocket *rocket : rockets) {
        delete(rocket);
    }

    enemies.clear();
    rockets.clear();
}
```

The `reset()` function will be called when the player presses the fire button, which is the down arrow button on the keyboard, again to restart the game. In the `reset()` function, we need to set the `score`, `currentTime`, and `prevTime` to 0.

When the game resets, remove any instantiated enemy and rocket objects by deleting and thus freeing the memory and afterward you also clear the vectors that were holding a reference to the now deleted objects. After setting up the variables and the `reset` function, let's use them in the game to reset the values when we restart the game.

In the `UpdateInput` function, in the `while` loop where we check whether the down arrow button on the keyboard was pressed, we will add a check if the game is over. If it is over, we'll set the `gameover` bool to false so that the game is ready to start, and we'll reset the variables by calling the `reset` function, as follows:

```
if (event.key.code == sf::Keyboard::Down) {  
  
    if (gameover) {  
        gameover = false;  
        reset();  
    }  
    else {  
        shoot();  
    }  
}
```

Here, the `shoot` is moved into an `else` statement so that the player can only shoot if the game is running.

Next, we will set the `gameover` condition to true when an enemy goes beyond the left side of the screen.

When we update the enemies, the enemy will be deleted when it disappears from the screen, and we will also set the `gameover` condition to true.

Add this code when we update the enemies in the `update()` function:

```
// Update Enemies  
for (int i = 0; i < enemies.size(); i++) {  
  
    Enemy* enemy = enemies[i];  
  
    enemy->update(dt);  
  
    if (enemy->getSprite().getPosition().x < 0) {
```

```
        enemies.erase(enemies.begin() + i);
        delete(enemy);
        gameover = true;
    }
}
```

3. Update the game only if `gameover` is `false`. In the main function, before we update the game, we will add a check to see whether the game is over. If the game is over, we will not update the game. To do this, use the following code:

```
while (window.isOpen()) {
    ///////////////////////////////////////////////////
    //update input
    updateInput();
    ///////////////////////////////////////////////////
    sf::Time dt = clock.restart();
    if(!gameover)
        update(dt.asSeconds());
    ///////////////////////////////////////////////////
    //Draw Game Here ++
    window.clear(sf::Color::Red);
    draw();

    // Show everything we just drew
    window.display();
}
```

We will also increase the score when the rocket collides with an enemy. So in the `update()` function, when we delete the rocket and enemy after the intersection, we will also update the score:

```
// Check collision between Rocket and Enemies

for (int i = 0; i < rockets.size(); i++) {
    for (int j = 0; j < enemies.size(); j++) {

        Rocket* rocket = rockets[i];
        Enemy* enemy = enemies[j];

        if (checkCollision(rocket->getSprite(), enemy-
                           >getSprite())) {

            score++;

            rockets.erase(rockets.begin() + i);
            enemies.erase(enemies.begin() + j);

            delete(rocket);
            delete(enemy);
```

```
        printf(" rocket intersects enemy \n");
    }

}

}
```

Now when you run the game, start the game by pressing the down arrow. When one of the enemies goes past the left of the screen, the game will end. When you press the down arrow again, the game will restart.

The gameloop is now complete, but we still can't see the score. To do this, let's add some text to the game.

Adding text

These steps will guide you through how to add the text:

1. Create an `sf::Font` called `headingFont` so that we can load the font, and then use this font to display the name of the game. At the top of the screen, where we created all the variables, create the `headingFont` variable, shown as follows:

```
int score = 0;
bool gameover = true;

// Text
sf::Font headingFont;
```

2. In the `init()` function, right after we loaded the `bgSprite`, load the font using the `loadFromFile` function:

```
// Create Sprite and Attach a Texture
bgSprite.setTexture(bgTexture);

// Load font

headingFont.loadFromFile("Assets/fonts/SnackerComic.ttf");
```

Since we will need a font loaded from the system, we have to place the font in the `fonts` directory, found under the `Assets` directory. Make sure you place the font file there. The font we will be using for the heading is the `SnackerComic.ttf` file. I have also included the `arial.ttf` file, which we will use to display the score, so make sure you add that as well.

3. Create the `headingText` variable using the `sf::Text` type, so that we can display the heading of the game. At the top, create `headingText` using the `sf::Text` type:

```
sf::Text headingText;
```

4. In the `init()` function, after we have loaded the `headingFont`, we will add the code to create the heading for the game:

```
// Set Heading Text
headingText.setFont(headingFont);
headingText.setString("Tiny Bazooka");
headingText.setCharacterSize(84);
headingText.setFillColor(sf::Color::Red);
```

We need to set the font for the heading text using the `setFont` function. In `setFont`, pass the `headingFont` variable that we just created.

Tell the `headingText` what needs to be displayed. For that, we use the `setString` function and pass in the `TinyBazooka` string as that is the name of the game we just made. Pretty cool name, huh?

Let's set the size of the font itself. To do this, we will use the `setCharacterSize` function and pass in 84 as the size in pixels so that it is clearly visible. We can then set the color to red using the `setFillColor` function.

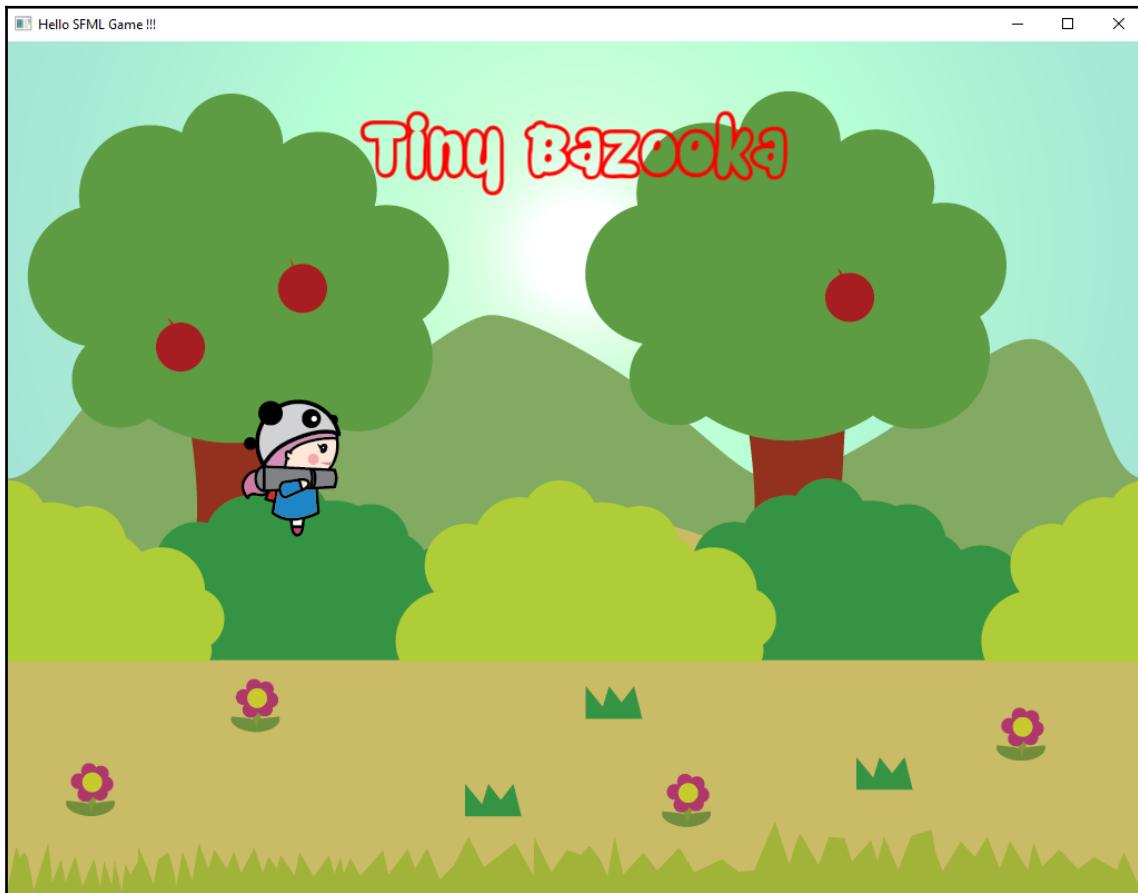
5. We want the heading to be centered to the viewport, so we will get the bounds of the text and set its origin to the center of the viewport in the *x* and *y* direction. Set the position of the text at the center of the *x* direction and 0.10 of the height from the top along the *y* direction:

```
sf::FloatRect headingbounds = headingText.getLocalBounds();
headingText.setOrigin(headingbounds.width/2,
headingbounds.height / 2);
headingText.setPosition(sf::Vector2f(viewSize.x * 0.5f,
viewSize.y * 0.10f));
```

6. To display the text, call `window.draw` and pass `headingText` into it. We also want the text to be drawn when the game is over. To do this, add an `if` statement which checks whether the game is over:

```
if (gameover) {
    window.draw(headingText);
}
```

7. Run the game and you will see the name of the game displayed at the top:



8. We still can't see the score yet, so let's add another Font and Text variable and call them `scoreFont` and `scoreText`. In the `scoreFont` variable, load the `arial.ttf` font, and set the text for the score using the `scoreText` variable:

```
sf::Font headingFont;
sf::Text headingText;

sf::Font scoreFont;
sf::Text scoreText;
```

9. Load the `ScoreFont` and then set the `ScoreText`:

```
scoreFont.loadFromFile("Assets/fonts/arial.ttf");

// Set Score Text

scoreText.setFont(scoreFont);
scoreText.setString("Score: 0");
scoreText.setCharacterSize(45);
scoreText.setFillColor(sf::Color::Red);

sf::FloatRect scorebounds = scoreText.getLocalBounds();
scoreText.setOrigin(scorebounds.width / 2, scorebounds.height /
2);
scoreText.setPosition(sf::Vector2f(viewSize.x * 0.5f, viewSize.y
* 0.10f));
```

Here, we set the `scoreText` string to a score of 0 which we will change once the score increases. Set the size of the font to 45.

Set the score in the same position as `headingText` as it will only be displayed when the game is over. When the game is running, `scoreText` will be displayed.

10. In the `update` function, where we update the scoring, update `scoreText`:

```
score++;
std::string finalScore = "Score: " + std::to_string(score);
scoreText.setString(finalScore);
sf::FloatRect scorebounds = scoreText.getLocalBounds();
scoreText.setOrigin(scorebounds.width / 2, scorebounds.height /
2);
scoreText.setPosition(sf::Vector2f(viewSize.x * 0.5f, viewSize.y
* 0.10f));
```

For convenience, we created a new string, called `finalScore`. To it, we set the "Score: " string and concatenate it with the score, which is an int converted to the string using the `toString` property of the string class. Then we used the `setString` function of `sf::Text` to set the string. Get the new bounds of the text since the text would have changed. Set the origin, center, and position of the updated text .

11. In the draw function, create a new else statement, and if the game is not over, draw the scoreText:

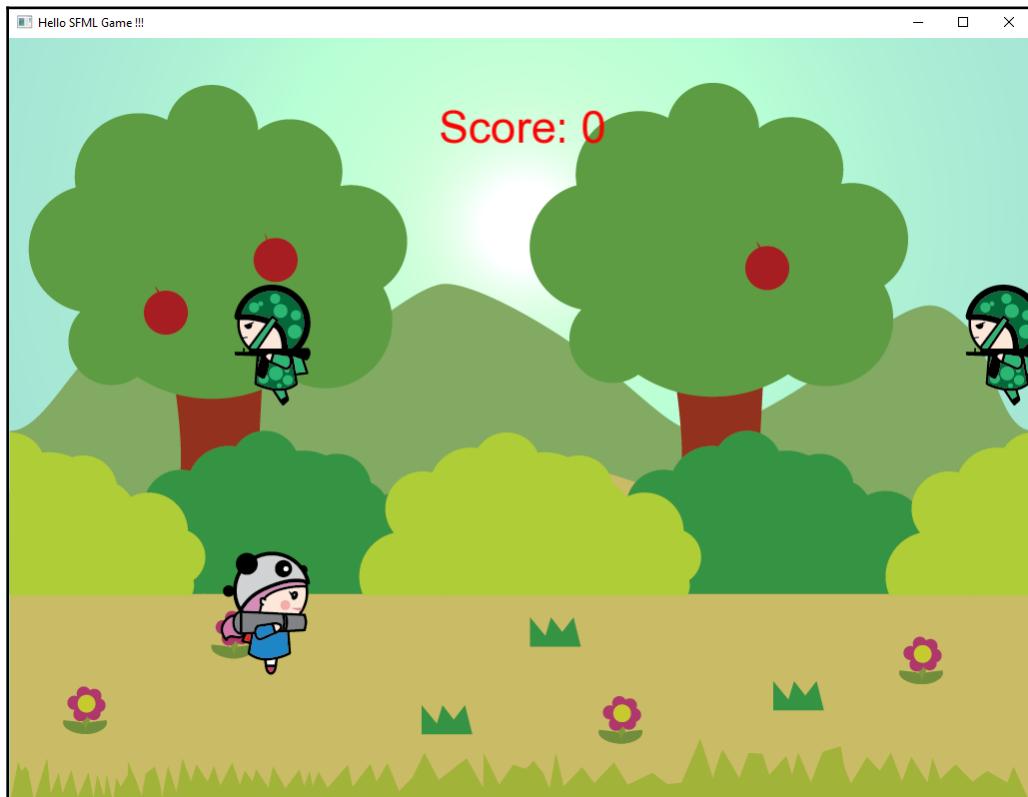
```
if (gameover) {  
    window.draw(headingText);  
} else {  
    window.draw(scoreText);  
}
```

12. Reset the scoreText in the reset() function:

```
prevTime = 0.0;  
scoreText.setString("Score: 0");
```

When you run the game now, the score display will continue to update. The values will reset when you restart the game.

The score system can be seen as follows:



13. Add a tutorial so that the player knows what to do when the game starts. Create a new sf::Text called tutorialText:

```
sf::Text tutorialText;
```

14. Initialize the text after the scoreText in the init() function:

```
// Tutorial Text

tutorialText.setFont(scoreFont);
tutorialText.setString("Press Down Arrow to Fire and Start Game,
Up Arrow to Jump");
tutorialText.setCharacterSize(35);
tutorialText.setFillColor(sf::Color::Red);

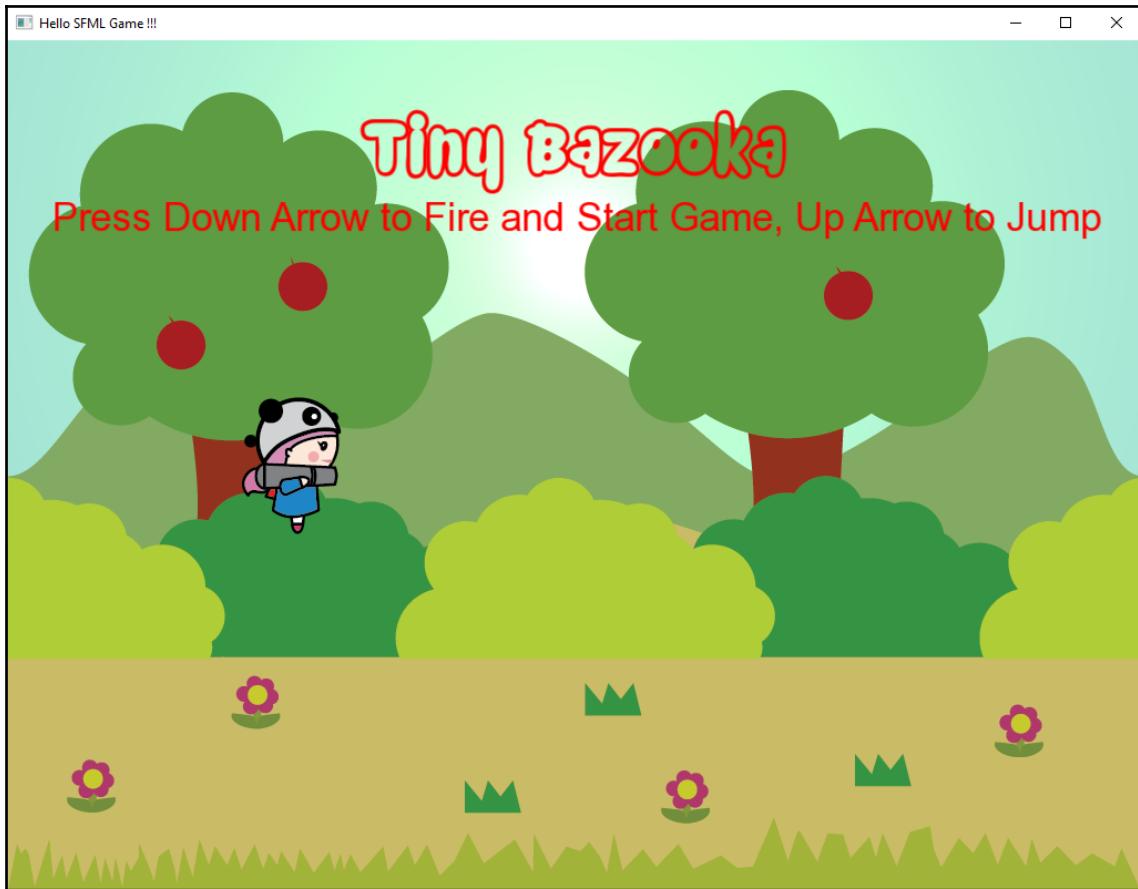
sf::FloatRect tutorialbounds = tutorialText.getLocalBounds();
tutorialText.setOrigin(tutorialbounds.width / 2,
tutorialbounds.height / 2);
tutorialText.setPosition(sf::Vector2f(viewSize.x * 0.5f,
viewSize.y * 0.20f));
```

15. We only want to show the tutorial at the start of the game, along with the heading text. So add the following code in the draw function:

```
if (gameover) {
    window.draw(headingText);
    window.draw(tutorialText);
}
else {
    window.draw(scoreText);
}
```

Now when you start the game, the player will see that by pressing the down arrow the game will start. Now when the game is running press the down arrow button again the player will shoot a rocket, and use the up arrow to make the player jump.

The following screenshot shows the tutorial:



Adding audio

Let's add some audio to the game to make it a little more interesting, and to give some audio feedback to the player to tell them whether the rocket was fired or an enemy was hit.

SFML supports .wav or .ogg files, but it doesn't support .mp3 files. For this project, all the files will be in the .ogg file format as it is good for compression and is also cross-platform compatible. To start, place the audio files in the `Audio` directory in the `Assets` folder of the system. With the audio files in place, we can start playing the audio files now.

Audio files can be of two types:

- The background music, which is of a longer duration and a much higher quality than other files in the game. These files are played using the `sf::Music` class.
- Other sound files, such as sound effects—which are smaller in size and sometimes of a lower quality—are played using the `sf::Sound` class. To play the files, you also need a `sf::SoundBuffer` class, which is used to store the file and play it later.

To add audio to the game, follows these steps:

1. Let's play the background music file, `bgMusic.ogg`. Audio files use the `Audio.hpp` header, which needs to be included at the top of `main.cpp`. This can be done as follows:

```
#include "SFML-2.5.1\include\SFML\Audio.hpp"
```

2. At the top of the `main.cpp` file, create a new instance of `sf::Music` and call it `bgMusic`:

```
sf::Music bgMusic;
```

3. In the `init()` function, add the following lines to open the `bgMusic.ogg` file and play the `bgMusic` file:

```
// Audio  
  
bgMusic.openFromFile("Assets/audio/bgMusic.ogg");  
bgMusic.play();
```

4. Run the game and you will hear the background music playing as soon as the game starts.
5. To add the sound files used for the rockets being fired and enemies being hit, we need two sound buffers to store both of the effects and two sound files to play the sound files. Create two variables of the `sf::SoundBuffer` type, called `fireBuffer` and `hitBuffer`:

```
sf::SoundBuffer fireBuffer;  
sf::SoundBuffer hitBuffer;
```

6. Create two `sf::Sound` variables, called `fireSound` and `hitSound`. Both are initialized by being passed into the respective buffers, as follows:

```
sf::Sound fireSound(fireBuffer);
sf::Sound hitSound(hitBuffer);
```

7. In the `init` function, initialize the buffers first, as follows:

```
bgMusic.openFromFile("Assets/audio/bgMusic.ogg");
bgMusic.play();

hitBuffer.loadFromFile("Assets/audio/hit.ogg");
fireBuffer.loadFromFile("Assets/audio/fire.ogg");
```

8. When the rocket intersects with the enemy, we will play the `hitSound` effect:

```
hitSound.play();
score++;

std::string finalScore = "Score: " +
std::to_string(score);

scoreText.setString(finalScore);

sf::FloatRect scorebounds = scoreText.getLocalBounds();
scoreText.setOrigin(scorebounds.width / 2,
scorebounds.height / 2);
scoreText.setPosition(sf::Vector2f(viewSize.x * 0.5f,
viewSize.y * 0.10f));
```

9. In the `shoot` function, we will play the `fireSound` file, as follows:

```
void shoot() {
    Rocket* rocket = new Rocket();

    rocket->init("Assets/graphics/rocket.png",
hero.getSprite().getPosition(), 400.0f);

    rockets.push_back(rocket);
    fireSound.play();
}
```

When you now play the game, you will hear a sound effect when you shoot the rocket and when the rocket hits the enemy.

Adding player animations

The game has now reached its final stages of development. Let's add some animation to the game to make it really come alive; let's animate the character. To do a 2D animation of sprites, we need a sprite sheet. We have other techniques for adding 2D animations, such as skeletal animation as well. But spritesheet-based 2D animations are faster to make. Hence, we will use sprite sheets to add animations to the main character.

A sprite sheet is an image file, but instead of just one single image, it contains a collection of images in a sequence so that we can loop them to create the animation. Each image in the sequence is called a frame.

For example here is the spritesheet we are going to be using to animate the player:



Looking from left to right, you can see that each frame is slightly different from the last. The main things animated here are the jet pack of the player character, and that the player character is blinking. Each picture will be shown as an animation frame when the game runs, just like in a flip-book animation, where one image is quickly replaced with another image to create the effect of animation.

SFML makes it really easy to animate 2D characters so that we can choose which frame to display in the `update` function. Let's begin animating the character:

1. Add the file to the `Assets/graphics` folder. We need to make some changes to the `Hero.h` and `Hero.cpp` files. Let's look at the changes in the `Hero.h` file first:

```
class Hero{  
  
public:  
    Hero();  
    ~Hero();  
  
    void init(std::string textureName, int frameCount, float
```

```
animDuration, sf::Vector2f position, float mass);  
void update(float dt);  
void jump(float velocity);  
sf::Sprite getSprite();  
  
private:  
  
    int jumpCount = 0;  
    sf::Texture m_texture;  
    sf::Sprite m_sprite;  
    sf::Vector2f m_position;  
    float m_mass;  
    float m_velocity;  
    const float m_gravity = 9.81f;  
    bool m_grounded;  
  
    int m_frameCount;  
    float m_animDuration;  
    float m_elapsedTime;;  
    sf::Vector2i m_spriteSize;  
  
};
```

We need to add two more parameters to the `init` function. The first is an `int` called `frameCount`, which is in the number of frames in the animation. In our case, there are four frames in the hero sprite sheet. The other parameter is a `float`, called `animDuration`, which basically sets how long you want the animation to be played, which will determine the speed of the animation.

We will also create some variables. The first two variables are for storing `frameCount` and `animDuration` locally, so create two variables called `m_frameCount` and `m_animDuration`. We will also create a `float` called `m_elapsedTime`, which will keep track of how long the game has been running, and a `vector2 int` to store the size of each frame, called `m_spriteSize`.

2. Let's move on to the `Hero.cpp` file to see what changes are needed. Here is the modified `init` function:

```
void Hero::init(std::string textureName, int frameCount, float  
animDuration, sf::Vector2f position, float mass) {  
  
    m_position = position;  
    m_mass = mass;  
    m_grounded = false;  
  
    m_frameCount = frameCount;
```

```
m_animDuration = animDuration;
    // Load a Texture
    m_texture.loadFromFile(textureName.c_str());

    m_spriteSize = sf::Vector2i(92, 126);

    // Create Sprite and Attach a Texture
    m_sprite.setTexture(m_texture);
    m_sprite.setTextureRect(sf::IntRect(0, 0, m_spriteSize.x,
    m_spriteSize.y));
    m_sprite.setPosition(m_position);
    m_sprite.setOrigin(m_spriteSize.x / 2, m_spriteSize.y / 2);

}
```

In the init function, we set `m_frameCount` and `m_animationDuration` locally. We need to hardcode the value of the width (as 92) and height (as 126) of each frame. If you are loading your own images, these values will be different.

After calling `setTexture`, we will call the `setTextureRect` function of the `Sprite` class to set which part of the sprite sheet we want to display. Start at the origin of the sprite and get the first frame of the sprite sheet by just passing the width and height of `spriteSize`.

Set the position and origin, which is equal to the center of the `spriteSize` width and height.

3. Let's make some changes to the update function, which is where the major magic happens:

```
void Hero::update(float dt){
    // Animate Sprite
    M_elapsedTime += dt;
    int animFrame = static_cast<int> ((m_elapsedTime /
m_animDuration) * m_frameCount) % m_frameCount;
    m_sprite.setTextureRect(sf::IntRect(animFrame * m_spriteSize.x,
0, m_spriteSize.x, m_spriteSize.y));

    // Update Position
    m_velocity -= m_mass * m_gravity * dt;

    m_position.y -= m_velocity * dt;

    m_sprite.setPosition(m_position);

    if (m_position.y >= 768 * 0.75) {
```

```
    m_position.y = 768 * 0.75;
    m_velocity = 0;
    m_grounded = true;
    jumpCount = 0;
}

}
```

In the update function, increase the elapsed time by the delta time. Then calculate the current animation frame number.

Update the part of the spritesheet to be shown by calling `setTextureRect` and move the origin of the frame to the x-axis depending upon `animFrame` by multiplying it by the width of the frame. The height of the new frame doesn't change so we set it to 0. The width and height of the frame remain the same so we pass in the size of the frame itself.

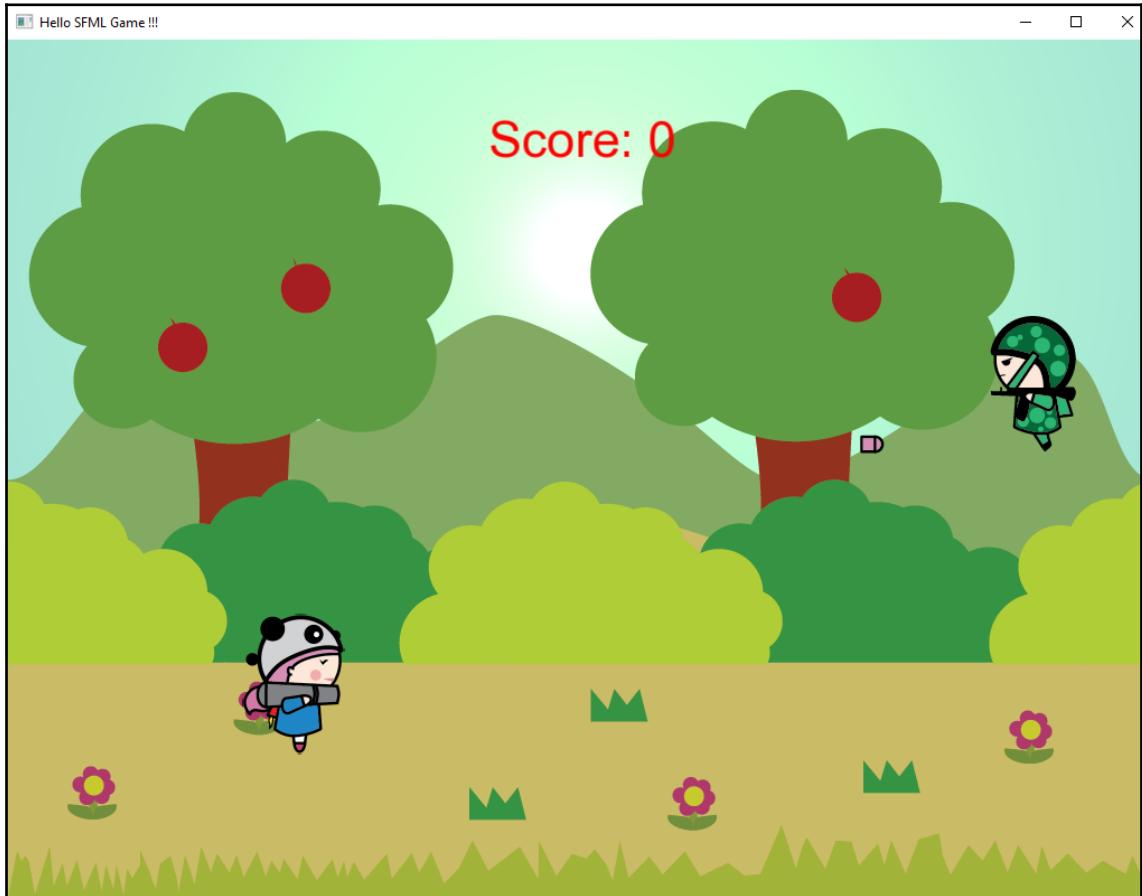
The rest of the functions in `Hero.cpp` remain as they are, and no changes need to be made to them.

4. Go back to `main.cpp` and we'll change how we call `hero.init`. In the `init` function, make the required change:

```
hero.init("Assets/graphics/heroAnim.png", 4, 1.0f,
sf::Vector2f(viewSize.x * 0.25f, viewSize.y * 0.5f), 200);
```

Here, we passed the new `heroAnim.png` file instead of the single-frame `.png` we previously loaded. Set the number of frames to 4 and set `animDuration` to `1.0f`.

5. Run the game and you can see the player character getting animated and blinking every four frames:



Summary

In this chapter, we completed the gameloop and added the gameover condition. We added scoring so that the player knows how many points they have scored. We also added text so that the name of the game is displayed, a tutorial can be seen to tell users how to play the game, and to show the score. We learned how to place these elements in the centre of the viewport. Finally, added sound effects and animations to make our game come to life.

In the next chapter, we will see how to render 3D and 2D objects in a scene. Instead of using a framework, we will start to create a basic engine and begin our journey into understanding the basics of rendering.

3

Section 3: Modern OpenGL 3D Game Development

Using the game-development concepts learned in Section 2, in this section we will create a 3D physics puzzle game using modern OpenGL and the bullet physics engine. We will learn about the graphics pipeline and the creation of 3D objects using vertex and index buffers. Then we will add them to the scene using vertex and fragment shaders, add textures, and use text rendering, including the bullet physics library, lighting models, post-processing effects, and 3D particle-system generation. The following chapters are covered in this section:

Chapter 6, Getting Started with OpenGL

Chapter 7, Building on the Game Objects

Chapter 8, Enhancing Your Game with Collision, Loop, and Lighting

6

Getting Started with OpenGL

In the previous three chapters, we rendered 2D objects called sprites in our tiny Bazooka game using the **Simple and Fast Media Library (SFML)**. At the core of SFML is OpenGL; this is used to render anything on screen, including 2D objects.

SFML does a great job of putting everything in a nice little package, and this enables us to get a 3D game going very quickly. However, in order to understand how a graphics library actually works, we need to learn how OpenGL works by delving deeper into how to use it to render anything on the screen.

In this chapter, we will discover how to use a graphics library, such as OpenGL, in order to render 3D objects in any scene; the following topics will be covered:

- What is OpenGL?
- Creating our first OpenGL project
- Creating a window and ClearScreen
- Creating a Mesh class
- Camera classes
- Shaderloader classes
- Light Renderer classes
- Drawing the object

What is OpenGL?

So, what is this OpenGL that we speak of? Well, OpenGL is a collection of graphics APIs; essentially, this is a collection of code that allows the user to gain access to the features of your graphics hardware. The current version of OpenGL is 4.6, but any graphics hardware that is capable of running OpenGL 4.5 can run 4.6 as well.

OpenGL is entirely hardware and operating system independent, so it doesn't matter if you have NVIDIA or AMD GPU; it will work the same on both hardware. The way in which OpenGL's features work is defined by a specification that is used by the graphics hardware manufacturers while developing the drivers for their hardware. This is why, sometimes, we have to update the graphics hardware drivers if something is not looking right or if the game is not performing well.

Furthermore, OpenGL runs the same whether you are running a Windows or a Linux machine. It is, however, deprecated on macOS Mojave, but if you are running a macOS earlier than Mojave, then it is still compatible.

OpenGL is only responsible for rendering objects in the scene. Unlike SFML, which allows you to create a window and then gain access to the keyboard and mouse input, we will need to add a separate library that will handle all of this for us.

So, let's start preparing our project for rendering a 3D OpenGL object in the scene.

Creating our first OpenGL project

Now that we have gained an understanding of what OpenGL is, let's examine how to create our first OpenGL project, as follows:

1. Create a new empty C++ project in Visual Studio and call it **OpenGLProject**.
2. Then, download GLEW; GLEW is a C/C++ extension loader library. OpenGL supports extensions that various GPU vendors can use to write and extend the functionality of OpenGL. This library will determine what extensions are supported on the platform.
3. Go to <http://glew.sourceforge.net/> and download the Windows 32-bit and 64-bit **Binaries**:

Downloads

GLEW is distributed as source and precompiled binaries.
The latest release is [2.1.0\[07-31-17\]](#):

[Source ZIP | TGZ](#)

[Binaries Windows 32-bit and 64-bit](#)

4. Next, we need to download GLFW; GLFW is a platform-independent API that is used for creating a window, reading inputs, and handling events. Go to <https://www.glfw.org/download.html> and download the 64-bit Windows binary as, in this book, we will primarily look at implementing it on the Windows platform:

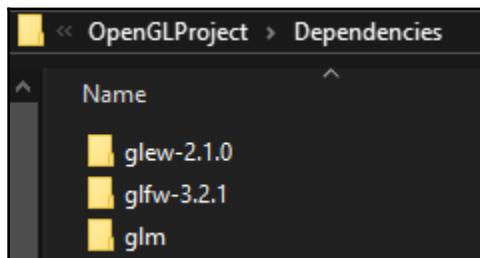
Windows pre-compiled binaries

These packages contain complete GLFW header file, [documentation](#) and release mode DLL and static library binaries for Visual C++ 2010 (32-bit only), Visual C++ 2012, Visual C++ 2013, Visual C++ 2015, MinGW (32-bit only) and MinGW-w64.

[32-bit Windows binaries](#)

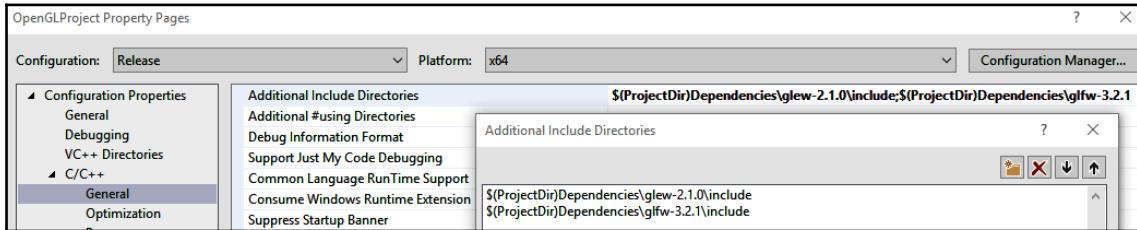
[64-bit Windows binaries](#)

5. Next, we need to download `glm`, which is used to do all the math for our graphics calculations. Go to <https://glm.g-truc.net/0.9.9/index.html> and download GLM from the site.
6. Now that we have downloaded all the required libraries and headers, we can start adding them into our project.
7. In the root directory (where the Visual Studio project file is stored) of the project, create a new directory called `Dependencies`.
8. In it, extract `glew`, `glfw`, and `glm`; the `Dependencies` directory should now look as follows:

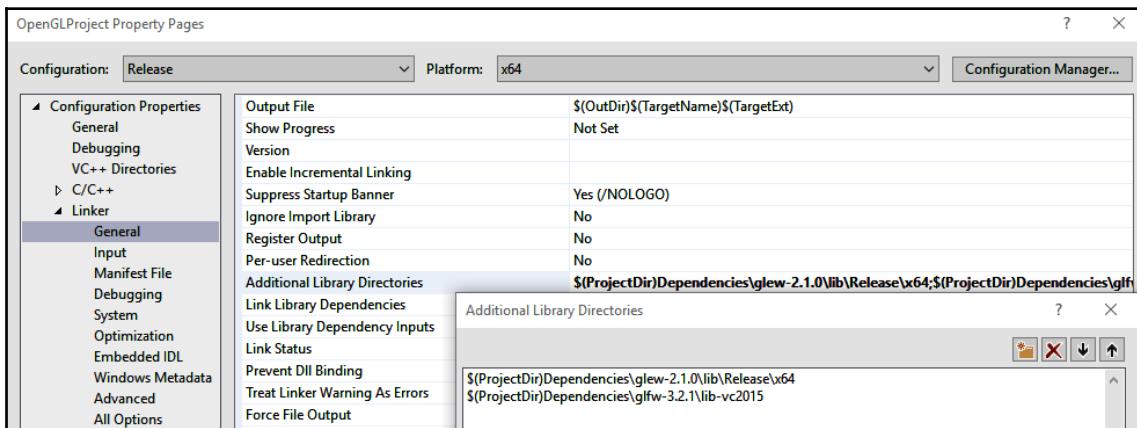


9. Open the Visual Studio project. We will now set the location of the headers and library files. To do this, open the project properties of `OpenGLProject` and set **Configuration** to **Release** and **Platform** to **x64**.

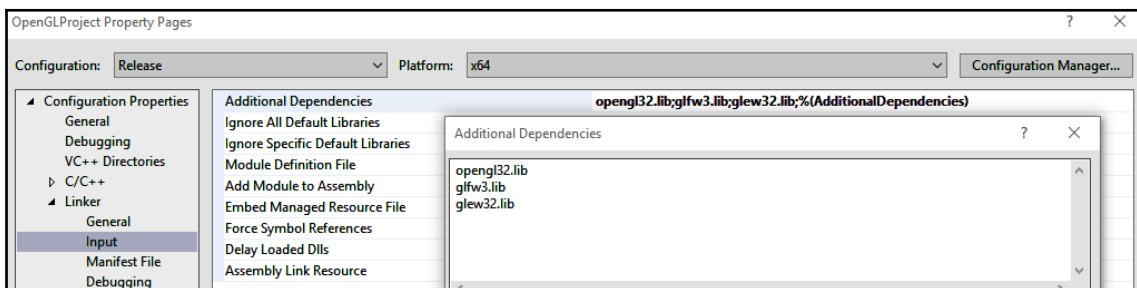
10. Under C/C++ | General, select Additional Include Directories, and then select the following directories of GLEW and GLFW:



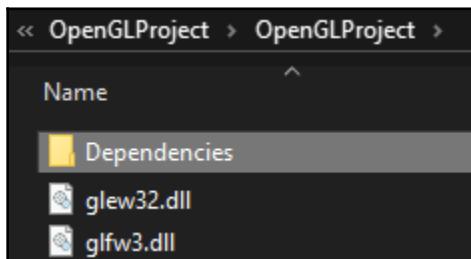
11. Next, under Linker | General, select Additional Library Directories, and then select the location of the .lib files in the glew and glfw directories, as follows:



12. Next, we have to go to Linker | Input and specify which .lib files we are using.
 13. Under Linker | Input, select Additional Dependencies and then add **opengl32.lib**, **glfw3.lib**, and **glew32.lib**, as follows:



14. Although we didn't specifically download `opengl32.lib`, it is included when you update the driver of the graphics hardware. Therefore, make sure that you are running the most recent drivers for your GPU and, if not, then you can download it from the manufacturer's website.
15. Finally, we have to add the `glew32.dll` and `glfw3.dll` files to the root directory of the project. `glew32.dll` is inside `glew-2.1.0/bin/Release/64` while `glfw3.dll` is inside `glfw-3.2.1/lib-vc2015`.
16. The root directory of the project file should now look like the following screenshot:



17. With all that out of the way, we can finally start working on the project.

Creating a window and ClearScreen

Let's now explore how we can work with this OpenGL project that we created:

1. The first thing we have to do is create a window so that we can start rendering the game objects to the screen.
2. Create a new .cpp file; Visual Studio will automatically call this `source.cpp`, so keep it as it is.
3. At the top of the file, include the `glew` and `glfw` header. Make sure that you include `glew.h` first, as it contains the correct OpenGL header files to be included:

```
#include <GL/glew.h>
#include <GLFW/glfw3.h>
```

Then create a main function and add the following to it.

```
int main(int argc, char **argv)
{
```

```
        glfwInit();  
  
        GLFWwindow* window = glfwCreateWindow(800,  
600,  
"Hello OpenGL ",  
NULL,  
NULL);  
        return 0;  
    }
```

4. The first thing we do is initialize `glfw` by calling `glfwInit()`.
5. Once initialized, we can create the window that our game scene will be rendered to. In order to create a window, we need to create a new instance of `GLFWwindow` called `window` and call `glfwCreateWindow`. This takes five parameters, including the width and height of the window, along with the name of the window. The final two parameters—`monitor` and `share`—are set to `NULL`. The `monitor` parameter takes a specific monitor on which the window will be created. If it is set to `null`, then the default monitor is chosen. The `share` parameter will let us share the window resource with. Here, we set it to `NULL` as we don't want to share the window resources.
6. Now run the project; you will see that a window briefly appears before the application closes.
7. Well, that's not very fun; so let's now add the rest of the code so that we can see something rendered on the viewport.
8. The first thing we need to do is initialize OpenGL Context. OpenGL Context is a collection of all the current states of OpenGL. We will discuss the different states in future sections.
9. So, in order to do this, we call `glfwMakeCurrentContext` and pass in the window that we just created:

```
glfwMakeCurrentContext(window);
```

10. We can now initialize GLEW by calling `glewInit()`.

11. Next, we will add the following code between `glewInit()` and `return 0` in the main function:

```
while (!glfwWindowShouldClose(window)) {
    // render our scene

    glfwSwapBuffers(window);
    glfwPollEvents();
}

glfwTerminate();
```

12. We create a while loop, call `glfwWindowShouldClose`, and then pass it in the current window. So, while the window is open, the `glfwSwapBuffers(window);` and `glfwPollEvents();` commands will be executed.
13. In the while loop, we will render our scene. Then, we will swap display buffers. The display buffer is where the current frame is rendered and stored. While the current frame is being shown, the next frame is actually being rendered in the background, which we don't get to see. When the next frame is ready, the current frame is swapped with the new frame. This swapping of frames is done by `glfwSwapBuffer` and is managed by OpenGL.
14. After we swap the display buffer, we then check for any events that were triggered, such as the closing of the window in `glfwPollEvents()`. Once the window is closed, `glfw` is terminated at the end.
15. If you run the project now, you will see a black window; while it doesn't vanish, it is still not very impressive. We can use OpenGL to clear the viewport with the color of our choice, so let's do that next.
16. Create a new function called `void renderScene()`. Whatever we render to the scene from now on, we will also add in this function. So, add a new prototype for `void renderScene()` at the top of the `source.cpp` file.
17. In the `renderScene` function, add the following lines of code:

```
void renderScene() {

    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glClearColor(1.0, 0.0, 0.0, 1.0); //clear yellow

    // Draw game objects here
}
```

In the first function, we call `glClear()`. All OpenGL functions start with the `gl` prefix; the `glClear` function clears the buffer. In this case, we are asking OpenGL to clear the color buffer and the depth buffer. The color buffer is where all the color information is stored for the scene. The depth buffer stores whichever pixel is in front; this means that if a pixel is behind another pixel, then that pixel will not be stored. This is especially important for 3D scenes, where some objects can be behind other objects and get occluded by the objects that are in front of it. We only require the pixel information regarding the objects that are in front as we will get to see only those objects and not the objects that are behind them.

Next, we call the `glClearColor` function and pass in an RGBA value; in this case, it is red. The `glClearColor` function clears the color buffer with the specific color in every frame. The buffers need to be cleared in every frame, otherwise, the previous frame will be overwritten with the image in the current frame. Imagine this to be like clearing the blackboard before drawing anything on it in every frame.

The depth buffer is also cleared after every frame using a default white color. This means that we don't have to clear it manually as it will be done by default.

18. Now call `renderScene` before we swap the buffer and run the project again. You should see a nice yellow viewport, as follows:



Before drawing the objects, we first have to create some additional classes that will help us to define the shape that we want to draw. We also need to create a camera class in order to set up a virtual camera through which we can view the scene. Furthermore, we need to write a basic vertex, a shader fragment, and a `Shaderloader` class, which will create a shader program that we can use to render our shape.

So, let's first create the `Mesh` class in which we will define the different shapes that we want to draw.

Creating a Mesh class

The following steps will guide you on how to create a Mesh class:

1. Create new .h and .cpp files called `Mesh.h` and `Mesh.cpp`, respectively. These will be used to create a new `Mesh` class; in the `Mesh.h` file, add the following code:

```
#include <vector>
#include "Dependencies/glm/glm/glm.hpp"

enum MeshType {

    kTriangle = 0,
    kQuad = 1,
    kCube = 2,
    kSphere = 3

};

struct Vertex {

    glm::vec3 pos;
    glm::vec3 normal;
    glm::vec3 color;
    glm::vec2 texCoords;

};

class Mesh {

public:
    static void setTriData(std::vector<Vertex>& vertices,
                           std::vector<uint32_t>& indices);
    static void setQuadData(std::vector<Vertex>& vertices,
                           std::vector<uint32_t>& indices);
    static void setCubeData(std::vector<Vertex>& vertices,
                           std::vector<uint32_t>& indices);
    static void setSphereData(std::vector<Vertex>& vertices,
                           std::vector<uint32_t>& indices);

};

};
```

2. At the top of the `Mesh.h` file, we include a vector so that we can store points in a vector and include `glm.hpp`. This will help us to define points in space using the `vec3` variable.
3. We create a new `enum` type called `MeshType` and create four types: `Mesh`, `Triangle`, `Quad`, `Cube`, and `Sphere`. This is done in order to specify the kind of mesh we are using so that the data will be populated accordingly.
4. We then create a new `struct` type called `Vertex`, which has a `vec3s` property called `pos`, `Color`, `Normal`, and a `vec2` property called `textCoords`.

Each vertex has certain properties, such as `Position`, `Color`, `Normal`, and `Texture Coordinate`. `Position` and `Color` stores the position and color information for each vertex. `Normal` specifies which direction the normal attribute is pointing to while `Texture Coordinate` specifies how a texture needs to be laid out. We will cover the normal and texture coordinate attributes when we cover lighting and how to apply texture to our object.

5. The `Mesh` class is created next. This has four functions for setting the vertex and index data per vertex.
6. In the `Mesh.cpp` file, we include the `Mesh.h` file and then set the data for the four shapes. Here is an example of how `setTriData` sets the values for the vertices and indices:

```
#include "Mesh.h"

void Mesh::setTriData(std::vector<Vertex>& vertices,
                      std::vector<uint32_t>& indices) {

    std::vector<Vertex> _vertices = {

        { { 0.0f, -1.0f, 0.0f }, // Position
          { 0.0f, 0.0f, 1.0 }, // Normal
          { 1.0f, 0.0f, 0.0 }, // Color
          { 0.0, 1.0 } }, // Texture Coordinate
        { 0.0 }, // 0

        { { 1.0f, 1.0f, 0.0f }, { 0.0f, 0.0f, 1.0 }, { 0.0f, 1.0f,
          0.0 }, { 0.0, 0.0 } }, // 1

        { { -1.0f, 1.0f, 0.0f }, { 0.0f, 0.0f, 1.0 }, { 0.0f, 0.0f,
          1.0 }, { 1.0, 0.0 } }, // 2
    };

    std::vector<uint32_t> _indices = {
```

```
    0, 1, 2,  
};  
  
vertices.clear(); indices.clear();  
  
vertices = _vertices;  
indices = _indices;  
}
```

7. For each of the three vertices of the triangle, we set the position, normal, color, and texture coordinate information in the vertices vector.

Next, we set the indices in the indices vector. For definitions of the other functions, you can refer to the project along with the book. Then, we set the `_vertices` and `_indices` vectors to the reference vertices and indices.

Creating a Camera class

The following steps will help you to create a Camera class:

1. We will create the camera class next. Create a `Camera.h` and `Camera.cpp` file. In the `Camera.h` file, include the following code:

```
#include <GL/glew.h>  
  
#include "Dependencies/glm/glm/glm.hpp"  
#include "Dependencies/glm/glm/gtc/matrix_transform.hpp"
```

2. Then create the camera class itself, as follows:

```
class Camera  
{  
public:  
    Camera(GLfloat FOV, GLfloat width, GLfloat height, GLfloat  
nearPlane, GLfloat farPlane, glm::vec3 camPos);  
    ~Camera();  
  
    glm::mat4 getViewMatrix();  
    glm::mat4 getProjectionMatrix();  
    glm::vec3 getCameraPosition();  
  
private:  
  
    glm::mat4 viewMatrix;  
    glm::mat4 projectionMatrix;
```

```
glm::vec3 cameraPos;
```

```
};
```

3. In the constructor and the public region of the camera class, we get the **field of view (FOV)**, width and height of the viewport, the distance to `nearPlane`, the distance to `farPlane`, and the position that we want to set the camera at.
4. We also add three getters to get the view matrix, projection matrix, and the camera position.
5. In the private section, we create three variables, two 4×4 matrices for setting the view and projection matrices, and a `vec3` property to specify the camera position.
6. In the `Camera.cpp` file, we include the `Camera.h` file at the top and create the camera constructor, as follows:

```
#include "Camera.h"

Camera::Camera(GLfloat FOV, GLfloat width, GLfloat height, GLfloat
nearPlane, GLfloat farPlane, glm::vec3 camPos){

    cameraPos = camPos;
    glm::vec3 cameraFront = glm::vec3(0.0f, 0.0f, 0.0f);
    glm::vec3 cameraUp = glm::vec3(0.0f, 1.0f, 0.0f);

    viewMatrix = glm::lookAt(cameraPos, cameraFront, cameraUp);
    projectionMatrix = glm::perspective(FOV, width /height,
    nearPlane, farPlane);
}
```

7. In the constructor, we set the camera position to the local variable and set up two `vec3` properties called `cameraFront` and `cameraUp`. Our camera is going to be a stationary camera that will always be looking toward the center of the world coordinates, and the up vector will always be pointing toward the positive `y` axis.
8. For creating `viewMatrix`, we call the `glm::lookAt` function and pass in the `cameraPos`, `cameraFront`, and `cameraUp` vectors.
9. We create the projection matrix by setting the `FOV` value of the `FOV`; this is an aspect ratio that is given by the `width` value over the `height`, `nearPlane`, and `farPlane` values.

- With the view and projection matrix set, we can now create the getter functions as follows:

```
glm::mat4 Camera::getViewMatrix() {  
  
    return viewMatrix;  
}  
glm::mat4 Camera::getProjectionMatrix() {  
  
    return projectionMatrix;  
}  
  
glm::vec3 Camera::getCameraPosition() {  
  
    return cameraPos;  
}
```

Next, we create the `shaderLoader` class, which will let us create the shader program.

The ShaderLoader class

The following steps will show you how to implement the `ShaderLoader` class in an OpenGL project:

- In the `ShaderLoader` class, we create a public function called `createProgram` that takes a vertex and fragment shader file.
- We also create two private functions called `readShader`, which return a string and a `createShader` object, which returns an unsigned GL int:

```
#include <GL/glew.h>  
  
class ShaderLoader {  
  
public:  
  
    GLuint CreateProgram(const char* vertexShaderFilename,  
                         const char*  
                         fragmentShaderFilename);  
  
private:  
  
    std::string readShader(const char *filename);  
    GLuint createShader(GLenum shaderType, std::string source,  
    const char* shaderName);  
};
```

3. In the `ShaderLoader.cpp` file, we include our `ShaderLoader.h` header file, the `iostream` system header file, and the `fstream` vector, as follows:

```
#include "ShaderLoader.h"

#include<iostream>
#include<fstream>
#include<vector>
```

`iostream` is used when you want to print something to the console; `fstream` is used for reading a file as we will be passing in vertex and shader files for it to read, as well as vectors for storing character strings.

4. First, we create the `readerShader` function; this will be used to read the shader file that we passed in, as follows:

```
std::string ShaderLoader::readShader(const char *filename)
{
    std::string shaderCode;
    std::ifstream file(filename, std::ios::in);

    if (!file.good()){
        std::cout << "Can't read file " << filename << std::endl;
        std::terminate();
    }

    file.seekg(0, std::ios::end);
    shaderCode.resize((unsigned int)file.tellg());
    file.seekg(0, std::ios::beg);
    file.read(&shaderCode[0], shaderCode.size());
    file.close();
    return shaderCode;
}
```

The contents of the shader file are then stored in a string and returned.

5. Next, we create the `createShader` function, which will actually compile the shader, as follows:

```
GLuint ShaderLoader::createShader(GLenum shaderType,
std::string source, const char* shaderName)
{

    int compile_result = 0;

    GLuint shader = glCreateShader(shaderType);
    const char *shader_code_ptr = source.c_str();
```

```
        const int shader_code_size = source.size();

        glShaderSource(shader, 1, &shader_code_ptr,
&shader_code_size);
        glCompileShader(shader);
        glGetShaderiv(shader, GL_COMPILE_STATUS,
&compile_result);

        //check for errors
        if (compile_result == GL_FALSE)
        {

            int info_log_length = 0;
            glGetShaderiv(shader, GL_INFO_LOG_LENGTH,
&info_log_length);
            std::vector<char> shader_log(info_log_length);
            glGetShaderInfoLog(shader, info_log_length, NULL,
&shader_log[0]);
            std::cout << "ERROR compiling shader: " <<
shaderName << std::endl <<&shader_log[0] << std::endl;
            return 0;
        }
        return shader;
    }
```

6. The `CreateShader` function takes the following three parameters:

- The first parameter is the `enum` parameter, called `shaderType`, which specifies the type of shader being sent to compile. In this case, it could be a vertex shader or a fragment shader.
 - The second parameter is the string that contains the shader code.
 - The final parameter is a string with the `shader type`, which will be used to specify whether there is a problem compiling the said `shader type`.
7. In the `CreateShader` function, we first call `glCreateShader` in order to specify the type of shader that is being created; then, `glCompileShader` is called to compile the shader. We then get the compile result of the shader.
8. If there is a problem with compiling the shader, then we send out a message stating that there is an error compiling the shader alongside `shaderLog`, which will detail the compilation error. If there are no errors during the compiling of the shader, then it is returned.

9. The final function is the `createProgram` function, which takes the vertex and fragment shaders:

```
GLuint ShaderLoader::createProgram (const char*
vertexShaderFilename, const char* fragmentShaderFilename) {

    std::string vertex_shader_code = readShader
(vertexShaderFilename);

    std::string fragment_shader_code = readShader
(fragmentShaderFilename);

    GLuint vertex_shader = createShader (GL_VERTEX_SHADER,
vertex_shader_code, "vertex shader");

    GLuint fragment_shader = createShader (GL_FRAGMENT_SHADER,
fragment_shader_code, "fragment shader");

    int link_result = 0;
    //create the program handle, attach the shaders and link it
    GLuint program = glCreateProgram();
    glAttachShader(program, vertex_shader);
    glAttachShader(program, fragment_shader);

    glLinkProgram(program);
    glGetProgramiv(program, GL_LINK_STATUS, &link_result);
    //check for link errors
    if (link_result == GL_FALSE) {

        int info_log_length = 0;
        glGetProgramiv(program, GL_INFO_LOG_LENGTH, &info_log_length);
        std::vector<char> program_log(info_log_length);

        glGetProgramInfoLog(program, info_log_length, NULL,
&program_log[0]);
        std::cout << "Shader Loader : LINK ERROR" << std::endl
<<&program_log[0] << std::endl;
        return 0;
    }
    return program;
}
```

10. This function takes the vertex and fragment shader files, reads them, and then compiles both files.
11. Then, a new `shaderProgram` function is created by calling `glCreateProgram()` and assigned to `program`.

12. We then have to attach both shaders to the program by calling `glAttachShader` and passing the program and the shader.
13. Finally, we link the program by calling `glLinkProgram`. We then pass in the program and check for any linking errors.
14. If there are any linking errors, then we send out an error message to the console along with a program log that will detail the linking error. If not, then the program is returned.

The Light Renderer class

It's time to finally draw our first object; to do so, perform the following steps:

1. We will draw a basic light source that will appear above the current scene so that we can visualize the location of the light source in the scene.
2. We will later use this location of the light source to calculate the lighting on our object.
3. A flat shaded object doesn't have any lighting calculation done on it.
4. We create a `LightRenderer.h` file and a `.cpp` file, and then we create the `LightRenderer` class.
5. At the top of the `LightRenderer.h` file, include the following headers:

```
#include <GL/glew.h>

#include "Dependencies/glm/glm/glm.hpp"
#include "Dependencies/glm/glm/gtc/type_ptr.hpp"

#include "Mesh.h"
#include "ShaderLoader.h";
#include "Camera.h"
```

6. We will need `glew.h` for calling the OpenGL commands. We will then need the `glm` headers for defining `vec3` and the matrices.
7. We will also need `Mesh.h` for defining the shape of the light in the light source. You can use `ShaderLoader` class for loading the shaders in order to render the object, and `Camera.h` for getting the camera location, view, and projection matrices to the scene.

8. We will create the `LightRenderer` class next; create this class as follows:

```
class LightRenderer
{
    };

};
```

In the class, we will add the following public section first:

```
public:
    LightRenderer(MeshType meshType, Camera* camera);
    ~LightRenderer();

    void draw();

    void setPosition(glm::vec3 _position);
    void setColor(glm::vec3 _color);
    void setProgram(GLuint program);

    glm::vec3 getPosition();
    glm::vec3 getColor();
```

9. In the public section, we create the constructor in which we pass `MeshType`, which will be used to set the shape of the object that we want to render. Then, we have the destructor. There is a function called `draw`, which will be used to draw the mesh. Then, we have a couple of setters for setting the position, color, and shader program for the object.
10. After defining the public section, we set the private section as follows:

```
private:
    Camera* camera;

    std::vector<Vertex> vertices;
    std::vector<GLuint> indices;
    glm::vec3 position, color;
    GLuint vbo, ebo, vao, program;
```

11. In the private section, we have a private variable in order to store the camera locally. We create vectors to store the vertex and index data; we also create local variables to store the position and color information. Then, we have `GLuint` that will store `vbo`, `ebo`, `vao`, and the `program` variable.

The program variable will have the shader program that we want to use to draw the object. Then, we have vbo, which stands for Vertex Buffer Object; ebo, which stands for Element Buffer Object; and vao, which stands for Vertex Array Object. Let's examine what each of these buffer objects are and what they do:

- **Vertex Buffer Object (VBO):** This is the geometrical information; it includes attributes such as position, color, normal, and texture coordinates – these are stored on a per vertex basis on the GPU.
- **Element Buffer Object (EBO):** This is used to store the index of each vertex and will be used while drawing the mesh.
- **Vertex Array Object (VAO):** This is a helper container object that stores all the VBOs and attributes. This is because you might have more than one VBO per object, and it will be tedious to bind the VBOs all over again when you render each frame.

Buffers are used to store information in the GPU memory for fast and efficient access to the data. Modern GPUs have a memory bandwidth of approximately 600 GB/s, which is enormous compared to the current high-end CPUs that only have approximately 12 GB/s.

Buffer objects are used to store, retrieve, and move data. It is very easy to generate a buffer object in OpenGL. You can easily generate one by calling `glGenBuffers()`.

That is all for `LightRender.h`; let's now move on to `LightRenderer.cpp`, as follows:

1. At the top of `LightRenderer.cpp`, include `LightRenderer.h`; then, let's add the constructor as follows:

```
LightRenderer::LightRenderer(MeshType meshType, Camera* camera) {  
}
```

2. In the `LightRenderer` constructor, we will start adding code. First, we initialize the local camera as follows:

```
this->camera = camera;
```

3. Next, we will set the shape of the object that we want to draw depending upon the `MeshType` type. For this, we create a `switch` statement and call the appropriate `setData` function depending upon the type, as follows:

```
switch (modelType) {  
  
    case kTriangle: Mesh::setTriData(vertices, indices);  
    break;
```

```
        case kQuad: Mesh::setQuadData(vertices, indices); break;
        case kCube: Mesh::setCubeData(vertices, indices); break;
        case kSphere: Mesh::setSphereData(vertices, indices);
    break;
}
```

4. Next, we will generate and bind the vao buffer object, as follows:

```
glGenVertexArrays(1, &vao);
 glBindVertexArray(vao);
```

The `glGenVertexArrays` function takes two parameters; the first parameter is the number of vertex array object names that we want to generate. In this case, we just want to create one, so it is specified as such. The second parameter takes in an array in which the vertex array names are stored, so we pass in the `vao` buffer object.

5. The `glBindVertexArray` function is called next, and `vao` is passed into it in order to bind the `vao` buffer object. The `vao` buffer object will be bound for the duration of the application. A buffer is only an object managing a certain piece of memory; buffers can be of different types and, therefore, they need to be bound to a specific buffer target to give meaning to the buffer.
6. Once the `vao` buffer object is bound, we can generate the vertex buffer object to store the vertex attributes.
7. To generate the vertex buffer object, we call `glGenBuffers()`; this also takes two parameters. The first parameter is the number of buffers that we want to generate and the second is the array of `vbos`. Since, in this case, we have one `vbo` buffer object, we will just pass in `1` for the first parameter and pass in the `vbo` as the second parameter:

```
glGenBuffers(1, &vbo);
```

8. Next, we have to specify the buffer type. This is done by using the `glBindBuffer()` function; this takes two parameters again. The first is the buffer type and, in this case, it is of type `GL_ARRAY_BUFFER`, and the second parameter is the name of the buffer object, which is `vbo`; next, add the following line of code:

```
glBindBuffer(GL_ARRAY_BUFFER, vbo);
```

9. In the next step, we actually pass in the data that we want to store in the buffer. This is done by calling `glBufferData`; the `glBufferData` function takes four parameters. The first parameter is the buffer type, which, in this case, is `GL_ARRAY_BUFFER`; the second parameter is the size in bytes of the buffer data to store; the third parameter is the pointer to the data, which will be copied; and the fourth parameter is the expected usage of the data being stored. In our case, we will just modify the data once and use it many times, so it will be `GL_STATIC_DRAW`.

10. Now add the function to store the data, as follows:

```
glBufferData(GL_ARRAY_BUFFER,  
            sizeof(Vertex) * vertices.size(),  
            &vertices[0],  
            GL_STATIC_DRAW);
```

We now have to set the vertex attributes that we are going to use. While creating the `vertex` struct, we have attributes such as position, color, normal, and texture coordinates; however, we might not need all the attributes all of the time. Therefore, we specify only the ones that we need. For our current purpose, since we are not using any lighting calculation or applying any texture to the object, we don't need to specify this – we will just need position and color attribute for now. However, these attributes need to be enabled first.

11. So, we call `glEnableVertexAttribArray` and pass in the index that we want to enable. The position will be in the 0th index, so we will set the value as follows:

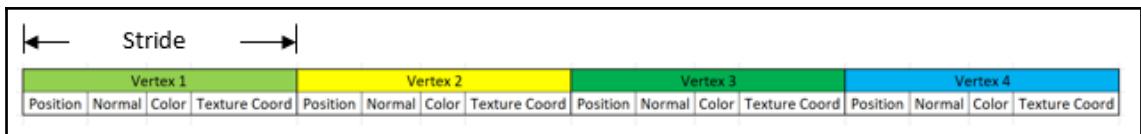
```
glEnableVertexAttribArray(0);
```

12. Next, we call `glVertexAttribPointer` to set the attribute that we want to use. So, the first attribute will be positioned at the 0th index; this takes 6 parameters, as follows:

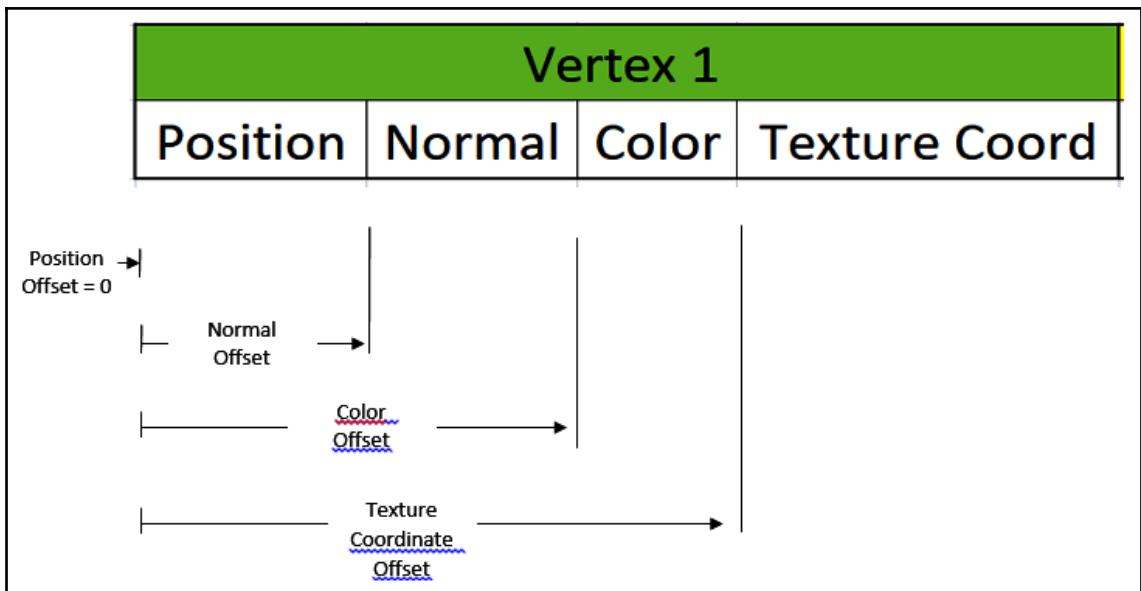
- The first parameter is the index of the vertex attribute, which, in this case, is 0.
- The second parameter is the size of the attribute; so, essentially, this is the number of components of the vertex attribute. In this case, it is position of the *x*, *y*, and *z* components, so it is specified as 3.
- The third parameter is the type of variable of the components; since they are specified in `GLfloat`, we specify `GL_FLOAT`.
- The fourth parameter is a Boolean that specifies whether the values should be normalized or whether they should be converted as fixed point values. Since we don't want the values to be normalized, we specify `GL_FALSE`.

- The fifth parameter is called the stride, which is the offset of consecutive vertex attributes.

Imagine the vertices being laid out in the memory as follows. The stride refers to the blocks of memory that you will have to go through to get to the next set of vertex attributes; this is the size of the vertex struct itself:



- The sixth parameter is the offset of the first component of the vertex attribute within the vertex struct. The attribute that we are looking at here is the position attribute, which is at the start of the vertex struct, so we will pass 0:



- Set the `glVertexAttribPointer` pointer as follows:

```
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, sizeof(Vertex),
(GLvoid*)0);
```

14. Let's create one more attribute pointer to color the object. So, as before, we enable the attribute and set the attrib pointer, as follows:

```
glEnableVertexAttribArray(1);  
glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, sizeof(Vertex),  
                      (void*) (offsetof(Vertex, Vertex::color)));
```

15. Since the next attribute index is 1, we enable the attribute array using 1. While setting the attribute pointer, the first parameter is 1, as it is the first index. Color has three components – r , g , and b – so the next parameter is 3. Colors are defined as floats and so we specify `GL_FLOAT` for the thirs parameter.
16. For the fourth parameter, since we don't want it to be normalized, we set the parameter to `GL_FALSE`. The fifth parameter is the stride and it is still equal to the size of the vertex struct. Finally, for the offset we use the `offsetof` function to set the offset of `vertex::color` in the vertex struct.

Next, we have to set the element buffer object. This is done in the same way as setting the vertex buffer object; so, we generate the element, set the binding, and then bind the data to the buffer, as follows:

1. We first generate the buffer by calling `glGenBuffers`. This is done by passing in the number of buffers that we want to create, which is 1, and then pass in the name of the buffer object in order to generate it:

```
glGenBuffers(1, &ebo);  
 glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, ebo);  
  
glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(GLuint) *  
             indices.size(), &indices[0], GL_STATIC_DRAW);
```

2. Next, we bind the buffer type to the buffer object, which, in this case, is `GL_ELEMENT_ARRAY_BUFFER` as it will store the element or index data.
3. Then, we set the index data itself by calling `glBufferData`. We pass in the buffer type first, set the size of the element data, and then pass in the data and the usage to be `GL_STATIC_DRAW` as before.
4. At the end of the constructor, we unbind the buffer and the vertex array as a precaution:

```
glBindBuffer(GL_ARRAY_BUFFER, 0);  
 glBindVertexArray(0);
```

5. Next, we will create the `draw` function; this will be used to draw the object itself.
To do this, add the `draw` function as follows:

```
void LightRenderer::draw() {  
  
}
```

6. Using this function we will add code to draw the object. The first thing we will do is create a `glm::mat4` function called `model` and initialize it; then, we will use the `glm::translate` function to translate the object to the required position:

```
glm::mat4 model = glm::mat4(1.0f);  
  
model = glm::translate(glm::mat4(1.0), position);
```

Next, we will set the model, view, and projection matrices to transform the object from its local space. This is covered in [Chapter 2, Mathematics and Graphics Concepts](#), so it is a good time to go and refresh your memory of graphics concepts.

The model, view, and projection matrices are set in the vertex shader. Information is sent to the shader by first calling `glUseProgram`, which takes in a shader program:

```
glUseProgram(this->program);
```

Then, we can send the information through the uniform variables. We will create a uniform data type in the shader using a name. In the `draw` function, we need to first get the location of this uniform variable by calling `glGetUniformLocation`, and then passing in the program and the variable string in the shader that we set, as follows:

```
GLint modelLoc = glGetUniformLocation(program, "model");
```

This will return a `GLuint` value with the location of the variable, which is the model matrix here.

Now we can set the value of the model matrix using the `glUniform` function. Since we are setting a matrix uniform, we use the `glUniformMatrix3fv` function; this takes four parameters. The first parameter is the location that we obtained in the previous step, and the second parameter is the amount of data that we are passing in; in this case, we are just passing in one matrix, so we specify this as 1. The third parameter is a Boolean value, which specifies whether the data needs to be transposed. We don't want the matrix to be transposed, so we specify it as `GL_FALSE`. Then, the final parameter is the pointer to the data for which we use `gl::ptr_value`, and then pass in the model matrix.

So, add the function to set the model matrix as follows:

```
glUniformMatrix4fv(modelLoc, 1, GL_FALSE, glm::value_ptr(model));
```

Similarly to the model matrix, we have to pass in the view and projection matrices to the shader as well. To do this, we get the view and projection matrices from the `camera` class. Then, we get the location of the uniform variable that we defined in the shader and set the value of the view and projection matrices using the `glUniformMatrix4fv` function:

```
glm::mat4 view = camera->getViewMatrix();
GLint vLoc = glGetUniformLocation(program, "view");
glUniformMatrix4fv(vLoc, 1, GL_FALSE, glm::value_ptr(view));

glm::mat4 proj = camera->getProjectionMatrix();
GLint pLoc = glGetUniformLocation(program, "projection");
glUniformMatrix4fv(pLoc, 1, GL_FALSE, glm::value_ptr(proj));
```

Once we have all the required data to draw the object, we can finally draw the object. At this point, we call `glBindVertexArray`, bind the `vao` buffer object, and then call the `glDrawElements` function to draw the object.

The `glDrawElements` function takes four parameters. The first parameter is the mode that we can use to draw the lines by calling `GL_LINES`. Alternatively, we can draw triangles by using `GL_TRIANGLES`. There are, in fact, many more types of modes that can be specified, but for our case we will only specify `GL_TRIANGLES`.

The second parameter is the number of elements or the number of indices that need to be drawn. This is specified when we created the object.

The third parameter is the type of index data that we will be passing, which is of type `GL_UNSIGNED_INT`.

The final parameter is the location where the indices are stored – this is set to 0.

So, now add the following lines:

```
glBindVertexArray(vao);
glDrawElements(GL_TRIANGLES, indices.size(), GL_UNSIGNED_INT, 0);
```

For safety purposes, we will unbind the vertex array and the program variable by setting their values to 0:

```
glBindVertexArray(0);
glUseProgram(0);
```

This marks the end of the draw function.

We will add the destructor and the rest of the setters and getters to finish the class, as follows:

```
LightRenderer::~LightRenderer() {  
}  
  
void LightRenderer::setPosition(glm::vec3 _position) {  
    position = _position;  
}  
  
void LightRenderer::setColor(glm::vec3 _color) {  
    this->color = _color;  
}  
  
void LightRenderer::setProgram(GLuint _program) {  
    this->program = _program;  
}  
  
//getters  
glm::vec3 LightRenderer::getPosition() {  
    return position;  
}  
  
glm::vec3 LightRenderer::getColor() {  
    return color;  
}
```

Drawing the object

Let's go back to `source.cpp` and render `LightRenderer`, as follows:

1. At the top of the file, include `ShaderLoader.h`, `Camera.h`, and `LightRenderer.h`, and then create an instance of the `Camera` and `LightRenderer` classes called `camera` and `light`, as follows:

```
#include "ShaderLoader.h"  
#include "Camera.h"  
#include "LightRenderer.h"
```

```
Camera* camera;
LightRenderer* light;
```

2. Create a new function called `initGame` and then add the prototype for it at the top. In the `gameInit` function, we will load the shader and initialize the camera and light.
3. Add the new function as follows:

```
void initGame() {
    ...
}
```

4. The first thing we do is enable depth testing so that only the pixels in the front are drawn. This is done by calling the `glEnable()` function and passing in the `GL_DEPTH_TEST` variable; this will enable the following depth test:

```
glEnable(GL_DEPTH_TEST);
```

5. Next, we will create a new instance of `ShaderLoader` called `shader` in the `init` function. We will then call the `createProgram` function and pass in the vertex and fragment shader files for shading the light source. The program will return a `GLuint` value, so we store it in a variable called `flatShaderProgram`, as follows:

```
ShaderLoader shader;

GLuint flatShaderProgram =
    shader.createProgram("Assets/Shaders/FlatModel.vs",
    "Assets/Shaders/FlatModel.fs");
```

6. The vertex and shader files are located in the `Assets` folder under `Shaders`; the `FlatModel.vs` file will be as follows:

```
#version 450 core

layout (location = 0) in vec3 Position;
layout (location = 1) in vec3 Color;

uniform mat4 projection;
uniform mat4 view;
uniform mat4 model;

out vec3 outColor;

void main() {
```

```
    gl_Position = projection * view * model * vec4(Position, 1.0);  
  
    outColor = Color;  
}
```

The `#version` specifies the version of GLSL that we are using, which is 450 of the OpenGL version 4.50. Next, `layout (location = 0)` and `layout (location = 1)` specify the location of the vertex attributes that are passed in; in this case, this is the position and color. The 0 and 1 indices correspond to the index number while setting `vertexAttribPointer`. In the variables specified, this data are inputs to the shader and are stored in a shader-specific `vec3` data type called `Position` and `Color`.

The three uniforms that we sent from the `draw` call for storing the model, view, and projection matrices are stored in a variable type called `uniform` and a `mat4` store data type, which are both matrices. We create another variable of type `out`, which specifies that this will be sent out of the vertex shader; this is of type `vec3` and is called `outColor`. Next, all the actual work is done inside the main function. For this, we transform the local coordinate system by multiplying the position with the model, view, and projection matrices and the result is stored in a GLSL intrinsic variable called `gl_Position`—this is the final position of the object. We then store the `Color` attribute in the `out` `vec3` variable that we created called `outColor`—and that's it for the vertex shader!

7. Next, let's take a look at the fragment shader `FlatModel.fs` file:

```
#version 450 core  
  
in vec3 outColor;  
  
out vec4 color;  
  
void main(){  
    color = vec4(outColor, 1.0f);  
}
```

In the fragment shader file, we also specify the version of GLSL that we are using.

Next, we specify an `in vec3` variable called `outColor`, which will be the color that was sent out of the vertex shader and can be used in the fragment shader. We also create an `out vec4` variable called `color`, which will be sent out of the fragment shader and will be used to color the object. The color being sent out of the fragment shader is expected to be a `vec4` variable. Then, in the main function, we convert `outColor` from a `vec3` variable to a `vec4` variable, and then set it to `color` variable.

In shaders, we can convert a `vec3` variable to a `vec4` variable by simply performing the following operation. This might look a bit strange, but for the sake of convenience this unique feature is available in shader programming to make our lives a little easier.

8. Going back to the `source.cpp` file, when we pass in the vertex and fragment shader files, they will create `flatShaderProgram`. Next, in the `initGame` function, we create and initialize the camera as follows:

```
camera = new Camera(45.0f, 800, 600, 0.1f, 100.0f, glm::vec3(0.0f,  
4.0f, 6.0f));
```

We create a new camera with an FOV of 45, width and height of 800 by 600, near and far plane of `0.1f` and `100.0f`, as well as a position of 0 along the `x` axis, 4.0 along the `y` axis, and 6.0 along the `z` axis.

9. Next, we create the light as follows:

```
light = new LightRenderer(MeshType::kTriangle, camera);  
light->setProgram(flatShaderProgram);  
light->setPosition(glm::vec3(0.0f, 0.0f, 0.0f));
```

10. This is done with the shape of a triangle and passed to the camera. We then set the shader to `flatShaderProgram` and set the position to the center of the world.
11. Now we call the draw function of the light in the `renderScene()` function, as follows:

```
void renderScene() {  
  
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);  
    glClearColor(1.0, 1.0, 0.0, 1.0); //clear yellow  
    light->draw();  
  
}
```

12. I changed the clear screen color to yellow so that the triangle can be seen clearly.
Next, call the initGame function in the main function, as follows:

```
int main(int argc, char **argv)
{
    glfwInit();

    GLFWwindow* window = glfwCreateWindow(800, 600, "Hello OpenGL",
    NULL, NULL);
    glfwMakeContextCurrent(window);

    glewInit();

    initGame();

    while (!glfwWindowShouldClose(window)) {

        renderScene();

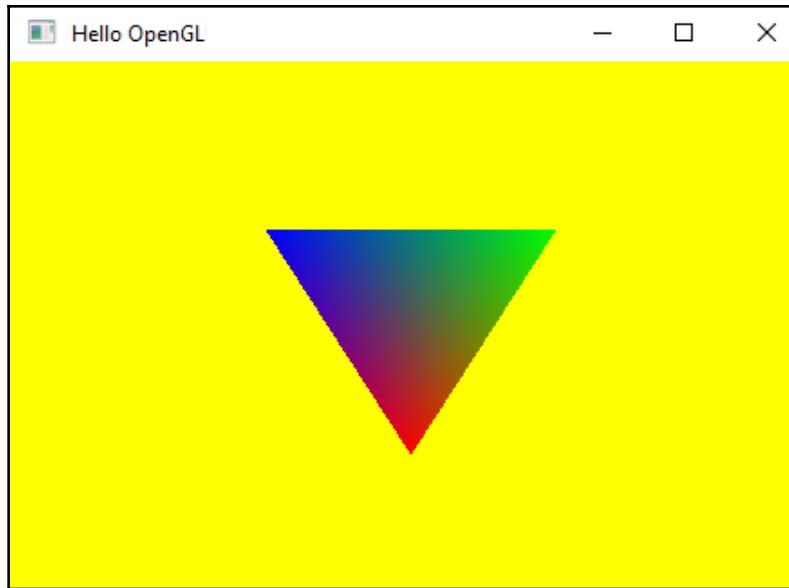
        glfwSwapBuffers(window);
        glfwPollEvents();
    }

    glfwTerminate();

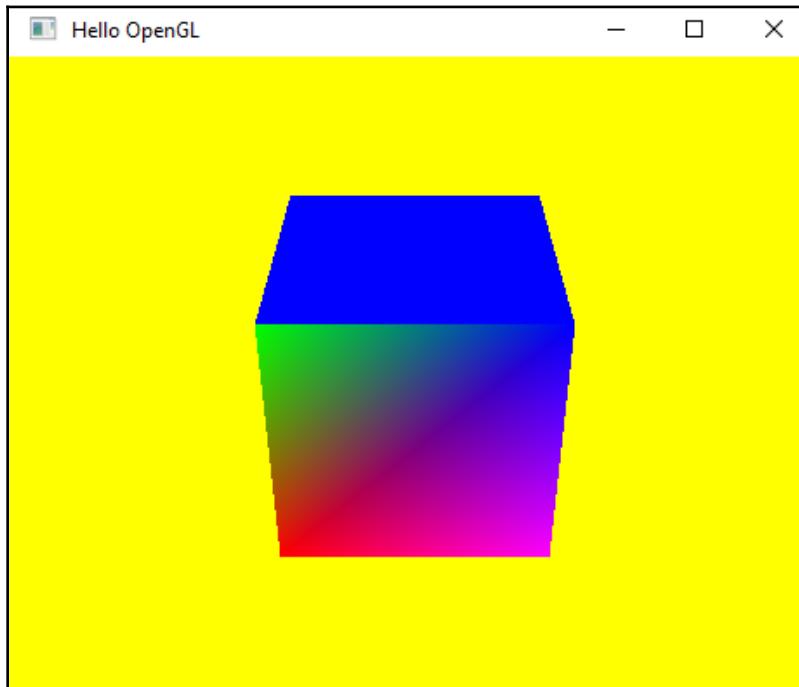
    delete camera;
    delete light;

    return 0;
}
```

13. Delete the camera and light at the end so that the system resource is released.
14. Now run the project to see the glorious triangle that we set as the shape of the
light source:



15. Change the `MeshType` type to `cube` to see a cube being drawn instead:



If, instead of the colored object as an output, you get an error, then this could either mean that you have done something incorrectly, or that your drivers are not updated and your GPU doesn't support OpenGL 4.5.

16. To make sure GLFW supports the version of the driver, add the following code, which checks for any GLFW errors. Then, run the project and look at the console output for any errors:

```
static void glfwError(int id, const char* description)
{
    std::cout << description << std::endl;
}

int main(int argc, char **argv)
{

    glfwSetErrorCallback(&glfwError);

    glfwInit();

    GLFWwindow* window = glfwCreateWindow(800, 600, "Hello OpenGL",
NULL, NULL);
    glfwMakeContextCurrent(window);

    glewInit();

    initGame();


    while (!glfwWindowShouldClose(window)) {

        renderScene();

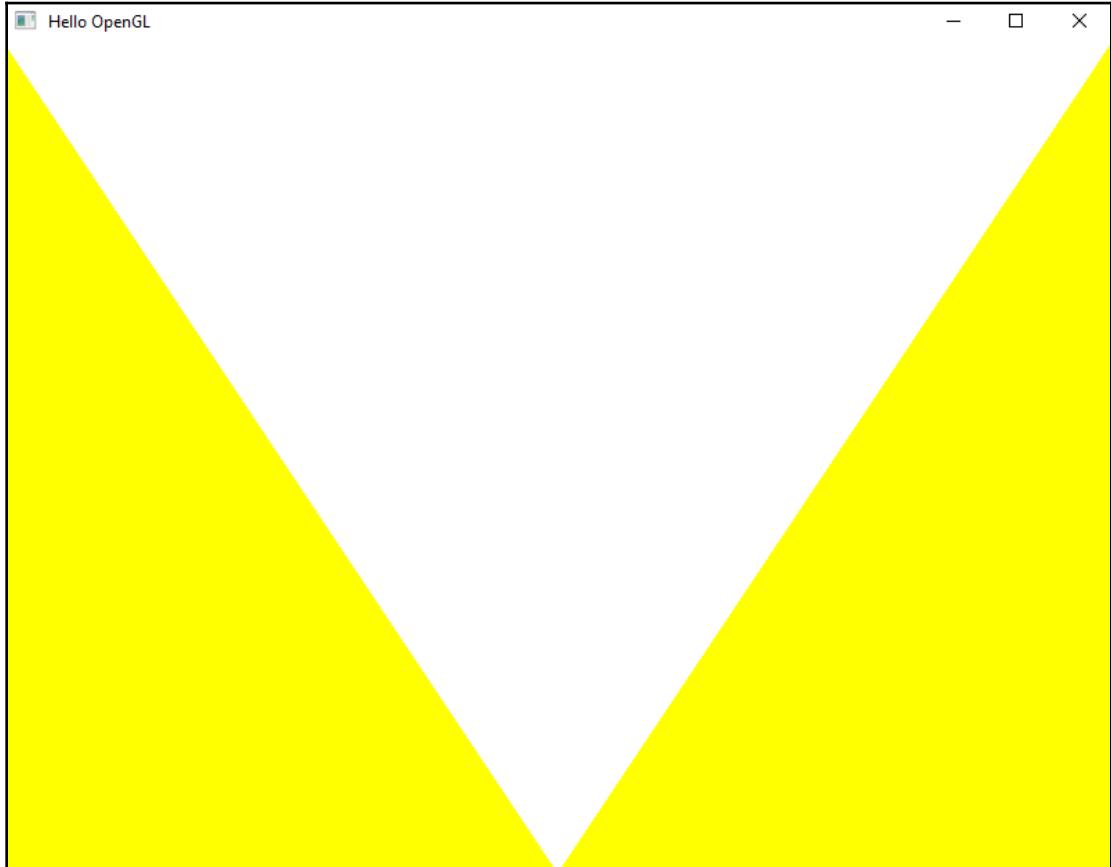
        glfwSwapBuffers(window);
        glfwPollEvents();
    }

    glfwTerminate();

    delete camera;
    delete light;

    return 0;
}
```

If you get the following output, then it could mean that the OpenGL version is not supported:



This will be accompanied by the following error suggesting that the GLSL version is not supported:

```
ERROR compiling shader: vertex shader
ERROR: 0:2: '' : incorrect GLSL version: 450
WARNING: 0:3: 'GL_ARB_explicit_attrib_location' : extension is not available
        current GLSL version
WARNING: 0:3: 'GL_ARB_explicit_attrib_location' : extension is not available
        current GLSL version
WARNING: 0:4: 'GL_ARB_explicit_attrib_location' : extension is not available
        current GLSL version
WARNING: 0:4: 'GL_ARB_explicit_attrib_location' : extension is not available
        current GLSL version
WARNING: 0:4: 'GL_ARB_explicit_attrib_location' : extension is not available
        current GLSL version
WARNING: 0:5: 'GL_ARB_explicit_attrib_location' : extension is not available
        current GLSL version

ERROR compiling shader: fragment shader
ERROR: 0:2: '' : incorrect GLSL version: 450

Shader Loader : LINK ERROR
Link called without any attached shader objects.
```

In this case, change the version of the shader code at the top of the shader to 330 instead of 450, and then try running the project again.

This should give you the desired output.

Summary

In this chapter, we created a new OpenGL project and added the necessary libraries to get the project working. Then, we created a new window to work with using GLFW. Then, using a couple more lines of code we were able to clear the viewport with the color of our choice.

Next, we started preparing classes to help us in drawing objects such as the `Mesh` class, which defined the shape of the object; and the `Camera` class that we use in order to views the object. Then, we created a `ShaderLoader` class that helped us in creating the shader program that was used to draw the object.

With the necessary preparation done, we finally created a `LightRenderer` class. This is used to draw an object that represents a light position that is defined by a shape – we used this class to draw our first object.

In the next chapter, we will explore how to draw other objects by adding texture and physics to the rendering engine.

7

Building on the Game Objects

In the last chapter, we looked at how to draw basic shapes using OpenGL. Now that we have covered the basics, let's improve our objects by adding some textures to them so that the objects don't just look like a plain cube and sphere.

We can write our physics like we did last time, but when dealing with 3D objects, writing our own physics can become difficult and time consuming. To simplify the process, we will use the help of an external physics library to handle the physics and collision detection.

We will cover the following topics in this chapter:

- Creating the `MeshRenderer` class
- Creating the `TextureLoader` class
- Adding Bullet Physics
- Adding rigid bodies

Creating a MeshRenderer class

For drawing regular game objects, we will create a separate class from the LightRenderer class by adding texture, and we will also add motion to the object by adding physical properties. We will draw a textured object and then add physics to this object in the next section of this chapter. To do this, we will create a new .h and .cpp file called MeshRenderer.

In the MeshRenderer.h file we will do the following:

1. First, we will add the includes, as follows:

```
#include <vector>

#include "Camera.h"
#include "LightRenderer.h"

#include <GL/glew.h>

#include "Dependencies/glm/glm/glm.hpp"
#include "Dependencies/glm/glm/gtc/matrix_transform.hpp"
#include "Dependencies/glm/glm/gtc/type_ptr.hpp"
```

2. Next, we will create the class itself, as follows:

```
Class MeshRenderer{  
  
};
```

3. In this, we will create the public section first, as follows:

```
public:  
    MeshRenderer(MeshType modelType, Camera* _camera);  
    ~MeshRenderer();  
    void draw();  
  
    void setPosition(glm::vec3 _position);  
    void setScale(glm::vec3 _scale);  
    void setProgram(GLuint _program);  
    void setTexture(GLuint _textureID);
```

In this section, we create the constructor, which takes a ModelType and the _camera. We add the destructor afterward. We have a separate function for drawing the object.

4. We then use some setter functions to set the position, scale, the shader program, and the `textureID` function itself, which we will be using to set the texture on the object.
5. Next, we will add the `private` section, as follows:

```
private:  
  
    std::vector<Vertex> vertices;  
    std::vector<GLuint> indices;  
    glm::mat4 modelMatrix;  
  
    Camera* camera;  
  
    glm::vec3 position, scale;  
    GLuint vao, vbo, ebo, texture, program;
```

In the `private` section, we have vectors to store the vertices and the indices. Then, we have a `glm::mat4` variable called `modelMatrix` to store the model matrix value in.

6. We create a local variable for the camera and `vec3`s for storing the position and scale value.
7. Finally, we have `GLuint` to store `vao`, `vbo`, `ebo`, `textureID`, and the shader program.

We will now move onto the `MeshRenderer.cpp` file, going through the following steps:

1. First, we will include the `MeshRenderer.h` file at the top of `MeshRenderer.cpp`.
2. Next, we will create the constructor for `MeshRenderer`, as follows:

```
MeshRenderer::MeshRenderer(MeshType modelType, Camera* _camera) {  
  
}
```

3. For this, we first initialize the `camera`, `position`, and `scale` local values, as follows:

```
    camera = _camera;  
  
    scale = glm::vec3(1.0f, 1.0f, 1.0f);  
    position = glm::vec3(0.0, 0.0, 0.0);
```

4. Create a switch statement, as we did in `LightRenderer`, to get the mesh data, as follows:

```
switch (modelType) {  
  
    case kTriangle: Mesh::setTriData(vertices, indices);  
        break;  
    case kQuad: Mesh::setQuadData(vertices, indices);  
        break;  
    case kCube: Mesh::setCubeData(vertices, indices);  
        break;  
    case kSphere: Mesh::setSphereData(vertices, indices);  
        break;  
}
```

5. Then, generate and bind `vao`, `vbo`, and `ebo`. In addition to this, set the data for `vbo` and `ebo` as follows:

```
glGenVertexArrays(1, &vao);  
glBindVertexArray(vao);  
  
glGenBuffers(1, &vbo);  
glBindBuffer(GL_ARRAY_BUFFER, vbo);  
glBufferData(GL_ARRAY_BUFFER, sizeof(Vertex) * vertices.size(),  
&vertices[0], GL_STATIC_DRAW);  
  
glGenBuffers(1, &ebo);  
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, ebo);  
glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(GLuint) *  
indices.size(), &indices[0], GL_STATIC_DRAW);
```

6. The next step is to set the attributes. In this case, we will be setting the position attribute, but instead of color, we will set the texture coordinate attribute, as it will be required to set the texture on top of the object.
7. The attribute at the 0th index will still be a vertex position, but the attribute of the first index will be a texture coordinate this time, as shown in the following code:

```
 glEnableVertexAttribArray(0);  
  
 glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, sizeof(Vertex),  
(GLvoid*)0);  
  
 glEnableVertexAttribArray(1);  
  
 glVertexAttribPointer(1, 2, GL_FLOAT, GL_FALSE, sizeof(Vertex),  
(void*)(offsetof(Vertex, Vertex::texCoords)));
```

Here, the attribute for the vertex position remains the same, but for the texture coordinate, the first index is enabled as before. The change occurs in the number of components. The texture coordinate is defined in the x and y axes, as this is a 2D texture. So, for the second parameter, we specify 2 instead of 3. The stride still remains the same, but the offset is changed to `texCoords`.

8. To close the constructor, we unbind the buffers and `vertexArray` as follows:

```
glBindBuffer(GL_ARRAY_BUFFER, 0);
glBindVertexArray(0);
```

9. We now add the `draw` function as follows:

```
void MeshRenderer::draw() {
}
```

10. In this `draw` function, we will first set the model matrix as follows:

```
glm::mat4 TranslationMatrix = glm::translate(glm::mat4(1.0f),
position);

glm::mat4 scaleMatrix = glm::scale(glm::mat4(1.0f), scale);

modelMatrix = glm::mat4(1.0f);

modelMatrix = TranslationMatrix * scaleMatrix;
```

11. We will create two matrices for storing `TranslationMatrix` and `scaleMatrix` and then we set the values.
12. We will then initialize the `modelMatrix` variable and the multiply scale and translation matrix and assign them to the `modelMatrix` variable.
13. Next, instead of creating a separate view and projection matrix, we can create a single matrix called `vp` and assign the multiplied view and projection matrices to it as follows:

```
glm::mat4 vp = camera->getprojectionMatrix() *
camera->getViewMatrix();
```

Obviously, the order in which the view and projection matrices are multiplied matters and cannot be reversed.

14. We can now send the values to the GPU.
15. Before we send the values to the shader, the first thing we have to do is call `glUseProgram` and set the shader program so that the data is sent to the correct program. Once this is complete, we can set the values for `vp` and `modelMatrix` as follows:

```
glUseProgram(this->program);
GLint vpLoc = glGetUniformLocation(program, "vp");
glUniformMatrix4fv(vpLoc, 1, GL_FALSE, glm::value_ptr(vp));

GLint modelLoc = glGetUniformLocation(program, "model");
glUniformMatrix4fv(modelLoc, 1, GL_FALSE,
glm::value_ptr(modelMatrix));
```

16. Next, we will bind the texture object. We use the `glBindTexture` function to bind the texture. The function takes two parameters, with the first being the texture target. We have a 2D texture, so we pass in `GL_TEXTURE_2D` as the first parameter and the second parameter is a texture ID. So, we add the following line to bind the texture:

```
glBindTexture(GL_TEXTURE_2D, texture);
```

You might be wonder why we aren't using `glUniformMatrix4fv` or something similar while setting the texture location, as we did for the matrices. Well, since we have just the one texture, the program sets the uniform location as the 0th index by default so we don't have to worry about it. That is all that is required to bind the texture.

17. Next, we can bind the `vao` and draw the object as follows:

```
glBindVertexArray(vao);
glDrawElements(GL_TRIANGLES, indices.size(), GL_UNSIGNED_INT, 0);
```

18. Unbind the `VertexArray` at the end as follows:

```
glBindVertexArray(0);
```

19. Next, we will add the definition for the destructor and `setters` as follows:

```
MeshRenderer::~MeshRenderer() {
}

// setters

void MeshRenderer::setTexture(GLuint textureID) {
```

```
        texture = textureID;

    }

    void MeshRenderer::setScale(glm::vec3 _scale) {

        this->scale = _scale;
    }

    void MeshRenderer::setPosition(glm::vec3 _position) {

        this->position = _position;
    }

    void MeshRenderer::setProgram(GLuint _program) {

        this->program = _program;
    }
```

Creating the TextureLoader class

We created the `MeshRenderer` class, but we still need to load the texture itself and set the texture ID, which can be passed to the `MeshRendered` object. For this, we will create a `TextureLoader` class that will be responsible for loading the textures. Let's see how to do so:

So, to do this, we need to create the new `.h` and `.cpp` file called `TextureLoader`.

For loading the JPEG or PNG image itself, we will use a header-only library called STB. This can be downloaded from <https://github.com/nothings/stb>. Clone or download the source from the link and place the `stb-master` folder in the `Dependencies` folder.

In the `TextureLoader` class, add the following:

```
#include <string>
#include <GL/glew.h>

class TextureLoader
{
public:
    TextureLoader();

    GLuint getTextureID(std::string texFileName);
```

```
    ~TextureLoader();  
};
```

We will then use `include string` and `glew.h`, as we will be passing the location of the file where the JPEG is located and `STB` will load the file from there. We will add a constructor and a destructor, as they are requirement; otherwise the compiler, will give an error. We will then create a function called `getTextureID`, which takes a string as an input and returns `GLuint`, which will be the texture ID.

In the `TextureLoader.cpp` file, we include `TextureLoader.h`. We then add the following to include `STB`:

```
#define STB_IMAGE_IMPLEMENTATION  
#include "Dependencies/stb-master/stb_image.h"
```

We add `#define`, as it is required in a `TextureLoader.cpp` file, navigate to `stb_image.h`, and include it in the project. We then add the constructor and destructor next, as follows:

```
TextureLoader::TextureLoader() {  
}  
  
TextureLoader::~TextureLoader() {  
}
```

Next, we create the `getTextureID` function as follows:

```
GLuint TextureLoader::getTextureID(std::string texFileName) {  
}
```

In the `getTextureID` function, we will first create three `int` variables to store the width, height, and number of channels. An image usually only has three channels: red, green, and blue; however, it could have a fourth channel, the alpha channel, which is used for transparency. JPEG pictures have only three channels, but the PNG file could have three or four channels.

In our game, we will only be using a JPEG file, so the `channels` parameter will always be three, as shown in the following code:

```
int width, height, channels;
```

We will use the `stbi_load` function to load the image data to an unsigned char pointer, as follows:

```
stbi_uc* image = stbi_load(texFileName.c_str(), &width, &height,
&channels,
STBI_rgb);
```

The function takes five parameters. The first is the string of the location of the file/filename. Then, it returns the width, height, and number of channels as the second, third, and fourth parameters, and in the fifth parameter, you set the required components. In this case, we want just the `r`, `g`, and `b` channels, so we specify `STBI_rgb`.

We then have to generate and bind the texture as follows:

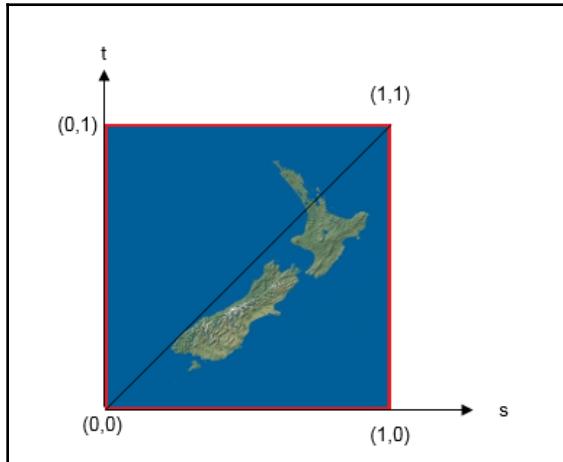
```
GLuint mtexture;
 glGenTextures(1, &mtexture);
 glBindTexture(GL_TEXTURE_2D, mtexture);
```

First, a texture ID is created called `mtexture` of the `GLuint` type. Then, we call the `glGenTextures` function and pass in the number of objects we want to create and pass in the array names, which is `mtexture`. We also have to bind the texture type by calling `glBindTexture` and passing in the texture type, which is `GL_TEXTURE_2D`, specifying that it is a 2D texture and stating the texture ID.

Next, we have to set the texture wrapping. Texture wrapping dictates what happens when the texture coordinate is greater or less than 1 in *x* and *y*.

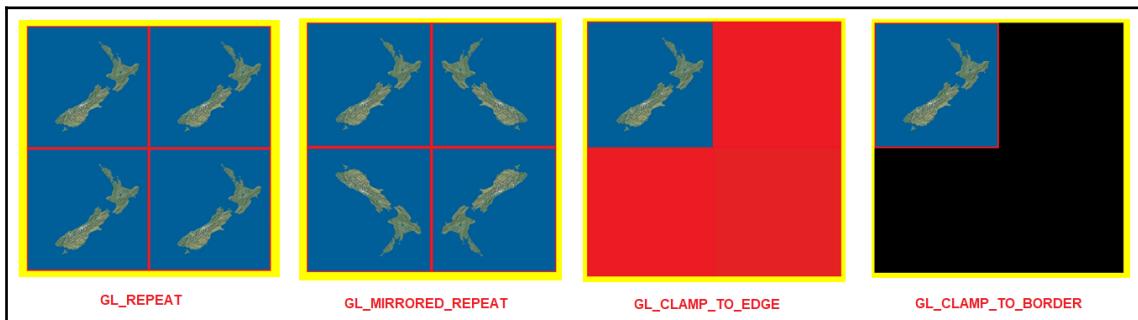
Textures can be wrapped in one of four ways: `GL_REPEAT`, `GL_MIRRORED_REPEAT`, `GL_CLAMP_TO_EDGE`, or `GL_CLAMP_TO_BORDER`.

If we imagine a texture applied to a quad then the positive *s* axis runs along horizontally and the *t* axis runs along vertically starting at the origin (the bottom left corner) as shown below:



Let's look at the different ways that the textures can be wrapped, as shown in the following list:

- `GL_REPEAT` just repeats the texture when applied to a quad.
- `GL_MIRRORED_REPEAT` repeats the texture, but also mirrors the texture the next time.
- `GL_CLAMP_TO_EDGE` takes the `rgb` value at the edge of the texture and repeats the value for the extent of the object. Here, the red border pixels get repeated.
- `GL_CLAMP_TO_BORDER` takes a user-specific value and applies it to the end of the object instead of applying the edge color, as shown in the following screenshot:



For our purposes, we need `GL_REPEAT`, which is set as the default anyway, but if you had to set it, you will need to add the following:

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);  
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
```

You use the `glTexParameteri` function, which takes three parameters. The first is the texture type, which is `GL_TEXTURE_2D`. The next parameter is the direction in which you want the wrapping to apply, which is `S` or `T`. `S` is the same as `x` and `T` is the same as `y`. The last parameter is the wrapping parameter itself.

Next, we can set the texture filtering. Sometimes, when you apply a low-quality texture to a big quad, if you zoom in closer, the texture will be pixilated, as seen in the following picture on the left:



The picture on the left is the output of setting the texture filtering to `GL_NEAREST`, and the picture on the right is the result of applying texture filtering to `GL_LINEAR`. The `GL_LINEAR` wrapping linearly interpolates with the texel value of the surrounding values to give a much smoother result when compared to `GL_NEAREST`.

When the texture is magnified, it is better to set the value to `GL_LINEAR` to get a smoother picture, and when the picture is minified, it can then be set to `GL_NEAREST`, as the texels (which are texture elements) will be so small that we won't be able to see them anyway.

For setting the texture filtering we use the same `glTexParameteri` function, but instead of passing in the wrapping direction as the second parameter, we specify `GL_TEXTURE_MIN_FILTER` and `GL_TEXTURE_MAG_FILTER` as the second parameter and pass in `GL_NEAREST` or `GL_LINEAR` as the third parameter, as follows:

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);  
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
```

It doesn't make sense if you load a huge image and the object is so far away that you can't even see it. So, for optimization purposes, you can create mipmaps. Mipmaps basically take the texture and it at lower resolutions, which will automatically change to a lower resolution image when the texture is too far from the camera. It will also change to a higher resolution texture when the camera is closer.

Here is the mipmap chain for the texture we are using:



The mipmap quality can be set using the `glTexParameter`i function again. This basically replaces `GL_NEAREST` with either `GL_NEAREST_MIPMAP_NEAREST`, `GL_LINEAR_MIPMAP_NEAREST`, `GL_NEAREST_MIPMAP_LINEAR`, or `GL_LINEAR_MIPMAP_LINEAR`.

The best option is `GL_LINEAR_MIPMAP_LINEAR`, because it linearly interpolates the value of the texel between two mipmaps, as well as samples, by linearly interpolating between the surrounding texels (a texel is the lowest unit of an image in the same way that pixel is the smallest unit of a screen to represent a color at a location on the screen. If a 1080p picture is shown on a 1080p screen then 1 texel is mapped to 1 pixel).

So, we will use the following as our new filtering/mipmap values:

```
glTexParameter(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,  
               GL_LINEAR_MIPMAP_LINEAR);  
glTexParameter(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
```

Once this has been set, we can finally create the texture using the `glTexImage2D` function, as follows:

```
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, width, height, 0, GL_RGB,  
            GL_UNSIGNED_BYTE, image);
```

The `glTexImage2D` function takes nine parameters. These are described as follows:

- The first is the texture type, which is `GL_TEXTURE_2D`.
- The second is the mipmap level. If we want to use a lower-quality picture, we can set this value to 1, 2, or 3. For our purposes, we will leave this value as 0, which is the base level.
- For the third parameter, we will specify which all-color channels we want to store from the image. Since we want to store all three channels, we specify `GL_RGB`.
- The fourth and fifth parameters that we specify are the width and height of the picture.
- The next parameter has to be set to 0, as specified in the documentation (which can be found at <https://www.khronos.org/registry/OpenGL-Refpages/gl4/html/glTexImage2D.xhtml>).
- The next parameter that we specify is the data format of the image source.
- The next parameter is the type of data that is passed in, which is `GL_UNSIGNED_BYTE`.
- Finally, we set the image data itself.

Now that the texture is created, we call `glGenerateMipmap` and pass in the `GL_TEXTURE_2D` texture type, as follows:

```
glGenerateMipmap(GL_TEXTURE_2D);
```

We then unbind the texture, free the picture, and finally return the `textureID` function, like so:

```
glBindTexture(GL_TEXTURE_2D, 0);
stbi_image_free(image);

return mtexture;
```

With all that done, we can finally add our texture to the game object.

In the `source.cpp`, include `MeshRenderer.h` and `TextureLoader.h`. To do so, follow these steps:

1. At the top, create a `MeshRenderer` pointer object called a `sphere`, as follows:

```
Camera* camera;
LightRenderer* light;
MeshRenderer* sphere;
```

2. In the `init` function, create a new shader program called `texturedShaderProgram` of the `GLuint` type, as follows:

```
GLuint flatShaderProgram =
    shader.CreateProgram("Assets/Shaders/FlatModel.vs",
    "Assets/Shaders/FlatModel.fs");
GLuint texturedShaderProgram =
    shader.CreateProgram("Assets/Shaders/TexturedModel.vs",
    "Assets/Shaders/TexturedModel.fs");
```

3. We will now load the two shaders called `TexturedModel.vs` and `TexturedModel.fs` as follows:

- Here is the `TexturedModel.vs` shader:

```
#version 450 core
layout (location = 0) in vec3 position;
layout (location = 1) in vec2 texCoord;

out vec2 TexCoord;

uniform mat4 vp;
uniform mat4 model;

void main(){
    gl_Position = vp * model *vec4(position, 1.0);
    TexCoord = texCoord;
}
```

The only difference between this and `FlatModel.vs` is that, here, the second location is a `vec2` called `texCoord`. We create an `out vec2` called `TexCoord`, into which we will store this value in the `main` function.

- Here is the `TexturedModel.fs` shader:

```
#version 450 core
in vec2 TexCoord;
out vec4 color;

// texture
uniform sampler2D Texture;
```

```
void main() {
    color = texture(Texture, TexCoord);
}
```

We create a new `vec2` called `TexCoord` to receive the value from the vertex shader.

We then create a new uniform type called `sampler2D` and call it `Texture`. The texture is received through a sampler that will be used to sample the texture depending upon the wrap and filtering parameters we set while creating the texture.

Then, the color is set depending upon the sampler and texture coordinates using the `texture` function. This function takes sampler and texture coordinates as parameters. The texel at a texture coordinate is sampled based on the sampler and that color value is returned and assigned to the object at that texture coordinate.

Let's continue creating the `MeshRenderer` object. Load the texture `globe.jpg` file using the `getTextureID` function of the `TextureLoader` class and set it to a `GLuint` called `sphereTexture`, as follows:

```
TextureLoader tLoader;
GLuint sphereTexture = tLoader.getTextureID("Assets/Textures/globe.jpg");
```

Create the sphere `MeshRederer` object, set the mesh type, and pass the camera. Set the program, texture, position, and scale as follows:

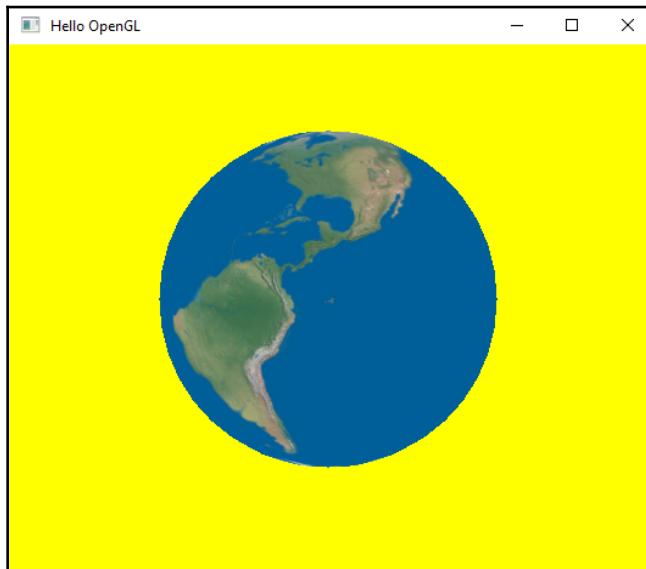
```
sphere = new MeshRenderer(MeshType::kSphere, camera);
sphere->setProgram(texturedShaderProgram);
sphere->setTexture(sphereTexture);
sphere->setPosition(glm::vec3(0.0f, 0.0f, 0.0f));
sphere->setScale(glm::vec3(1.0f));
```

In the `renderScene` function, draw the sphere object as follows:

```
void renderScene() {
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glClearColor(1.0, 1.0, 0.0, 1.0);

    sphere->draw();
}
```

You should now see the textured globe when you run the project as shown in the following screenshot:



The camera is created as follows, and is set at the z position of four units:

```
camera = new Camera(45.0f, 800, 600, 0.1f, 100.0f, glm::vec3(0.0f, 0.0f, 4.0f));
```

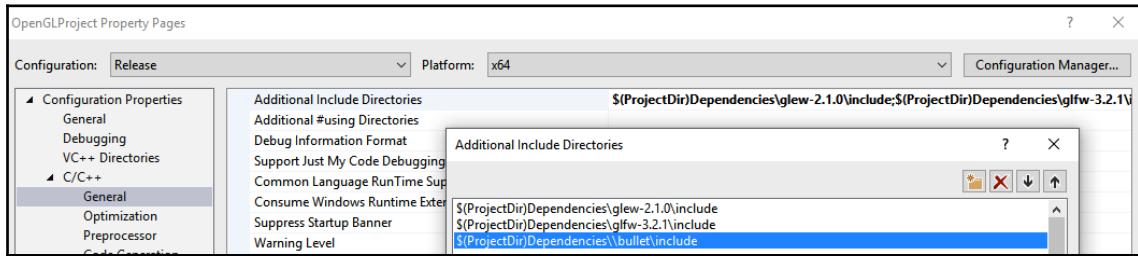
Adding Bullet Physics

To add physics to our game, we will be using the Bullet Physics engine. This is an open source project that is widely used in AAA games and movies. It is used for collision detection as well as soft and rigid-body dynamics. The library is free for commercial use.

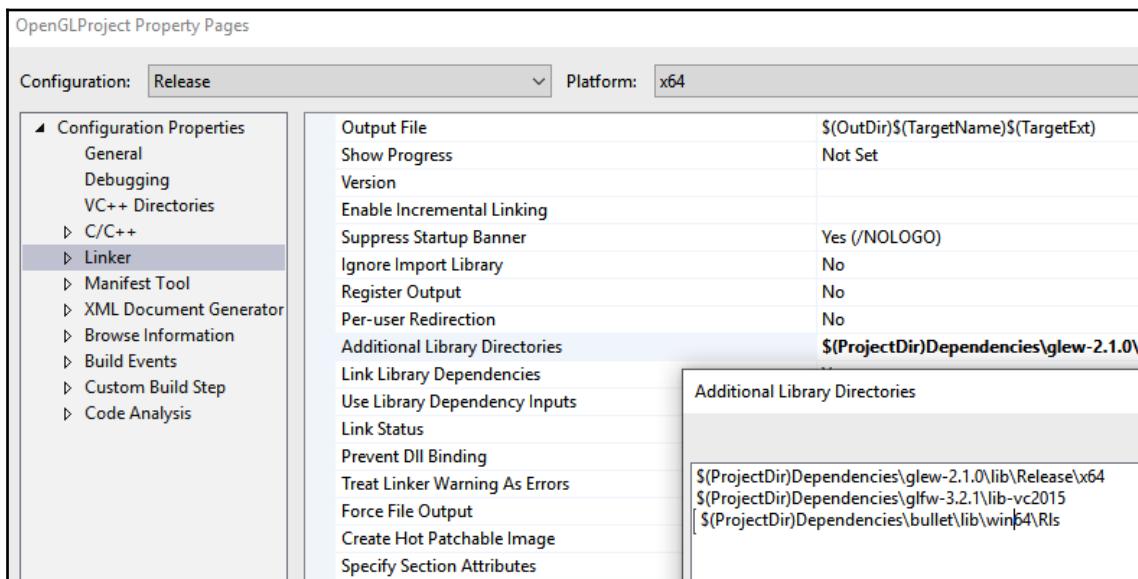
Download the source from <https://github.com/bulletphysics/bullet3> and, using CMake. You will need to build the project for the release version of x64. For your convenience, the header and lib files are included in the project for the chapter. You can take the folder and paste it into the dependencies folder.

Now that we have the folder, let's take a look at how to add Bullet Physics by following these steps:

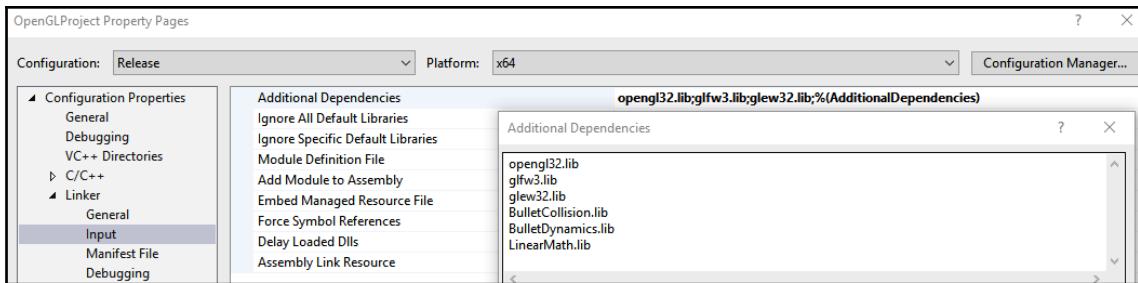
1. Add the `include` folder in C/C++ | General | Additional Include Directories shown as follows:



2. Add the `lib/win64/Rls` folder in Linker | General | Additional Library Directories:



3. Add `BulletCollision.lib`, `BulletDynamics.lib`, and `LinearMath.lib` to **Linker | Input | Additional Dependencies**, as shown in the following screenshot:



These libraries are responsible for the calculation of the movement of the game objects based on conditions such as gravity and external force, collision detection, and memory allocation.

4. With the prep work out of the way, we can start adding physics to the game. In the `source.cpp` file, include `btBulletDynamicsCommon.h` at the top of the file, as follows:

```
#include "Camera.h"  
#include "LightRenderer.h"  
#include "MeshRenderer.h"  
#include "TextureLoader.h"  
  
#include <btBulletDynamicsCommon.h>
```

5. After this, create a new pointer object to `btDiscreteDynamicsWorld` as follows:

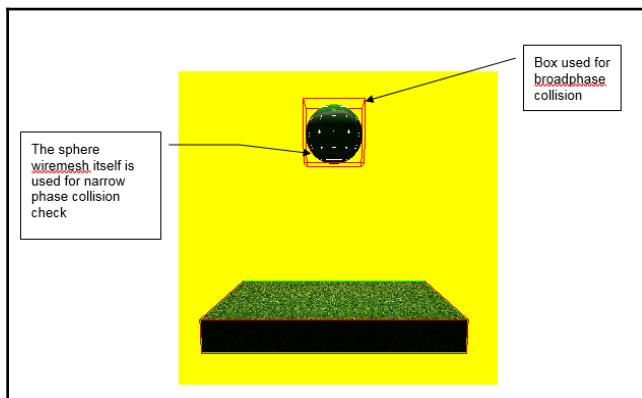
```
btDiscreteDynamicsWorld* dynamicsWorld;
```

6. This object keeps track of all the physics setting and objects in the current scene.

However, before we create `dynamicsWorld`, the Bullet Physics library requires some objects to be initialized first.

These required objects are listed as follows:

- **btBroadPhaseInterface:** Collision detection is actually done in two phases: the broadphase and narrowphase. In the broadphase, the physics engine eliminates all the objects that are unlikely to collide. This check is done using bounding boxes of the objects. Then, in the narrowphase, the actual shape of the object is used to check the likelihood of a collision. Pairs of objects are created with a strong likelihood for collision. In the following screenshot, the red box around the sphere is used for broadphase collision and the white wiremesh of the sphere is used for narrowphase collision:



- **btDefaultCollisionConfiguration:** This is used for setting up default memory.
- **btCollisionDispatcher:** A pair of objects that have a strong likelihood of colliding are tested for collision using actual shapes. This is used for getting details of the collision detection, such as which object collided with which other object.
- **btSequentialImpulseConstraintSolver:** You can create constraints, such as a hinge constraint or slider constraint, which can restrict the motion or rotation of one object about another object. For example, if there is a hinge joint between the wall and the door, then the door can only rotate around the joint and cannot be moved about, as it is fixed at the hinge joint. The constraint solver is responsible for calculating this correctly. The calculation is repeated a number of times to get close to the optimal solution.

In the `init` function, before we create the sphere object, we will initialize these objects as follows:

```
//init physics
btBroadphaseInterface* broadphase = new btDbvtBroadphase();
btDefaultCollisionConfiguration* collisionConfiguration = new
btDefaultCollisionConfiguration();
btCollisionDispatcher* dispatcher = new
btCollisionDispatcher(collisionConfiguration);
btSequentialImpulseConstraintSolver* solver = new
btSequentialImpulseConstraintSolver();
```

7. Then, we will create a new `dynamicsWorld` by passing the `dispatcher`, `broadphase`, `solver`, and `collisionConfiguration` as parameters to the `btDiscreteDynamicsWorld` function, as follows:

```
dynamicsWorld = new btDiscreteDynamicsWorld(dispatcher, broadphase,
solver, collisionConfiguration);
```

8. Now that our physics world is created, we can set the parameters for our physics. The basic parameter is gravity. We set its value to real-world conditions as follows:

```
dynamicsWorld->setGravity(btVector3(0, -9.8f, 0));
```

Adding rigid bodies

Now we can create rigid bodies or soft bodies and watch them interact with other rigid or soft bodies. A rigid body is an animate or inanimate object that doesn't change its shape or physical properties. Soft bodies, on the other hand, can be squishy and made to change shape.

In the following example, we will focus on the creation of a rigid body.

To create a rigid body, we have to specify the shape of the object and the motion state, and then set the mass and inertia of the objects. Shapes are defined using `btCollisionShape`. An object can have different shapes, or sometimes even a combination of shapes, called a compound shape. We use `btBoxShape` to create cubes and cuboids and `btSphereShape` to create spheres and other shapes, such as `btCapsuleShape`, `btCylinderShape`, and `btConeShape`, which will be used for narrowphase collision by the library.

In our case, we are going to create a sphere shape and see our Earth sphere bounce around. So, let's begin:

1. Using the following code create a `btSphere` for creating a sphere shape and set the radius as `1.0`, which is the radius of our rendered sphere as well:

```
btCollisionShape* sphereShape = new btSphereShape(1.0f);
```

2. Next, set the `btDefaultMotionState`, where we specify the rotation and position of the sphere, as follows:

```
btDefaultMotionState* sphereMotionState = new  
btDefaultMotionState(btTransform(btQuaternion(0, 0, 0, 1),  
btVector3(0, 10.0f, 0)));
```

We set the rotation to `0` and set the position of the rigid body to a distance of `10.0f` along the `y` axis. We should also set the mass and inertia and calculate the inertia of the `sphereShape`, as follows:

```
btScalar mass = 10.0;  
btVector3 sphereInertia(0, 0, 0);  
sphereShape->calculateLocalInertia(mass, sphereInertia);
```

3. To create the rigid body, we first have to create

`btRigidBodyConstructionInfo` and pass the variables to it, as follows:

```
btScalar mass = 10.0;  
btVector3 sphereInertia(0, 0, 0);  
sphereShape->calculateLocalInertia(mass, sphereInertia);  
  
btRigidBody::btRigidBodyConstructionInfo sphereRigidBodyCI(mass,  
sphereMotionState, sphereShape, sphereInertia);
```

4. Now, create the rigid body object by passing `btRigidBodConstructionInfo` into it using the following code:

```
btRigidBody* sphereRigidBody = new btRigidBody(sphereRigidBodyCI);
```

5. Now, set the physical properties of the rigid body, including friction and restitution:

```
sphereRigidBody->setRestitution(1.0f);  
sphereRigidBody->setFriction(1.0f);
```

These values are between `0.0f` and `1.0f`, meaning that the object is really smooth and has no friction, and has no restitution or bounciness. The `1.0` figure, on the other hand, means that the object is rough on the outside and extremely bouncy, like a bouncy ball.

6. After these necessary parameters are set, we need to add the rigid body to the `dynamicWorld` we created as follows, using the `addRigidBody` function of the `dynamicsWorld`:

```
dynamicsWorld->addRigidBody(sphereRigidBody);
```

Now, for our sphere mesh to actually behave like the sphere body, we have to pass the rigid body to the sphere mesh class and make some minor changes. Open the `MeshRenderer.h` and `.cpp` files. In the `MeshRenderer.h` file, include the `btBulletDynamicsCommon.h` header and add a local `btRigidBody` called `rigidBody` to the private section. Also, change the constructor to take a `rigidbody`, as follows:

```
#include <btBulletDynamicsCommon.h>

class MeshRenderer{
public:
    MeshRenderer(MeshType modelType, Camera* _camera, btRigidBody* _rigidBody);
    .
    .
    .
    private:
    .
    .
    .
    btRigidBody* rigidBody;
};
```

7. In the `MeshRenderer.cpp` file, change the constructor to take a `rigidBody` variable and set the local `rigidBody` variable to it as follows:

```
MeshRenderer::MeshRenderer(MeshType modelType, Camera* _camera,
                           btRigidBody* _rigidBody) {

    rigidBody = _rigidBody;
    camera = _camera;
    .
    .
}
```

8. Then, in the draw function, we have to replace the code where we set the `modelMatrix` variable with the code where we get the sphere rigid body value, as follows:

```
btTransform t;  
  
rigidBody->getMotionState()->getWorldTransform(t);
```

9. We use the `btTransform` variable to get the transformation from the rigid body's `getMotionState` function and then get the `WorldTrasform` variable and set it to our `brTransform` variable `t`, as follows:

```
btQuaternion rotation = t.getRotation();  
btVector3 translate = t.getOrigin();
```

10. We create two new variables of the `btQueternion` type to store rotation and `btVector3` to store the translation values using the `getRotation` and `getOrigin` functions of the `btTranform` class, as follows:

```
glm::mat4 RotationMatrix = glm::rotate(glm::mat4(1.0f),  
rotation.getAngle(), glm::vec3(rotation.getAxis().getX(), rotation.ge  
tAxis().getY(), rotation.getAxis().getZ()));  
  
glm::mat4 TranslationMatrix =  
glm::translate(glm::mat4(1.0f), glm::vec3(translate.getX(),  
translate.getY(), translate.getZ()));  
  
glm::mat4 scaleMatrix = glm::scale(glm::mat4(1.0f), scale);
```

11. Next, we create three `glm::mat4` variables, called `RotationMatrix`, `TranslationMatrix`, and `ScaleMatrix`, and set the values of rotation and translation using the `glm::rotate` and `glm::translate` function. We then pass in the rotation and translation values we stored earlier as shown in the following screenshot. We will keep the `ScaleMatrix` variable as is:

```
modelMatrix = TranslationMatrix * RotationMatrix * scaleMatrix;
```

The new `modelMatrix` variable will be the multiplication of the scale, rotation, and translation matrices in that order. The rest of the code will remain the same in the draw function.

12. In the `init` function, change the code to reflect the modified `MeshRenderer` constructor:

```
// Sphere Mesh

sphere = new MeshRenderer(MeshType::kSphere, camera,
sphereRigidBody);
sphere->setProgram(texturedShaderProgram);
sphere->setTexture(sphereTexture);
sphere->setScale(glm::vec3(1.0f));
```

13. We don't have to set the position, as that will be set by the rigid body. Set the camera as shown in the following code so that we can see the sphere:

```
camera = new Camera(45.0f, 800, 600, 0.1f, 100.0f, glm::vec3(0.0f,
4.0f, 20.0f));
```

14. Now, run the project. We can see the sphere being drawn, but it is not moving. That's because we have to update the physics bodies.
15. We have to use the `dynamicsWorld` and `stepSimulation` functions to update the simulation every frame. To do this, we have to calculate the delta time between the previous and current frames.
16. At the top of the `source.cpp`, include `<chrono>` so that we can calculate the tick update. Now, we have to make changes to the main function and the `while` loop, as follows:

```
auto previousTime = std::chrono::high_resolution_clock::now();

while (!glfwWindowShouldClose(window)) {

    auto currentTime =
std::chrono::high_resolution_clock::now();
    float dt = std::chrono::duration<float,
std::chrono::seconds::period>(currentTime - previousTime).count();

    dynamicsWorld->stepSimulation(dt);

    renderScene();

    glfwSwapBuffers(window);
    glfwPollEvents();

    previousTime = currentTime;
}
```

Just before the `while` loop, we create a variable called `previousTime` and initialize it with the current time. In the `while` loop, we get the current time and store it in the variable. Then, we calculate the delta time between the previous time and the current time by subtracting the two. We have the delta time now, so we call the `stepSimulation` and pass in the delta time. Then we render the scene, and swap the buffer and poll for events as usual. Finally, we set the current time as the previous time.

Now, when we run the project, we can see the sphere falling down, which is pretty cool. However, the sphere doesn't interact with anything.

Let's add a box rigid body at the bottom and watch the sphere bounce off of it. After the sphere `MeshRenderer` object, add the following code to create a box rigid body:

```
btCollisionShape* groundShape = new btBoxShape(btVector3(4.0f,
0.5f, 4.0f));
btDefaultMotionState* groundMotionState = new
btDefaultMotionState(btTransform(btQuaternion(0, 0, 0, 1),
btVector3(0, -2.0f, 0)));
btRigidBody::btRigidBodyConstructionInfo groundRigidBodyCI(0.0f,
new btDefaultMotionState(), groundShape, btVector3(0, 0, 0));

btRigidBody* groundRigidBody = new
btRigidBody(groundRigidBodyCI);
groundRigidBody->setFriction(1.0);
groundRigidBody->setRestitution(0.9);
groundRigidBody->setCollisionFlags(btCollisionObject::CF_STATIC_OBJECT);

dynamicsWorld->addRigidBody(groundRigidBody);
```

Here, we first create a shape of the `btBoxShape` type with the length, height, and depth set as `4.0`, `0.5`, and `4.0`. Next, we will set the motion state, where we set the rotation to zero and set the position at `-2.0` in the `y` axis and `0` along the `x` and `z` axis. For the construction information, we set the mass and inertia to `0`. We also set the default motion state and pass in the shape. Next, we create the rigid body by passing the rigid body information into it. Once the rigid body is created, we set the restitution and friction value. Next, we use the `setCollisionFlags` function of `rigidBody` to set the rigid body type as static. This means that it will be like a brick wall and won't move and get affected by forces from other rigid bodies, but other bodies will be affected by it.

Finally, we add the ground rigid body to the world so that the box rigid body will be part of the physics simulation as well. We now have to create a `MeshRenderer` cube to render the ground rigid body. Create a new `MeshRenderer` object called `Ground` at the top, under which you created the sphere `MeshRenderer` object. In the `init` function, under which we added the code for ground rigid body, add the following:

```
// Ground Mesh
GLuint groundTexture =
tLoader.getTextureID("Assets/Textures/ground.jpg");
ground = new MeshRenderer(MeshType::kCube, camera,
groundRigidBody);
ground->setProgram(texturedShaderProgram);
ground->setTexture(groundTexture);
ground->setScale(glm::vec3(4.0f, 0.5f, 4.0f));
```

We will create a new texture by loading `ground.jpg`, so make sure you add it to the `Assets/ Textures` directory. Call the constructor and set the `meshtype` to `cube`, and then set the `camera` and pass in the ground rigid body. We then set the shader program, texture, and scale of the object.

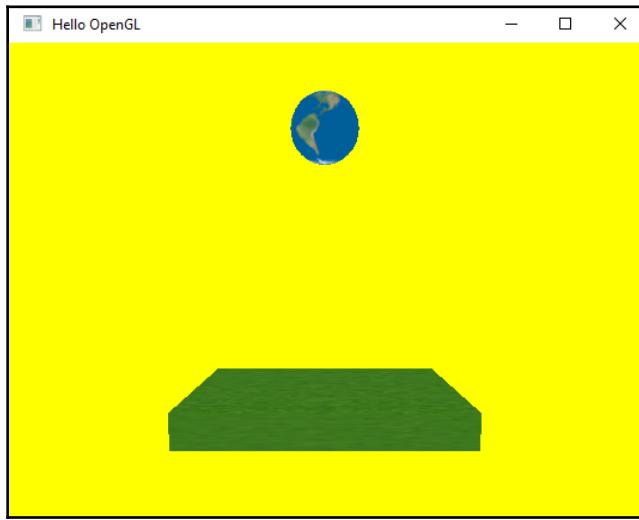
17. In the `renderScene` function, draw the ground `MeshRenderer` object as follows:

```
void renderScene() {

    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glClearColor(1.0, 1.0, 0.0, 1.0);

    sphere->draw();
    ground->draw();
}
```

18. Now, when you run the project, you will see the sphere bouncing on the ground box:



Summary

In this chapter, we created a new class called `MeshRenderer`, which will be used to render textured 3D objects to our scene. We created a texture-loaded class, which will be used to load the textures from the images provided. Then, we added physics to the object by adding the Bullet Physics library. We then initialized the physics world and created and added the rigid body to the mesh renderer by adding the body itself to the world, causing the rendered object to be affected by physics.

In the next chapter, we will add a Gameplay loop, as well as scoring and text rendering to display the score on the viewport. We will also add lighting to our world.

8

Enhancing Your Game with Collision, Loop, and Lighting

In this chapter, we will see how to add collision to detect contact between the ball and the enemy to determine the lose condition, and also check contact between the ball and the ground to check whether the player can jump or not. We will then finalize the gameplay loop.

Once the gameplay loop is complete, we can add in text rendering to show the player the score. For displaying the text, we will use the library FreeType for loading the characters from the font file.

We will also add in some basic lighting to the objects in the scene. Lighting will be calculated using the Phong lighting model, and we will cover how this is implemented in practice. To finish the gameplay loop, we have to add an enemy as well.

The topics covered in this chapter are as follows:

- Adding a `RigidBody` name
- Adding an enemy
- Moving the enemy
- Checking collision
- Adding keyboard controls
- Gameloop and scoring
- Text rendering
- Adding lighting

Adding a RigidBody name

To identify the different rigid bodies we are adding to the scene, we will add a property in the `MeshRenderer` class to specify each object with the name of the object being rendered, so we will add the property as well. Let's look at how to do this:

1. In the `MeshRenderer.h` class within the `MeshRenderer` class, change the constructor of the class to take in a string as the name for the object, as follows:

```
MeshRenderer(MeshType modelType, std::string _name, Camera *  
_camera, btRigidBody* _rigidBody)
```

2. Add a new public property called the name of the type `std::string` and initialize it as demonstrated as follows:

```
std::string name = "";
```

3. Next, in the `MeshRenderer.cpp` file, modify the constructor implementation, as follows:

```
MeshRenderer::MeshRenderer(MeshType modelType, std::string _name,  
Camera* _camera, btRigidBody* _rigidBody){  
    name = _name;  
    ...  
    ...  
}
```

4. We have now added the name property in the `MeshRenderer` class.

Adding an enemy

Before we add an enemy to the scene, let's clean up our code a little bit and create a new function called `addRigidBodies` in `main.cpp`, so that all the rigid bodies will be created in a single function. To do so, follow the given steps:

1. In the source of `main.cpp`, create a new function called `addRigidBodies` above the main function

2. Add the following in the `addRigidBodies` function for adding the sphere and ground, instead of putting all the game code in the main function:

```
// Sphere Rigid Body

btCollisionShape* sphereShape = new btSphereShape(1);
btDefaultMotionState* sphereMotionState = new
btDefaultMotionState(btTransform(btQuaternion(0, 0, 0, 1),
btVector3(0, 0.5, 0)));
btScalar mass = 13.0f;
btVector3 sphereInertia(0, 0, 0);
sphereShape->calculateLocalInertia(mass, sphereInertia);

btRigidBody::btRigidBodyConstructionInfo sphereRigidBodyCI(mass,
sphereMotionState, sphereShape, sphereInertia);

btRigidBody* sphereRigidBody = new
btRigidBody(sphereRigidBodyCI);

sphereRigidBody->setFriction(1.0f);
sphereRigidBody->setRestitution(0.0f);

sphereRigidBody->setActivationState(DISABLE_DEACTIVATION);

dynamicsWorld->addRigidBody(sphereRigidBody);

// Sphere Mesh

sphere = new MeshRenderer(MeshType::kSphere, "hero", camera,
sphereRigidBody);
sphere->setProgram(texturedShaderProgram);
sphere->setTexture(sphereTexture);
sphere->setScale(glm::vec3(1.0f));

sphereRigidBody->setUserPointer(sphere);

// Ground Rigid body

btCollisionShape* groundShape = new btBoxShape(btVector3(4.0f,
0.5f, 4.0f));
btDefaultMotionState* groundMotionState = new
btDefaultMotionState(btTransform(btQuaternion(0, 0, 0, 1),
btVector3(0, -1.0f, 0)));

btRigidBody::btRigidBodyConstructionInfo groundRigidBodyCI(0.0f,
groundMotionState, groundShape, btVector3(0, 0, 0));
```

```
btRigidBody* groundRigidBody = new  
btRigidBody(groundRigidBodyCI);  
  
groundRigidBody->setFriction(1.0);  
groundRigidBody->setRestitution(0.0);  
  
groundRigidBody->setCollisionFlags(btCollisionObject::CF_STATIC_OBJECT);  
  
dynamicsWorld->addRigidBody(groundRigidBody);  
  
// Ground Mesh  
ground = new MeshRenderer(MeshType::kCube, "ground", camera,  
groundRigidBody);  
ground->setProgram(texturedShaderProgram);  
ground->setTexture(groundTexture);  
ground->setScale(glm::vec3(4.0f, 0.5f, 4.0f));  
  
groundRigidBody->setUserPointer(ground);
```

Notice that some of the values have been changed to suit our game. We have also disabled deactivation on the sphere because, if we don't, then the sphere will be unresponsive when we want it to jump for us, as sometimes the sphere won't jump.

To access the name of the rendered mesh, we can set this instance as a property of the rigid body by using the `setUserPointer` property of the `RigidBody` class. The `setUserPointer` takes a void pointer so any kind of data can be passed into it. For the sake of convenience, we are just passing the instance of the `MeshRenderer` class itself. In this function, we will also add the enemy rigid body to the scene, as follows:

```
// Enemy Rigid body  
  
btCollisionShape* shape = new btBoxShape(btVector3(1.0f, 1.0f,  
1.0f));  
btDefaultMotionState* motionState = new  
btDefaultMotionState(btTransform(btQuaternion(0, 0, 0, 1),  
btVector3(18.0, 1.0f, 0)));  
btRigidBody::btRigidBodyConstructionInfo rbCI(0.0f, motionState,  
shape, btVector3(0.0f, 0.0f, 0.0f));  
  
btRigidBody* rb = new btRigidBody(rbCI);  
  
rb->setFriction(1.0);  
rb->setRestitution(0.0);
```

```
//rb->setCollisionFlags(btCollisionObject::CF_KINEMATIC_OBJECT);  
  
rb->setCollisionFlags(btCollisionObject::CF_NO_CONTACT_RESPONSE);  
  
dynamicsWorld->addRigidBody(rb);  
  
// Enemy Mesh  
enemy = new MeshRenderer(MeshType::kCube, "enemy", camera, rb);  
enemy->setProgram(texturedShaderProgram);  
enemy->setTexture(groundTexture);  
enemy->setScale(glm::vec3(1.0f, 1.0f, 1.0f));  
  
rb->setUserPointer(enemy);
```

3. Add the enemy in the same way that we added the sphere and the ground. Since the shape of the enemy object is a cube, we use the `btBoxShape` to set the shape of the box for the rigid body. We set the location at 18 units in the x axis and at one unit's distance in the y axis. We then set the friction and restitution values.

For the type of rigid body, we set its collision flag to `NO_CONTACT_RESPONSE` instead of `KINEMATIC_OBJECT`. This is because we could set the type of `KINEMATIC_OBJECT`, but then the enemy object would exert force on other objects, such as the sphere, when it comes in contact with it. To avoid this, we use `NO_CONTACT_RESPONSE`, which will just check if there was an overlap between the enemy rigid body and another body, instead of applying force to it.

You can uncomment the `KINEMATIC_OBJECT` line of code and comment the `NO_CONTACT_RESPONSE` to see how using either changes the way the object behaves in the physics simulation.

4. Once we have created the rigid body, we add the rigid body to the world, set the mesh renderer for the enemy object, and give it the name `enemy`.

Moving the enemy

To update the enemy movement, we will add a tick function that will be called by the rigid body world. In this tick function, we will update the position of the enemy so that the enemy cube moves from the right of the screen to the left. We will also check if the enemy has gone beyond the screen on the left.

If it has, then we will reset its position to the right of the screen. To do so, follow these steps:

1. In this update function, we will also update our gameplay logic, scoring, and checking for contact between the sphere and the enemy and the sphere and the ground. Add the tick function callback prototype at the top of the `Main.cpp` file, as follows:

```
void myTickCallback(btDynamicsWorld *dynamicsWorld, btScalar  
timeStep);
```

2. In the `TickCallback` function, we update the position of the enemy, as follows:

```
void myTickCallback(btDynamicsWorld *dynamicsWorld, btScalar  
timeStep) {  
  
    // Get enemy transform  
    btTransform t(enemy->rigidBody->getWorldTransform());  
  
    // Set enemy position  
    t.setOrigin(t.getOrigin() + btVector3(-15, 0, 0) *  
    timeStep);  
  
    // Check if offScreen  
    if(t.getOrigin().x() <= -18.0f) {  
        t.setOrigin(btVector3(18, 1, 0));  
    }  
    enemy->rigidBody->setWorldTransform(t);  
    enemy->rigidBody->getMotionState()->setWorldTransform(t);  
}
```

In the `myTickCallback` function, we first get the current transform and store it in a variable `t`. We then set the origin, which is the position of the transform, by getting the current position, moving it 15 units to the left, and multiplying it by the current timestep (which is the difference between the previous and current time).

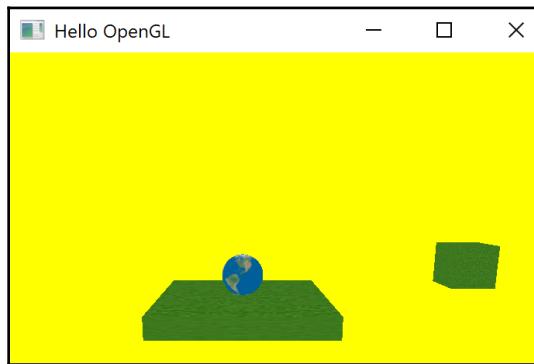
Once we get the updated location, we check that the current location is less than 18 units. If it is, then the current location is beyond the screen bounds on the left of the screen. Consequently, we set the current location back to the right of the viewport and make the object wrap around the screen.

We then update the location of the object itself to this new location by updating the `worldTransform` of the rigid body and the `motionstate` of the object.

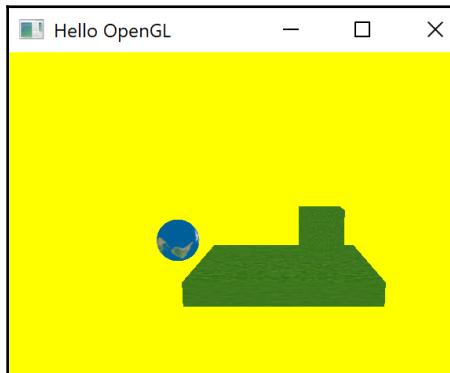
3. Set the tick function as the default TickCallback of the dynamic world in the init function, as shown as follows:

```
dynamicsWorld = new btDiscreteDynamicsWorld(dispatcher, broadphase,
solver, collisionConfiguration);
dynamicsWorld->setGravity(btVector3(0, -9.8f, 0));
dynamicsWorld->setInternalTickCallback(myTickCallback);
```

4. Build and run the project to see the cube enemy spawn at the right of the screen, followed by passing through the sphere and moving toward the left of the screen. When the enemy goes offscreen, it will get looped around to the right of the screen, as shown in the following screenshot:



5. If we set the collisionFlag of the enemy to KINEMATIC_OBJECT, you will see that the enemy doesn't go through the sphere but pushes it off the ground, as follows:



6. This is not what we want, so change the collision flag of the enemy back to NO_CONTACT_RESPONSE, as we don't want the enemy to physically interact with any object.

Checking collision

In the tick function, let's check for collision between the sphere and the enemy as well as the sphere and the ground. Follow the given steps:

1. To check the number of contacts between objects we will use the getNumManifolds property of the dynamic world object. The manifold will have information regarding all the contacts in the scene per update cycle.
2. We will check whether the number of contacts is greater than zero. If it is, then we check which pairs of objects were in contact with each other. After updating the enemy object, add the following code to check for contact between hero and enemy:

```
int numManifolds =
dynamicsWorld->getDispatcher()->getNumManifolds();

for (int i = 0; i < numManifolds; i++) {

    btPersistentManifold *contactManifold = dynamicsWorld-
    >getDispatcher()->getManifoldByIndexInternal(i);

    int numContacts = contactManifold->getNumContacts();

    if (numContacts > 0) {

        const btCollisionObject *objA = contactManifold-
        >getBody0();
        const btCollisionObject *objB = contactManifold-
        >getBody1();

        MeshRenderer* gModA = (MeshRenderer*)objA-
        >getUserPointer();
        MeshRenderer* gModB = (MeshRenderer*)objB-
        >getUserPointer();

        if ((gModA->name == "hero" && gModB->name ==
            "enemy") ||
            (gModA->name == "enemy" && gModB->name
            == "hero")) {
```

```
        printf("collision: %s with %s \n",
gModA->name, gModB->name);

        if (gModB->name == "enemy") {
            btTransform b(gModB->rigidBody-
>getWorldTransform());
            b.setOrigin(btVector3(18, 1, 0));
            gModB->rigidBody-
>setWorldTransform(b);
            gModB->rigidBody->
            getMotionState()->
            setWorldTransform(b);
        } else {
            btTransform a(gModA->rigidBody-
>getWorldTransform());
            a.setOrigin(btVector3(18, 1, 0));
            gModA->rigidBody-
>setWorldTransform(a);
            gModA->rigidBody->
            getMotionState()->
            setWorldTransform(a);
        }
    }

    if ((gModA->name == "hero" && gModB->name ==
"ground") ||
        (gModA->name == "ground" && gModB->name
== "hero")) {
        printf("collision: %s with %s \n",
gModA->name, gModB->name);

    }
}
```

3. We will first get the number of contact manifolds or contact pairs. Then, for each contact manifold, we check whether the number of contacts is greater than zero. If it is greater than zero, then it means there has been a contact in the current update.

4. We then get both the collision objects and assign them to `ObjA` and `ObjB`. After this, we get the user pointer for both the objects, and type caste it to `MeshRenderer` to access the name of the objects we assigned. When checking for contact between two objects, object A can be in contact with object B or the other way around. If there has been contact between the sphere and the enemy, we set the position of the enemy back to the right of the viewport. We also check the contact between the sphere and the ground. If there is contact, we just print out that there has been contact.

Adding keyboard controls

Let's add keyboard controls to interact with the sphere. We will set it so that when we press the up key on the keyboard, the sphere jumps. We will add the jump feature by applying an impulse on the sphere. To do so take the following steps:

1. We use `GLFW`, which has a keyboard callback function so that we can add interaction with the keyboard for the game. Before the main function, we will set the function shown, as follows:

```
void updateKeyboard(GLFWwindow* window, int key, int scancode, int action, int mods){  
  
    if (glfwGetKey(window, GLFW_KEY_ESCAPE) == GLFW_PRESS) {  
        glfwSetWindowShouldClose(window, true);  
    }  
  
    if (key == GLFW_KEY_UP && action == GLFW_PRESS) {  
        if (grounded == true) {  
            grounded = false;  
        sphere->rigidBody->applyImpulse(btVector3(0.0f, 100.0f, 0.0f),  
        btVector3(0.0f, 0.0f, 0.0f));  
            printf("pressed up key \n");  
        }  
    }  
}
```

The two main parameters that we are concerned with are the key and action. With key, we get which key is pressed and with action, we can retrieve what action was performed on that key. In the function, we first check whether the escape key was pressed using the `glfwGetKey` function. If so, then we close the window using the `glfwSetWindowShouldClose` function by passing true as the second parameter.

To make the sphere jump, we check whether the up key was pressed. If it was, we then create a new Boolean member variable called `grounded`, which describes a state if the sphere is touching the ground. If this is true, we set the Boolean value to `false` and apply an impulse of 100 units on the sphere's rigid body origin in the Y direction by calling the `applyImpulse` function of the `rigidbody`.

2. In the tick function, before we get the number of manifolds, we set the `grounded` Boolean to false, as follows:

```
grounded = false;

int numManifolds =
dynamicsWorld->getDispatcher()->getNumManifolds();
```

3. We set the `grounded` Boolean value to true when there is contact between the sphere and the ground, as follows:

```
if ((gModA->name == "hero" && gModB->name == "ground") ||
(gModA->name == "ground" && gModB->name == "hero")) {

//printf("collision: %s with %s \n", gModA->name, gModB->name);

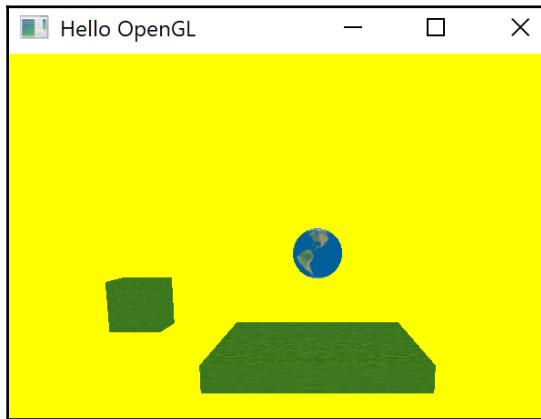
grounded = true;

}
```

4. In the main function, set the `updateKeyboard` as the callback using `glfwSetKeyCallback`, as follows:

```
int main(int argc, char **argv) {
...
    glfwMakeContextCurrent(window);
    glfwSetKeyCallback(window, updateKeyboard);
...
}
```

- Now build and run the application. Press the up key to see the sphere jump, only when it is grounded as follows:



Game loop and scoring

Let's now finish up by adding scoring and finishing the game loop.

- Along with the grounded Boolean, add another Boolean and check for gameover. After doing this, add an int called score and initialize it to 0 at the top of the `main.cpp` file, as follows:

```
GLuint sphereTexture, groundTexture;  
  
bool grounded = false;  
bool gameover = true;  
int score = 0;
```

- Next, in the tick function, the enemy should only move when the game is not over, so wrap the updating in an if condition to check whether or not the game is over. If the game is not over, then we update the position of the enemy, as follows:

```
void myTickCallback(btDynamicsWorld *dynamicsWorld, btScalar  
timeStep) {  
  
    if (!gameover) {  
  
        // Get enemy transform  
        btTransform t(enemy->rigidBody->getWorldTransform());  
    }  
}
```

```
// Set enemy position

t.setOrigin(t.getOrigin() + btVector3(-15, 0, 0) *
timeStep);

// Check if offScreen

if (t.getOrigin().x() <= -18.0f) {

    t.setOrigin(btVector3(18, 1, 0));
    score++;
    label->setText("Score: " + std::to_string(score));

}

enemy->rigidBody->setWorldTransform(t);
enemy->rigidBody->getMotionState()->setWorldTransform(t);
}

...
}
```

3. We also increment the score if the enemy goes beyond the left of the screen. Still in the tick function, if there is contact between the sphere and the enemy, we set the score to 0 and set gameover to true, as follows:

```
if ((gModA->name == "hero" && gModB->name == "enemy") ||
(gModA->name == "enemy" && gModB->name ==
"hero")) {

    if (gModB->name == "enemy") {
        btTransform b(gModB->rigidBody-
>getWorldTransform());
        b.setOrigin(btVector3(18, 1, 0));
        gModB->rigidBody-
>setWorldTransform(b);
        gModB->rigidBody->getMotionState()->
>setWorldTransform(b);
    }else {
        btTransform a(gModA->rigidBody-
>getWorldTransform());
        a.setOrigin(btVector3(18, 1, 0));
        gModA->rigidBody-
>setWorldTransform(a);
        gModA->rigidBody->getMotionState()->
>setWorldTransform(a);
    }
}
```

```
        gameover = true;
        score = 0;

    }
```

4. In the update keyboard function, when the up keyboard key is pressed, we first check if the game is over. If it is, we set gameover Boolean to false, which will start the game. Now, when the player presses the up key again, the character will jump. This way, the same key can be used for starting the game and also making the character jump.
5. Make the changes to the updateKeyboard function, as follows:

```
void updateKeyboard(GLFWwindow* window, int key, int scancode, int action, int mods){

    if (glfwGetKey(window, GLFW_KEY_ESCAPE) == GLFW_PRESS) {
        glfwSetWindowShouldClose(window, true);
    }

    if (key == GLFW_KEY_UP && action == GLFW_PRESS) {

        if (gameover) {
            gameover = false;
        } else {
            if (grounded == true) {

                grounded = false;
                sphere->rigidBody->applyImpulse(btVector3(0.0f, 100.0f, 0.0f),
                btVector3(0.0f, 0.0f, 0.0f));
                printf("pressed up key \n");
            }
        }
    }
}
```

6. Although we are calculating the score, the user still cannot see what the score is, so let's add text rendering to the game next.

Text rendering

For rendering text, we will use the library called FreeType, load the font, and read the characters from it. FreeType can load the popular font format called TrueType. TrueType fonts have the extension .ttf.

TTFs contain vector information called glyphs and that they can be just used to store any data but one use case is of course to represent characters with them.

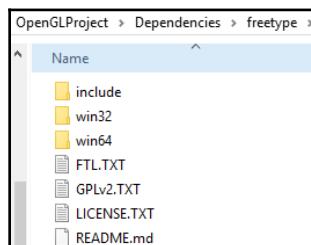
So, when we want to render a particular glyph, we load the character glyph by specifying the size, and the character will be generated without there being a loss in quality.



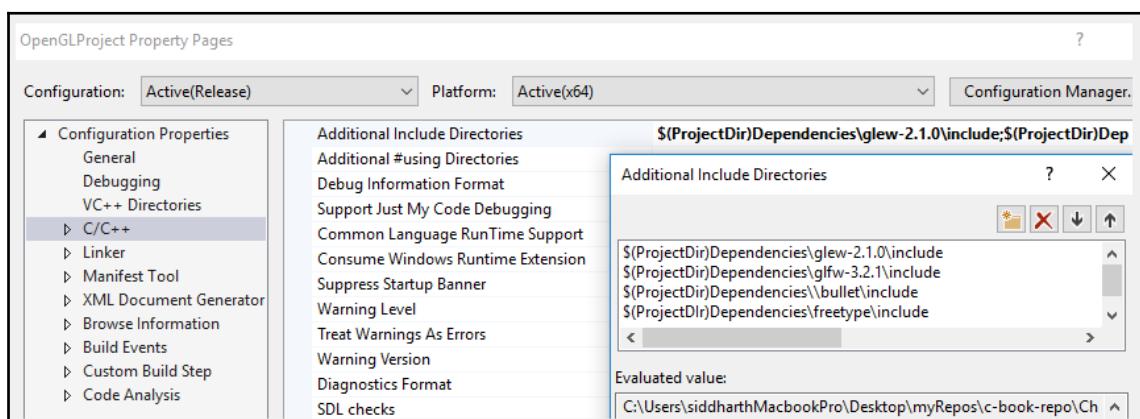
The source can be downloaded from their website here: <https://www.freetype.org/> and the library can be built from it. The precompiled libraries can also be downloaded from here: <https://github.com/ubawurinna/freetype-windows-binaries>.

Let us add the library to our project. Since we are developing for the 64 bit OS, we are interested in the include directory and the win64 directory, as it has the `freetype.lib` and `freetype.dll` for our version of the project.

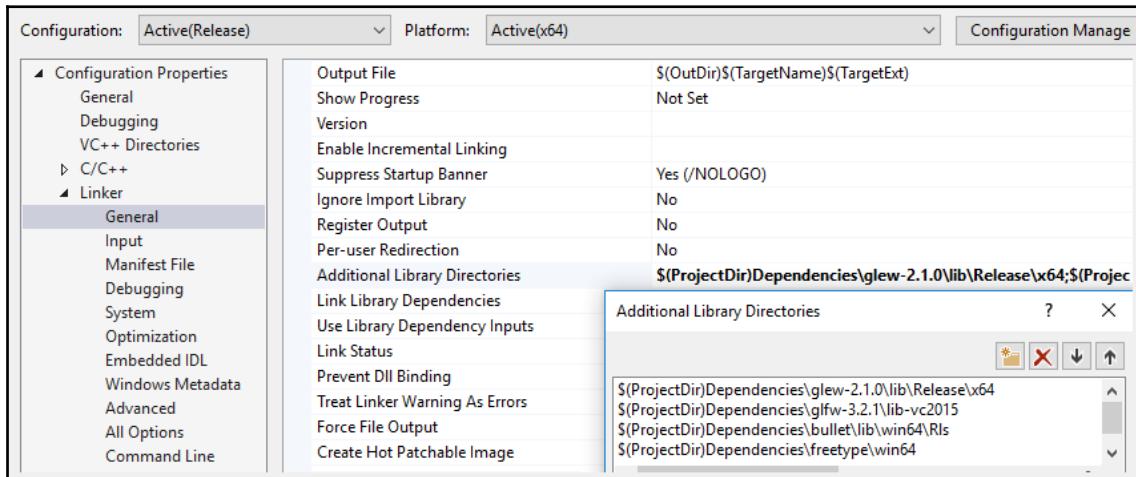
1. Create a folder called `freetype` in your dependencies folder and extract the files into it, as follows:



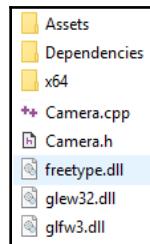
2. Open the project properties and, under **C/C++** in **Additional Include Directory**, add the `freetype` include directory location, as follows:



3. Under Configuration Properties | Linker | General | Additional Library Directories, add the freetype win64 directory, as follows:



4. In the project directory, copy the Freetype.dll from the win64 directory and paste it here:



With the prep work out of the way, we can start working on the project now.

5. Create a class called `TextRenderer`, as well as a file named `TextRenderer.h` and a file named `TextRenderer.cpp`. We will add the functionality to text rendering in these files. In `TextRenderer.h`, first include the usual include headers for GL and glm as b, as follows:

```
#include <GL/glew.h>

#include "Dependencies/glm/glm/glm.hpp"
#include "Dependencies/glm/glm/gtc/matrix_transform.hpp"
#include "Dependencies/glm/glm/gtc/type_ptr.hpp"
```

6. Next, we will include the headers for `freetype.h`, as follows:

```
#include <ft2build.h>
#include FT_FREETYPE_H
```

7. The `FT_FREETYPE_H` macro just includes `freetype.h` in the `freetype` directory. Then, we will include `<map>` as we will have to map each character's location, size, and other information. We will also include `<string>` and pass a string into the class to be rendered, as follows:

```
#include <string>
```

8. For each glyph, we will need to keep track of certain properties. For this, we create a struct called `Character`, as follows:

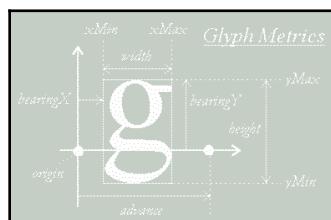
```
struct Character {
    GLuint TextureID; // Texture ID of each glyph texture
    glm::ivec2 Size; // glyph Size
    glm::ivec2 Bearing; // baseline to left/top of glyph
    GLuint Advance; // id to next glyph
};
```

For each glyph, we will store the texture ID of the texture we create for each character. We store the size of it, the bearing, which is the distance from the top left corner of the glyph from the baseline of the glyph, and the ID of the next glyph in the font file.

9. This is how a font file looks with all the character glyphs in it:



Information regarding each character is stored in relation to the character adjacent to it, as follows:



Each of these properties can be accessed on a per glyph basis after loading the font face of type `FT_Face`. The width and height of each glyph can be accessed using the `glyph` property per font face `face->glyph` as `face->glyph->bitmap.width` and `face->glyph->bitmap.rows`.

The image data is available per glyph using the `bitmap.buffer` property, which we will be using when creating the texture for each glyph as the following code shows how all of this is implemented.

The next glyph in the font file can be accessed using the `advance.x` property of the `glyph` if the font is horizontally aligned.

That is enough theory about the library. If you are interested in more information, the documentation is available at FreeType's website at the following link: <https://www.freetype.org/freetype2/docs/tutorial/step2.html#section-1>.

Let's continue with the `TextRenderer.h` file and create the `TextRenderer` class. In the file, create the class as shown as follows:

```
class TextRenderer{  
  
public:  
    TextRenderer(std::string text, std::string font, int size, glm::vec3  
color, GLuint program);  
    ~TextRenderer();  
  
    void draw();  
    void setPosition(glm::vec2 _position);  
    void setText(std::string _text);  
  
private:  
    std::string text;  
    GLfloat scale;  
    glm::vec3 color;  
    glm::vec2 position;  
  
    GLuint VAO, VBO, program;  
    std::map<GLchar, Character> Characters;  
};
```

In the class under the public section, we add the constructor and destructor. In the constructor, we pass in the string we want to draw, the file we want to use, the size and the color of the text we want to draw in, and we pass in a shader program to use while drawing the font.

Then, we have the `draw` function to draw the text, a couple of setters to set the position, and a `setText` function to set a new string to draw if needed. In the private section, we have local variables for text string, scale, color, and position. We also have member variables for VAO, VBO, and program to draw the text string. At the end of the class, we create a map to store all the loaded characters and assign each `GLchar` to a character `struct` in the map. This is all we need to do for the `TextRenderer.h` file.

In the `TextRenderer.cpp` file, include the `TextRenderer.h` file at the top of the file and take the following steps:

1. Add the `TextRenderer` constructor implementation as follows:

```
TextRenderer::TextRenderer(std::string text, std::string font, int  
size, glm::vec3 color, GLuint program)  
{  
}
```

In the constructor, we will add the functionality to load all the characters and prep the class for drawing the text.

2. We first initialize the local variables, as follows:

```
this->text = text;  
this->color = color;  
this->scale = 1.0;  
this->program = program;  
this->setPosition(position);
```

3. Next, the projection matrix is set. For text, we specify the orthographic projection as it doesn't have any depth, as follows:

```
glm::mat4 projection = glm::ortho(0.0f,  
static_cast<GLfloat>(800), 0.0f, static_cast<GLfloat>(600));  
glUseProgram(program);  
glUniformMatrix4fv(glGetUniformLocation(program, "projection"),  
1, GL_FALSE, glm::value_ptr(projection));
```

The projection is created using the `glm::ortho` function, which takes the origin x, window width, origin y, and window height only as the parameters for creating the orthographic projection matrix. We will use the current program and pass the value for the projection matrix to a location called `projection` to the shader. Since this value will never change, it is called and assigned once in the constructor.

4. Before we load the font itself, we have to initialize the FreeType library itself, as follows:

```
// FreeType
FT_Library ft;

// Initialise freetype
if (FT_Init_FreeType(&ft))
    std::cout << "ERROR::FREETYPE: Could not init FreeType Library" <<
    std::endl;
```

5. Now we can load the font face itself as follows:

```
// Load font
FT_Face face;
if (FT_New_Face(ft, font.c_str(), 0, &face))
    std::cout << "ERROR::FREETYPE: Failed to load font" <<
    std::endl;
```

6. Now set the font size in pixels and disable byte alignment restriction as follows. If we don't restrict the bite alignment, the font will be drawn jumbled, so don't forget to add the line:

```
// Set size of glyphs
FT_Set_Pixel_Sizes(face, 0, size);

// Disable byte-alignment restriction
glPixelStorei(GL_UNPACK_ALIGNMENT, 1);
```

7. We will then load the first 128 characters in the font we loaded and create and assign texture ID, size, bearing, and advance, and store it in the characters map as follows:

```
for (GLubyte i = 0; i < 128; i++) {

    // Load character glyph
    if (FT_Load_Char(face, i, FT_LOAD_RENDER)){
        std::cout << "ERROR::FREETYTPPE: Failed to load
        Glyph"
        <<
        std::endl;
        continue;
    }

    // Generate texture
    GLuint texture;
    glGenTextures(1, &texture);
```

```
glBindTexture(GL_TEXTURE_2D, texture);
glTexImage2D(
    GL_TEXTURE_2D,
    0,
    GL_RED,
    face->glyph->bitmap.width,
    face->glyph->bitmap.rows,
    0,
    GL_RED,
    GL_UNSIGNED_BYTE,
    face->glyph->bitmap.buffer
);

// Set texture filtering options
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S,
GL_CLAMP_TO_EDGE);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T,
GL_CLAMP_TO_EDGE);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,
GL_LINEAR);
// Create a character
Character character = {
    texture,
    glm::ivec2(face->glyph->bitmap.width,
               face->glyph->bitmap.rows),
    glm::ivec2(face->glyph->bitmap_left,
              face->glyph->bitmap_top),
    face->glyph->advance.x
};
// Store character in characters map
Characters.insert(std::pair<GLchar, Character>(i,
character));
}
```

- Once the characters are loaded, we can unbind the texture and destroy the font face and FreeType library, as follows:

```
glBindTexture(GL_TEXTURE_2D, 0);

// Destroy FreeType once we're finished
FT_Done_Face(face);
FT_Done_FreeType(ft);
```

9. Each character will be drawn as a texture on a separate quad, so set the VAO/VBO for a quad, create a position attribute, and enable it, as follows:

```
glGenVertexArrays(1, &VAO);
glGenBuffers(1, &VBO);

 glBindVertexArray(VAO);
 glBindBuffer(GL_ARRAY_BUFFER, VBO);
 glBufferData(GL_ARRAY_BUFFER, sizeof(GLfloat) * 6 * 4, NULL,
 GL_DYNAMIC_DRAW);
 glEnableVertexAttribArray(0);
 glVertexAttribPointer(0, 4, GL_FLOAT, GL_FALSE, 4 *
 sizeof(GLfloat), 0);
```

10. We now unbind the VBO and VAO once done, as follows:

```
glBindBuffer(GL_ARRAY_BUFFER, 0);
 glBindVertexArray(0);
```

That's all for the constructor. Now we can move onto the draw function. Let's see how:

1. So, create the draw function implementation as follows:

```
void TextRenderer::draw() {
}
```

2. We will add the functionality to draw in this function. First, we get the position from where the text needs to start drawing, as follows:

```
glm::vec2 textPos = this->position;
```

3. Then, we have to enable blending. If we don't enable blending, the whole quad for the text will be colored instead of just the area where the text is present, as shown in the image on the left:



In the image on the left where the S is supposed to be, we see the whole quad colored in red, including the pixels where it is supposed to be transparent.

By enabling blending, we set the final color value at a pixel using the following equation:

$$Color_{final} = Color_{Source} * Alpha_{Source} + Color_{Destination} * 1 - Alpha_{Source}$$

Here source color and source alpha are the color and alpha value of the text at a certain pixel location and the destination color and alpha are the values of color and alpha at the color buffer.

In this example, since we draw the text later, the destination color will be yellow and the source color, which is the text, is red. The destination alpha value is 1.0 as the yellow color is opaque. For the text, if we take at the S glyph for example, within the S, which is the red area it is opaque, but it is transparent.

So, using this formula, let's calculate the final pixel color around the S where it is transparent, using the following equation:

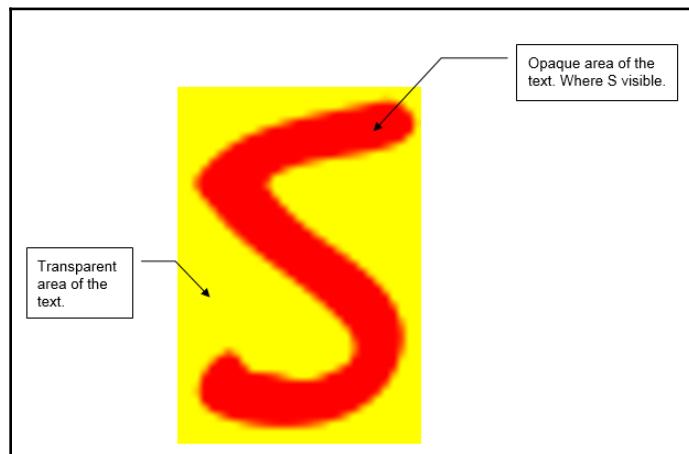
$$\begin{aligned} \text{Color}_{\text{final}} &= (1.0f, 0.0f, 0.0f, 0.0f) * 0.0 + (1.0f, 1.0f, 0.0f, 1.0f) * (1.0f - 0.0f) \\ &= (1.0f, 1.0f, 0.0f, 1.0f); \end{aligned}$$

This is just the yellow background color, which is why, when we enable blending, we see the red colored S instead of the whole quad looking red.

Conversely, within the S glyph, it is not transparent, so the alpha value is 1 at that pixel location. So, when we apply the same formula, we get the final color as follows:

$$\begin{aligned} \text{Color}_{\text{final}} &= (1.0f, 0.0f, 0.0f, 1.0f) * 1.0 + (1.0f, 1.0f, 0.0f, 1.0f) * (1.0f - 1.0f) \\ &= (1.0f, 0.0f, 0.0f, 1.0f) \end{aligned}$$

This is just the red text color, for example:



Let's see how this is implemented in practice.

4. The blend function to get blending enabled is shown as follows:

```
glEnable(GL_BLEND);
```

Then we set the source and destination blending factors, which is `GL_SRC_ALPHA`, so, for the source pixel, we use its alpha value as is and, for the destination, we set the alpha as `GL_ONE_MINUS_SRC_ALPHA`, which is the source alpha minus one, as follows:

```
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
```

By default, the value source and destination values are added. You could subtract, add, and divide as well.

5. We then call the `glUseProgram` function to set the program and set the text color to the uniform location, and set the default texture as follows:

```
glUseProgram(program);
glUniform3f(glGetUniformLocation(program, "textColor"),
this->color.x, this->color.y, this->color.z);
glActiveTexture(GL_TEXTURE0);
```

6. We now bind the VAO as follows:

```
glBindVertexArray(VAO);
```

7. We now go through all the characters in the text we want to draw and get the size, bearing to set the position, and the texture ID of each glyph we want to draw, as follows:

```
std::string::const_iterator c;

for (c = text.begin(); c != text.end(); c++) {

    Character ch = Characters[*c];

    GLfloat xpos = textPos.x + ch.Bearing.x * this->scale;
    GLfloat ypos = textPos.y - (ch.Size.y - ch.Bearing.y) *
        this->scale;

    GLfloat w = ch.Size.x * this->scale;
    GLfloat h = ch.Size.y * this->scale;
    // Per Character Update VBO
    GLfloat vertices[6][4] = {
        { xpos, ypos + h, 0.0, 0.0 },
        { xpos, ypos, 0.0, 1.0 },
        { xpos + w, ypos, 1.0, 1.0 },
        { xpos + w, ypos + h, 1.0, 0.0 },
        { xpos, ypos + h, 0.0, 1.0 },
        { xpos, ypos, 1.0, 1.0 }
    };
}
```

```

        { xpos, ypos + h, 0.0, 0.0 },
        { xpos + w, ypos, 1.0, 1.0 },
        { xpos + w, ypos + h, 1.0, 0.0 }
    };

    // Render glyph texture over quad
    glBindTexture(GL_TEXTURE_2D, ch.TextureID);
    // Update content of VBO memory
    glBindBuffer(GL_ARRAY_BUFFER, VBO);
    // Use glBufferSubData and not glBufferData
    glBufferSubData(GL_ARRAY_BUFFER, 0, sizeof(vertices),
    vertices);

    glBindBuffer(GL_ARRAY_BUFFER, 0);
    // Render quad
    glDrawArrays(GL_TRIANGLES, 0, 6);
    // Now advance cursors for next glyph (note that advance
    is number of 1/64 pixels)
    // Bitshift by 6 to get value in pixels (2^6 = 64 (divide
    amount of 1/64th pixels by 64 to get amount of pixels))
    textPos.x += (ch.Advance >> 6) * this->scale;
}

```

We will bind the one VBO and pass in the vertex data for all the quads to be drawn using `glBufferSubData`. Once bound, the quads are drawn using `glDrawArrays` and pass in 6 for the number of vertices to be drawn.

Then, we calculate the `textPos.x`, which will determine where the next glyph will be drawn. We get this distance by multiplying the advance of the current glyph by the scale and adding it to the current text position's `x` component. A bit shift of 6 is done to the advance, to get the value in pixels.

8. At the end of the draw function, we unbind the vertex array and the texture, and then disable blending as follows:

```

glBindVertexArray(0);
glBindTexture(GL_TEXTURE_2D, 0);

glDisable(GL_BLEND);

```

9. Finally, we add in the implementation of the `setPosition` and `setString` functions, as follows:

```

void TextRenderer::setPosition(glm::vec2 _position) {

    this->position = _position;
}

```

```
void TextRenderer::setText(std::string _text) {
    this->text = _text;
}
```

We are now done with the `TextRenderer` class. Let's now see how to display the text in our game.

1. In the `main.cpp`, include the `TextRenderer.h` at the top of the file and create a new object of the class called `label`, as follows:

```
#include "TextRenderer.h"

TextRenderer* label;
```

2. Create a new `GLuint` for the text shader program, as follows:

```
GLuint textProgram
```

3. And create the new shaded program for the text, as follows:

```
textProgram = shader.CreateProgram("Assets/Shaders/text.vs",
"Assets/Shaders/text.fs");
```

4. The `text.vs` and `text.fs` files are placed in the `Assets` directory under `Shaders`.`text.vs`, as follows:

```
#version 450 core
layout (location = 0) in vec4 vertex;
uniform mat4 projection;

out vec2 TexCoords;

void main(){
    gl_Position = projection * vec4(vertex.xy, 0.0, 1.0);
    TexCoords = vertex.zw;
}
```

We get the vertex position as an attribute and projection matrix as a uniform. The texture coordinate is set in the main function and is sent out to the next shader stage. The position of the vertex of the quad is set by multiplying the local coordinates with the orthographic projection matrix in the main function.

5. Next, we move on to the fragment shader, as follows:

```
#version 450 core

in vec2 TexCoords;

uniform sampler2D text;
uniform vec3 textColor;

out vec4 color;

void main(){
    vec4 sampled = vec4(1.0, 1.0, 1.0, texture(text, TexCoords).r);
    color = vec4(textColor, 1.0) * sampled;
}
```

We get the texture coordinate from the vertex shader and the texture and color as uniforms. A new `out vec4` is created called `color` to send out color information. In the `main` function, we create a new `vec4` called `sampled` and store the r,g, and b value to 1. We also store the red color as the alpha value to draw only the opaque part of the text. Then a new `vec4` called `color` is created, in which the white color is replaced with the color we want the text to be drawn in, and assign the `color` variable.

6. Let's continue with the text label implementation. After the `addRigidBody` function in the `init` function, initialize the `label` object, as follows:

```
label = new TextRenderer("Score: 0", "Assets/fonts/gooddog.ttf",
64, glm::vec3(1.0f, 0.0f, 0.0f), textProgram);
label->setPosition(glm::vec2(320.0f, 500.0f));
```

In the constructor, we set the string we want to render, pass in the location of the font file, and pass in the text height, the text color, and the text program. We then use the `setPosition` function to set the position of the text.

7. Next, in the `tick` function where we update the score, we update the text as well, as follows:

```
if (t.getOrigin().x() <= -18.0f) {

    t.setOrigin(btVector3(18, 1, 0));
    score++;
    label->setText("Score: " + std::to_string(score));
}
```

8. In the tick function, we reset the string when the game is over, as follows:

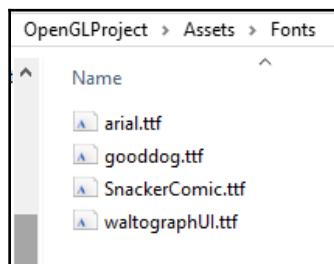
```
gameover = true;
score = 0;
label->setText("Score: " + std::to_string(score));
```

9. In the render function, we finally call the draw function to draw the text, as follows:

```
void renderScene(float dt){  
  
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);  
    glClearColor(1.0, 1.0, 0.0, 1.0);  
  
    // Draw Objects  
  
    //light->draw();  
  
    sphere->draw();  
    enemy->draw();  
    ground->draw();  
  
    label->draw();  
}
```

Because of alpha blending, the text has to be drawn at the end after drawing all the other objects.

10. Finally, make sure the font file is added in the Assets folder under Fonts, as follows:

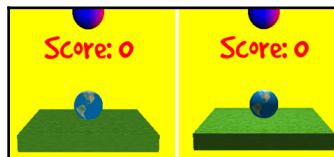


A few font files have been provided for you to experiment with. More free fonts can be downloaded from <https://www.1001freefonts.com/> and <https://www.dafont.com/>. Build and run the game to see the text getting drawn and updated; for example:



Adding lighting

Finally, let's add lighting to the objects in the scene just to make the objects more interesting to look at, by enabling the drawing of the light renderer in the scene. Now, the light is originating from the center of this sphere. Using the position of the light source, we will calculate if a pixel is lit or not; for example:



The picture on the left shows the scene unlit. In contrast, the right scene is lit with the earth sphere and the ground affected by the light source. The surface that is facing the light is brightest, for example, at the top of the sphere. This creates a **Specular** at the top of the sphere. As the surface is farther from or at an angle to the light source, those pixel values slowly diffuse. Then, there are surfaces that are not facing the light source at all, like the side of the ground facing us. However, they are still not completely black as they are still lit by the light from the source, which bounces around and becomes part of the ambient light. The **Ambient**, **Diffuse**, and **Specular** become major parts of the lighting model to light an object. Lighting models are used to simulate lighting in computer graphics as, unlike the real world, we are limited by the processing power of our hardware.

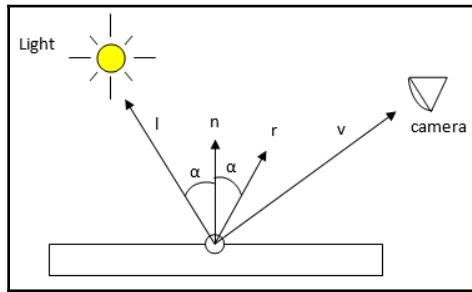
The formula for the final color of the pixel according to the Phong shading model is as follows:

$$C = k_a * L_c + L_c * \max(0, n \cdot l) + k_s * L_c * \max(0, v \cdot r)^p$$

Where,

- k_a is the ambient strength
- L_c is the light color
- n is the surface normal
- l is the light direction
- k_s is the specular strength
- v is the view direction
- r is the reflected light direction about the normal of the surface
- p is the Phong exponent, which will determine how shiny a surface is.

For n , l , v and r vectors refer to the diagram below.



Let's now look at how to implement this in practice.

1. All the lighting calculation is done in the fragment shader of the object, as it affects the final color of the object, depending upon the light source and camera position. For each object to be lit, we also need to pass in the light color, diffuse, and specular strength. In the `MeshRenderer.h` file, change the constructor to take the light source and diffuse and specular strengths, as follows:

```
MeshRenderer(MeshType modelType, std::string _name, Camera *_camera, btRigidBody* _rigidBody, LightRenderer* _light, float _specularStrength, float _ambientStrength);
```

2. Include the `LightRenderer.h` at the top of the file, as follows:

```
#include "LightRenderer.h"
```

3. In the private section of the class, add an object for the LightRenderer and floats to store the ambient and specular Strength, as follows:

```
GLuint vao, vbo, ebo, texture, program;
LightRenderer* light;
float ambientStrength, specularStrength;
```

4. In the MeshRenderer.cpp file, change the implementation of the constructor and assign the variables passed in to the local variables, as follows:

```
MeshRenderer::MeshRenderer(MeshType modelType, std::string _name,
    Camera* _camera, btRigidBody* _rigidBody, LightRenderer* _light,
    float _specularStrength, float _ambientStrength) {
    name = _name;
    rigidBody = _rigidBody;
    camera = _camera;
    light = _light;
    ambientStrength = _ambientStrength;
    specularStrength = _specularStrength;
    ...
}
```

5. In the constructor, we also need to add a new normal attribute as we will need the surface normal information for lighting calculations, as follows:

```
glEnableVertexAttribArray(0);
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, sizeof(Vertex),
(GLvoid*)0);

glEnableVertexAttribArray(1);
glVertexAttribPointer(1, 2, GL_FLOAT, GL_FALSE, sizeof(Vertex),
(void*)(offsetof(Vertex, Vertex::texCoords)));
glEnableVertexAttribArray(2);
glVertexAttribPointer(2, 3, GL_FLOAT, GL_FALSE, sizeof(Vertex),
(void*)(offsetof(Vertex, Vertex::normal))));
```

6. In the Draw function, we pass the camera position, light position, light color, specular strength, and ambient strength as uniforms to the shader, as follows:

```
// Set Texture
glBindTexture(GL_TEXTURE_2D, texture);

// Set Lighting
GLuint cameraPosLoc = glGetUniformLocation(program, "cameraPos");
glUniform3f(cameraPosLoc, camera->getCameraPosition().x,
camera->getCameraPosition().y, camera->getCameraPosition().z);
```

```

        GLuint lightPosLoc = glGetUniformLocation(program, "lightPos");
        glUniform3f(lightPosLoc, this->light->getPosition().x,
this->light->getPosition().y, this->light->getPosition().z);

        GLuint lightColorLoc = glGetUniformLocation(program,
"lightColor");
        glUniform3f(lightColorLoc, this->light->getColor().x,
this->light->getColor().y, this->light->getColor().z);

        GLuint specularStrengthLoc = glGetUniformLocation(program,
"specularStrength");
        glUniform1f(specularStrengthLoc, specularStrength);

        GLuint ambientStrengthLoc = glGetUniformLocation(program,
"ambientStrength");
        glUniform1f(ambientStrengthLoc, ambientStrength);

        glBindVertexArray(vao);
        glDrawElements(GL_TRIANGLES, indices.size(), GL_UNSIGNED_INT,
0);
        glBindVertexArray(0);
    }
}

```

7. We also need to create new vertex and fragment shaders for the effect to take place. We create a new vertex shader `LitTexturedModel.vs`, as follows:

```

#version 450 core
layout (location = 0) in vec3 position;
layout (location = 1) in vec2 texCoord;
layout (location = 2) in vec3 normal;

out vec2 TexCoord;
out vec3 Normal;
out vec3 fragWorldPos;

uniform mat4 vp;
uniform mat4 model;

void main() {

    gl_Position = vp * model *vec4(position, 1.0);
    TexCoord = texCoord;
    fragWorldPos = vec3(model * vec4(position, 1.0));
    Normal = mat3(transpose(inverse(model))) * normal;
}

```

8. We add the new location layout to receive the normal attribute.

9. Create a new out vec3 to send the normal information to the fragment shader. We also create a new out vec3 to send the world coordinates of a fragment. In the main function, we calculate the world position of the fragment by multiplying the local position with the world matrix and store it in the fragWorldPos variable. The normal is also converted to world space. Unlike how we multiplied local position, the model matrix to convert to world space normal needs to be treated differently. The normal is multiplied by the inverse of the model matrix and is stored in the normal variable. That is all for the vertex shader. Let's now look at the `LitTexturedModel.fs`.
10. In the fragment shader, we first get the texture coordinate, normal, and fragment world position. Next, we get the camera position, light position and color, specular and ambient strength uniforms, and the texture as uniform as well. The final pixel value will be stored in the out vec4 called color, as follows:

```
#version 450 core

in vec2 TexCoord;
in vec3 Normal;
in vec3 fragWorldPos;

uniform vec3 cameraPos;
uniform vec3 lightPos;
uniform vec3 lightColor;

uniform float specularStrength;
uniform float ambientStrength;

// texture
uniform sampler2D Texture;

out vec4 color;
```

11. In the main function of the shader, we add the lighting calculation as shown in the following formula:

```
void main() {
    vec3 norm = normalize(Normal);
    vec4 objColor = texture(Texture, TexCoord);

    /**ambient
    vec3 ambient = ambientStrength * lightColor;
    /**diffuse
    vec3 lightDir = normalize(lightPos - fragWorldPos);
    float diff = max(dot(norm, lightDir), 0.0);
    vec3 diffuse = diff * lightColor;
```

```

    //**specular
    vec3 viewDir = normalize(cameraPos - fragWorldPos);
    vec3 reflectionDir = reflect(-lightDir, norm);
    float spec = pow(max(dot(viewDir,
    reflectionDir),0.0),128);
    vec3 specular = specularStrength * spec * lightColor;
    // lighting shading calculation
    vec3 totalColor = (ambient + diffuse + specular) *
    objColor.rgb;

    color = vec4(totalColor, 1.0f);
}

```

12. We get the normal and object color first. Then, as per the formula equation, we first calculate the ambient part of the equation by multiplying the ambient strength and light color and store it in a `vec3` called `ambient`. For the diffuse part of the equation, we first calculate the light direction from the position of the pixel in the world space by subtracting the two positions. The resulting vector is normalized and saved in `vec3 lightDir`. We then get the dot product of the normal and light direction.
13. After this, we get the resultant value or the 0, whichever is bigger, and store it in `float diff`. `diff` is multiplied by the light color and stored in `vec3` to get the diffuse color. For the specular part of the equation, we calculate the view direction by subtracting the camera position from the fragment world position.
14. The resulting vector is normalized and stored in `vec3 specDir`. Then, the reflected light vector about the surface normal is calculated by using the `reflect glsl` intrinsic function and by passing in the `viewDir` and surface normal.
15. Then, the dot product of the view and reflected vector is calculated. The bigger value of the calculated value and 0 is chosen. The resulting float value is raised to the power of 128. The value can be from 0 to 256. The bigger the value, the shinier the object will appear. The specular value is calculated by multiplying the specular strength, the calculated `spec` value, and the light color stored in the specular `vec3`.
16. Then finally, the total shading is calculated by adding the three ambient, diffuse, and specular values together and then multiplying this by the object color. The object color is a `vec4` so we convert it to a `vec3`. The total color is assigned to the color variable by converting the `totalColor` to a `vec4`. For implementing in the project, create a new shader program called `litTexturedShaderProgram`, as follows:

```

GLuint litTexturedShaderProgram;
Create the shader program and assign it to it in the init function

```

```
in main.cpp.
    litTexturedShaderProgram =
        shader.CreateProgram("Assets/Shaders/LitTexturedModel.vs",
        "Assets/Shaders/LitTexturedModel.fs");

17. Finally, in the addRigidBody function, change the shaders for the sphere,
ground and enemy, as follows:

// Sphere Rigid Body

btCollisionShape* sphereShape = new btSphereShape(1);
btDefaultMotionState* sphereMotionState = new
btDefaultMotionState(btTransform(btQuaternion(0, 0, 0, 1),
btVector3(0, 0.5, 0)));

btScalar mass = 13.0f;
btVector3 sphereInertia(0, 0, 0);
sphereShape->calculateLocalInertia(mass, sphereInertia);

btRigidBody::btRigidBodyConstructionInfo sphereRigidBodyCI(mass,
sphereMotionState, sphereShape, sphereInertia);

btRigidBody* sphereRigidBody = new
btRigidBody(sphereRigidBodyCI);

sphereRigidBody->setFriction(1.0f);
sphereRigidBody->setRestitution(0.0f);

sphereRigidBody->setActivationState(DISABLE_DEACTIVATION);

dynamicsWorld->addRigidBody(sphereRigidBody);

// Sphere Mesh

sphere = new MeshRenderer(MeshType::kSphere, "hero", camera,
sphereRigidBody, light, 0.1f, 0.5f);
sphere->setProgram(litTexturedShaderProgram);
sphere->setTexture(sphereTexture);
sphere->setScale(glm::vec3(1.0f));

sphereRigidBody->setUserPointer(sphere);

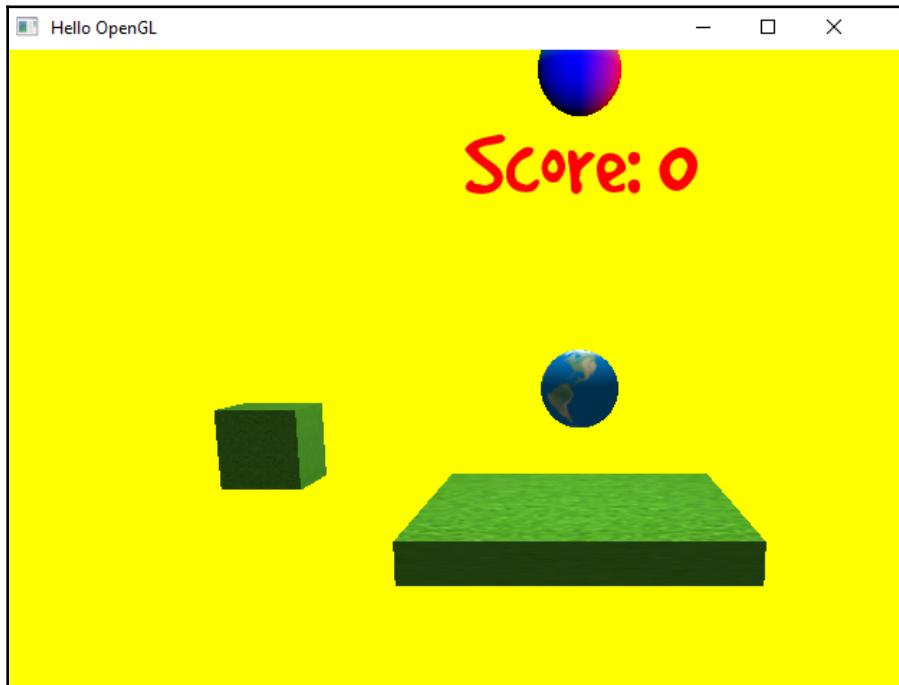
// Ground Rigid body

btCollisionShape* groundShape = new btBoxShape(btVector3(4.0f,
0.5f, 4.0f));
btDefaultMotionState* groundMotionState = new
btDefaultMotionState(btTransform(btQuaternion(0, 0, 0, 1),
```

```
btVector3(0, -1.0f, 0));  
  
btRigidBody::btRigidBodyConstructionInfo groundRigidBodyCI(0.0f,  
groundMotionState, groundShape, btVector3(0, 0, 0));  
  
btRigidBody* groundRigidBody = new  
btRigidBody(groundRigidBodyCI);  
  
groundRigidBody->setFriction(1.0);  
groundRigidBody->setRestitution(0.0);  
  
groundRigidBody->setCollisionFlags(btCollisionObject::CF_STATIC_OBJECT);  
  
dynamicsWorld->addRigidBody(groundRigidBody);  
  
// Ground Mesh  
ground = new MeshRenderer(MeshType::kCube, "ground", camera,  
groundRigidBody, light, 0.1f, 0.5f);  
ground->setProgram(litTexturedShaderProgram);  
ground->setTexture(groundTexture);  
ground->setScale(glm::vec3(4.0f, 0.5f, 4.0f));  
  
groundRigidBody->setUserPointer(ground);  
  
// Enemy Rigid body  
  
btCollisionShape* shape = new btBoxShape(btVector3(1.0f, 1.0f,  
1.0f));  
btDefaultMotionState* motionState = new  
btDefaultMotionState(btTransform(btQuaternion(0, 0, 0, 1),  
btVector3(18.0, 1.0f, 0)));  
btRigidBody::btRigidBodyConstructionInfo rbCI(0.0f, motionState,  
shape, btVector3(0.0f, 0.0f, 0.0f));  
  
btRigidBody* rb = new btRigidBody(rbCI);  
  
rb->setFriction(1.0);  
rb->setRestitution(0.0);  
  
//rb->setCollisionFlags(btCollisionObject::CF_KINEMATIC_OBJECT);  
  
rb->setCollisionFlags(btCollisionObject::CF_NO_CONTACT_RESPONSE);  
  
dynamicsWorld->addRigidBody(rb);  
  
// Enemy Mesh  
enemy = new MeshRenderer(MeshType::kCube, "enemy", camera, rb,
```

```
    light, 0.1f, 0.5f);  
    enemy->setProgram(litTexturedShaderProgram);  
    enemy->setTexture(groundTexture);  
    enemy->setScale(glm::vec3(1.0f, 1.0f, 1.0f));  
  
    rb->setUserPointer(enemy);
```

18. Build and run the project to see the lighting shader take effect; for example:



As an exercise, try adding a texture in the background as in the SFML game.

Summary

In this chapter, we saw how to add collision detection between the game objects, and then we finished the game loop by adding controls and scoring. Using the font loading library FreeType, we loaded the TrueType font in our game to add scoring text to the game, so that the user can see the score. Finally, to top it all off, we added lighting to the scene by adding the Phong lighting model to the objects by adding it to the shaders.

There is still a lot that can be added graphically to add more realism to the game, such as framebuffers to add post-processing effects. We could also add particles effects to add dust and rain effects. For additional learning, I would highly recommend learnopengl.com, as it is an amazing source for learning more about OpenGL.

In the next chapter, we will start exploring the Vulkan Rendering API and see how it is different from OpenGL.

4

Section 4: Rendering 3D Objects with Vulkan

Using the knowledge of 3D graphics programming that we gained in the previous section, we can now build on it to develop a basic project with Vulkan. OpenGL is a high-level graphics library. There are a lot things that OpenGL does in the background that the user is generally not aware of. With Vulkan, we will see the inner workings of a graphics library. We will see the why and the how of creating SwapChains, image views, renderpasses, Framebuffers, command buffers, as well as rendering and presenting objects to a scene.

The following chapters are in this section:

Chapter 9, Getting Started with Vulkan

Chapter 10, *Preparing the Clear Screen*

Chapter 11, *Creating Object Resources*

Chapter 12, *Drawing Vulkan Objects*

9

Getting Started with Vulkan

In the previous three chapters, we did our rendering using OpenGL. Although OpenGL is good for developing prototypes and getting your rendering going faster, it does have its weaknesses. For one thing, OpenGL is very driver-dependent, which makes it slower and less predictable when it comes to performance, which is why we prefer Vulkan for rendering.

In this chapter, we will cover the following topics:

- About Vulkan
- How to create and configure Vulkan in Visual Studio
- Creating validation layers and extensions
- Creating a Vulkan context class
- Accessing the GPU

About Vulkan

With OpenGL, developers have to depend upon vendors such as NVIDIA, AMD, and Intel to release appropriate drivers in order to increase the performance of their games before they are released. This is only possible if the developer is working closely with the vendor. If not, the vendor will only be able to release optimized drivers after the release of the game, and it could take a couple of days to release the new drivers.

Furthermore, if you want to port your PC game to a mobile platform and you are using OpenGL as your renderer, you will need to port the renderer to OpenGL ES, which is a subset of OpenGL, where the ES stands for Embedded Systems. Although there are a lot of similarities between OpenGL and OpenGL ES, there is still additional work to be done to get it to work on other platforms. To alleviate these issues, VULKAN was introduced. This gives the developer more control by reducing driver impact and providing explicit developer control to make the game perform better.

Vulkan has been developed from the ground up and is therefore not backward compatible with OpenGL. Using Vulkan, you have complete access to the GPU.

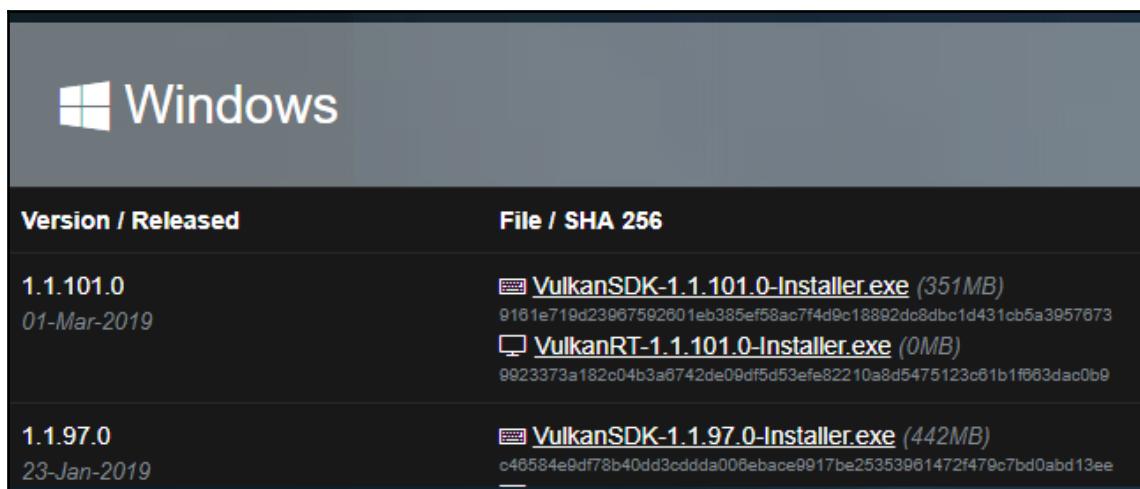
With complete GPU access, you also get complete responsibility for implementing the rendering API. Consequently, the downside of using Vulkan is that you have to specify each thing when developing with it.

This makes Vulkan a very verbose API in which you have to specify everything. However, this also makes it easy to create extensions to the API spec for Vulkan when GPUs add newer features to the cards.

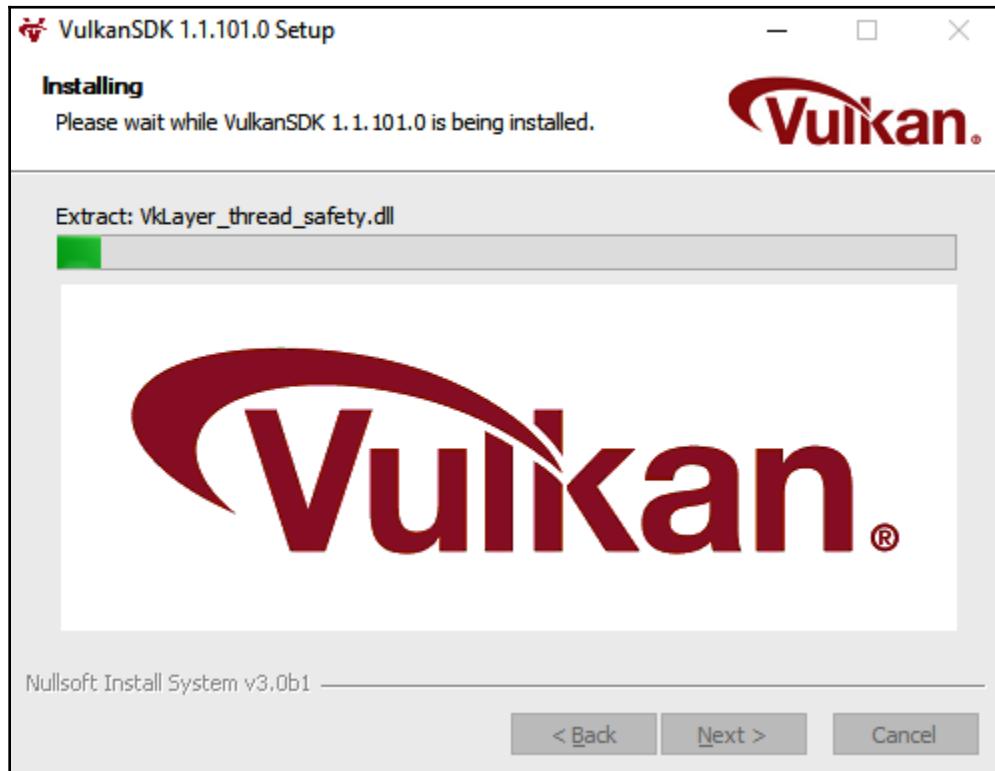
Configuring Visual Studio

Vulkan is just a rendering API, so we need to create a window and do math. For both, we use GLFW and GLM, as we did when creating an OpenGL project. To do this, follow these steps:

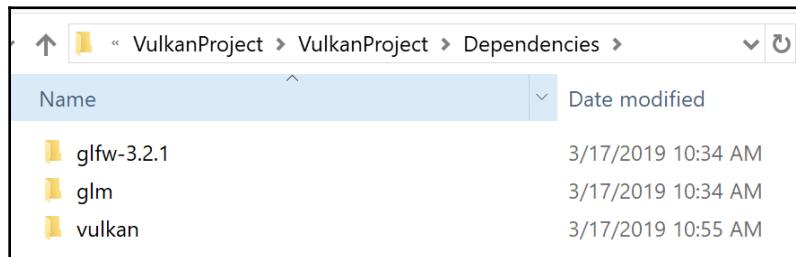
1. Create a new Visual Studio C++ project and call it `VulkanProject`.
2. Copy the `GLFW` and `GLM` folders from the OpenGL project and place it inside the `VulkanProject` folder under a folder named `Dependencies`.
3. Download the Vulkan SDK. Go to <https://vulkan.lunarg.com/sdk/home> and download the Windows version of the SDK, as shown in the following screenshot:



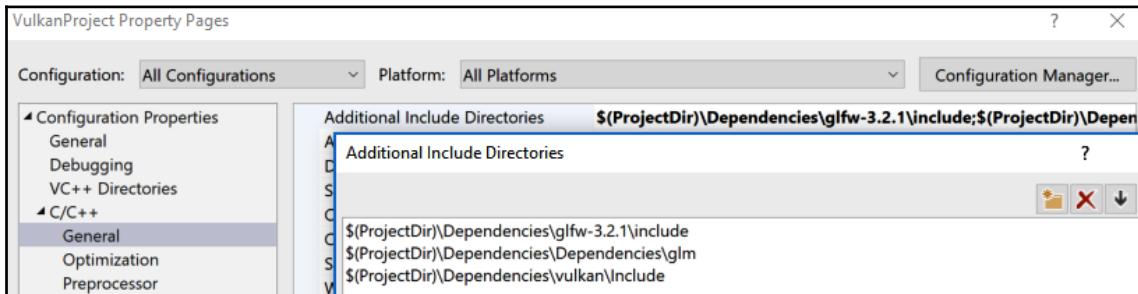
4. Install the SDK, as shown in the following screenshot:



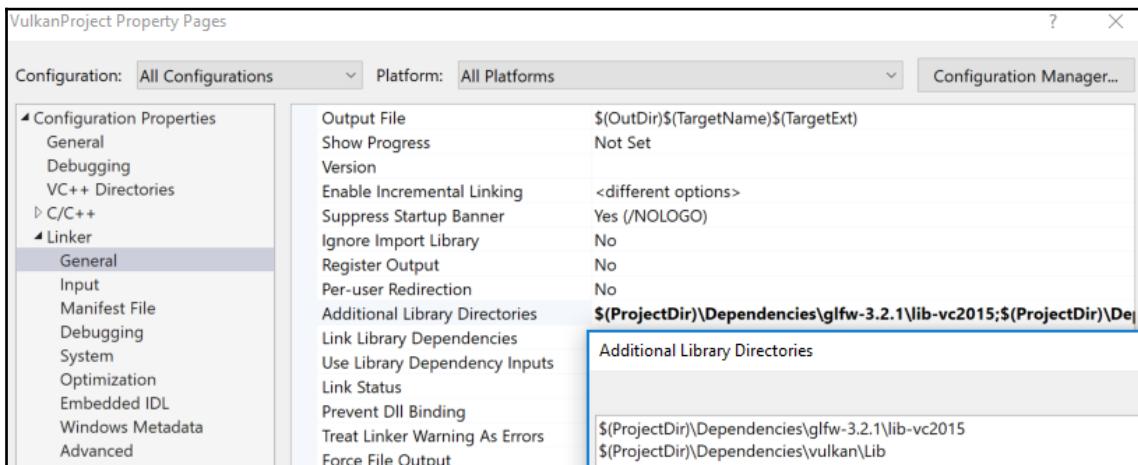
5. In the Dependencies directory, create a new folder called Vulkan. Copy and paste the Lib and include the folder from the Vulkan SDK folder in C:\ drive, as shown in the following screenshot:



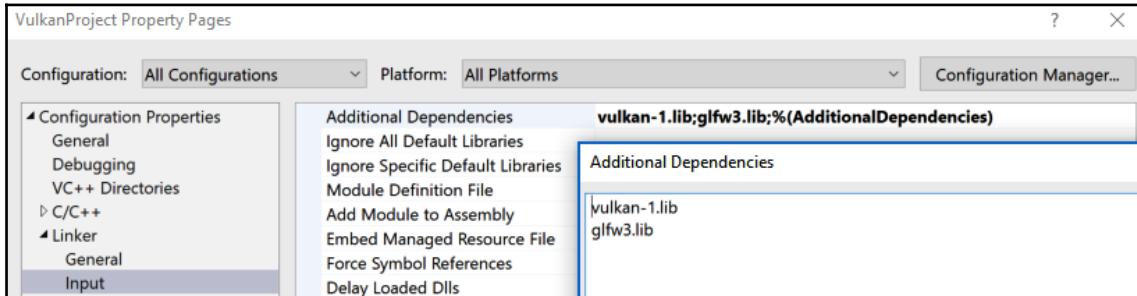
6. In the Visual Studio project, create a new blank `source.cpp` file. Open up the **VulkanProject properties** and add the `include` directory in the **C/C++ | General | Additional Include Directory**.
7. Make sure **All Configurations** is selected and **All Platforms** are selected in the **Configuration** and **Platform** drop-down lists, as shown in the following screenshot:



8. Add the Library Directories under the **Linker | General** section, as shown in the following screenshot:



9. In **Linker | Input**, set the libraries that you want to use, as shown in the following screenshot:



With this prep work out of the way, let's check whether our window creation works properly:

1. In `source.cpp`, add the following:

```
#define GLFW_INCLUDE_VULKAN
#include<GLFW/glfw3.h>

int main() {

    glfwInit();

    glfwWindowHint(GLFW_CLIENT_API, GLFW_NO_API);
    glfwWindowHint(GLFW_RESIZABLE, GLFW_FALSE);

    GLFWwindow* window = glfwCreateWindow(1280, 720, "HELLO VULKAN",
                                         nullptr, nullptr);

    while (!glfwWindowShouldClose(window)) {

        glfwPollEvents();
    }

    glfwDestroyWindow(window);
    glfwTerminate();

    return 0;
}
```

First, we include `glfw3.h` and ask GLFW to include Vulkan-related headers. Then, in the main function, we initialize GLFW by calling `glfwInit()`. We then call the `glfwWindowHint` functions. The first `glfwWindowHint` function doesn't create the OpenGL context, as it is created by default by GLFW by default otherwise. In the next function, we disable the resizing of the window we are about to create.

Then we create the 1280 x 720 window in a similar way to when we created the window in the OpenGL project. We create a while loop that checks whether the window should be closed. If the window does not need to be closed, we will poll the system events. Once this is done, we will destroy the window, terminate `glfw`, and return 0.

2. This should just give us a window to work with. So, run the application in **debug** mode as an **x64** executable to see the window being displayed and saying **HELLO VULKAN**, as shown in the following screenshot:



Vulkan validation layers and extensions

Before we jump into creating the Vulkan application, we have to check for application validation layers and extensions:

- **Validation layers:** Since so much control is given to developers, it is also possible for the developers to implement the Vulkan applications in an incorrect manner. The Vulkan validation layers check for such errors and tell the developer that they are doing something wrong and need to fix it.
- **Extensions:** Over the course of the development of the Vulkan API, there might be new features introduced in newer GPUs. In order to keep Vulkan up to date, these features can be added by extending its functionality by adding extensions.

One example of this is the introduction of Ray Tracing in the RTX series of GPUs. In Vulkan, a new extension was created to support this change in the hardware, called `Vk_NV_ray_tracing` by Nvidia. If our game uses this extension, we can check whether the hardware supports it.

Similar extensions can be added and checked at the application level as well. One such extension is the Debug report extension, which we can generate if something goes wrong when implementing Vulkan. Our first class will add this functionality into the application to check for Application validation layers and extensions. Let's begin creating our first class: Create a new class called `AppValidationLayersAndExtensions`. In `AppValidationLayersAndExtensions.h`, add the following:

```
#pragma once

#include<vulkan\vulkan.h>
#include<vector>
#include<iostream>

#define GLFW_INCLUDE_VULKAN
#include<GLFW\glfw3.h>

class AppValidationLayersAndExtensions {
public:
    AppValidationLayersAndExtensions();
    ~AppValidationLayersAndExtensions();

    const std::vector<const char*> requiredValidationLayers = {
        "VK_LAYER_LUNARG_standard_validation"
    };

    bool checkValidationLayerSupport();
    std::vector<const char*> getRequiredExtensions(bool isValidationLayersEnabled);
};

// Debug Callback
VkDebugReportCallbackEXT callback;

void setupDebugCallback(bool isValidationLayersEnabled,
VkInstance* vkInstance);
void destroy(VkInstance* instance, bool isValidationLayersEnabled);

// Callback

* pCreateInfo, VkResult createDebugReportCallbackEXT(
    VkInstance* instance,
    const VkDebugReportCallbackCreateInfoEXT
    const VkAllocationCallbacks* pAllocator,
    VkDebugReportCallbackEXT* pCallback) {

    auto func =
```

```
(PFN_vkCreateDebugReportCallbackEXT)vkGetInstanceProcAddr(instance,
"vkCreateDebugReportCallbackEXT");

    if (func != nullptr) {
        return func(instance, pCreateInfo, pAllocator, pCallback);
    }
    else {
        returnVK_ERROR_EXTENSION_NOT_PRESENT;
    }

}

void DestroyDebugReportCallbackEXT(
    VkInstanceinstance,
    VkDebugReportCallbackEXTcallback,
    constVkAllocationCallbacks* pAllocator) {

    auto func =
(PFN_vkDestroyDebugReportCallbackEXT)vkGetInstanceProcAddr(instance,
"vkDestroyDebugReportCallbackEXT");
    if (func != nullptr) {
        func(instance, callback, pAllocator);
    }
}

};
```

We include `vulkan.h`, `iostream`, `vector`, and `glfw`. Then we create a vector called `requiredValidationLayers`, and in it we pass `VK_LAYER_LUNARG_standard_validation`. For our application, we will need the standard validation layer, which has all the validation layers in it. If there are only specific validation layers that we need, then we can specify them individually as well. We then create two functions: one for checking the support for validation layers and one for getting the required extensions.

For generating a report if there is an error, we need a debug callback. Then we add two functions: one to set up the debug callback and one to destroy it. These functions will call the `debug`, `create`, and `destroy` functions. These functions will then call `vkGetInstanceProcAddress` to get the pointer to the `vkCreateDebugReportCallbackEXT` and `vkDestroyDebugReportCallbackEXT` pointer functions for calling these functions.

It would be better if it were less confusing to generate a debug report, but unfortunately this is how it is done. However, now is the only time we have to do this. So let's move on to `AppValidationLayersAndExtensions.cpp`.

1. First, we add the constructor and destructor, as follows:

```
AppValidationLayersAndExtensions::AppValidationLayersAndExtensions() {}  
  
AppValidationLayersAndExtensions::~AppValidationLayersAndExtensions() {}  
Then we add the implementation to checkValidationLayerSupport().  
  
bool AppValidationLayersAndExtensions::checkValidationLayerSupport() {  
  
    uint32_t layerCount;  
  
    // Get count of validation layers available  
    vkEnumerateInstanceLayerProperties(&layerCount, nullptr);  
  
    // Get the available validation layers names  
    std::vector<VkLayerProperties>availableLayers(layerCount);  
    vkEnumerateInstanceLayerProperties(&layerCount,  
    availableLayers.data());  
  
    for (const char* layerName : requiredValidationLayers) { //layers we  
        require  
  
        // boolean to check if the layer was found  
        bool layerFound = false;  
  
        for (const auto& layerproperties : availableLayers) {  
  
            // If layer is found set the layer found boolean to true  
            if (strcmp(layerName, layerproperties.layerName) == 0) {  
                layerFound = true;  
                break;  
            }  
        }  
  
        if (!layerFound) {  
            return false;  
        }  
  
        return true;  
    }  
}
```

To check the supported validation layers, call the `vkEnumerateInstanceLayerProperties` function twice. We call it first to get the number of validation layers available—once we have the count, we call it again to populate it with the names of the layers.

We create an `int` called `layerCount` and pass it in the first time we call `vkEnumerateInstanceLayerProperties`. The function takes two parameters: the first is the count and the second is initially kept `null`. Once the function is called, we will know the number of validation layers available. For the names of the layers, we create a new vector called `availableLayers` of the `VkLayerProperties` type and initialize it with the `layerCount`. The function is then called again, and this time we pass in the `layerCount` and the vector as parameters to store the information. We then make a check between the required layers and the available layers. If the validation layer was found, the function will return `true`. If not, it will return `false`.

2. Next, we add the `getRequiredInstanceExtentions` function, as follows:

```
std::vector<constchar*>AppValidationLayersAndExtensions::getRequire  
dExtensions(bool isValidationLayersEnabled) {  
  
    uint32_t glfwExtensionCount = 0;  
    constchar** glfwExtensions;  
  
    // Get extensions  
    glfwExtensions =  
        glfwGetRequiredInstanceExtensions(&glfwExtensionCount);  
  
    std::vector<constchar*>extensions(glfwExtensions, glfwExtensions  
+ glfwExtensionCount);  
  
    //debug report extention is added.  
  
    if (isValidationLayersEnabled) {  
        extensions.push_back("VK_EXT_debug_report");  
    }  
  
    return extensions;  
}
```

The `getRequiredInstanceExtensions` phrase will get all the extensions that are supported by GLFW. It takes a Boolean to check whether the validation layers are enabled, and returns a vector with the names of the supported extensions. In this function, we create a `uint32_t` called `glfwExtensionCount` and a `const char` to store the names of the extensions. We call `glfwGetRequiredExtentions`, pass in `glfwExtentionCount`, and set it as equal to `glfwExtensions`. This will store all the required extensions in `glfwExtensions`.

We create a new extensions vector and store the `glfwExtention` names. If we have enabled the validation layer, then we add an additional extension layer called `VK_EXT_debug_report`, which is the extension for generating a debug report. This extension vector is returned at the end of the function.

3. We then add the debug report callback function, which will generate a report message whenever there is an error, as follows:

```
static VKAPI_ATTR VkBool32 VKAPI_CALL debugCallback(
    VkDebugReportFlagsEXT flags,
    VkDebugReportObjectTypeEXT objType,
    uint64_t obj,
    size_t location,
    int32_t code,
    const char* layerPrefix,
    const char* msg,
    void* userData) {

    std::cerr << "validation layer: " << msg << std::endl << std::endl;

    return false;
}
```

4. We create the `setupDebugCallback` function, which will call the `createDebugReportCallbackExt` function, as follows:

```
voidAppValidationLayersAndExtensions::setupDebugCallback(bool isValidationLayersEnabled, VkInstance vkInstance) {

    if (!isValidationLayersEnabled) {
        return;
    }

    printf("setup call back \n");
```

```
VkDebugReportCallbackCreateInfoEXT info = {};  
  
    info.sType =  
VK_STRUCTURE_TYPE_DEBUG_REPORT_CALLBACK_CREATE_INFO_EXT;  
    info.flags = VK_DEBUG_REPORT_ERROR_BIT_EXT |  
VK_DEBUG_REPORT_WARNING_BIT_EXT;  
    info.pfnCallback = debugCallback; // callback function  
  
    if (createDebugReportCallbackEXT(vkInstance, &info, nullptr,  
&callback) != VK_SUCCESS) {  
  
        throw std::runtime_error("failed to set debug callback!");  
    }  
  
}
```

This function takes a Boolean, which will check that the validation layer is enabled, and it also takes a Vulkan instance, which we will create after this class.

When creating a Vulkan object, we usually have to first populate a struct with the required parameters. So, to create `DebugReportCallback`, we have to first populate the `VkDebugReportCallbackCreateInfoExt` struct first. In the struct, we pass in the `sType`, which specifies the structure type. We also pass in any flags for both error and warning reporting. Finally, we pass in the callback function itself. We then call the `createDebugReportCallbackExt` function and pass in the instance, the struct, a null pointer for memory allocation, and the callback function. Even though we pass in a null pointer for memory allocation, Vulkan will take care of memory allocation by itself. This function is available if you have a memory-allocation function of your own.

5. We now create the `destroy` function to destroy the debug report callback function, as follows:

```
voidAppValidationLayersAndExtensions::destroy(VkInstanceinstance,  
boolisValidationLayersEnabled){  
  
    if (isValidationLayersEnabled) {  
        DestroyDebugReportCallbackEXT(instance, callback,  
nullptr);  
    }  
  
}
```

Vulkan instances

To use the `AppValidationLayerAndExtension` class, we have to create a Vulkan instance. To do so:

1. We will create another class called `VulkanInstance`. In `VulkanInstance.h`, add the following:

```
#pragmaonce
#include<vulkan\vulkan.h>

#include "AppValidationLayersAndExtensions.h"

class VulkanInstance
{
public:
    VulkanInstance();
    ~VulkanInstance();

    VkInstance vkInstance;

    void createAppAndVkInstance(bool enableValidationLayers
        AppValidationLayersAndExtensions *valLayersAndExtentions);

};

};
```

We include `vulkan.h` and `AppValidationLayersAndExtentions.h`, as we will need the required validation layers and extensions when creating the Vulkan instance. We add the constructor, destructor, and instance of `VkInstance`, as well as a function called `ceateAppAndVkInstance`. This function takes a Boolean which checks whether the validation layers are enabled, as well as `AppValidationLayersAndExtensions`. That's it for the header.

2. In the `.cpp` file, add the following code:

```
#include "VulkanInstance.h"

VulkanInstance::VulkanInstance() {}

VulkanInstance::~VulkanInstance() {}
```

3. Then add the `createAppAndVkInstance` function, which will allow us to create the Vulkan instance, as follows:

```
void VulkanInstance::createAppAndVkInstance(bool enableValidationLayers, AppValidationLayersAndExtensions *valLayersAndExtentions) {

    // links the application to the Vulkan library

    VkApplicationInfo appInfo = {};
    appInfo.sType = VK_STRUCTURE_TYPE_APPLICATION_INFO;
    appInfo.pApplicationName = "Hello Vulkan";
    appInfo.applicationVersion = VK_MAKE_VERSION(1, 0, 0);
    appInfo.pEngineName = "SidTechEngine";
    appInfo.engineVersion = VK_MAKE_VERSION(1, 0, 0);
    appInfo.apiVersion = VK_API_VERSION_1_0;

    VkInstanceCreateInfo vkInstanceInfo = {};
    vkInstanceInfo.sType = VK_STRUCTURE_TYPE_INSTANCE_CREATE_INFO;
    vkInstanceInfo.pApplicationInfo = &appInfo;

    // specify extensions and validation layers
    // these are global meaning they are applicable to whole program
    not just the device

    auto extensions =
    valLayersAndExtentions->getRequiredExtensions(enableValidationLayers);

    vkInstanceInfo.enabledExtensionCount =
    static_cast<uint32_t>(extensions.size());
    vkInstanceInfo.ppEnabledExtensionNames = extensions.data();

    if (enableValidationLayers) {
        vkInstanceInfo.enabledLayerCount =
        static_cast<uint32_t>(valLayersAndExtentions->requiredValidationLayers.size());
        vkInstanceInfo.ppEnabledLayerNames =
        valLayersAndExtentions->requiredValidationLayers.data();
    }
    else {
        vkInstanceInfo.enabledLayerCount = 0;
    }
    if (vkCreateInstance(&vkInstanceInfo, nullptr, &vkInstance) != VK_SUCCESS) {
        throw std::runtime_error("failed to create vkInstance ");
    }
}
```

```
    }  
}
```

In the preceding function, we first have to populate `VkApplicationInfo` struct, which will be required when we create `VkInstance`. We then create the `appInfo` struct. In this, the first parameter we specify is the `struct` type, which is of `VK_STRUCTURE_TYPE_APPLICATION_INFO` type. The next parameter is the application name itself, in which we specify the application version, which is 1.0. We then also specify the engine name and version. Finally, we specify the Vulkan API version to use.

Once the application `struct` is populated, we can create the `vkInstanceCreateInfo` struct for creating the Vulkan instance. In the struct instance we created—just like all the structs before this—we have to specify the `struct` with the `struct` type, which is `VK_STRUCTURE_TYPE_INSTANCE_CREATE_INFO`.

Then we have to pass in the application info struct. We then have to specify the Vulkan extension and validation layers and counts. This information is retrieved from the `AppValidationLayersAndExtensions` class. The validation layers are only enabled if it is in debug mode; otherwise, it is not enabled.

Now we can create the Vulkan instance by calling the `vkCreateInstance` function. This takes three parameters: the create info instance, an allocator, and the instance variable used to store the Vulkan instance. For allocation, we again specify `nullptr` and let Vulkan take care of memory allocation. If the Vulkan instance wasn't created a runtime error will be printed to the console to say that the function failed to create the the Vulkan instance.

Now to use this `ValidationAndExtensions` class and the Vulkan instance class, we will create a new Singleton class called `VulkanContext`, as we will need access to some Vulkan objects in this class when creating our `ObjectRenderer`.

The Vulkan Context class

The Vulkan Context class will include all the functionality for creating our Vulkan renderer. In this class, we will create the validation layer, create the Vulkan application and instance, select the GPU we want to use, create the swapchain, create render targets, create the render pass, and add the command buffers to send our draw commands to the GPU.

We will also add a `drawBegin` and `drawEnd` function. In the `drawBegin` function, we will add the functionality to prepare before drawing something. The `drawEnd` function will be called after we have drawn an object and prepared it to be presented to the viewport.

Create the new .h class and .cpp file. In the .h file, we include the following:

```
#define GLFW_INCLUDE_VULKAN
#include<GLFW\glfw3.h>

#include<vulkan\vulkan.h>

#include"AppValidationLayersAndExtensions.h"
#include"VulkanInstance.h"
```

Next, we will create a Boolean called `isValidationsEnabled`. This will be set to `true` if the application is running in **debug** mode and `false` if running in **release** mode:

```
#ifdef _DEBUG
bool const bool isValidationsEnabled = true;
#else
const bool isValidationsEnabled = false;
#endif
```

Next, we create the class itself, as follows:

```
class VulkanContext {

public:
    static VulkanContext* instance;
    static VulkanContext* getInstance();

    ~VulkanContext();

    void initVulkan();

private:

    // My Classes
    AppValidationLayersAndExtensions *valLayersAndExt;
    VulkanInstance* vInstance;

};
```

In the public section, we create a static instance and the `getInstance` variable and function, which sets and gets the instance of this class. We add the destructor and add a `initVulkan` function, which will be used to initialize the Vulkan context. In the private section, we create an instance of the `AppValidationLayersAndExtensions` and `VulkanInstance` class. In the `VulkanContext.cpp` file, we first set the instance variable to null, and in the `getInstance` function, we check whether the instance was created. If it is not created, we create a new instance and return it and we add the destructor:

```
#include "VulkanContext.h"

VulkanContext* VulkanContext::instance = NULL;

VulkanContext* VulkanContext::getInstance() {
    if (!instance) {
        instance = new VulkanContext();
    }
    return instance;
}

VulkanContext::~VulkanContext()
```

Then we add the functionality for the `initVulkan` function, as follows:

```
void VulkanContext::initVulkan() {

    // Validation and Extension Layers
    valLayersAndExt = new AppValidationLayersAndExtensions();

    if (isValidationLayersEnabled &&
    !valLayersAndExt->checkValidationLayerSupport()) {
        throw std::runtime_error("Validation Layers Not Available !");
    }

    // Create App And Vulkan Instance()
    vInstance = new VulkanInstance();
    vInstance->createAppAndVkInstance(isValidationLayersEnabled,
    valLayersAndExt);

    // Debug CallBack
    valLayersAndExt->setupDebugCallback(isValidationLayersEnabled,
    vInstance->vkInstance);

}
```

We first create a new `AppValidationLayersAndExtensions` instance. Then we check whether the validation layers are enabled and then check whether the validation layers are supported. If `ValidationLayers` is not available, a runtime error is sent out saying that the Validation layers are not available.

If the validation layers are supported, a new instance of the `VulkanInstance` class is created and the `createAppAndVkInstance` function is called, which creates a new `vkInstance`.

Once this is complete, we call the `setupDebugCallBack` function by passing in the Boolean and `vkInstance`. In the `source.cpp` file, include the `VulkanContext.h` file and call `initVulkan` after the window is created, as follows:

```
#define GLFW_INCLUDE_VULKAN
#include<GLFW/glfw3.h>

#include "VulkanContext.h"

int main() {

    glfwInit();

    glfwWindowHint(GLFW_CLIENT_API, GLFW_NO_API);
    glfwWindowHint(GLFW_RESIZABLE, GLFW_FALSE);

    GLFWwindow* window = glfwCreateWindow(1280, 720, "HELLO VULKAN",
                                         nullptr, nullptr);

    VulkanContext::getInstance()->initVulkan();

    while (!glfwWindowShouldClose(window)) {

        glfwPollEvents();
    }

    glfwDestroyWindow(window);
    glfwTerminate();

    return 0;
}
```

Hopefully, you won't get any errors in the console window when you build and run the application. If you do get errors, go through each line of code and make sure there are no spelling mistakes:

setup call back

Creating the window surface

We need an interface to the window we created for the current platform to present the images we will render. We use the `VkSurfaceKHR` property to get access to the window surface for the window created on the current platform. To store the surface information that the OS supports, we will call the `glfw` function, `glfwCreateWindowSurface`, to create the surface supported by the OS.

In `VulkanContext.h`, add a new variable of the `VkSurfaceKHR` type called `surface`, as follows:

```
private:  
    //surface  
    VkSurfaceKHR surface;
```

Since we need access to the window instance we created in `source.cpp`, change the `initVulkan` function to accept a `GLFWwindow`, as follows:

```
void initVulkan(GLFWwindow* window);
```

In `VulkanContext.cpp`, change the `initVulkan` implementation as follows and call the `glfwCreateWindowSurface` function, which takes in the Vulkan instance and the window. Next, pass in `null` for the allocator and the surface to create the surface object:

```
void VulkanContext::initVulkan(GLFWwindow* window) {  
  
    // -- Platform Specific  
  
    // Validation and Extension Layers  
    valLayersAndExt = new AppValidationLayersAndExtensions();  
  
    if (isValidationLayersEnabled &&  
        !valLayersAndExt->checkValidationLayerSupport()) {  
        throw std::runtime_error("Requested Validation Layers Not
```

```
Available !");  
}  
  
// Create App And Vulkan Instance()  
vInstance = new VulkanInstance();  
vInstance->createAppAndVkInstance(isValidationLayersEnabled,  
valLayersAndExt);  
  
// Debug CallBack  
valLayersAndExt->setupDebugCallback(isValidationLayersEnabled,  
vInstance->vkInstance);  
  
// Create Surface  
if (glfwCreateWindowSurface(vInstance->vkInstance, window, nullptr,  
&surface) != VK_SUCCESS) {  
  
    throw std::runtime_error(" failed to create window surface !");  
}  
}
```

Finally, in `source.cpp`, change the `initVulkan` function as follows:

```
GLFWwindow* window = glfwCreateWindow(WIDTH, HEIGHT, "HELLO VULKAN ",  
nullptr, nullptr);  
  
VulkanContext::getInstance()->initVulkan(window);
```

Picking a physical device and creating a logical device

Now we will create the `Device` class, which will be used to go through the different physical devices we have and we will choose one to render our application. To check whether your GPU is compatible with Vulkan, check the compatibility list from your GPU vendor site or at [https://en.wikipedia.org/wiki/Vulkan_\(API\)](https://en.wikipedia.org/wiki/Vulkan_(API)).

Basically, any Nvidia GPU from the Geforce 600 series and AMD GPU from the Radeon HD 2000 series and later should be supported. To access the physical device and create the logical device, we will create a new class, which enable us to access it whenever we want. So, create a new class called `Device`. In `Device.h`, add the following includes:

```
#include<vulkan\vulkan.h>
#include<stdexcept>

#include<iostream>
#include<vector>
#include<set>

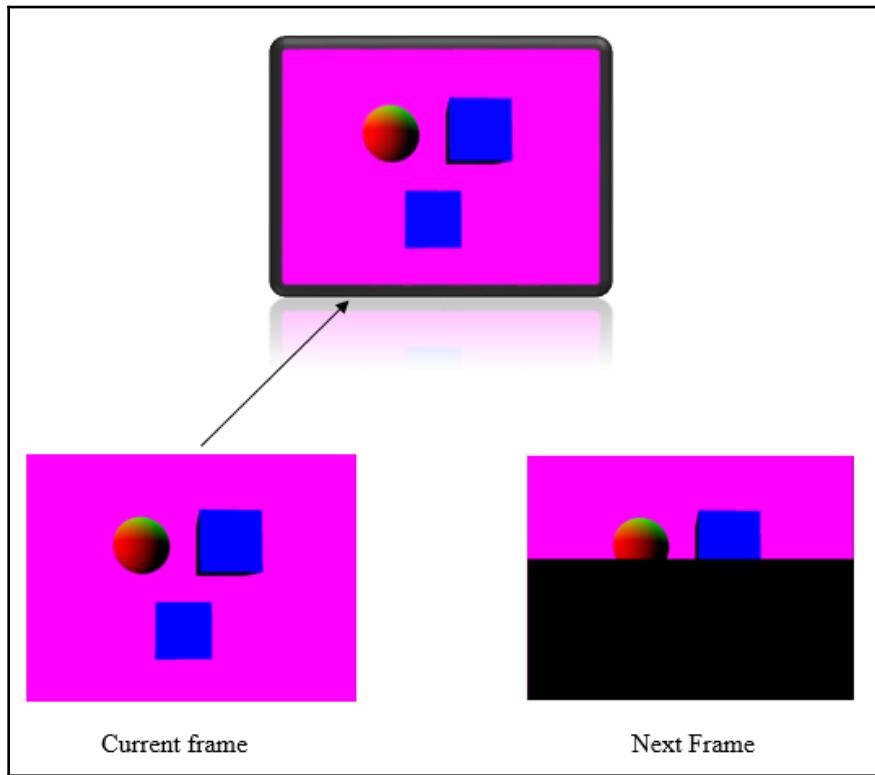
#include"VulkanInstance.h"
#include"AppValidationLayersAndExtensions.h"
```

We will also add a couple of structs for the sake of convenience. The first is called `SwapChainSupportDetails`, which has access to `VkSurfaceCapabilitiesKHR` which has all the required details about the surface. We'll also add the `surfaceFormats` vector of the `VkSurfaceFormatKHR` type, which keeps track of all the different image formats the surface supports, and the `presentModes` vector of the `VkPresentModeKHR` type, which stores the presentation modes that the GPU supports.

Rendered images will be sent to the window surface to get displayed. This is how we are able to see the final rendered image using a renderer, such as OpenGL or Vulkan. Now we can show these images to the window one at a time, which is fine if we want to look at a still image forever. But when we run a game that is updated every 16 milliseconds (so 60 times in a second), there might be cases where the image has not been fully rendered, but it would be time to display it. At this point, we will see half-rendered images, which leads to screen tearing.

To avoid this, we use double buffering. This allows us to render the image to different images, called the front buffer and the back buffer, and pingpong between the two. We then preset the one buffer that has finished rendering and display it to the viewport while the next frame is still being rendered, as shown in the following diagram.

There are different ways to present the image as well. We will look at these different presentation modes when we create the swapchain:



Create a struct to track the surface properties, format, and presentation modes, as follows:

```
struct SwapChainSupportDetails {  
    VkSurfaceCapabilitiesKHR surfaceCapabilities; // size and images in  
    swapchain  
    std::vector<VkSurfaceFormatKHR> surfaceFormats;  
    std::vector<VkPresentModeKHR> presentModes;  
};
```

A GPU also has what is called `QueueFamilies`. Commands to the GPU are sent and executed using queues. There are separate queues for different kinds of work. Render commands are sent to render queues, compute commands are sent to compute queues, and there are also presentation queues for presenting images. We also need to know which queues the GPU supports and how many of the queues are present.

The renderer, compute, and presentation queues can be combined and are known as Queue families. The queues could be combined in different ways to form a number of queue families. So there can be a combination of render and present queues to form one queue family and another family might just contain compute queues. So we have to check whether we have at least one queue family with graphics and presentation queues, as we need a graphics queue to pass our rendering commands into and a presentation queue for presenting the image after rendering to it.

We will add one more struct to check for both, as follows:

```
struct QueueFamilyIndices {  
  
    int graphicsFamily = -1;  
    int presentFamily = -1;  
  
    bool arePresent() {  
        return graphicsFamily >= 0 && presentFamily >= 0;  
    }  
};
```

We will now create the `Device` class itself. After creating the class, we add in the constructor and destructor, as follows:

```
{  
  
public:  
    Device();  
    ~Device();
```

We will then add variables to store the physical device, the `SwapChainSupportDetails`, and the `QueueFamilyIndices`, as follows

```
VkPhysicalDevice physicalDevice;  
SwapChainSupportDetails swapchainSupport;  
QueueFamilyIndices queueFamilyIndices;
```

To create double buffering, we first have to check that the device supports it. This is done using the `VK_KHR_SWAPCHAIN_EXTENSION_NAME` extension to check for swapchain. So, we create a vector of the `char*` const and pass in the extension name, as follows:

```
std::vector<const char*> deviceExtensions = { VK_KHR_SWAPCHAIN_EXTENSION_NAME };
```

Then we add the `pickPhysicalDevice` function, which will be selected depending on whether the device is suitable. While checking for suitability, we will check whether the selected device supports the swapchain extension, get the swapchain support details, and get the queue family indices, as follows:

```
void pickPhysicalDevice (VulkanInstance* vInstance, VkSurfaceKHR  
surface);  
  
bool isDeviceSuitable(VkPhysicalDevice device, VkSurfaceKHR surface);  
  
bool checkDeviceExtensionSupported(VkPhysicalDevice device) ;  
SwapChainSupportDetails querySwapChainSupport(VkPhysicalDevice device,  
VkSurfaceKHR surface);  
QueueFamilyIndices findQueueFamilies(VkPhysicalDevice device,  
VkSurfaceKHR surface);
```

We will also add a getter function to get the queue families of the current device, as follows:

```
QueueFamilyIndices getQueueFamiliesIndicesOfCurrentDevice();
```

Once we have the physical device we want to use, we will create an instance of the logical device. The logical device is an interface for the physical device itself. We will use the logical device to create buffers and so on. We will also store the current device graphics and present a queue to send the graphics and presentation commands. Finally, we will add a `destroy` function, which is used to destroy the physical and logical devices we created, as follows:

```
// ++++++  
// Logical device  
// ++++++  
  
void createLogicalDevice(VkSurfaceKHRsurface,  
bool isValidationLayersEnabled, AppValidationLayersAndExtensions  
*appValLayersAndExtentions);  
VkDevice logicalDevice;  
  
// handle to the graphics queue from the queue families of the gpu  
VkQueue graphicsQueue; // we can also have separate queue for compute,  
memory transfer, etc.  
VkQueue presentQueue; // queue for displaying the framebuffer  
  
void destroy();  
}; // End of Device class
```

That's all for the `Device.h` file. Moving on to `Device.cpp`, we first include `Device.h` and add the constructor and the destructor, as follows:

```
#include "Device.h"
Device::Device() {}

Device::~Device() {
}
```

Now the real work begins. We will create the `pickPhysicalDevice` function, which takes a Vulkan instance and the `VkSurface`, as follows:

```
void Device::pickPhysicalDevice(VulkanInstance* vInstance,
VkSurfaceKHR surface) {
    uint32_t deviceCount = 0;

    vkEnumeratePhysicalDevices(vInstance->vkInstance, &deviceCount,
    nullptr);

    if (deviceCount == 0) {
        throw std::runtime_error("failed to find GPUs with vulkan support
    !");
    }

    std::cout << "Device Count: " << deviceCount << std::endl;

    std::vector<VkPhysicalDevice> devices(deviceCount);
    vkEnumeratePhysicalDevices(vInstance->vkInstance, &deviceCount,
    devices.data());

    std::cout << std::endl;
    std::cout << "DEVICE PROPERTIES" << std::endl;
    std::cout << "======" << std::endl;

    for (const auto& device : devices) {

        VkPhysicalDeviceProperties deviceProperties;

        vkGetPhysicalDeviceProperties(device, &deviceProperties);
        std::cout << std::endl;
        std::cout << "Device name: " << deviceProperties.deviceName <<
        std::endl;

        if (isDeviceSuitable(device, surface))
            physicalDevice = device;
    }
}
```

```
        break;  
    }  
  
    if (physicalDevice == VK_NULL_HANDLE) {  
        throw std::runtime_error("failed to find suitable GPU !");  
    }  
  
}
```

We create an `int32` to store the count of the number of physical devices. We get the number of available GPUs using `vkEnumeratePhysicalDevices` and pass the Vulkan instance and the count and `null` for the third parameter. This will get the number of devices available. If `deviceCount` is zero, this means that there are no GPUs available. Then we also print the available number of devices to the console.

To get the physical devices themselves, we create a vector called `devices`, which will store the data type `VkPhysicalDevice`, which will store devices itself. We will call the `vkEnumeratePhysicalDevices` function again, but this time—apart from passing in the Vulkan instance and the device count—we will also store the device information in the vector passed in as the third parameter. We will then print out the number of devices with the `DEVICE_PROPERTIES` heading.

To get the properties of the available devices, we will go through the number of devices and get the properties using `vkGetPhysicalDeviceProperties` and store it in the variable of the `VkPhysicalDeviceProperties` type.

We will then print out the name of the device and call `DeviceSuitable` on the device. If the device is suitable, we will store it as `physicalDevice` and break out of the loop. Note that we set the first available device as the device we will use for our purpose.

If there is no suitable device, we throw a runtime error to say that a suitable device wasn't found. Let's look at the `DeviceSuitable` function shown in the following code:

```
bool Device::isDeviceSuitable(VkPhysicalDevice device, VkSurfaceKHR  
surface)  {  
  
    // find queue families the device supports  
  
    QueueFamilyIndices qFamilyIndices = findQueueFamilies(device, surface);  
  
    // Check device extensions supported  
    bool extensionSupported = checkDeviceExtensionSupported(device);  
  
    bool swapChainAdequate = false;
```

```
// If swapchain extension is present
// Check surface formats and presentation modes are supported
if (extensionSupported) {

    swapchainSupport = querySwapChainSupport(device, surface);
    swapChainAdequate = !swapchainSupport.surfaceFormats.empty() &&
!swapchainSupport.presentModes.empty();

}

VkPhysicalDeviceFeatures supportedFeatures;
vkGetPhysicalDeviceFeatures(device, &supportedFeatures);

return qFamilyIndices.arePresent() && extensionSupported &&
swapChainAdequate && supportedFeatures.samplerAnisotropy;

}
```

In this function, we first get the queue family indices by calling `findQueueFamilies`. Then we check whether `VK_KHR_SWAPCHAIN_EXTENSION_NAME` extension is supported. After this, we check for swapchain support on the device. If the surface formats and presentation modes are not empty, `swapChainAdequate` boolean is set to true. Finally, we get the physical device features by calling `vkGetPhysicalDeviceFeatures`.

In the end, we return `true` if the queue families are present, the swapchain extension is supported, the swapchain is adequate, and the device supports anisotropic filtering. Anisotropic filtering is a mode that enables the pixels in the distance to be clearer when the mode is enabled.

Anisotropic filtering is a mode which, when enabled, helps to sharpen textures viewed from extreme angles.

In the following example, the image on the right has anisotropic filtering enabled and the image on the left has it disabled. In the image on the right, the white dashed line is still relatively visible further down the road. However, on the left, the dashed line becomes blurry and pixelated. Therefore, anisotropic filtering is required:



(Taken from <https://i.imgur.com/jzCq5ST.jpg>)

Let's look at the three functions we called in the previous function. First, let's check out the `findQueueFamilies` function, as shown in the following code:

```
QueueFamilyIndicesDevice::findQueueFamilies(VkPhysicalDevice device,  
VkSurfaceKHR surface) {  
  
    uint32_t queueFamilyCount = 0;  
  
    vkGetPhysicalDeviceQueueFamilyProperties(device, &queueFamilyCount,  
    nullptr);  
  
    std::vector<VkQueueFamilyProperties> queueFamilies(queueFamilyCount);  
  
    vkGetPhysicalDeviceQueueFamilyProperties(device, &queueFamilyCount,  
    queueFamilies.data());  
  
    int i = 0;  
  
    for (const auto& queueFamily : queueFamilies) {  
  
        if (queueFamily.queueCount > 0 && queueFamily.queueFlags  
&VK_QUEUE_GRAPHICS_BIT) {  
            queueFamiliyIndices.graphicsFamily = i;  
        }  
  
        VkBool32 presentSupport = false;  
        vkGetPhysicalDeviceSurfaceSupportKHR(device, i, surface,  
&presentSupport);  
  
        if (queueFamily.queueCount > 0 && presentSupport) {  
            queueFamiliyIndices.presentFamily = i;  
        }  
  
        if (queueFamiliyIndices.arePresent()) {  
            break;  
        }  
  
        i++;  
    }  
  
    return queueFamiliyIndices;  
}
```

To get the queue family properties, we call the `vkGetPhysicalDeviceQueueFamilyProperties` function, and in the physical device we pass an `int` to store the number of queue families and the `null` pointer. This will give us the number of queue families available.

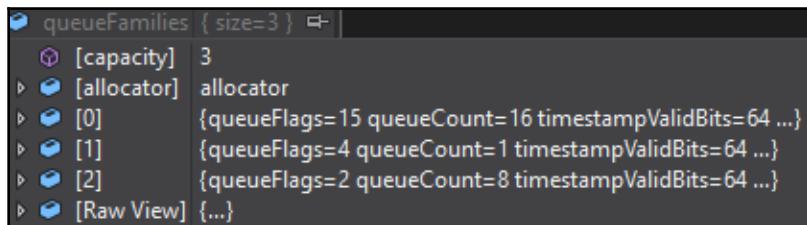
Next, for the properties themselves, we create a vector of the `VkQueueFamilyProperties` type, called `queueFamilies`, to store the information. Then we call `vkGetPhysicalDeviceQueueFamilyProperties` and pass in the physical device, the count, and `queueFamilies` itself to populate it with the required data. We create an `int`, `i`, and initialize it to 0. This will store the index of the graphics and presentation indices.

In the `for` loop, we check whether each of the queue families supports a graphics queue by looking for `VK_QUEUE_GRAPHICS_BIT`. If they do, we set the graphics family index.

We then check for presentation support by passing in the index to check whether the same family supports presentation as well. If it supports presentation, we set `presentFamily` to that index.

If the queue family supports graphics and presentation, the graphics and presentation index will be the same.

The following screenshot shows the number of queue families by device and the number of queues in each queue family:



There are three queue families on my GPU. The first queue family at the 0th index has 16 queues, the second at the 1st index has 1 queue, and the third queue at the 2nd index supports 8 queues.

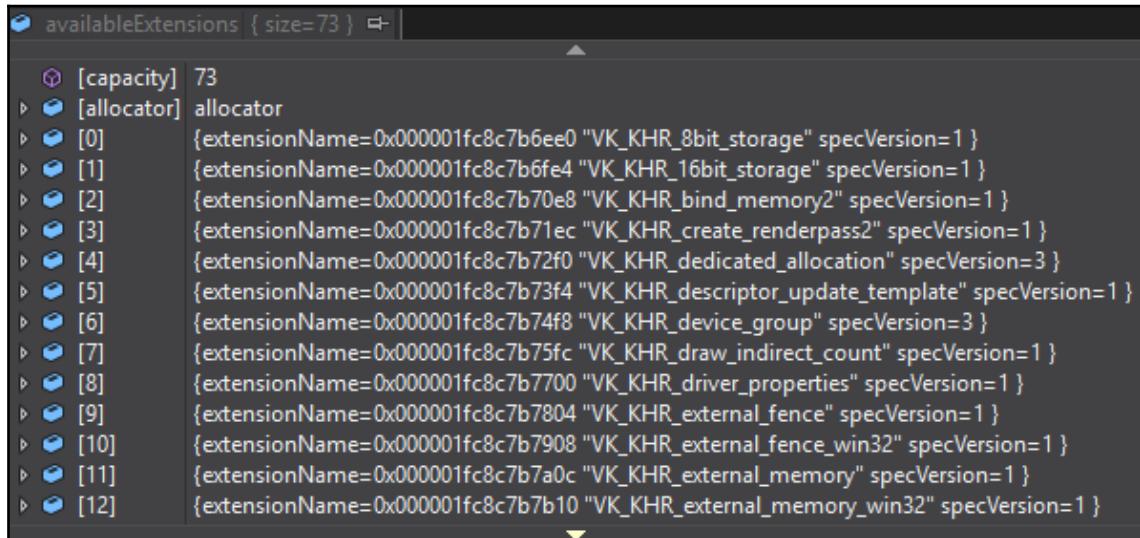
The `queueFlags` specify the queues in the queue family. The queues supported could be graphics, compute, transfer, or sparse binding.

After this, we check that both the graphics and presentation indices were found, and then we break out of the loop. Finally, we return the `queueFamilyIndices`. I am running the project on aIntel(R) Iris(R) Plus Graphics 650. This integrated intel GPU has one queue family that supports both graphics and the presentation queue. Different GPUs have different numbers of queue families and each family may support more than one queue type. Next, let's look at the device extension that is supported. The device extensions that are supported are checked by the `checkDeviceExtensionSupported` function, which takes in a physical device, as shown in the following code:

```
boolDevice::checkDeviceExtensionSupported(VkPhysicalDevice device) {  
  
    uint32_t extensionCount;  
  
    // Get available device extenions count  
    vkEnumerateDeviceExtensionProperties(device, nullptr, &extensionCount,  
    nullptr);  
  
    // Get available device extenions  
    std::vector<VkExtensionProperties> availableExtensions(extensionCount);  
  
    vkEnumerateDeviceExtensionProperties(device, nullptr, &extensionCount,  
    availableExtensions.data());  
  
    // Populate with required device extenions we need  
    std::set<std::string> requiredExtensions(deviceExtensions.begin(),  
    deviceExtensions.end());  
  
    // Check if the required extention is present  
    for (const auto& extension : availableExtensions) {  
        requiredExtensions.erase(extension.extensionName);  
    }  
  
    // If device has the required device extention then return  
    return requiredExtensions.empty();  
}
```

We get the number of extensions supported by the device by calling `vkEnumerateDeviceExtensionProperties` and passing in the physical device, null pointer, an `int` to store the count in it, and `null` as the last parameter. The actual properties are stored inside the `availableExtensions` vector, which stores the `VkExtensionProperties` data type. We call `vkEnumerateDeviceExtensionProperties` again and this time get the device's extension properties.

We populate the `requiredExtensions` vector with the extension we require. Then we check the available extension vector with the required extensions. If the required extension is found, we remove it from the vector. This means that the device supports the extension and returns the value from the function, as shown in the following code:



The current device I am running has 73 available extensions, that are available as shown in the following code. You can set a break point and peek into the device extension properties to see the supported extension of the device. The third function we will look at is the `querySwapChainSupport` function, which populates the surface capabilities, surface formats, and presentation modes available:

```
SwapChainSupportDetailsDevice::querySwapChainSupport (VkPhysicalDevice device
, VkSurfaceKHR surface) {

    SwapChainSupportDetails details;

    vkGetPhysicalDeviceSurfaceCapabilitiesKHR(device, surface,
&details.surfaceCapabilities);

    uint32_t formatCount;
    vkGetPhysicalDeviceSurfaceFormatsKHR(device, surface, &formatCount,
nullptr);

    if (formatCount != 0) {
        details.surfaceFormats.resize(formatCount);
        vkGetPhysicalDeviceSurfaceFormatsKHR(device, surface,
&formatCount, details.surfaceFormats.data());
    }
}
```

```
    }

    uint32_t presentModeCount;
    vkGetPhysicalDeviceSurfacePresentModesKHR(device, surface,
&presentModeCount, nullptr);

    if (presentModeCount != 0) {

        details.presentModes.resize(presentModeCount);
        vkGetPhysicalDeviceSurfacePresentModesKHR(device, surface,
&presentModeCount, details.presentModes.data());
    }

    return details;
}
```

To get the surface capabilities, we call `vkGetPhysicalDeviceSurfaceCapabilitiesKHR` and pass in the device, which will be `surface` to get the surface capabilities. To get the surface format and presentation modes, we call

`vkGetPhysicalDeviceSurfaceFormatKHR` and

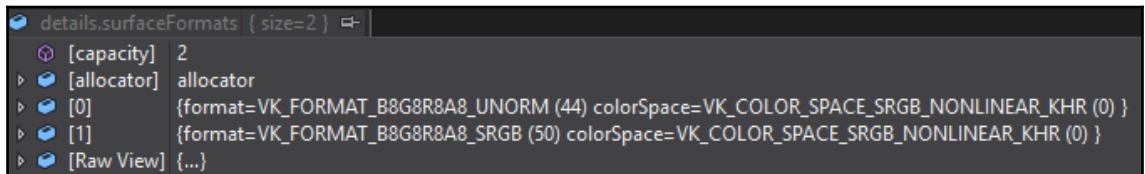
`vkGetPhysicalDeviceSurfacePresentModeKHR` twice.

The first time we call the `vkGetPhysicalDeviceSurfacePresentModeKHR` function, we get the number of formats and modes present, and then we call the function to get the formats and the modes populated and stored in the vectors in the struct.

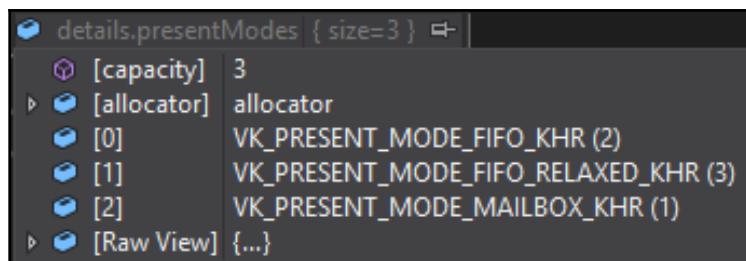
Here are the capabilities of my device surface:

details.surfaceCapabilities	{minImageCount=2 maxImageCount=8 currentExtent={width=1280 height=720}}
minImageCount	2
maxImageCount	8
currentExtent	{width=1280 height=720}
minImageExtent	{width=1280 height=720}
maxImageExtent	{width=1280 height=720}
maxImageArrayLayers	1
supportedTransforms	1
currentTransform	VK_SURFACE_TRANSFORM_IDENTITY_BIT_KHR (1)
supportedCompositeAlpha	1
supportedUsageFlags	159

So, the minimum image count is two, meaning that we can add double buffering. These are the surface formats and the color space that my device supports:



Here are the presentation modes that are supported by my device:



So, it seems that the device only supports the immediate mode. We will see which will be this information in the coming chapters. After getting the physical device properties, we set the getter function for the queueFamilyIndices as follows:

```

QueueFamilyIndicesDevice::getQueueFamiliesIndicesOfCurrentDevice() {
    return queueFamilyIndices;
}

```

Now we can also create the logical device by using the `createLogicalDevice` function, as shown in the following steps.

To create the logical device, we have to populate the `VkDeviceCreateInfo` struct, which requires the `queueCreateInfo` struct. Let's see how to do so:

1. Create a vector to store `VkDeviceQueueCreateInfo` and to store the information for both the graphics and presentation queues.
2. Create another vector of the `int` type to store the indices of the graphics and presentation queues.

3. For each queue family, populate `VkDeviceQueueCreateInfo`. Create a local struct and pass in the struct type, the queue family index, queue count, and priority (which is 1), and then push it into the `queueCreateInfos` vector, as shown in the following code:

```
void Device::createLogicalDevice(VkSurfaceKHRsurface,
bool isValidationLayersEnabled, AppValidationLayersAndExtensions
*appValLayersAndExtentions) {

    // find queue families like graphics and presentation
    QueueFamilyIndices indices = findQueueFamilies(physicalDevice,
surface);

    std::vector<VkDeviceQueueCreateInfo> queueCreateInfos;

    std::set<int> uniqueQueueFamilies = { indices.graphicsFamily,
indices.presentFamily };

    float queuePriority = 1.0f;

    for (int queueFamily : uniqueQueueFamilies) {

        VkDeviceQueueCreateInfo queueCreateInfo = {};
        queueCreateInfo.sType =
VK_STRUCTURE_TYPE_DEVICE_QUEUE_CREATE_INFO;
        queueCreateInfo.queueFamilyIndex = queueFamily;
        queueCreateInfo.queueCount = 1; // we only require 1 queue
        queueCreateInfo.pQueuePriorities = &queuePriority;
        queueCreateInfos.push_back(queueCreateInfo);
    }
}
```

4. To create the device, specify the device features that we will be using. For the device features, we create a variable of the `VkPhysicalDeviceFeatures` type and set `samplerAnisotropy` to `true`, as follows:

```
//specify device features
VkPhysicalDeviceFeatures deviceFeatures = {};

deviceFeatures.samplerAnisotropy = VK_TRUE;
```

5. Create the `VkDeviceCreateInfo` struct, which is required to create the logical device. Set the type to `VK_STRUCTURE_TYPE_DEVICE_CREATE_INFO`, and then set `queueCreateInfos`, the count, and the device features that are to be enabled.

6. Set the device extension count and names. If the validation layer is enabled, we set the validation layers count and names. Create the `logicalDevice` by calling `vkCreateDevice` and passing in the physical device, the create device information, `null` for the allocator, and then create the logical device, as shown in the following code. If the creation fails, then we throw a runtime error:

```
VkDeviceCreateInfo createInfo = {};
createInfo.sType = VK_STRUCTURE_TYPE_DEVICE_CREATE_INFO;
createInfo.pQueueCreateInfos = queueCreateInfos.data();
createInfo.queueCreateInfoCount =
    static_cast<uint32_t>(queueCreateInfos.size());

createInfo.pEnabledFeatures = &deviceFeatures;

createInfo.enabledExtensionCount =
    static_cast<uint32_t>(deviceExtensions.size());
createInfo.ppEnabledExtensionNames = deviceExtensions.data();

if (isValidationLayersEnabled) {
    createInfo.enabledLayerCount =
        static_cast<uint32_t>(appValLayersAndExtentions->requiredValidation
        Layers.size());
    createInfo.ppEnabledLayerNames =
        appValLayersAndExtentions->requiredValidationLayers.data();
}
else {
    createInfo.enabledLayerCount = 0;
}

//create logical device

if (vkCreateDevice(physicalDevice, &createInfo, nullptr,
&logicalDevice) != VK_SUCCESS) {
    throw std::runtime_error("failed to create logical device
!");
}
```

7. Get the device graphics and presentation queue, as shown in the following code. We are now done with the Device class:

```
//get handle to the graphics queue of the gpu  
vkGetDeviceQueue(logicalDevice, indices.graphicsFamily, 0,  
&graphicsQueue);  
  
//get handle to the presentation queue of the gpu  
vkGetDeviceQueue(logicalDevice, indices.presentFamily, 0,  
&presentQueue);  
  
}
```

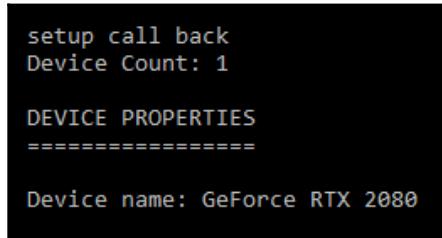
8. This wraps up the Device class. Include the `Device.h` in `VulkanContext.h` and add a new device object of the `Device` type in the `VulkanContext` class's private section, as follows:

```
// My Classes  
AppValidationLayersAndExtensions *valLayersAndExt;  
VulkanInstance* vInstance;  
Device* device;
```

9. In the `VulkanContext.cpp` file in the `VulkanInit` function, add the following after creating the surface:

```
device = new Device();  
device->pickPhysicalDevice(vInstance, surface);  
device->createLogicalDevice(surface, isValidationLayersEnabled,  
valLayersAndExt);
```

10. This will create a new instance of the `device` class and choose from the available physical devices. You will then be able to create the logical device. Run the application to see which device the application will run on. On my desktop, the following device count and name were found:



11. On my laptop, the application found one device with the following device name:

```
setup call back
Device Count: 1

DEVICE PROPERTIES
=====
Device name: Intel(R) Iris(R) Plus Graphics 650
-
```

12. Set a break point inside `findQueueFamilies`,
`checkDeviceExtensionSupport`, and `querySwapChainSupport` to check the
number of queue family device extensions and to check for swapchain support
for your GPUs.

Summary

We are about a quarter of the way through the process of seeing something rendered to the viewport. In this chapter, we set the validation layers and the extension that will be needed to set up Vulkan rendering. We created a Vulkan application and instance, and created a device class to select the physical device. We also created the logical device to interact with the GPU.

In the next chapter, we will create the swapchain itself to swap between the buffers, and we will create the render and the depth texture to draw the scene into. We will create a render pass to set how the render textures are to be used, and create the draw command buffers, which will execute our draw commands.

10

Preparing the Clear Screen

In the last chapter, we enabled the Vulkan validation layers and extensions, created the Vulkan application and instance, chose the device, and created the logical device. In this chapter, we will continue the journey toward creating a clear screen picture and presenting it to the viewport.

Before drawing pictures, we first clear and erase the previous picture with a color value. If we don't do this, the new picture will be written on top of the previous picture, which will create a psychedelic effect.

Each picture is cleared and rendered and then presented to the screen. While the current one is being shown, the next picture is in the process of being drawn in the background. Once that is rendered, the current picture will be swapped with the new picture. This swapping of pictures is taken care of by the **SwapChain**.

In our case, each picture in the SwapChain we are drawing simply stores the color information. This target picture is rendered and hence, it is called the render target. We can also have other target pictures. For example, we can have a depth target/picture which will store the depth information of each pixel per frame. Consequently, we create these render targets as well.

Each of these target pictures per frame is set as an attachment and used to create the Framebuffer. Since we have double buffering (meaning we have two sets of pictures to swap between), we create a Framebuffer each for both frames. Consequently, we will create two Framebuffers—one for each frame—and we will add the picture as an attachment.

The commands that we give to the GPU—for example, the draw command—are sent to the GPU with each frame using a command buffer. The command buffer stores all the commands to be submitted to the GPU using the graphics queue of the device. So, for each frame, we create a command buffer to carry all our commands as well.

Once the commands are submitted and the scene is rendered, instead of presenting the drawn picture to the screen we can save it and add any post-processing effects to it, such as motion blur. In the renderpass, we can specify how the render target is to be used.

Although in our case we are not going to add any post-processing effects, we still need to create a renderpass. Consequently, we create a renderpass which will specify to the device how many swap chain pictures and buffers we will be using, what kind of buffers they are and how they are to be used.

The different stages the picture will go through are as follows:



The topics covered in this chapter are as follows:

- Creating the SwapChain
- Creating the Renderpass
- Using render views and Framebuffers
- Creating CommandBuffer
- Beginning and ending Renderpass
- Creating the clear screen

Creating the SwapChain

While the scene is rendered, the buffers are swapped and presented to the window surface. The surface is platform-dependent and, depending upon the operating system, we have to choose the surface format accordingly. For the scene to be presented properly, we create the SwapChain depending upon the surface format, presentation mode, and the extent, meaning the width and height of the picture that the window can support.

In Chapter 10, *Drawing Vulkan Objects*, when we chose the GPU device to use, we retrieved the properties of the device, such as the surface format and the presentation modes it supports. While we create the SwapChain, we match and check the surface format and the presentation that is available from the device, and that is also supported by the window to create the SwapChain object itself.

We create a new class called SwapChain and add the following includes in SwapChain.h:

```
#include <vulkan\vulkan.h>
#include <vector>
#include <set>
#include <algorithm>
```

We then create the class, as follows:

```
class SwapChain {
public:
    SwapChain();
    ~SwapChain();

    VkSwapchainKHR swapChain;
    VkFormat swapChainImageFormat;
    VkExtent2D swapChainImageExtent;
    std::vector<VkImage> swapChainImages;
```

```

VkSurfaceFormatKHRchooseSwapChainSurfaceFormat (const
std::vector<VkSurfaceFormatKHR>&availableFormats);
VkPresentModeKHRchooseSwapPresentMode (const
std::vector<VkPresentModeKHR>availablePresentModes);
VkExtent2DchooseSwapExtent (const VkSurfaceCapabilitiesKHR&capabilities);
void create (VkSurfaceKHRsurface);

void destroy ();
}

```

In the public section of the class, we create the constructor and the destructor. Then we create the variables of the `VkSwapchainKHR`, `VkFormat`, and `VkExtent2D` types to store the swapchain itself. When we create the surface, we store the format of the picture itself, which is supported as well as the extent of the picture, which is the width and height of the viewport. This is because when the viewport is stretched or changed, the size of the swapchain picture will also be changed accordingly.

We create a vector of the `VkImage` type, called `swapChainImages`, to store the SwapChain pictures. Three helper functions, `chooseSwapChainSurfaceFormat`, `chooseSwapPresentMode`, and `chooseSwapExtent`, are created to get the most suitable surface format, present mode, and SwapChain extent. Finally, the `create` function takes the surface in which we will create the swapchain itself. We also add a function to destroy and release the resources back to the system.

This is it for the `SwapChain.h` file. We will now move on to `SwapChain.cpp` to add in the implementation of the functions.

In the `SwapChain.cpp` file, add the following includes:

```

#include "SwapChain.h"

#include "VulkanContext.h"

```

We will need to include `VulkanContext.h` to get the device's `SwapChainSupportDetails` struct, which we populated in the last chapter when we selected the Physical Device and created the Logical Device. Before we create the swapchain, let's first look at the three helper functions and see how each is created.

The first of the three functions is `chooseSwapChainSurfaceFormat`. This function takes in a vector of `VkSurfaceFormatKHR`, which is the available format supported by the device. Using this function, we will choose the surface format that is most suitable. The function is created as follows:

```

VkSurfaceFormatKHRSwapChain::chooseSwapChainSurfaceFormat (const

```

```
std::vector<VkSurfaceFormatKHR>&availableFormats) {  
  
    if (availableFormats.size() == 1 &&availableFormats[0].format ==  
VK_FORMAT_UNDEFINED) {  
        return{VK_FORMAT_B8G8R8A8_UNORM, VK_COLOR_SPACE_SRGB_NONLINEAR_KHR  
};  
    }  
  
    for (constauto& availableFormat : availableFormats) {  
        if (availableFormat.format == VK_FORMAT_B8G8R8A8_UNORM&&  
availableFormat.colorSpace == VK_COLOR_SPACE_SRGB_NONLINEAR_KHR) {  
            return availableFormat;  
        }  
    }  
    returnavailableFormats[0];  
}
```

First, we check whether the available format is just 1 and it is undefined by the device. This means that there is no preferred format, so we choose the one that is most convenient for us.

The values returned are the color format and the color space. The color format specifies the format of the color itself, `VK_FORMAT_B8G8R8A8_UNORM`, which tells us that we store 32 bits of information in each pixel. The colors are stored in the Blue, Green, Red, and Alpha channels, and in that order. Each channel is stored in 8 bits, so that means 2^8 , which is 256 color values. `UNORM` suggests that each color value is normalized so the color values, instead of being from 0-255, are normalized between 0 and 1.

We choose the SRGB color space as the second parameter, as we want more of a range of colors to be represented. If there is no preferred format, we go through the formats available and then check and return the ones that we need. We choose this color space, as most surfaces support this format because it is widely available. Otherwise, we just return the first available format.

The next function is `chooseSwapPresentMode` which takes in a vector of `VkPresentModeKHR` called `availablePresentModes`. Presentation modes specify how the final rendered picture is presented to the viewport. Here are the available modes:

- `VK_PRESENT_MODE_IMMEDIATE_KHR`: In this case, the picture will be displayed as soon as a picture is available to present. Pictures are not queued to be displayed. This causes picture tearing.

- **VK_PRESENT_MODE_FIFO_KHR:** The acquired pictures to be presented are put in a queue. The size of the queue is one minus the swap chain size. At vsync, the first picture to be displayed gets displayed in the **First In, First Out (FIFO)** manner. There is no tearing as pictures are displayed in the same order in which they were added to the queue and vsync is enabled. This mode needs to always be supported.
- **VK_PRESENT_MODE_FIFO_RELAXED_KHR:** This is a variation of the **FIFO** mode. In this mode, if the rendering is faster than the refresh rate of the monitor, it is fine, but if the drawing is slower than the monitor, there will be screen tearing as the next available picture is presented immediately.
- **VK_PRESENTATION_MODE_MAILBOX_KHR:** The presentation of the pictures is put in a queue but it has just one element in it, unlike **FIFO** which has more than one element in the queue. The next picture to be displayed will wait for the queue to be displayed and then the presentation engine will display the picture. This doesn't cause tearing.

With this information, let's create the `chooseSwapPresentMode` function:

```
VkPresentModeKHRSwapChain::chooseSwapPresentMode(const
std::vector<VkPresentModeKHR>availablePresentModes) {

    VkPresentModeKHR bestMode = VK_PRESENT_MODE_FIFO_KHR;

    for (const auto& availablePresentMode : availablePresentModes) {

        if (availablePresentMode == VK_PRESENT_MODE_MAILBOX_KHR) {
            return availablePresentMode;
        }
        elseif (availablePresentMode == VK_PRESENT_MODE_IMMEDIATE_KHR) {
            bestMode = availablePresentMode;
        }
    }

    return bestMode;
}
```

Since the **FIFO** mode is our most preferred mode, we set it in the function so that we can compare it with the available modes of the device. If it is not available, we go for the next best mode, which is the **MAILBOX** mode, so that the presentation queue will have at least one more picture to avoid screen tearing. If neither mode is available, we go for the **IMMEDIATE** mode, which is least desirable.

The third function is the `chooseSwapExtent` function. In this function, we get the resolution of the window that we drew to set the resolution of the swapchain pictures. It is added as follows:

```
VkExtent2DSwapChain::chooseSwapExtent (const VkSurfaceCapabilitiesKHR& capabilities) {

    if (capabilities.currentExtent.width != std::numeric_limits<uint32_t>::max()) {
        return capabilities.currentExtent;
    }
    else {

        VkExtent2D actualExtent = { 1280, 720 };

        actualExtent.width = std::max(capabilities.minImageExtent.width,
            std::min(capabilities.maxImageExtent.width, actualExtent.width));
        actualExtent.height = std::max(capabilities.minImageExtent.height,
            std::min(capabilities.maxImageExtent.height, actualExtent.height));

        return actualExtent;
    }
}
```

The resolution of this window should match the swapchain pictures. Some window managers allow the resolution to be different between the pictures and the window. This is indicated by setting the value to the maximum of `uint32_t`. If not then in that case we return the current extent that we retrieved by the capabilities of the hardware, or pick the resolution that best matches the resolution between the maximum and minimum values available, as compared to the actual resolution we set which is 1,280 x 720.

Let's now look at the `create` function in which we actually create the SwapChain itself. To create this function, we will add the functionality to create the SwapChain:

```
void SwapChain::create(VkSurfaceKHR surface) {
    ...
}
```

The first thing we do is get the device support details, which we retrieved for our device when we created the `Device` class:

```
SwapChainSupportDetails swapChainSupportDetails =
VulkanContext::getInstance() -> getDevice() -> swapchainSupport;
```

Then using the helper function we created, we get the surface format, present mode, and the extent:

```
VkSurfaceFormatKHR surfaceFormat =
chooseSwapChainSurfaceFormat(swapChainSupportDetails.surfaceFormats);
VkPresentModeKHR presentMode =
chooseSwapPresentMode(swapChainSupportDetails.presentModes);
VkExtent2D extent =
chooseSwapExtent(swapChainSupportDetails.surfaceCapabilities);
```

We then set the minimum number of pictures required to make the swapchain:

```
uint32_t imageCount =
swapChainSupportDetails.surfaceCapabilities.minImageCount;
```

We should also make sure that we don't exceed the maximum available picture count, so if the `imageCount` is more than the maximum amount, we set `imageCount` to the maximum count:

```
if (swapChainSupportDetails.surfaceCapabilities.maxImageCount > 0 &&
imageCount > swapChainSupportDetails.surfaceCapabilities.maxImageCount) {
    imageCount =
swapChainSupportDetails.surfaceCapabilities.maxImageCount;
}
```

To create the swapchain, we have to populate the `VkSwapchainCreateInfoKHR` struct first, so let's create it. Create a variable called `createInfo` of the type and specify the type of the structure:

```
VkSwapchainCreateInfoKHR createInfo = {};
createInfo.sType = VK_STRUCTURE_TYPE_SWAPCHAIN_CREATE_INFO_KHR;
```

In this, we have to specify the surface to use, minimum picture count, picture format, space, and extent. We need to specify the picture array layers. Since we are not going to create a stereoscopic application like a virtual reality game in which there will be two surfaces: one for the left eye and one for the right eye. Instead, we just set the value for it as 1. We also need to specify what the picture will be used for. Here it will be used to show the color information using the color attachment:

```
createInfo.surface = surface;
createInfo.minImageCount = imageCount;
createInfo.imageFormat = surfaceFormat.format;
createInfo.imageColorSpace = surfaceFormat.colorSpace;
createInfo.imageExtent = extent;
createInfo.imageArrayLayers = 1; // this is 1 unless you are making
a stereoscopic 3D application
createInfo.imageUsage = VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT;
```

We now specify the graphics, presentation indices, and the count. We also specify the sharing mode. It is possible for the presentation and graphics family to be the same or different.

If the presentation and graphics family is different, the sharing mode is said to be of the `VK_SHARING_MODE_CONCURRENT` type. This means that the picture can be used across multiple queue families. However, if the picture is in the same queue family, the sharing mode is said to be of the `VK_SHARING_MODE_EXCLUSIVE` type:

```
if (indices.graphicsFamily != indices.presentFamily) {

    createInfo.imageSharingMode = VK_SHARING_MODE_CONCURRENT;
    createInfo.queueFamilyIndexCount = 2;
    createInfo.pQueueFamilyIndices = queueFamilyIndices;

}

else {

    createInfo.imageSharingMode = VK_SHARING_MODE_EXCLUSIVE;
    createInfo.queueFamilyIndexCount = 0;
    createInfo.pQueueFamilyIndices = nullptr;
}
```

If we want, we can apply a pre-transform to the picture to either flip it or mirror it. In this case, we just keep the current transform. We can also alpha-blend the picture with other window systems but we just keep it opaque and ignore the alpha channel, set the present mode, and set whether the pixel should be clipped if there is a window in front. We can also specify an old `SwapChain` if the current one becomes invalid when we resize the window. Since we don't resize the window, we don't have to specify an older swapchain.

After setting the info struct, we can create the swapchain itself:

```
if
(vkCreateSwapchainKHR(VulkanContext::getInstance() -> getDevice() -> logicalDevice,
    &createInfo, nullptr, &swapChain) != VK_SUCCESS) {
    throw std::runtime_error("failed to create swap chain !");
}
```

We create the swapchain using the `vkCreateSwapchainKHR` function, which takes the logical device, the create info struct, an allocator callback, and the swapchain itself. If it doesn't create the `SwapChain` because of an error, we will send out an error. Now that the swapchain is created, we will obtain the swapchain pictures.

Depending upon the picture count, we call the `vkGetSwapchainImagesKHR` function, which we use to first get the picture count and then call the function again to populate the `vkImage` vector with the pictures:

```
vkGetSwapchainImagesKHR(VulkanContext::getInstance() -> getDevice() -> logicalDevice, swapChain, &imageCount, nullptr);
    swapChainImages.resize(imageCount);
vkGetSwapchainImagesKHR(VulkanContext::getInstance() -> getDevice() -> logicalDevice, swapChain, &imageCount, swapChainImages.data());
```

The creation of pictures is a bit more involved, but Vulkan automatically creates color pictures. We can set the picture format and extent as well:

```
swapChainImageFormat = surfaceFormat.format;
swapChainImageExtent = extent;
```

Then we add the destroy function, which destroys the swapchain by calling the `vkDestroySwapchainKHR` function:

```
void SwapChain::destroy() {
    // Swapchain
    vkDestroySwapchainKHR(VulkanContext::getInstance() -> getDevice() -> logicalDevice, swapChain, nullptr);
}
```

In the `VulkanApplication.h` file, include the `SwapChain` header and create a new `SwapChain` instance in the `VulkanApplication` class. In `VulkanApplication.cpp`, in the `initVulkan` function after creating the logical device, create the `SwapChain` as follows:

```
swapChain = new SwapChain();
swapChain->create(surface);
```

Build and run the application to make sure the swapchain is created without any errors.

Creating the Renderpass

After creating the swapchain, we move on to the render pass. Here, we specify how many color attachments and depth attachments are present and how many samples to use for each of them for each Framebuffer.

As mentioned at the start of the Preparing Clear Screen chapter, a Framebuffer is a collection of target attachments. Attachments can be of type color, depth, and so on. The color attachment stores the color information that is presented to the viewport. There are other attachments that the end user doesn't see but are used internally. This includes depth, for example, which has all the depth information per pixel. In the render pass, apart from the type of attachments, we also specify how the attachments are used.

For this book, we will be presenting what is rendered in a scene to the viewport, so we will just use a single pass. In case we add a post-processing effect, we will take the rendered picture and apply this effect to it, for which we will need to use multiple passes. We will create a new class called `Renderpass`, in which we will create the render pass.

In the `Renderpass.h` file, add the following includes and class:

```
#include <vulkan\vulkan.h>
#include <array>

class Renderpass
{
public:
    Renderpass();
    ~Renderpass();

    VkRenderPass renderPass;

    void createRenderPass(VkFormat swapChainImageFormat);

    void destroy();
};
```

In the class, add the constructor, destructor, and the `VkRenderPass` and `renderPass` variables. Add a new function called `createRenderPass` to create the `Renderpass` itself, which takes in the picture format. Also add a function to destroy the `Renderpass` object after use.

In the `Renderpass.cpp` file, add the following includes and the constructor and destructor:

```
#include "Renderpass.h"
#include "VulkanContext.h"
Renderpass::Renderpass() {}

Renderpass::~Renderpass() {}
```

We now add the `createRenderPass` function, in which we will add the functionality to create the Renderpass for the current scene to be rendered:

```
void Renderpass::createRenderPass(VkFormat swapChainImageFormat) {  
    ...  
}
```

When we create the render pass, we have to specify the number and the type of attachments that we are using. So for our project we want only color attachments as we will only be drawing color information. We could also have a depth attachment, which stores depth information. We need to provide subpasses, and if so then how many as we could be using subpasses for adding post-processing effects to the current frame.

For the attachments and subpasses, we have to populate structs to pass to them at the time of creating the render pass.

So, let's populate the structs. First, we create the attachments:

```
VkAttachmentDescription colorAttachment = {};  
colorAttachment.format = swapChainImageFormat;  
colorAttachment.samples = VK_SAMPLE_COUNT_1_BIT;  
colorAttachment.loadOp = VK_ATTACHMENT_LOAD_OP_CLEAR;  
colorAttachment.storeOp = VK_ATTACHMENT_STORE_OP_STORE;  
colorAttachment.stencilLoadOp = VK_ATTACHMENT_LOAD_OP_DONT_CARE;  
colorAttachment.stencilStoreOp = VK_ATTACHMENT_STORE_OP_DONT_CARE;  
  
colorAttachment.initialLayout = VK_IMAGE_LAYOUT_UNDEFINED;  
colorAttachment.finalLayout = VK_IMAGE_LAYOUT_PRESENT_SRC_KHR;
```

We create the struct and specify the format to be used, which is the same as the `swapChainImage` format. We have to provide the sample count as 1 as we are not going to be using multi-sampling. In the loadop and storeop, we specify what to do with the data before and after rendering. We specify that at the time of loading the attachment, we will clear the data to a constant at the start. After the render process, we store the data so we can read from it later. We then decide what to do with the data before and after the stencil operation. Since we are not using the stencil buffer we specify DON'T CARE during loading and storing. We also have to specify the data layout before and after processing the picture. The previous layout of the picture doesn't matter, but after rendering, the picture needs to be changed to the layout for it to be ready for presenting.

Now we go through the subpass. Each subpass references the attachments that need to be specified as a separate structure:

```
VkAttachmentReference colorAttachRef = {};
colorAttachRef.attachment = 0;
colorAttachRef.layout = VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL;
```

In the subpass reference, we specify the attachment index, which is the 0th index and specify the layout, which is a color attachment with optimal performance. Next, we create the subpass structure:

```
VkSubpassDescription subpass = {};
subpass.pipelineBindPoint = VK_PIPELINE_BIND_POINT_GRAPHICS;
subpass.colorAttachmentCount = 1;
subpass.pColorAttachments = &colorAttachRef;
```

In the pipeline bind point, we specify that this is a graphics subpass, as it could have been a compute subpass. Specify the attachment count to be 1 and provide the color attachment. Now we can create the renderpass info struct:

```
std::array<VkAttachmentDescription, 1> attachments = { colorAttachment };

VkRenderPassCreateInfo rpCreateInfo = {};
rpCreateInfo.sType = VK_STRUCTURE_TYPE_RENDER_PASS_CREATE_INFO;
rpCreateInfo.attachmentCount =
static_cast<uint32_t>(attachments.size());
rpCreateInfo.pAttachments = attachments.data();
rpCreateInfo.subpassCount = 1;
rpCreateInfo.pSubpasses = &subpass;
```

We create an array of one element of the `VkAttachmentDescription` type, and then we create the info struct and pass in the type. The attachment count and the attachments are passed in, and then the subpass count and the subpass is passed in as well. Create the renderpass itself by calling `vkCreateRenderPass` and passing in the logical device, the create info, and the allocator callback to get the renderpass:

```
if (vkCreateRenderPass(VulkanContext::getInstance() ->
getDevice() -> logicalDevice, &rpCreateInfo, nullptr, &renderPass) !=
VK_SUCCESS) {
    throw std::runtime_error(" failed to create renderpass !! ");
}
```

Finally, in the destroy function, we call `vkDestroyRenderPass` to destroy it after we are done:

```
void Renderpass::destroy() {
    vkDestroyRenderPass(VulkanContext::getInstance() ->
        getDevice() -> logicalDevice, renderPass, nullptr);

}
```

In `VulkanApplication.h`, include `RenderPass.h` and create an object of render pass. In `VulkanApplication.cpp`, after creating the swapchain, create the renderpass:

```
renderPass = new Renderpass();
renderPass->createRenderPass(swapChain->swapChainImageFormat);
```

Now build and run the project to make sure there are no errors.

Using render targets and Framebuffers

To use a picture, we have to create an `ImageView`. The picture doesn't have any information, such as mipmap levels, and you can't access a portion of the picture. Mipmaps are explained in OpenGL when we covered texture filtering while loading textures.

By using picture views, we specify the type of the texture and whether it has mipmaps. In addition, in `renderpass` we specified the attachments per frame buffer. We will create Framebuffers here and pass in the picture views as attachments.

Create a new class called `RenderTexture`. In the `RenderTexture.h` file, add the following headers and then create the class itself:

```
#include <vulkan/vulkan.h>
#include<array>

class RenderTexture
{
public:
    RenderTexture();
    ~RenderTexture();
    std::vector<VkImage> _swapChainImages;
    VkExtent2D _swapChainImageExtent;

    std::vector<VkImageView> swapChainImageViews;
    std::vector<VkFramebuffer> swapChainFramebuffers;
```

```
void createViewsAndFramebuffer(std::vector<VkImage> swapChainImages,
                               VkFormat swapChainImageFormat,
                               VkExtent2D swapChainImageExtent,
                               VkRenderPass
renderPass);

void createImageViews(VkFormat swapChainImageFormat);
void createFrameBuffer(VkExtent2D swapChainImageExtent, VkRenderPass
renderPass);

void destroy();

};

};
```

In the class, we add the constructor and destructor as usual. We will store `swapChainImages` and the extent to use it locally. We create two vectors to store the created `ImageViews` and `Framebuffers`. For creating the views and `Framebuffers`, we will call the `createViewsAndFramebuffers` function which take the pictures, picture format, extent, and the renderpass as the input. This function intern will call `createImageViews` and `CreateFramebuffer` to create them. We will add the `destroy` function, which destroys and releases the resources back to the system.

In the `RenderTexture.cpp` file, we will add the following includes as well as the constructor and destructor:

```
#include "RenderTexture.h"
#include "VulkanContext.h"
RenderTexture::RenderTexture() {}

RenderTexture::~RenderTexture()
```

Then add the `createViewAndFramebuffer` function:

```
void RenderTexture::createViewAndFramebuffer(std::vector<VkImage>
swapChainImages, VkFormat swapChainImageFormat,
VkExtent2D swapChainImageExtent,
VkRenderPass renderPass){

    _swapChainImages = swapChainImages;
    _swapChainImageExtent = swapChainImageExtent;

    createImageViews(swapChainImageFormat);
    createFrameBuffer(swapChainImageExtent, renderPass);
}
```

We first assign the images and `imageExtent` to the local variables. Then we call the `imageViews` function followed by `createFramebuffer` in order to create both of them. To create the image views, use the `createImageViews` function:

```
void RenderTexture::createImageViews(VkFormat swapChainImageFormat) {  
  
    swapChainImageViews.resize(_swapChainImages.size());  
  
    for (size_t i = 0; i < _swapChainImages.size(); i++) {  
  
        swapChainImageViews[i] =  
            vkTools::createImageView(_swapChainImages[i],  
                swapChainImageFormat,  
                VK_IMAGE_ASPECT_COLOR_BIT);  
    }  
}
```

We specify the vector size depending upon the swapchain image count first. For each of the image counts, we create image views using the `createImageView` function in the `vkTool` namespace. The `createImageView` function takes in the image itself, the image format, and `ImageAspectFlag`. This will be `VK_IMAGE_ASPECT_COLOR_BIT` or `VK_IMAGE_ASPECT_DEPTH_BIT` depending upon the kind of view that you want to create for the image. The `createImageView` function is created in the Tools file under the `vkTools` namespace. The `Tools.h` file is as follows:

```
#include <vulkan\vulkan.h>  
#include <stdexcept>  
#include <vector>  
  
namespace vkTools {  
  
    VkImageView createImageView(VkImage image, VkFormat format,  
        VkImageAspectFlags aspectFlags);  
  
}
```

The implementation of the function is created in the `Tools.cpp` file as follows:

```
#include "Tools.h"  
#include "VulkanContext.h"  
  
namespace vkTools {  
    VkImageView createImageView(VkImage image, VkFormat format,  
        VkImageAspectFlags aspectFlags) {
```

```
VkImageViewCreateInfo viewInfo = {};
viewInfo.sType = VK_STRUCTURE_TYPE_IMAGE_VIEW_CREATE_INFO;
viewInfo.image = image;
viewInfo.viewType = VK_IMAGE_VIEW_TYPE_2D;
viewInfo.format = format;

viewInfo.subresourceRange.aspectMask = aspectFlags;
viewInfo.subresourceRange.baseMipLevel = 0;
viewInfo.subresourceRange.levelCount = 1;
viewInfo.subresourceRange.baseArrayLayer = 0;
viewInfo.subresourceRange.layerCount = 1;

VkImageView imageView;
if
(vkCreateImageView(VulkanContext::getInstance() ->getDevice() ->logicalDevice
, &viewInfo, nullptr, &imageView) != VK_SUCCESS) {
    throw std::runtime_error("failed to create texture image
view !");
}

return imageView;
}
```

}

To create the `imageView`, we have to populate the `VkImageViewCreateInfo` struct and then use the `vkCreateImageView` function to create the view itself. To populate the view info, we specify the structure type, the picture itself, the view type which is `VK_IMAGE_VIEW_TYPE_2D` and a 2D texture, and then specify the format. We pass in the `aspectFlags` for the aspect mask. We create the image view without any mipmap level or layers so we set them to 0. We would only need multiple layers if we were making something like a VR game.

We then create an `imageView` of the `VkImage` type and create it using the `vkCreateImageView` function, which takes in the logical device, the view info struct, and then the picture view is created and returned. That's all for the Tools file.

We will use the Tools file and add more functions to it when we want functions that can be reused. Now, let's go back to the `RenderTexture.cpp` file and add in the function to create the Framebuffer.

We will create Framebuffers for each frame in the swapchain. `createFramebuffer` requires the picture extent and the renderpass itself:

```
void RenderTexture::createFrameBuffer(VkExtent2D swapChainImageExtent,
VkRenderPass renderPass) {

    swapChainFramebuffers.resize(swapChainImageViews.size());

    for (size_t i = 0; i < swapChainImageViews.size(); i++) {

        std::array<VkImageView, 2> attachments = {
            swapChainImageViews[i]
        };

        VkFramebufferCreateInfo fbInfo = {};
        fbInfo.sType = VK_STRUCTURE_TYPE_FRAMEBUFFER_CREATE_INFO;
        fbInfo.renderPass = renderPass;
        fbInfo.attachmentCount =
            static_cast<uint32_t>(attachments.size());
        fbInfo.pAttachments = attachments.data();
        fbInfo.width = swapChainImageExtent.width;
        fbInfo.height = swapChainImageExtent.height;
        fbInfo.layers = 1;

        if
            (vkCreateFramebuffer(VulkanContext::getInstance()>getDevice()>logicalDevice,
            &fbInfo, NULL, &swapChainFramebuffers[i]) != VK_SUCCESS) {

                throw std::runtime_error(" failed to create framebuffers
!!!");
            }
        }
    }
}
```

For each frame that we create, the Framebuffer first populates the `framebufferInfo` struct and then calls `vkCreateFramebuffer` to create the Framebuffer itself. For each frame, we create a new info struct and specify the type of struct. We then pass the renderpass, the attachment count, and the attachment views, specify the width and height of the Framebuffer, and set the layers to 1.

Finally, we create the Framebuffer by calling the `vkCreateFramebuffer` function:

```
void RenderTexture::destroy() {

    // image views
    for (auto imageView : swapChainImageViews) {
```

```

vkDestroyImageView(VulkanContext::getInstance() -> getDevice() -> logicalDevice
, imageView, nullptr);
}

// Framebuffers
for (auto framebuffer : swapChainFramebuffers) {
vkDestroyFramebuffer(VulkanContext::getInstance() -> getDevice() -> logicalDevice,
framebuffer, nullptr);
}

}

```

In the destroy function, we destroy each of the picture views and Framebuffers we created by calling `vkDestroyImageView` and `vkDestroyFramebuffer`. And that is all for the `RenderTexture` class.

In `VulkanApplication.h`, include the `RenderTexture.h` and create an instance of it called `renderTexture` in the `VulkanApplication` class. In the `VulkanApplication.cpp` file, include the `initVulkan` function and create a new `RenderTexture`:

```

renderTexture = new RenderTexture();
renderTexture->createViewsAndFramebuffer(swapChain->swapChainImages,
swapChain->swapChainImageFormat,
swapChain->swapChainImageExtent,
renderPass->renderPass);

```

Creating CommandBuffer

In Vulkan, the drawing and other operations done on the GPU are performed using command buffers. The command buffers contain the draw commands, which are recorded and then executed. Draw commands are to be recorded and executed in every frame. To create a command buffer, we have to first create a command pool and then allocate command buffers from the command pool. Then the commands are recorded per frame.

Let's create a new class for creating the command buffer pool and then allocate the command buffers. We also create a function to start and stop recording and to destroy the command buffers. Create a new class, called `DrawCommandBuffer`, and `DrawCommandBuffer.h` as follows:

```

#include <vulkan\vulkan.h>
#include <vector>

```

```
class DrawCommandBuffer
{
public:
    DrawCommandBuffer();
    ~DrawCommandBuffer();

    VkCommandPool commandPool;
    std::vector<VkCommandBuffer> commandBuffers;

    void createCommandPoolAndBuffer(size_t imageCount);
    void beginCommandBuffer(VkCommandBuffer commandBuffer);
    void endCommandBuffer(VkCommandBuffer commandBuffer);

    void createCommandPool();
    void allocateCommandBuffers(size_t imageCount);

    void destroy();
};
```

In the class, we create the constructor and destructor. We create variables to store the command pool and a vector to store `VkCommandBuffer`. We create one function initially to create the command pool and allocate the command buffers. The next two functions `beginCommandBuffer` and `endCommandBuffer` will be called when we want to start and stop recording the command buffer. The `createCommandPool` and `allocateCommandBuffers` functions will be called by `createCommandPoolAndBuffer`.

We will create the `destroy` function to destroy the command buffers when we want the resources to be released to the system. In `CommandBuffer.cpp`, add the necessary includes and the constructor and destructor:

```
#include "DrawCommandBuffer.h"
#include "VulkanContext.h"

DrawCommandBuffer::DrawCommandBuffer() {}

DrawCommandBuffer::~DrawCommandBuffer() {}
```

Then we add `createCommandPoolAndBuffer`, which takes in the picture count:

```
void DrawCommandBuffer::createCommandPoolAndBuffer(size_t imageCount) {

    createCommandPool();
    allocateCommandBuffers(imageCount);
}
```

The `createCommandPoolAndBuffer` function will call the `createCommandPool` and `allocateCommandBuffers` functions. First, we create the `createCommandPool` function. Commands have to be sent to a certain queue. We have to specify the queue when we create the command pool:

```
void DrawCommandBuffer::createCommandPool() {
    QueueFamilyIndices qFamilyIndices =
        VulkanContext::getInstance()->getDevice()->getQueueFamiliesIndicesOfCurrentDevice();

    VkCommandPoolCreateInfo cpInfo = {};

    cpInfo.sType = VK_STRUCTURE_TYPE_COMMAND_POOL_CREATE_INFO;
    cpInfo.queueFamilyIndex = qFamilyIndices.graphicsFamily;
    cpInfo.flags = VK_COMMAND_POOL_CREATE_RESET_COMMAND_BUFFER_BIT;

    if (vkCreateCommandPool(VulkanContext::getInstance()->getDevice()->logicalDevice, &cpInfo, nullptr, &commandPool) != VK_SUCCESS) {
        throw std::runtime_error(" failed to create command pool !!");
    }
}
```

To start, we get the queue family indices for the current device. To create the command pool, we have to populate the `VkCommandPoolCreateInfo` struct. As usual, we specify the type. Then we set the queue family index in which the pool has to be created. After that, we set the `VK_COMMAND_POOL_CREATE_RESET_COMMAND_BUFFER_BIT` flags, which will reset the values of the command buffer every time. We then use the `vkCreateCommandPool` function by passing in the logical device and the info struct to get the command pool. Next, we create the `allocateCommandBuffers` function:

```
void DrawCommandBuffer::allocateCommandBuffers(size_t imageCount) {
    commandBuffers.resize(imageCount);

    VkCommandBufferAllocateInfo cbInfo = {};
    cbInfo.sType = VK_STRUCTURE_TYPE_COMMAND_BUFFER_ALLOCATE_INFO;
    cbInfo.commandPool = commandPool;
    cbInfo.level = VK_COMMAND_BUFFER_LEVEL_PRIMARY;
    cbInfo.commandBufferCount = (uint32_t)commandBuffers.size();

    if (vkAllocateCommandBuffers(VulkanContext::getInstance()->getDevice()->logicalDevice, &cbInfo, commandBuffers.data()) != VK_SUCCESS) {
```

```
        throw std::runtime_error(" failed to allocate command buffers  
        !!");  
    }  
  
}
```

We resize the `commandBuffers` vector. Then, to allocate the command buffers, we have to populate `VkCommandBufferAllocateInfo`. We first set the type of the struct and the command pool. Then we have to specify the level of the command buffers. You can have a chain of command buffers with the primary command buffer containing the secondary command buffer. For our use, we will set the command buffers as primary. We then set `commandBufferCount`, which is equal to the swapchain pictures.

Then we allocate the command buffers using the `vkAllocateCommandBuffers` function. We pass in the logical device, the info struct, and the command buffers to allocate memory for the command buffers.

Then we add `beginCommandBuffer`. This takes in the current command buffer to start recording command into it:

```
void DrawCommandBuffer::beginCommandBuffer(VkCommandBuffer commandBuffer) {  
  
    VkCommandBufferBeginInfo cbBeginInfo = {};  
  
    cbBeginInfo.sType = VK_STRUCTURE_TYPE_COMMAND_BUFFER_BEGIN_INFO;  
    cbBeginInfo.flags = VK_COMMAND_BUFFER_USAGE_SIMULTANEOUS_USE_BIT;  
  
    if (vkBeginCommandBuffer(commandBuffer, &cbBeginInfo) != VK_SUCCESS) {  
  
        throw std::runtime_error(" failed to begin command buffer !!");  
    }  
  
}
```

To record command buffers, we also have to populate the `VkCommandBufferBeginInfoStruct`. Once again we specify the struct type and the `VK_COMMAND_BUFFER_USAGE_SIMULTANEOUS_USE_BIT` flag. This enables us to schedule the command buffer for the next frame while the last frame is still in use. `vkBeginCommandBuffer` is called to start recording the commands by passing in the current command buffer.

Next we add in the `endCommandBuffer` function. This function just calls `vkEndCommandBuffer` to stop recording to the command buffer:

```
void DrawCommandBuffer::endCommandBuffer(VkCommandBuffer commandBuffer) {  
  
    if (vkEndCommandBuffer(commandBuffer) != VK_SUCCESS) {  
  
        throw std::runtime_error(" failed to record command buffer");  
    }  
  
}
```

We can then destroy the command buffers and the pool using the `Destroy` function. Here, we just destroy the pool, which will destroy the command buffers as well:

```
void DrawCommandBuffer::destroy() {  
  
    vkDestroyCommandPool(VulkanContext::getInstance()->getDevice()->logicalDevice, commandPool, nullptr);  
  
}
```

In the `VulkanApplication.h` file, include `DrawCommandBuffer.h` and create an object of this class. In `VulkanApplication.cpp`, in the `VulkanInit` function after creating the `renderViewsAndFrameBuffers`, create `DrawCommandBuffer`:

```
renderTexture = new RenderTexture();  
renderTexture->createViewsAndFramebuffer(swapChain->swapChainImages,  
    swapChain->swapChainImageFormat,  
    swapChain->swapChainImageExtent,  
    renderPass->renderPass);  
  
drawComBuffer = new DrawCommandBuffer();  
drawComBuffer->createCommandPoolAndBuffer(swapChain->swapChainImages.size());
```

Begining and ending Renderpass

Along with the commands being recorded in each frame, the renderpass is also processed for each frame where the the color and the depth information is reset. So since we only have color attachments in each frame, we have to clear the color information for each frame as well. Go back to the `Renderpass.h` file and add two new functions, called `beginRenderPass` and `endRenderPass`, in the class, as follows:

```
class Renderpass
{
public:
    Renderpass();
    ~Renderpass();

    VkRenderPass renderPass;

    void createRenderPass(VkFormat swapChainImageFormat);

    void beginRenderPass(std::array<VkClearValue, 1> clearValues,
                         VkCommandBuffer commandBuffer, VkFramebuffer swapChainFrameBuffer,
                         VkExtent2D swapChainImageExtent);
    void endRenderPass(VkCommandBuffer commandBuffer);

    void destroy();
};
```

In `RenderPass.cpp`, add the implementation of the `beginRenderPass` function:

```
void Renderpass::beginRenderPass(std::array<VkClearValue, 1> clearValues,
                                 VkCommandBuffer commandBuffer,
                                 VkFramebuffer swapChainFrameBuffer,
                                 VkExtent2D swapChainImageExtent) {

    VkRenderPassBeginInfo rpBeginInfo = {};
    rpBeginInfo.sType = VK_STRUCTURE_TYPE_RENDER_PASS_BEGIN_INFO;
    rpBeginInfo.renderPass = renderPass;
    rpBeginInfo.framebuffer = swapChainFrameBuffer;
    rpBeginInfo.renderArea.offset = { 0,0 };
    rpBeginInfo.renderArea.extent = swapChainImageExtent;

    rpBeginInfo.pClearValues = clearValues.data();
    rpBeginInfo.clearValueCount = static_cast<uint32_t>(clearValues.size());

    vkCmdBeginRenderPass(commandBuffer, &rpBeginInfo, VK_SUBPASS_CONTENTS_INLINE)
    ;
}
```

We then populate the `VkRenderPassBeginInfo` struct. In this, we specify the struct type, pass in the renderpass and the current Framebuffer, set the render area as the whole viewport, and pass in the clear value and the count. The clear value is the color value we want to clear the screen with, and the count would be 1 as we would like to clear only the color attachment.

To begin the renderpass, we pass in the current command buffer, the info struct, and specify the third parameter as `VK_SUBPASS_CONTENTS_INLINE`, specifying that the renderpass commands are bound to the primary command buffer.

In the `endCommandBuffer` function, we finish the Renderpass for the current frame:

```
void Renderpass::endRenderPass(VkCommandBuffer commandBuffer) {
    vkCmdEndRenderPass(commandBuffer);
}
```

To end the Renderpass, the `vkCmdEndRenderPass` function is called and the current command buffer is passed in.

We have the required classes to get the clear screen going. Now let's go to the Vulkan Application class and add some lines of code to get it working.

Creating the clear screen

In the `VulkanApplication.h` file, we will add three new functions, called `drawBegin`, `drawEnd`, and `cleanup`. `drawBegin` will be called before we pass in any draw commands, and `drawEnd` will be called once the drawing is done and the frame is ready to be presented to the viewport. In the `cleanup` function, we will destroy all the resources.

We will also create two variables. The first is `uint32_t` to get the current picture from the swapchain and the second is `currentCommandBuffer` of type `VkCommandBuffer` to get the current command buffer:

```
public:
    static VulkanApplication* getInstance();
    static VulkanApplication* instance;

    ~VulkanApplication();

    void initVulkan(GLFWwindow* window);

    void drawBegin();
    void drawEnd();
```

```
void cleanup();  
  
private:  
  
    uint32_t imageIndex = 0;  
    VkCommandBuffer currentCommandBuffer;  
  
    //surface  
    VkSurfaceKHR surface;
```

In the `VulkanApplication.cpp` file we add the implementation of the `drawBegin` and `drawEnd` functions:

```
void VulkanApplication::drawBegin() {  
  
    vkAcquireNextImageKHR(VulkanContext::getInstance()->getDevice()->logicalDevice,  
                          swapChain->swapChain,  
                          std::numeric_limits<uint64_t>::max(),  
                          NULL,  
                          VK_NULL_HANDLE,  
                          &imageIndex);  
  
    currentCommandBuffer = drawComBuffer->commandBuffers[imageIndex];  
  
    // Begin command buffer recording  
    drawComBuffer->beginCommandBuffer(currentCommandBuffer);  
  
    // Begin renderpass  
    VkClearColorValue clearcolor = { 1.0f, 0.0f, 1.0f, 1.0f };  
  
    std::array<VkClearColorValue, 1> clearValues = { clearcolor };  
  
    renderPass->beginRenderPass(clearValues,  
                                 currentCommandBuffer,  
                                 renderTexture->swapChainFramebuffers[imageIndex],  
                                 renderTexture->_swapChainImageExtent);  
  
}
```

First, we acquire the next picture from the swap chain. This is done using the Vulkan `vkAcquireNextImageKHR` API call. To this, we pass in the logical device, the swapchain instance. Next we need to pass in timeout, for which we pass in the maximum numerical value as we don't care about the time limit. The next two variables are kept as null. These require a semaphore and fence, which we will discussed in a later chapter. Finally, we pass in the `imageIndex` itself.

Then we get the current command buffer from the command buffers vector. We begin recording the command buffer by calling `beginCommandBuffer` and the commands will be stored in the `currentCommandBuffer` object. We now start the renderpass. In this, we pass the clear color value which is the purple color, because why not? Pass in the current commandbuffer, the frame buffer, and the picture extent.

We can now implement the `drawEnd` function:

```
void VulkanApplication::drawEnd() {

    // End render pass commands
    renderPass->endRenderPass(currentCommandBuffer);

    // End command buffer recording
    drawComBuffer->endCommandBuffer(currentCommandBuffer);

    // submit command buffer
    VkSubmitInfo submitInfo = {};
    submitInfo.sType = VK_STRUCTURE_TYPE_SUBMIT_INFO;
    submitInfo.commandBufferCount = 1;
    submitInfo.pCommandBuffers = &currentCommandBuffer;

    vkQueueSubmit(VulkanContext::getInstance()->getDevice()->graphicsQueue,
    1, &submitInfo, NULL);

    // Present frame
    VkPresentInfoKHR presentInfo = {};
    presentInfo.sType = VK_STRUCTURE_TYPE_PRESENT_INFO_KHR;
    presentInfo.swapchainCount = 1;
    presentInfo.pSwapchains = &swapChain->swapChain;
    presentInfo.pImageIndices = &imageIndex;

    vkQueuePresentKHR(VulkanContext::getInstance()->getDevice()->presentQueue,
    &presentInfo);
    vkQueueWaitIdle(VulkanContext::getInstance()->getDevice()->presentQueue);

}
```

We end the renderpass and stop recording to the command buffer. Then we have to submit the command buffer and present the frame. To submit the command buffer, we create a `VkSubmitInfo` struct and populate it with the struct type, the buffer count which is 1 per frame, and the command buffer itself. The command is submitted to the graphics queue by calling `vkQueueSubmit` and passing in the graphics queue, the submission count, and the submit info. Once the frame is rendered, it is presented to the viewport using the present queue.

To present the scene once it is drawn, we have to create and populate the `VkPresentInfoKHR` struct. For presentation, the picture is sent back to the swapchain. When we create the info and set the type of the struct, we also have to set the swap chain, the image index, and the swapchain count which is 1.

We then present the picture using `vkQueuePresentKHR` by passing in the present queue and the present info to the function. At the end, we wait for the host to finish the presentation operation of a given queue using `vkQueueWaitIdle` function which takes in the present queue. Also, it is better to clean up the resources when you are done with them, so add the cleanup function as well:

```
void VulkanApplication::cleanup() {
    vkDeviceWaitIdle(VulkanContext::getInstance() -> getDevice() -> logicalDevice);

    drawComBuffer->destroy();
    renderTexture->destroy();
    renderPass->destroy();
    swapChain->destroy();

    VulkanContext::getInstance() -> getDevice() -> destroy();

    valLayersAndExt->destroy(vInstance->vkInstance,
        isValidationLayersEnabled);

    vkDestroySurfaceKHR(vInstance->vkInstance, surface, nullptr);
    vkDestroyInstance(vInstance->vkInstance, nullptr);
}

delete drawComBuffer;
delete renderTarget;
delete renderPass;
delete swapChain;
delete device;

delete valLayersAndExt;
delete vInstance;

if (instance) {
    delete instance;
    instance = nullptr;
}
```

When we destroy, we have to call `vkDeviceWaitIdle` to stop using the device. Then we destroy the objects in the reverse order. So we destroy the command buffer first, then the render texture resources, then renderpass, and then the swapchain. We now destroy the device, validation layer, surface, and finally the Vulkan instance. At the end we also delete the class instances we created for `DrawCommandBuffer`, `RenderTarget`, `Renderpass`, `Swapchain`, `Device`, `ValidationLayersAndExtensions`, and `VulkanInstance`.

And finally we also delete the instance of the `VulkanContext` as well and set it to `nullptr` after deleting it.

In the `source.cpp` file, in the while loop, call the `drawBegin` and `drawEnd` functions. Call the `cleanup` function after the loop:

```
#define GLFW_INCLUDE_VULKAN
#include<GLFW/glfw3.h>

#include "VulkanApplication.h"

int main() {

    glfwInit();

    glfwWindowHint(GLFW_CLIENT_API, GLFW_NO_API);
    glfwWindowHint(GLFW_RESIZABLE, GLFW_FALSE);

    GLFWwindow* window = glfwCreateWindow(1280, 720, "HELLO VULKAN ",
                                         nullptr, nullptr);

    VulkanApplication::getInstance()->initVulkan(window);

    while (!glfwWindowShouldClose(window)) {

        VulkanApplication::getInstance()->drawBegin();

        // draw command

        VulkanApplication::getInstance()->drawEnd();

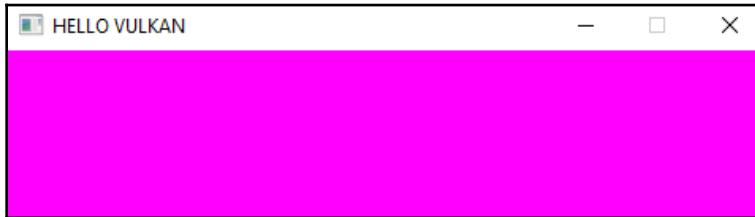
        glfwPollEvents();
    }

    VulkanApplication::getInstance()->cleanup();

    glfwDestroyWindow(window);
    glfwTerminate();
}
```

```
    return 0;  
}
```

You will see a purple viewport, as follows, when you build and run the command:



The screen looks OK, but if you look at the console, you will see the following error which says that when we call `vkAcquireNextImageKHR`, the semaphore and fence cannot both be `NULL`:

```
C:\Users\siddharthMacbookPro\Desktop\myRepos\c-book-repo\Chapters\First Draft\Draft\Chapter 11\VulkanProject\x64\Debug\VulkanProject.exe  
setup call back  
Device Count: 1  
  
DEVICE PROPERTIES  
=====  
  
Device name: Intel(R) Iris(R) Plus Graphics 650  
validation layer: [ VUID-vkAcquireNextImageKHR-semaphore-01780 ] Object: 0x1d4e3e17718 (Type = 3) | vkAcquireNextImageK  
HR: Semaphore and fence cannot both be VK_NULL_HANDLE. There would be no way to determine the completion of this operati  
on. The Vulkan spec states: semaphore and fence must not both be equal to VK_NULL_HANDLE (https://www.khronos.org/regis  
ry/vulkan/specs/1.1-extensions/html/vkspec.html#VUID-vkAcquireNextImageKHR-semaphore-01780)  
  
validation layer: [ VUID-vkAcquireNextImageKHR-semaphore-01780 ] Object: 0x1d4e3e17718 (Type = 3) | vkAcquireNextImageK  
HR: Semaphore and fence cannot both be VK_NULL_HANDLE. There would be no way to determine the completion of this operati  
on. The Vulkan spec states: semaphore and fence must not both be equal to VK_NULL_HANDLE (https://www.khronos.org/regis  
ry/vulkan/specs/1.1-extensions/html/vkspec.html#VUID-vkAcquireNextImageKHR-semaphore-01780)  
  
validation layer: [ VUID-vkAcquireNextImageKHR-semaphore-01780 ] Object: 0x1d4e3e17718 (Type = 3) | vkAcquireNextImageK  
HR: Semaphore and fence cannot both be VK_NULL_HANDLE. There would be no way to determine the completion of this operati  
on. The Vulkan spec states: semaphore and fence must not both be equal to VK_NULL_HANDLE (https://www.khronos.org/regis  
ry/vulkan/specs/1.1-extensions/html/vkspec.html#VUID-vkAcquireNextImageKHR-semaphore-01780)
```

Summary

In this chapter, we saw the creation of the swapchain, renderpass, render views, Framebuffers, and the command buffers. We also looked at what each does and why they are important for rendering a clear screen.

In the next chapter, we will create the resources that will enable us to render geometry to the viewport. Once we have the object resources ready, we will render the objects. We will then explore semaphores and fences and why they are needed.

11

Creating Object Resources

In the previous chapter, we got our clear screen working and created the Vulkan instance. We also created the logical device, the swapchain, the render targets, and the views, as well as the draw command buffer to record and submit commands to the GPU. Using it, we were able to have a purple clear screen. We haven't drawn any geometry yet, but we are now ready to do so.

In this chapter, we will get most of the things that we need ready to render the geometries. We have to create the vertex, index, and uniform buffers. The vertex, index, and uniform buffers will have information regarding the vertex attributes, such as position, color, normal, and texture coordinates; index information will have the indices of the vertices we want to draw; and uniform buffers will have information such as a novel view projection matrix.

We will need to create a descriptor set and layout, which will specify to which shader stage the uniform buffers are bound.

We also have to generate the shaders that will be used to draw the geometry.

For both creating the object buffers and descriptor sets and layouts, we will create new classes so that they are compartmentalized so we can understand how they are related. Before we hop on the object buffers class, we will add the `Mesh` class that we created in the OpenGL project and we will use the same class and add minor changes to it. The `Mesh` class has information regarding the vertex and index information for the different geometry shapes we want to draw.

We will cover the following topics in this chapter:

- Updating the `Mesh` class for Vulkan
- Creating the `ObjectBuffers` class
- Creating the `Descriptor` class
- Creating the SPIR-V shader binary

Updating the Mesh class for Vulkan

In the `Mesh.h` file, we just have to add a few lines of code to specify `InputBindingDescription` and `InputAttributeDescription`. In `InputBindingDescription`, we specify the binding location, the stride of the data itself, and the input rate, which specifies whether the data is per vertex or per instance. In the `Mesh.h` file in the OpenGL project, we will just add functions to the `Vertex` struct:

```
struct Vertex {

    glm::vec3 pos;
    glm::vec3 normal;
    glm::vec3 color;
    glm::vec2 texCoords;

};
```

So, in the `Vertex` struct, add the function to retrieve `AttributeDescription`:

```
static VkVertexInputBindingDescription getBindingDescription() {

    VkVertexInputBindingDescription bindingDescription = {};

    bindingDescription.binding = 0;
    bindingDescription.stride = sizeof(Vertex);
    bindingDescription.inputRate = VK_VERTEX_INPUT_RATE_VERTEX;

    return bindingDescription;
}
```

In the function, `VertexInputBindingDescriptor` specifies that the binding is at the 0th index, the stride is equal to the size of the `Vertex` struct itself, and the input rate is `VK_VERTEX_INPUT_RATE_VERTEX`, which is per vertex. The function just returns the created binding description.

Since we have four attributes in the vertex struct, we have to create an attribute descriptor for each one. Add the following function in the `Vertex` struct as well, which returns an array of four input attribute descriptors. For each attribute descriptor, we have to specify the binding location, which is 0, as specified in the binding description, the layout location for each attribute, the format of the data type, and the offset from the start of the `Vertex` struct:

```
static std::array<VkVertexInputAttributeDescription, 4>
getAttributeDescriptions() {

    std::array<VkVertexInputAttributeDescription, 4> attributeDescriptions =
```

```
    attributeDescriptions[0].binding = 0; // binding index, it is 0 as
    specified above
    attributeDescriptions[0].location = 0; // location layout
    attributeDescriptions[0].format = VK_FORMAT_R32G32B32_SFLOAT; // data
    format
    attributeDescriptions[0].offset = offsetof(Vertex, pos); // bytes since
    the start of the per vertex data

    attributeDescriptions[1].binding = 0;
    attributeDescriptions[1].location = 1;
    attributeDescriptions[1].format = VK_FORMAT_R32G32B32_SFLOAT;
    attributeDescriptions[1].offset = offsetof(Vertex, normal);

    attributeDescriptions[2].binding = 0;
    attributeDescriptions[2].location = 2;
    attributeDescriptions[2].format = VK_FORMAT_R32G32B32_SFLOAT;
    attributeDescriptions[2].offset = offsetof(Vertex, color);

    attributeDescriptions[3].binding = 0;
    attributeDescriptions[3].location = 3;
    attributeDescriptions[3].format = VK_FORMAT_R32G32_SFLOAT;
    attributeDescriptions[3].offset = offsetof(Vertex, texCoords);

    return attributeDescriptions;
}
```

We will also create a new struct in the `Mesh.h` file to organize the uniform data information. So, create a new struct called `UniformBufferObject`:

```
struct UniformBufferObject {

    glm::mat4 model;
    glm::mat4 view;
    glm::mat4 proj;

};
```

At the top of the `Mesh.h` file, we will also include two define statements to tell GLM to use radians instead of degrees, and to use the normalized depth value:

```
#define GLM_FORCE_RADIAN
#define GLM_FORCE_DEPTH_ZERO_TO_ONE
```

That is all for `Mesh.h`. The `Mesh.cpp` file doesn't get modified at all.

Creating the ObjectBuffers class

To create object-related buffers, such as vertex, index, and uniform, we will create a new class called `ObjectBuffers`. In the `ObjectBuffers.h` file, we will add the required include statements:

```
#include <vulkan\vulkan.h>
#include <vector>

#include "Mesh.h"
```

Then we will create the class itself. In the public section, we will add the constructor and the destructor and add the required data types for creating vertex, index, and uniform buffers. We add a vector of the data `vertex` to set the vertex information of the geometry, create a `VkBuffer` instance called `vertexBuffer` to store the vertex buffer, and create a `VkDeviceMemory` instance called `vertexBufferMemory`:

- `VkBuffer`: This is the handle to the object buffer itself.
- `VkDeviceMemory`: Vulkan operates on memory data in the device's memory through the `DeviceMemory` object.

Similarly, we create a vector to store indices, and create an `indexBuffer` and `indexBufferMemory` object, just as we did for `vertex`.

For the uniform buffer, we only create `uniformBuffer` and `uniformBufferMemory` as a vector is not required.

We add a `createVertexIndexUniformBuffers` function, which takes in a `Mesh` type and the vertices and indices will be set based on it.

We also add a destroy function to destroy the Vulkan object we created.

In the private section, we add three functions, which `createVertexIndexUniformBuffers` will call to create the buffers. That is all for the `ObjectBuffers.h` file. So, the `ObjectBuffers` class should be like this:

```
class ObjectBuffers
{
public:
    ObjectBuffers();
    ~ObjectBuffers();

    std::vector<Vertex> vertices;
    VkBuffer vertexBuffer;
```

```
VkDeviceMemory vertexBufferMemory;

std::vector<uint32_t> indices;
VkBuffer indexBuffer;
VkDeviceMemory indexBufferMemory;

VkBuffer uniformBuffers;
VkDeviceMemory uniformBuffersMemory;

void createVertexIndexUniformsBuffers(MeshType modelType);
void destroy();

private:
    void createVertexBuffer();
    void createIndexBuffer();
    void createUniformBuffers();

};
```

Next, let's go on to the `ObjectBuffers.cpp` file. In this file, we include the headers and create the constructor and destructor:

```
#include "ObjectBuffers.h"
#include "Tools.h"
#include "VulkanContext.h"

ObjectBuffers::ObjectBuffers() {}

ObjectBuffers::~ObjectBuffers() {}
```

`Tools.h` is included as we will be adding some more functionality to it that we will use. Next, we will create the `createVertexIndexUniformsBuffers` function:

```
void ObjectBuffers::createVertexIndexUniformsBuffers(MeshType modelType) {
    switch (modelType) {

        case kTriangle: Mesh::setTriData(vertices, indices); break;
        case kQuad:     Mesh::setQuadData(vertices, indices); break;
        case kCube:      Mesh::setCubeData(vertices, indices); break;
        case kSphere:   Mesh::setSphereData(vertices, indices); break;

    }

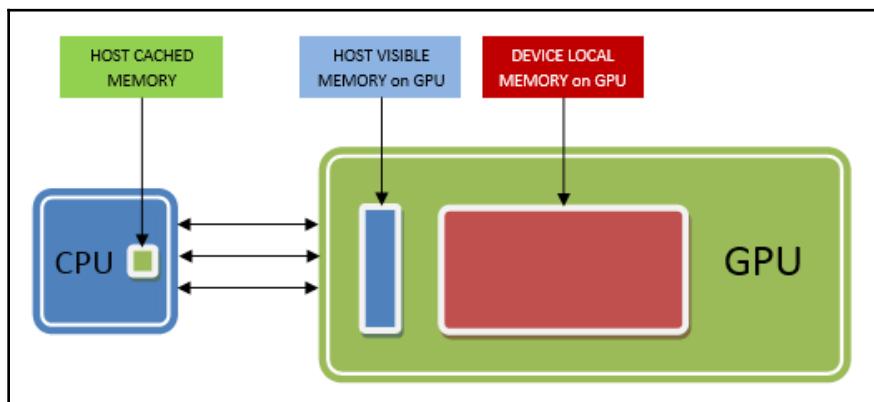
    createVertexBuffer();
    createIndexBuffer();
    createUniformBuffers();
}
```

Similar to the OpenGL project, we will add a switch statement to set the vertex and index data depending upon the mesh type. Then we call the `createVertexBuffer`, `createIndexBuffer`, and `createUniformBuffers` functions to set the respective buffers. We will create the `createVertexBuffer` function first.

To create the vertex buffer, it is better if we create the buffer on the device that is on the GPU itself. Now, the GPU has two types of memories: **HOST VISIBLE** and **DEVICE LOCAL**. **HOST VISIBLE** is a part of the GPU memory that the CPU has access to. This memory is not very large, so it is used for storing <250 MB of data.

For larger chunks of data, such as vertex and index data, it is better to use the **DEVICE LOCAL** memory, which the CPU doesn't have access to.

So, how do you transfer to the **DEVICE LOCAL** memory? Well, we first have to copy the data to the **HOST VISIBLE** section on the GPU, then copy it to the **DEVICE LOCAL** memory. So, we first create what is called a staging buffer, copy the vertex data into it, and then copy the staging buffer to the actual vertex buffer:



(Source: <https://www.youtube.com/watch?v=rXSdDE7NWmA>)

Let's add functionality into the `VkTool` file to create the different kinds of buffers. With this, we can create both the staging buffer and vertex buffer itself. So, in the `VkTools.h` file in the `VkTools` namespace, add a new function called `createBuffer`. This function takes in five parameters:

- The first is `VkDeviceSize`, which is the size of the data for which the buffer is to be created.
- The second is the `usage` flag, which tells us what this buffer is going to be used for.

- The third is the memory properties where we want to create the buffer; this is where we will specify whether we want it in the HOST VISIBLE section or the DEVICE LOCAL area.
- The fourth is the buffer itself.
- The fifth is the buffer memory to bind the buffer to:

```
namespace vkTools {

    VkImageView createImageView(VkImage image,
                               VkFormat format,
                               VkImageAspectFlags aspectFlags);

    void createBuffer(VkDeviceSize size,
                      VkBufferUsageFlags usage,
                      VkMemoryPropertyFlags properties,
                      VkBuffer &buffer,
                      VkDeviceMemory& bufferMemory);
}
```

In the `VKTools.cpp` file, we add the functionality for creating the buffer and binding it to `bufferMemory`. In the namespace, add the new function:

```
void createBuffer(VkDeviceSize size,
                  VkBufferUsageFlags usage,
                  VkMemoryPropertyFlags properties,
                  VkBuffer &buffer, // output
                  VkDeviceMemory& bufferMemory) {

    // code
}
```

Before binding the buffer, we create the buffer itself. Hence, we populate the `VkBufferCreateInfo` struct as follows:

```
VkBufferCreateInfo bufferInfo = {};
bufferInfo.sType = VK_STRUCTURE_TYPE_BUFFER_CREATE_INFO;
bufferInfo.size = size;
bufferInfo.usage = usage;
bufferInfo.sharingMode = VK_SHARING_MODE_EXCLUSIVE;

if
(vkCreateBuffer(VulkanContext::getInstance()->getDevice()->logicalDevice,
&bufferInfo, nullptr, &buffer) != VK_SUCCESS) {

    throw std::runtime_error(" failed to create vertex buffer ");
}
```

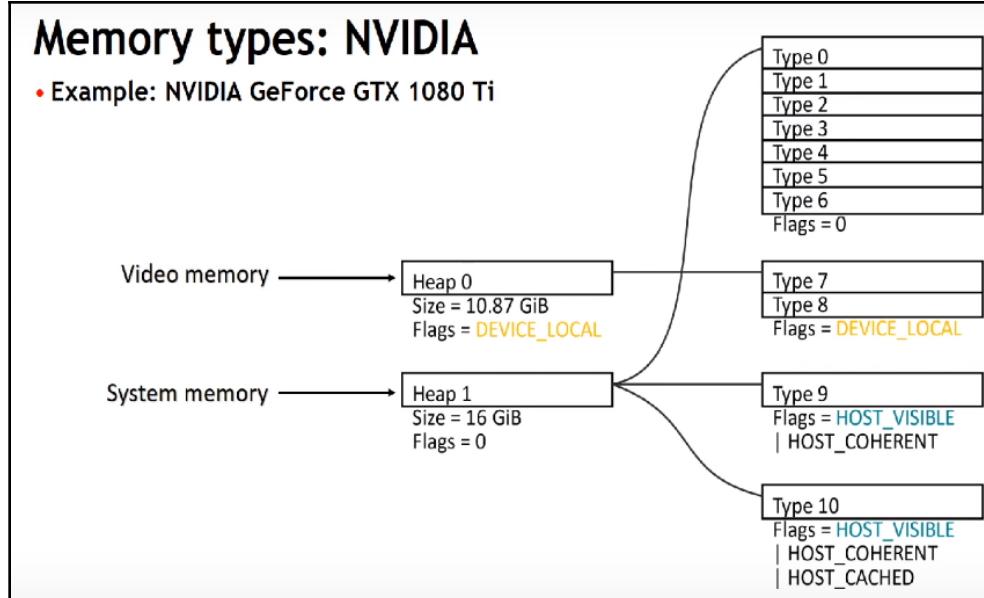
The struct takes the usual type first, then we set the buffer size and usage. We also need to specify the buffer sharing mode, because a buffer can be shared between queues, such as graphics and compute, or could be exclusive to one queue. So, here we specify that the buffer is exclusive to the current queue.

Then the buffer is created by calling `vkCreateBuffer` and passing in `logicalDevice` and `bufferInfo`. Next, to bind the buffer, we have to get the suitable memory type for our specific use of the buffer. So, first we have to get the memory requirements for the kind of buffer we are creating. The required memory requirement is received by calling the `vkGetBufferMemoryRequirements` function, which takes in the logical device, buffer and the memory requirements gets stored in a variable type called `VkMemoryRequirements`.

We get the memory requirements as follows:

```
VkMemoryRequirements memrequirements;
vkGetBufferMemoryRequirements(VulkanContext::getInstance() -> getDevice() -> logicalDevice, buffer, &memrequirements);
```

To bind memory, we have to populate the `VkMemoryAllocateInfo` struct. It requires the allocation size and the memory index of the type of the memory. Each GPU has different memory type index, with a different heap index and memory type. These are the corresponding values for 1080Ti:



We will now add a new function in `VkTools` to get the correct kind of memory index for our buffer usage. So, add a new function in `VkTool.h` under the `vkTools` namespace, called `findMemoryTypeIndex`:

```
uint32_t findMemoryTypeIndex(uint32_t typeFilter, VkMemoryPropertyFlags properties);
```

It takes two parameters, which are the memory type bits available and the memory properties that we need. Add the implementation for the `findMemoryTypeIndex` function in the `VkTools.cpp` file. Under the namespace, add the following function:

```
uint32_t findMemoryTypeIndex(uint32_t typeFilter, VkMemoryPropertyFlags properties) {

    //-- Properties has two arrays -- memory types and memory heaps
    VkPhysicalDeviceMemoryProperties memProperties;
    vkGetPhysicalDeviceMemoryProperties(VulkanContext::getInstance() -> getDevice()
        () -> physicalDevice, &memProperties);

    for (uint32_t i = 0; i < memProperties.memoryTypeCount; i++) {

        if ((typeFilter & (1 << i)) &&
            (memProperties.memoryTypes[i].propertyFlags &
            properties)
            == properties) {

            return i;
        }
    }

    throw std::runtime_error("failed to find suitable memory type!");
}
```

This function gets the device's memory properties using the `vkGetPhysicalDeviceMemoryProperties` function, and populates the memory properties of the physical device.

The memory properties get information regarding the memory heap and memory type for each index. From all the available indices, we choose what is required for our purposes and return the values. Once the function has been created, we can go back to binding the buffer. So, continuing with our `createBuffer` function, add the following to it to bind the buffer to the memory:

```
VkMemoryAllocateInfo allocInfo = {};
allocInfo.sType = VK_STRUCTURE_TYPE_MEMORY_ALLOCATE_INFO;
allocInfo.allocationSize = memrequirements.size;
```

```

    allocInfo.memoryTypeIndex =
findMemoryTypeIndex(memrequirements.memoryTypeBits, properties);

    if
(vkAllocateMemory(VulkanContext::getInstance()->getDevice()->logicalDevice,
&allocInfo, nullptr, &bufferMemory) != VK_SUCCESS) {

        throw std::runtime_error("failed to allocate vertex buffer
memory");
    }

vkBindBufferMemory(VulkanContext::getInstance()->getDevice()->logicalDevice
, buffer, bufferMemory, 0);

```

After all that, we can go back to `ObjectBuffers` to actually create the `createVertexBuffers` function. So, create the function as follows:

```

void ObjectBuffers::createVertexBuffer() {
// code
}

```

In it, we will create the staging buffer first, copy the vertex data into it, and then copy the staging buffer into the vertex buffer. In the function, we first get the total buffer size, which is the number of vertices and the size of the data stored per vertex:

```
VkDeviceSize bufferSize = sizeof(vertices[0]) * vertices.size();
```

Next, we create the staging buffer and `stagingBufferMemory` to bind the staging buffer to it:

```

VkBuffer stagingBuffer;
VkDeviceMemory stagingBufferMemory;

```

And we call the newly created `createBuffer` in `vkTools` to create the buffer:

```

vkTools::createBuffer(bufferSize,
VK_BUFFER_USAGE_TRANSFER_SRC_BIT,
VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT |
VK_MEMORY_PROPERTY_HOST_COHERENT_BIT,
stagingBuffer,
stagingBufferMemory);

```

In it, we pass in the size, usage, memory type we want, and the buffer and buffer memory. `VK_BUFFER_USAGE_TRANSFER_SRC_BIT` indicates that the buffer is going to be used as part of a source transfer command when data is transferred.

`VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT` specifies that we want this to be allocated in the host-visible (CPU) memory space on the GPU.

`VK_MEMORY_PROPERTY_HOST_COHERENT_BIT` means that CPU cache management is not done by us, but by the system. This will make sure the mapped memory matches the allocated memory. Next, we use `vkMapMemory` to get a host pointer to the staging buffer and create a void pointer called `data`. Then call `vkMapMemory` to get the pointer to the mapped memory:

```
void* data;

vkMapMemory(VulkanContext::getInstance() -> getDevice() -> logicalDevice,
            stagingBufferMemory,
            0, // offset
            bufferSize, // size
            0, // flag
            &data);
```

`VkMapMemory` takes the logical device, the staging buffer binding, we specify 0 for the offset and pass the buffer size. There are no special flags, so we pass in 0 and get the pointer to the mapped memory. We use `memcpy` to copy the vertex data to the data pointer:

```
memcpy(data, vertices.data(), (size_t)bufferSize);
```

We unmap the staging memory once host access to it is not required:

```
vkUnmapMemory(VulkanContext::getInstance() -> getDevice() -> logicalDevice,
               stagingBufferMemory);
```

Now that the data is stored in the staging buffer, let's next create the vertex buffer and bind it to `vertexBufferMemory`:

```
// Create Vertex Buffer
vkTools::createBuffer(bufferSize,
                      VK_BUFFER_USAGE_TRANSFER_DST_BIT |
                      VK_BUFFER_USAGE_VERTEX_BUFFER_BIT,
                      VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT,
                      vertexBuffer,
                      vertexBufferMemory);
```

We use the `createBuffer` function to create the vertex buffer. We pass in the buffer size. For the buffer usage, we specify that it is used as the destination of the transfer command when we transfer the staging buffer to it, and it will be used as the vertex buffer. For the memory property, we want this to be created in `DEVICE_LOCAL` for the best performance. Pass the vertex buffer and vertex buffer memory to bind the buffer to the memory. Now we have to copy the staging buffer to the vertex buffer.

Copying buffers on the GPU has to be done using transfer queues and command buffers. We could get the transfer queue to do the transfer in the same way as we retrieved the graphics and presentation queues. The good news is that we don't need to, because all graphics and compute queues also support transfer functionality, so we will use the graphics queue for it.

We will create two helper functions in the `vkTools` namespace for creating and destroying temporary command buffers. So, in the `VkTools.h` file, add two functions in the namespace for the beginning and ending single-time commands:

```
VkCommandBuffer beginSingleTimeCommands(VkCommandPool commandPool);  
void endSingleTimeCommands(VkCommandBuffer commandBuffer, VkCommandPool  
commandPool);
```

`beginSingleTimeCommands` returns a command buffer for us to use, and `endSingleTimeCommands` destroys the command buffer. In the `VkTools.cpp` file, under the namespace, add these two functions:

```
VkCommandBuffer beginSingleTimeCommands(VkCommandPool commandPool) {  
  
    //-- Alloc Command buffer  
    VkCommandBufferAllocateInfo allocInfo = {};  
  
    allocInfo.sType = VK_STRUCTURE_TYPE_COMMAND_BUFFER_ALLOCATE_INFO;  
    allocInfo.level = VK_COMMAND_BUFFER_LEVEL_PRIMARY;  
    allocInfo.commandPool = commandPool;  
    allocInfo.commandBufferCount = 1;  
  
    VkCommandBuffer commandBuffer;  
    vkAllocateCommandBuffers(VulkanContext::getInstance() -> getDevice() -> logical  
    Device, &allocInfo, &commandBuffer);  
  
    //-- Record command buffer  
  
    VkCommandBufferBeginInfo beginInfo = {};  
    beginInfo.sType = VK_STRUCTURE_TYPE_COMMAND_BUFFER_BEGIN_INFO;  
    beginInfo.flags = VK_COMMAND_BUFFER_USAGE_ONE_TIME_SUBMIT_BIT;  
  
    //start recording
```

```
    vkBeginCommandBuffer(commandBuffer, &beginInfo);

    return commandBuffer;

}

void endSingleTimeCommands(VkCommandBuffer commandBuffer, VkCommandPool
commandPool) {

    //-- End recording
    vkEndCommandBuffer(commandBuffer);

    //-- Execute the Command Buffer to complete the transfer
    VkSubmitInfo submitInfo = {};
    submitInfo.sType = VK_STRUCTURE_TYPE_SUBMIT_INFO;
    submitInfo.commandBufferCount = 1;
    submitInfo.pCommandBuffers = &commandBuffer;

    vkQueueSubmit(VulkanContext::getInstance()->getDevice()->graphicsQueue, 1,
    &submitInfo, VK_NULL_HANDLE);
    vkQueueWaitIdle(VulkanContext::getInstance()->getDevice()->graphicsQueue);

    vkFreeCommandBuffers(VulkanContext::getInstance()->getDevice()->logicalDevice,
    commandPool, 1, &commandBuffer);

}
```

We have already looked at how to create and destroy command buffers. If you have any questions, you can refer to Chapter 11, *Preparing the Clear Screen*. Next, in the `Vktools.h` file, we will add the functionality to copy a buffer. Add a new function under the namespace:

```
VkCommandBuffer beginSingleTimeCommands(VkCommandPool commandPool);
void endSingleTimeCommands(VkCommandBuffer commandBuffer, VkCommandPool
commandPool);

void copyBuffer(VkBuffer srcBuffer,
    VkBuffer dstBuffer,
    VkDeviceSize size);
```

The `copyBuffer` function takes a source buffer, a destination buffer, and the buffer size as input. Now add this new function in the `VkTools.cpp` file:

```
void copyBuffer(VkBuffer srcBuffer,
                VkBuffer dstBuffer,
                VkDeviceSize size) {

    QueueFamilyIndices qFamilyIndices =
        VulkanContext::getInstance()->getDevice()->getQueueFamiliesIndicesOfCurrent
        Device();

    // Create Command Pool
    VkCommandPool commandPool;

    VkCommandPoolCreateInfo cpInfo = {};

    cpInfo.sType = VK_STRUCTURE_TYPE_COMMAND_POOL_CREATE_INFO;
    cpInfo.queueFamilyIndex = qFamilyIndices.graphicsFamily;
    cpInfo.flags = 0;

    if
        (vkCreateCommandPool(VulkanContext::getInstance()->getDevice()->logicalDev
        ce, &cpInfo, nullptr,
        &commandPool) != VK_SUCCESS) {
            throw std::runtime_error(" failed to create command pool
        !!");
    }

    // Allocate command buffer and start recording
    VkCommandBuffer commandBuffer = beginSingleTimeCommands(commandPool);

    //-- Copy the buffer
    VkBufferCopy copyregion = {};
    copyregion.srcOffset = 0;
    copyregion.dstOffset = 0;
    copyregion.size = size;
    vkCmdCopyBuffer(commandBuffer, srcBuffer, dstBuffer, 1, &copyregion);

    // End recording and Execute command buffer and free command buffer
    endSingleTimeCommands(commandBuffer, commandPool);

    vkDestroyCommandPool(VulkanContext::getInstance()->getDevice()->logicalDev
    ce, commandPool, nullptr);
}
```

In the function, we first get the queue family indices from the device. We then create a new command pool, and then we create a new command buffer using the `beginSingleTimeCommands` function. To copy the buffer, we create the `VkBufferCopy` struct. We set the source and destination offset to be 0 and set the buffer size.

To actually copy the buffers, we call the `vkCmdCopyBuffer` function, which takes in a command buffer, the source command buffer, the destination command buffer, the copy region count (which is 1 in this case), and the copy region struct. Once the buffers are copied, we call `endSingleTimeCommands` to destroy the command buffer and call `vkDestroyCommandPool` to destroy the command pool itself.

Now we can go back to the `createVertexBuffers` function in `ObjectsBuffers` and copy the staging buffer to the vertex buffer. We also destroy the staging buffer and the buffer memory:

```
vkTools::copyBuffer(stagingBuffer,
                    vertexBuffer,
                    bufferSize);

vkDestroyBuffer(VulkanContext::getInstance()->getDevice()->logicalDevice,
                stagingBuffer, nullptr);
vkFreeMemory(VulkanContext::getInstance()->getDevice()->logicalDevice,
                stagingBufferMemory, nullptr);
```

The index buffers are created the same way, using the `createIndexBuffer` function:

```
void ObjectBuffers::createIndexBuffer() {

    VkDeviceSize bufferSize = sizeof(indices[0]) * indices.size();

    VkBuffer stagingBuffer;
    VkDeviceMemory stagingBufferMemory;

    vkTools::createBuffer(bufferSize,
                          VK_BUFFER_USAGE_TRANSFER_SRC_BIT,
                          VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT |
                          VK_MEMORY_PROPERTY_HOST_COHERENT_BIT,
                          stagingBuffer, stagingBufferMemory);

    void* data;
    vkMapMemory(VulkanContext::getInstance()->getDevice()->logicalDevice,
                stagingBufferMemory, 0, bufferSize, 0, &data);
    memcpy(data, indices.data(), (size_t)bufferSize);
    vkUnmapMemory(VulkanContext::getInstance()->getDevice()->logicalDevice,
                  stagingBufferMemory);
```

```
    vkTools::createBuffer(bufferSize,
        VK_BUFFER_USAGE_TRANSFER_DST_BIT |
        VK_BUFFER_USAGE_INDEX_BUFFER_BIT,
        VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT,
        indexBuffer,
        indexBufferMemory);

    vkTools::copyBuffer(stagingBuffer,
        indexBuffer,
        bufferSize);

vkDestroyBuffer(VulkanContext::getInstance()->getDevice()->logicalDevice,
stagingBuffer, nullptr);
    vkFreeMemory(VulkanContext::getInstance()->getDevice()->logicalDevice,
stagingBufferMemory, nullptr);

}
```

Creating UniformBuffer is easier, because we will just be using the HOST_VISIBLE GPU memory, so staging buffers are not required:

```
void ObjectBuffers::createUniformBuffers() {

    VkDeviceSize bufferSize = sizeof(UniformBufferObject);

    vkTools::createBuffer(bufferSize,
        VK_BUFFER_USAGE_UNIFORM_BUFFER_BIT,
        VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT |
        VK_MEMORY_PROPERTY_HOST_COHERENT_BIT,
        uniformBuffers,
        uniformBuffersMemory);

}
```

Finally, we destroy the buffers and memories in the destroy function:

```
void ObjectBuffers::destroy(){

vkDestroyBuffer(VulkanContext::getInstance()->getDevice()->logicalDevice,
uniformBuffers, nullptr);
    vkFreeMemory(VulkanContext::getInstance()->getDevice()->logicalDevice,
uniformBuffersMemory, nullptr);

vkDestroyBuffer(VulkanContext::getInstance()->getDevice()->logicalDevice,
indexBuffer, nullptr);
    vkFreeMemory(VulkanContext::getInstance()->getDevice()->logicalDevice,
indexBufferMemory, nullptr);
```

```
vkDestroyBuffer(VulkanContext::getInstance() -> getDevice() -> logicalDevice,
vertexBuffer, nullptr);
vkFreeMemory(VulkanContext::getInstance() -> getDevice() -> logicalDevice,
vertexBufferMemory, nullptr);

}
```

Creating the Descriptor class

Unlike OpenGL, where we had uniform buffers to pass in the model, view, projection, and other kinds of data, Vulkan has descriptors. In descriptors, we have to first specify the layout of the buffer, as well as the binding location, count, type of descriptor, and the shader stage it is associated with.

Once the descriptor layout is created with the different types of descriptors, we have to create a descriptor pool for the number-swapchain image count, because the uniform buffer will be set for each time per frame.

After that, we can allocate and populate the descriptor sets for both the frames. The allocation of data will be done from the pool.

We will create a new class for creating the descriptor set, layout binding, pool, and allocating and populating the descriptor sets. Create a new class called `Descriptor`. In the `Descriptor.h` file, add the following code:

```
#pragma once
#include <vulkan\vulkan.h>
#include <vector>

class Descriptor
{
public:
    Descriptor();
    ~Descriptor();

    // all the descriptor bindings are combined into a single layout

    VkDescriptorSetLayout descriptorsetLayout;
    VkDescriptorPool descriptorPool;
    VkDescriptorSet descriptorSet;

    void createDescriptorLayoutSetPoolAndAllocate(uint32_t
_swapChainImageCount);
    void populateDescriptorSets(uint32_t _swapChainImageCount,
```

```
        VkBuffer uniformBuffers);

    void destroy();

private:

    void createDescriptorSetLayout();
    void createDescriptorPoolAndAllocateSets(uint32_t _swapChainImageCount);

};
```

We include the usual `Vulkan.h` and `vector`. In the public section, we create the class with the constructor and the destructor. We also create three variables, called `descriptorsetLayout`, `descriptorPool`, and `descriptorSets`, of the `VkDescriptorSetLayout`, `VkDescriptorPool`, and `VkDescriptorSet` types for easy access to the set. The `createDescriptorLayoutSetPoolAndAllocate` function will call the private `createDescriptorSetLayout` and `createDescriptorPoolAndAllocateSets` functions, which will create the layout set and then create the descriptor pool and allocate to it. The `populateDescriptorSets` function will be called when we set the uniform buffer to populate the sets with the data.

We also have a `destroy` function to destroy the Vulkan objects that have been created. In the `Descriptor.cpp` file, we will add the implementations of the functions. Add the necessary includes first, and then add the constructor, destructor, and the `createDescriptorLayoutAndPool` function:

```
#include "Descriptor.h"

#include<array>
#include "VulkanContext.h"

#include "Mesh.h"

Descriptor::Descriptor(){

}

Descriptor::~Descriptor(){

}

void Descriptor::createDescriptorLayoutSetPoolAndAllocate(uint32_t
_swapChainImageCount) {

    createDescriptorSetLayout();
```

```
    createDescriptorPoolAndAllocateSets(_swapChainImageCount);

}
```

The `createDescriptorLayoutSetPoolAndAllocate` function calls the `createDescriptorsetLayout` and `createDescriptorPoolAndAllocateSets` functions. Let's add the `createDescriptorsetLayout` function:

```
void Descriptor::createDescriptorsetLayout() {

    VkDescriptorSetLayoutBinding uboLayoutBinding = {};
    uboLayoutBinding.binding = 0; // binding location
    uboLayoutBinding.descriptorCount = 1;
    uboLayoutBinding.descriptorType = VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER;
    uboLayoutBinding.stageFlags = VK_SHADER_STAGE_VERTEX_BIT;
    std::array<VkDescriptorSetLayoutBinding, 1> layoutBindings = {
        uboLayoutBinding };

    VkDescriptorSetLayoutCreateInfo layoutCreateInfo = {};
    layoutCreateInfo.sType =
        VK_STRUCTURE_TYPE_DESCRIPTOR_SET_LAYOUT_CREATE_INFO;
    layoutCreateInfo.bindingCount = static_cast<uint32_t>
        (layoutBindings.size());
    layoutCreateInfo.pBindings = layoutBindings.data();

    if
        (vkCreateDescriptorSetLayout(VulkanContext::getInstance() ->getDevice() ->logicalDevice, &layoutCreateInfo, nullptr, &descriptorsetLayout) !=
        VK_SUCCESS) {

            throw std::runtime_error("failed to create descriptor set
layout");
        }
    }
}
```

For our project, the layout set will just have one layout binding, which is the one struct with the model, view, and projection matrix information.

We have to populate the `VkDescriptorSetLayout` struct and specify the binding location index, the count, the type of information we will be passing in, and to which shader stage the uniform buffer will be sent. After creating the set layout, we populate `VkDescriptorSetLayoutCreateInfo`, in which we specify the binding count and the bindings itself.

Then we call the `vkCreateDescriptorSetLayout` function to create the descriptor set layout by passing in the logical device and the layout-creation info. Next, we add the `createDescriptorPoolAndAllocateSets` function:

```
void Descriptor::createDescriptorPoolAndAllocateSets(uint32_t
    _swapChainImageCount) {

    // create pool
    std::array<VkDescriptorPoolSize, 1> poolSizes = {};
    poolSizes[0].type = VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER;
    poolSizes[0].descriptorCount = _swapChainImageCount;

    VkDescriptorPoolCreateInfo poolInfo = {};
    poolInfo.sType = VK_STRUCTURE_TYPE_DESCRIPTOR_POOL_CREATE_INFO;
    poolInfo.poolSizeCount = static_cast<uint32_t>(poolSizes.size());
    poolInfo.pPoolSizes = poolSizes.data();

    poolInfo.maxSets = _swapChainImageCount;

    if
    (vkCreateDescriptorPool(VulkanContext::getInstance() -> getDevice() -> logicalDevice, &poolInfo, nullptr, &descriptorPool) != VK_SUCCESS) {

        throw std::runtime_error("failed to create descriptor pool ");
    }

    // allocate
    std::vector<VkDescriptorSetLayout> layouts(_swapChainImageCount,
    descriptorsetLayout);

    VkDescriptorSetAllocateInfo allocInfo = {};
    allocInfo.sType = VK_STRUCTURE_TYPE_DESCRIPTOR_SET_ALLOCATE_INFO;
    allocInfo.descriptorPool = descriptorPool;
    allocInfo.descriptorSetCount = _swapChainImageCount;
    allocInfo.pSetLayouts = layouts.data();

    if
    (vkAllocateDescriptorSets(VulkanContext::getInstance() -> getDevice() -> logicalDevice, &allocInfo, &descriptorSet) != VK_SUCCESS) {

        throw std::runtime_error("failed to allocate descriptor sets ! ");
    }
}
```

To create the descriptor pool, we have to specify the pool size using `VkDescriptorPoolSize`. We create an array of it and call it `poolSizes`. Since in the layout set we just have the uniform buffer, we set its type and set the count equal to the swap-chain-image count. To create the descriptor pool, we have to specify the type, pool-size count, and the pool-size data. We also have to set the `maxSets`, which is the maximum number of sets that can be allocated from the pool, which is equal to the swap-chain-image count. We create the descriptor pool by calling `vkCreateDescriptorPool` and passing in the logical device and the pool-creation info. Next, we have to specify allocation parameters for the description sets.

We create a vector of the descriptor set layout. Then we create the `VkDescriptorAllocationInfo` struct to populate it. We pass in the description pool, the descriptor set count (which is equal to the swap-chain-images count), and pass in the layout data. Then we allocate the descriptor sets by calling `vkAllocateDescriptorSets` and passing in the logical device and the create info struct.

Finally, we will add the `populateDescriptorSets` function, as follows:

```
void Descriptor::populateDescriptorSets(uint32_t _swapChainImageCount,
                                         VkBuffer uniformBuffers) {

    for (size_t i = 0; i < _swapChainImageCount; i++) {

        // Uniform buffer info

        VkDescriptorBufferInfo uboBufferDescInfo = {};
        uboBufferDescInfo.buffer = uniformBuffers;
        uboBufferDescInfo.offset = 0;
        uboBufferDescInfo.range = sizeof(UniformBufferObject);

        VkWriteDescriptorSet uboDescWrites;
        uboDescWrites.sType = VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET;
        uboDescWrites.pNext = NULL;
        uboDescWrites.dstSet = descriptorSet;
        uboDescWrites.dstBinding = 0; // binding index of 0
        uboDescWrites.dstArrayElement = 0;
        uboDescWrites.descriptorType = VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER;
        uboDescWrites.descriptorCount = 1;
        uboDescWrites.pBufferInfo = &uboBufferDescInfo; // uniforms
        buffers
        uboDescWrites.pImageInfo = nullptr;
        uboDescWrites.pTexelBufferView = nullptr;

        std::array<VkWriteDescriptorSet, 1> descWrites = { uboDescWrites};
```

```
vkUpdateDescriptorSets(VulkanContext::getInstance() -> getDevice() -> logicalDevice,
                      static_cast<uint32_t>(descWrites.size()),
                      descWrites.data(),
                      0,
                      nullptr);
}

}
```

This function takes in the swapchain image count and the uniform buffer as parameters. For both the images of the swapchain, the configuration of the descriptor needs to be updated by calling `vkUpdateDescriptorSets`. This function takes in an array of `VkWriteDescriptorSet`. Now `VkWriteDescriptorSet` takes in either a buffer, image struct, or a `TexelBufferView` as a parameter. Since we are going to use the uniform buffer, we will have to create it and pass it in. `VkDescriptorBufferInfo` takes in a buffer (which will be the uniform buffer we created), takes an offset (which is none in this case), and then takes the range (which is the size of the buffer itself).

After creating it, we can start specifying `VkWriteDescriptorSet`. This takes in the type, `descriptorSet`, and the binding location (which is the 0th index). It has no array elements in it, and it takes the descriptor type (which is the uniform buffer type); the descriptor count is 1, and we pass in the buffer info struct. For the image info and the texel buffer view, we specify none, as it is not being used.

We then create an array of `VkWriteDescriptorSet` and add the uniform buffer descriptor writes info we created, called `uboDescWrites`, to it. We update the descriptor set by calling `vkUpdateDescriptorSets` and pass in the logical device, the descriptor writes size, and the data. That's it for the `populateDescriptorSets` function. We finally add the `destroy` function, which destroys the descriptor pool and the descriptor set layout. Add the function as follows:

```
void Descriptor::destroy() {
    vkDestroyDescriptorPool(VulkanContext::getInstance() -> getDevice() -> logicalDevice, descriptorPool, nullptr);
    vkDestroyDescriptorsetLayout(VulkanContext::getInstance() -> getDevice() -> logicalDevice, descriptorsetLayout, nullptr);
}
```

Creating the SPIR-V shader binary

Unlike OpenGL, which takes in GLSL human-readable files for shaders, Vulkan takes in shaders in binary or byte code format. All shaders, whether vertex, fragment, or compute, have to be in byte code format.

SPIR-V is also good for cross-compilation, making porting shader files a lot easier. If you have a Direct3D HLSL shader code, it can be compiled to SPIR-V format and can be used in a Vulkan application, making it very easy to port Direct3D games to Vulkan. The shader is initially written in GLSL, with some minor changes to how we wrote it for OpenGL. A compiler is provided, which compiles the code from GLSL to SPIR-V format. The compiler is included with the Vulkan SDK installation. The basic vertex-shader GLSL code is as follows:

```
#version 450
#extension GL_ARB_separate_shader_objects : enable

layout (binding = 0) uniform UniformBufferObject{
    mat4 model;
    mat4 view;
    mat4 proj;
} ubo;

layout(location = 0) in vec3 inPosition;
layout(location = 1) in vec3 inNormal;
layout(location = 2) in vec3 inColor;
layout(location = 3) in vec2 inTexCoord;

layout(location = 0) out vec3 fragColor;

void main() {
    gl_Position = ubo.proj * ubo.view * ubo.model * vec4(inPosition, 1.0);
    fragColor = inColor;
}
```

The shader should look very familiar, with some minor changes. For example, the GLSL version is still specified at the top. In this case, it is `#version 450`. But we also see new things, such as `#extension GL_ARB_separate_shader_objects : enable`. This specifies the extension that the shader uses. In this case, an old extension is needed, which basically lets us use vertex and fragment shaders as separate files. Extensions need to be approved by the **Architecture Review Board (ARB)**.

Apart from the inclusion of the extension, you may have noticed that there is a location layout specified for all data types. And you may have noticed that, when creating vertex and uniform buffers, we had to specify the binding index for the buffers. In Vulkan, there is no equivalent to `GLgetUniformLocation` to get the location index of a uniform buffer. This is because it takes quite a bit of system resources to get the location. Instead, we specify and sort of hardcode the value of the index. The uniform buffer, in and out buffer can all be assigned an index of 0 as they are of different data types. Since the uniform buffer will be sent as a struct with model, view, and projection matrices, a similar struct is created in the shader and assigned to the 0th index of the uniform layout.

All four attributes are also assigned layout index 0, 1, 2, and 3, as specified in the `Mesh.h` file under the `Vertex` struct when setting `VkVertexInputAttributeDescription` for the four attributes. The `out` is also assigned a layout location index of 0, and the data type is specified as `vec3`. Then, in the main function of the shader, we set the `gl_Position` value by multiplying the local coordinate of the object by the model, view, and projection matrix received from the uniform buffer struct. `outColor` is set as `inColor` received.

The fragment shader is as follows: open a `.txt` file, add the shader code to it, and name the file `basic`. Also change the extension to `*.vert` from `*.txt`:

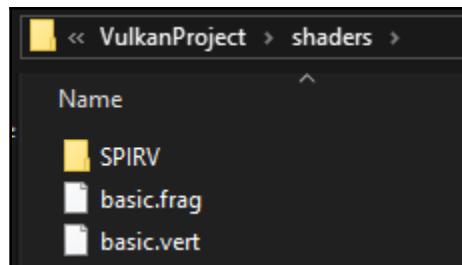
```
#version 450
#extension GL_ARB_separate_shader_objects : enable

layout(location = 0) in vec3 fragColor;

layout(location = 0) out vec4 outColor;

void main() {
    outColor = vec4(fragColor, 1.0f);
}
```

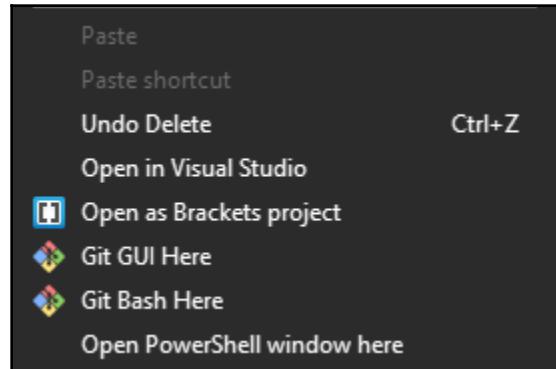
Here, we specify the GLSL version and the extension to use. There are `in` and `out`, both of which have a location layout of 0. `in` is a `vec3` called `fragColor`, which is what we sent out of the vertex shader. `outColor` is a `vec4`. In the main function of the shader file, we convert `vec3` to `vec4` and set the resultant color to `outColor`. Add the fragment shader to a file called `basic.frag`. In the `VulkanProject` root directory, create a new folder called `shaders` and add the two shader files to it:



Create a new folder called `SPIRV`, as this is where we will put the compiled SPIRV bytecode file. To compile `.glsl` files, we will use the `glslValidator.exe` file, which is installed when we installed the Vulkan SDK. Now, to compile the code, we can use the following command:

```
glslangValidator.exe -V basic.frag -o basic.frag.spv
```

Hold *Shift* on the keyboard, right-click in the `shaders` folder, and click the Open PowerShell window here:



In Powershell, type in the following command:

```
\VulkanProject\shaders> glslangValidator.exe -V basic.frag -o basic.frag.spv
```

Make sure the `V` is capitalized and the `o` is lowercase, otherwise it will give compile errors. This will create a new `spirv` file in the folder. Change `frag` to `vert` to compile the SPIRV vertex shader:

```
\VulkanProject\shaders> glslangValidator.exe -V basic.frag -o basic.frag.spv  
\VulkanProject\shaders> glslangValidator.exe -V basic.vert -o basic.vert.spv
```

This will create the vertex and fragment shader SPIRV binaries in the folder:

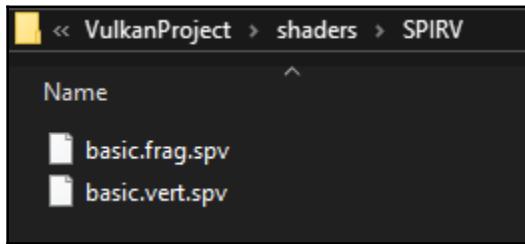
Name	Date modified	Type	Size
SPIRV	3/18/2019 5:12 PM	File folder	
basic.frag	3/18/2019 5:16 PM	FRAG File	1 KB
basic.frag.spv	4/1/2019 2:34 PM	SPV File	1 KB
basic.vert	3/18/2019 5:12 PM	VERT File	1 KB
basic.vert.spv	4/1/2019 2:34 PM	SPV File	3 KB

Instead of compiling the code manually each time, we can create a `.bat` file that can do this for us, and put the compiled SPIRV binaries in the `SPIRV` folder. In the `shaders` folder, create a new `.txt` file and name it `glsl_spirv_compiler.bat`.

In the `.bat` file, add the following:

```
@echo off  
echo compiling glsl shaders to spirv  
for /r %%i in (*.vert;*.frag) do %VULKAN_SDK%\Bin32\glslangValidator.exe -V  
"%%i" -o "%~dp!SPIRV\%~nx!%.spv
```

Save and close the file. Now double-click on the .bat file to execute it. This will compile the shaders and place the compiled binary in the SPIRV shader files:



You can delete the SPIRV files in the `shaders` folder that we compiled earlier using the console command, because we will be using the shader files in the SPIRV subfolder.

Summary

In this chapter, we created all the resources required to render the geometry. First of all, we added the `Mesh` class, which has vertex and index information for all the mesh types, including triangle, quad, cube, and sphere. Then we created the `ObjectBuffers` class, which was used to store and bind the buffers to the GPU memory using the `VkTool` file. We also created a separate descriptor class, which has our descriptor set layout and pool. We created descriptor sets. Finally, we created SPIRV bytecode shader files, which were compiled from the GLSL shader.

In the next chapter, we will use the resources we created here to draw our first colored geometry.

12

Drawing Vulkan Objects

In the previous chapter, we created all the resources required for the object to be drawn. In this chapter, we will make the object renderer class that will draw the object on the viewport so that we can have actual geometry to draw as well, along with viewing our awesome purple viewport.

We will also look at how to synchronize the CPU and the GPU operation at the end, which will also remove the validation error that we got in Chapter 11, *Creating Object Resources*.

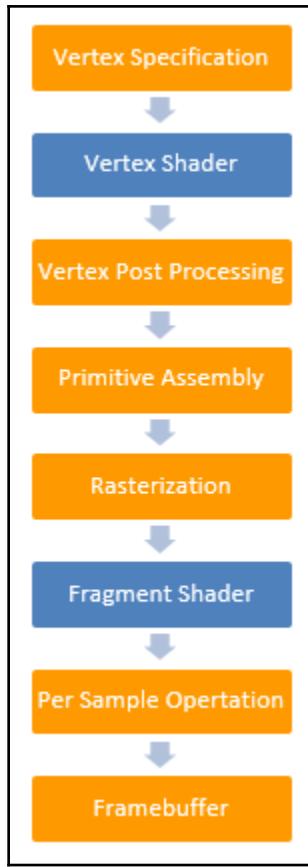
Before we set up the scene for rendering, we will have to prepare one last thing for the geometry render, which is the graphics pipeline. We will start setting this up next.

The following topics are covered in this chapter:

- Preparing the graphics pipeline class
- Object renderer class
- Changes to the Vulkan context class
- Camera class
- Drawing the object
- Synchronization

Preparing the graphics pipeline class

The graphics pipeline defines the pipeline an object should follow when it is drawn. As we saw in Chapter 2, *Mathematics and Graphics Concepts*, there is a series of steps required to draw an object:



In OpenGL, the pipeline states can be changed at any time, just like we enabled and disabled blending when drawing text in Chapter 8, *Enhancing Your Game with Collision, Loop, and Lighting*. But changing states takes a lot of system resources, so Vulkan discourages changing states at will. Hence, you will have to set the pipeline states in advance for each object. Before you create a pipeline's state, you also need to create a pipeline layout that takes the descriptor set layout we created in the previous chapter. So we will create that first.

Then we also need to provide the shader SPIRV files, which will need to be read to create the shader modules. So, add the functionality in the graphics pipeline class. We then populate the graphics pipeline info, which will take the shader stages. We also specify the vertex input state, which will have information regarding the buffer's bindings and attributes, which we created earlier when defining the vertex struct.

The input assembly state also needs to be specified, which describes the kind of geometry to be drawn with the vertices, because we can draw points, lines, or triangles with the set of vertices.

We need to specify the viewport state, which describes the region of the framebuffer that will be rendered to, because we can display part of the framebuffer to the viewport if necessary. In our case, we will be displaying the whole region to the viewport. We specify the rasterization state, which will perform depth testing, and back face culling, and convert geometry to rasterized lines, which will be colored as specified in the fragment shader.

The multisampling state will specify if you want to enable multisampling to enable anti-aliasing. The depth and stencil state specify if the depth and stencil tests are enabled and are to be performed on the object. The color blending state specifies if blending is enabled or not. Finally, the dynamic state enables us to change some pipeline states dynamically without creating the pipeline again. We won't be using dynamic states for our implementation. With all this set, we create the graphics pipeline for the object.

Let's begin by creating a new class for the graphics pipeline. In the `GraphicsPipeline.h` file, add the following:

```
#include <vulkan\vulkan.h>
#include <vector>

#include <fstream>

class GraphicsPipeline
{
public:
    GraphicsPipeline();
    ~GraphicsPipeline();

    VkPipelineLayout pipelineLayout;
    VkPipeline graphicsPipeline;

    void createGraphicsPipelineLayoutAndPipeline(VkExtent2D
        swapChainImageExtent, VkDescriptorSetLayout descriptorsetLayout,
        VkRenderPass renderPass);

    void destroy();
}
```

```

private:

    std::vector<char> readfile(const std::string& filename);
    VkShaderModule createShaderModule(const std::vector<char> & code);

    void createGraphicsPipelineLayout(VkDescriptorSetLayout
descriptorsetLayout);
    void createGraphicsPipeline(VkExtent2D swapChainImageExtent,
VkRenderPass renderPass);

};

```

We include the usual headers, and also include `fstream`, because we will need it for reading the shader files. We then create the class itself. In the public section, we will add the constructor and destructor. We create objects for storing `pipelineLayout` and `graphicsPipeline` of type `VkPipelineLayout` and `VkPipeline`.

We create a new function called `createGraphicsPipelineLayoutAndPipeline`, which takes `VkExtent2D`, `VkDescriptorSetLayout`, and `VkRenderPass`, because this is required for creating both the layout and the pipeline itself. The function will internally be calling `createGraphicsPipelineLayout` and `createGraphicsPipeline`, which will create the layout and the pipeline. These functions are added in the private section.

In the public section, we also have a function called `destroy`, which will destroy all the created resources. In the private section, we also have two more functions. One is the `readFile` function, which reads the SPIR-V file, and the second is `createShaderModule`, which will create the shader module from the read shader file. Let's move on to the `GraphicsPipeline.cpp` file:

```

#include "GraphicsPipeline.h"

#include "VulkanContext.h"
#include "Mesh.h"

GraphicsPipeline::GraphicsPipeline() {}

GraphicsPipeline::~GraphicsPipeline() {}

```

In this, we include the `GraphicsPipeline.h`, `VulkanContext.h`, and `Mesh.h` files, because they are required. We also add the implementation for the constructor and the destructor. We then add the `createGraphicsPipelineLayoutAndPipeline` function, as follows:

```

void GraphicsPipeline::createGraphicsPipelineLayoutAndPipeline(VkExtent2D
swapChainImageExtent, VkDescriptorSetLayout descriptorsetLayout,

```

```
VkRenderPass renderPass) {  
  
    createGraphicsPipelineLayout(descriptorSetLayout);  
    createGraphicsPipeline(swapChainImageExtent, renderPass);  
  
}
```

The `createPipelineLayout` function is created as follows. We have to create a `CreateInfo` struct with the structure type and set the `descriptorLayout` and count, and create the pipeline layout using the `vkCreatePipelineLayout` function:

```
void GraphicsPipeline::createGraphicsPipelineLayout(VkDescriptorSetLayout  
descriptorSetLayout) {  
  
    VkPipelineLayoutCreateInfo pipelineLayoutInfo = {};  
    pipelineLayoutInfo.sType =  
VK_STRUCTURE_TYPE_PIPELINE_LAYOUT_CREATE_INFO;  
  
    // used for passing uniform objects and images to the shader  
    pipelineLayoutInfo.setLayoutCount = 1;  
    pipelineLayoutInfo.pSetLayouts = &descriptorSetLayout;  
  
  
    if  
(vkCreatePipelineLayout(VulkanContext::getInstance() -> getDevice() -> logicalD  
evice, &pipelineLayoutInfo, nullptr, &pipelineLayout) != VK_SUCCESS) {  
  
        throw std::runtime_error(" failed to create pipeline layout !");  
    }  
  
}
```

Before we add the `createPipeline` function, we will add the `readFile` and `createShaderModule` functions:

```
std::vector<char> GraphicsPipeline::readfile(const std::string& filename) {  
  
    std::ifstream file(filename, std::ios::ate | std::ios::binary);  
  
    if (!file.is_open()) {  
        throw std::runtime_error(" failed to open shader file");  
    }  
    size_t filesize = (size_t)file.tellg();  
    std::vector<char> buffer(filesize);  
  
    file.seekg(0);
```

```

        file.read(buffer.data(), filesize);

        file.close();

        return buffer;

    }

```

`readFile` takes a SPIR-V code file, opens and reads it, saves the contents of the file into a vector of `char` called `buffer`, and returns it. We then add the `createShaderModule` function, as follows:

```

VkShaderModule GraphicsPipeline::createShaderModule(const std::vector<char>
& code) {

    VkShaderModuleCreateInfo cInfo = {};

    cInfo.sType = VK_STRUCTURE_TYPE_SHADER_MODULE_CREATE_INFO;
    cInfo.codeSize = code.size();
    cInfo.pCode = reinterpret_cast<const uint32_t*>(code.data());

    VkShaderModule shaderModule;
    if
        (vkCreateShaderModule(VulkanContext::getInstance() ->getDevice() ->logicalDevice,
        &cInfo, nullptr, &shaderModule) != VK_SUCCESS) {
            throw std::runtime_error(" failed to create shader module !");
        }

    return shaderModule;
}

```

To create the shader module, which is required for shader stage create info for creating the pipeline, we need to populate the shader module create info, which takes the code and the size from the buffer to create the shader module. The shader module is created using the `vkCreateShaderModule` function, which takes the device and the create info. Once the shader module is created, it is returned. To create the pipeline, we have to create the following infos: the shader stage info, the vertex input info, the input assembly struct, the viewport info struct, the rasterization info struct, the multisample state struct, the depth stencil struct (if required), the color blending struct, and the dynamic state struct.

So, let's create each, one after the other, starting with the shader stage struct. Add the `createGraphicsPipeline` function, and in it we will create the pipeline:

```

void GraphicsPipeline::createGraphicsPipeline(VkExtent2D
swapChainImageExtent, VkRenderPass renderPass) {

```

```
...
```

```
}
```

In this function, we will now add the following, which will create the graphics pipeline.

Shader stage create info

To create the vertex shader, `ShaderStageCreateInfo`, we need to read the shader code first and create the shader module for it:

```
auto vertexShaderCode = readfile("shaders/SPIRV/basic.vert.spv");  
  
VkShaderModule vertexShaderModule = createShaderModule(vertexShaderCode);
```

To read the shader file, we pass in the location of the shader file location. Then we pass the read code in to the `createShaderModule` function, which will give us `vertexShaderModule`. We create the shader stage info struct for the vertex shader and pass in the stage, the shader module, and the name of the function to be used in the shader, which is `main` in our case:

```
VkPipelineShaderStageCreateInfo vertShaderStageCreateInfo = {};  
vertShaderStageCreateInfo.sType =  
VK_STRUCTURE_TYPE_PIPELINE_SHADER_STAGE_CREATE_INFO;  
vertShaderStageCreateInfo.stage = VK_SHADER_STAGE_VERTEX_BIT;  
vertShaderStageCreateInfo.module = vertexShaderModule;  
vertShaderStageCreateInfo.pName = "main";
```

Similarly, we will create the `shaderstage create info` for the fragment shader:

```
auto fragmentShaderCode = readfile("shaders/SPIRV/basic.frag.spv");  
VkShaderModule fragShaderModule =  
createShaderModule(fragmentShaderCode);  
  
VkPipelineShaderStageCreateInfo fragShaderStageCreateInfo = {};  
  
fragShaderStageCreateInfo.sType =  
VK_STRUCTURE_TYPE_PIPELINE_SHADER_STAGE_CREATE_INFO;  
fragShaderStageCreateInfo.stage = VK_SHADER_STAGE_FRAGMENT_BIT;  
fragShaderStageCreateInfo.module = fragShaderModule;  
fragShaderStageCreateInfo.pName = "main";
```

Note that the shader stage is set to `VK_SHADER_STAGE_FRAGMENT_BIT` to show that this is the fragment shader, and we also pass in `basic.frag.spv` as the file to read, which is the fragment shader file. We then create an array of `shaderStageCreateInfo` and add the two shaders to it for convenience:

```
VkPipelineShaderStageCreateInfo shaderStages[] = {
    vertShaderStageCreateInfo, fragShaderStageCreateInfo };
```

Vertex input state create info

In this info, we specify the input buffer binding and the attribute description:

```
auto bindingDescription = Vertex::getBindingDescription();
auto attributeDescriptions = Vertex::getAttributeDescriptions();

VkPipelineVertexInputStateCreateInfo vertexInputInfo = {};
vertexInputInfo.sType =
VK_STRUCTURE_TYPE_PIPELINE_VERTEX_INPUT_STATE_CREATE_INFO;

vertexInputInfo.vertexBindingDescriptionCount = 1; // initially was 0 as
vertex data was hardcoded in the shader
vertexInputInfo.pVertexBindingDescriptions = &bindingDescription;

vertexInputInfo.vertexAttributeDescriptionCount = static_cast<uint32_t>
(attributeDescriptions.size());
vertexInputInfo.pVertexAttributeDescriptions =
attributeDescriptions.data();
```

This is specified in the `Mesh.h` file under the `vertex` struct.

Input assembly create info

In this, we specify the geometry we want to create, which is a triangle list. Add it as follows:

```
VkPipelineInputAssemblyStateCreateInfo inputAssemblyInfo = {};
inputAssemblyInfo.sType =
VK_STRUCTURE_TYPE_PIPELINE_INPUT_ASSEMBLY_STATE_CREATE_INFO;
inputAssemblyInfo.topology = VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST;
inputAssemblyInfo.primitiveRestartEnable = VK_FALSE;
```

Rasterization state create info

In this struct, we specify that depth clamping is enabled, which, instead of discarding the fragments if they are beyond the near and far plane, still keeps the value and sets the value equal to the near or far plane, even if that pixel is beyond the near or far plane.

Discard the pixel in the rasterization stage by setting the value of `rasterizerDiscardEnable` to true or false. Set the polygon mode, which could be `VK_POLYGON_MODE_FILL` or `VK_POLYGON_MODE_LINE`. If it is line, then only a wireframe is drawn; otherwise, the insides are also rasterized.

We can set the line width with the `lineWidth` parameter. We can enable or disable back face culling and then set the front face winding order by setting the `cullMode` and `frontFace` parameters.

We can alter the depth value by enabling it and adding a constant value to the depth, clamping it, or adding a slope factor. Depth biases are used in shadow maps, which we won't be using, so we won't enable depth bias. Add the struct and populate it as follows:

```
VkPipelineRasterizationStateCreateInfo rastStateCreateInfo = {};
rastStateCreateInfo.sType =
VK_STRUCTURE_TYPE_PIPELINE_RASTERIZATION_STATE_CREATE_INFO;
rastStateCreateInfo.depthClampEnable = VK_FALSE;
rastStateCreateInfo.rasterizerDiscardEnable = VK_FALSE;
rastStateCreateInfo.polygonMode = VK_POLYGON_MODE_FILL;
rastStateCreateInfo.lineWidth = 1.0f;
rastStateCreateInfo.cullMode = VK_CULL_MODE_BACK_BIT;
rastStateCreateInfo.frontFace = VK_FRONT_FACE_CLOCKWISE;
rastStateCreateInfo.depthBiasEnable = VK_FALSE;
rastStateCreateInfo.depthBiasConstantFactor = 0.0f;
rastStateCreateInfo.depthBiasClamp = 0.0f;
rastStateCreateInfo.depthBiasSlopeFactor = 0.0f;
```

MultiSample state create info

For our project, we won't be enabling multisampling for anti-aliasing. But we will still need to create the struct:

```
VkPipelineMultisampleStateCreateInfo msStateInfo = {};
msStateInfo.sType =
VK_STRUCTURE_TYPE_PIPELINE_MULTISAMPLE_STATE_CREATE_INFO;
msStateInfo.sampleShadingEnable = VK_FALSE;
msStateInfo.rasterizationSamples = VK_SAMPLE_COUNT_1_BIT;
```

We disable it by setting `sampleShadingEnable` to false and set the sample count to 1.

Depth and stencil create info

Since we don't have a depth or stencil buffer, we don't need to create it. But when you have a depth buffer, you will need to add it to use the depth texture.

Color blend state create info

We set the color blending to false because it is not required for our project. To populate it, we have to first create the `ColorBlend` attachment state, which contains the configuration per attachment. Then we create `ColorBlendStateInfo`, which contains the overall blend state.

Create the `ColorBlendAttachment` state as follows. In this, we still specify the color write mask, which is the red, green, blue, and alpha bits, and set the attachment state to false, which disables blending for the framebuffer attachment:

```
VkPipelineColorBlendAttachmentState cbAttach = {};
cbAttach.colorWriteMask = VK_COLOR_COMPONENT_R_BIT |
VK_COLOR_COMPONENT_G_BIT | VK_COLOR_COMPONENT_B_BIT |
VK_COLOR_COMPONENT_A_BIT;
cbAttach.blendEnable = VK_FALSE;
```

We create the actual blend struct, which takes the blend attachment info created, and we set the attachment count to 1 because we have a single attachment:

```
cbCreateInfo.sType =
VK_STRUCTURE_TYPE_PIPELINE_COLOR_BLEND_STATE_CREATE_INFO;
cbCreateInfo.attachmentCount = 1;
cbCreateInfo.pAttachments = &cbAttach;
```

Dynamic state info

Since we don't have any dynamic states, this is not created.

Viewport state create info

In the viewport state create info, we can specify the region of the framebuffer in which the output will be rendered to the viewport. So, we can render the scene but then only show some of it to the viewport. We can also specify a scissor rectangle, which will discard the pixels being rendered to the viewport.

But we won't be doing anything fancy like that, because we will render the whole scene to the viewport as it is. To define the viewport size and scissor size, we have to create the respective structs, as follows:

```
VkViewport viewport = {};
viewport.x = 0;
viewport.y = 0;
viewport.width = (float)swapChainImageExtent.width;
viewport.height = (float)swapChainImageExtent.height;
viewport.minDepth = 0.0f;
viewport.maxDepth = 1.0f;

VkRect2D scissor = {};
scissor.offset = { 0,0 };
scissor.extent = swapChainImageExtent;
```

For the viewport size and extent, we set them to the size of the swapchain image size in terms of width and height, starting from $(0, 0)$. We also set the min and max depth, which is normalized between 0 and 1.

For the scissor, since we want to show the whole viewport, we set the offset to $(0, 0)$, which says that we don't want to offset and start from where the viewport starts and set the extent of the size of the swapchain image.

Now we can create the viewport state create info, as follows:

```
VkPipelineViewportStateCreateInfo vpStateInfo = {};
vpStateInfo.sType =
VK_STRUCTURE_TYPE_PIPELINE_VIEWPORT_STATE_CREATE_INFO;
vpStateInfo.viewportCount = 1;
vpStateInfo.pViewports = &viewport;
vpStateInfo.scissorCount = 1;
vpStateInfo.pScissors = &scissor;
```

Graphics pipeline create info

To create the graphics pipeline, we have to create the final info struct, which we will populate with the info structs we have created so far. So, add the struct, as follows:

```
VkGraphicsPipelineCreateInfo gpInfo = {};
gpInfo.sType = VK_STRUCTURE_TYPE_GRAPHICS_PIPELINE_CREATE_INFO;
gpInfo.stageCount = 2;
gpInfo.pStages = shaderStages;
gpInfo.pVertexInputState = &vertexInputInfo;
gpInfo.pInputAssemblyState = &inputAssemblyInfo;
gpInfo.pRasterizationState = &rastStateCreateInfo;
gpInfo.pMultisampleState = &msStateInfo;
gpInfo.pDepthStencilState = nullptr;
gpInfo.pColorBlendState = &cbCreateInfo;
gpInfo.pDynamicState = nullptr;

gpInfo.pViewportState = &vpStateInfo;
```

We also need to pass in the pipeline layout, render pass, and specify if there are any subpasses:

```
gpInfo.layout = pipelineLayout;
gpInfo.renderPass = renderPass;
gpInfo.subpass = 0;
```

Now we can create the pipeline, as follows:

```
if
(vkCreateGraphicsPipelines(VulkanContext::getInstance() -> getDevice() -> logicalDevice,
VK_NULL_HANDLE, 1, &gpInfo, nullptr, &graphicsPipeline) != VK_SUCCESS)
    throw std::runtime_error("failed to create graphics pipeline !!");
```

Also, make sure to destroy the shader modules, because they are not required anymore:

```
vkDestroyShaderModule(VulkanContext::getInstance() -> getDevice() -> logicalDevice,
vertexShadeModule, nullptr);
vkDestroyShaderModule(VulkanContext::getInstance() -> getDevice() -> logicalDevice,
fragShaderModule, nullptr);
```

And that is all for the `createGraphicsPipeline` function. Finally, add the `destroy` function, which will destroy the pipeline and the layout:

```
void GraphicsPipeline::destroy()
{
    vkDestroyPipeline(VulkanContext::getInstance() -> getDevice() -> logicalDevice,
```

```
graphicsPipeline, nullptr);
vkDestroyPipelineLayout(VulkanContext::getInstance() -> getDevice() -> logicalDevice,
    pipelineLayout, nullptr);

}
```

Object renderer class

With all the necessary classes created, we can finally create our `ObjectRenderer` class, which will render the mesh object to the scene.

Let's create a new class called `ObjectRenderer`. In `ObjectRenderer.h`, add the following:

```
#include "GraphicsPipeline.h"
#include "ObjectBuffers.h"
#include "Descriptor.h"

#include "Camera.h"

class ObjectRenderer
{
public:
    void createObjectRenderer(MeshType modelType, glm::vec3 _position,
        glm::vec3 _scale);
    void updateUniformBuffer(Camera camera);

    void draw();
    void destroy();

private:
    GraphicsPipeline gPipeline;
    ObjectBuffers objBuffers;
    Descriptor descriptor;

    glm::vec3 position;
    glm::vec3 scale;

};
```

We will include the descriptor, pipeline, and object buffer headers, because they are required for the class. In the public section of the class, we will add objects of the three classes to define the pipeline, object buffers, and descriptors. We add four functions:

- The first one is the `createObjectRenderer` function, which takes the model type, the position where the object needs to be created, and the scale of the object.
- Then we have `updateUniformBuffer`, which will update the uniform buffer every frame and pass it to the shader. This takes the camera as a parameter, because it is required to get the view and perspective matrix. So include the camera header as well.
- We then have the `draw` function, which will be used to bind the pipeline, vertex, index, and descriptors to make the draw call.
- We also have a `destroy` function to call the `destroy` functions of the pipeline, descriptors, and object buffers.

In the `object Renderer.cpp` file, add the following include and add the `createObjectRenderer` function:

```
#include "ObjectRenderer.h"
#include "VulkanContext.h"
void ObjectRenderer::createObjectRenderer(MeshType modelType, glm::vec3 _position, glm::vec3 _scale) {

    uint32_t swapChainImageCount =
        VulkanContext::getInstance() -> getSwapChain() -> swapChainImages.size();

    VkExtent2D swapChainImageExtent =
        VulkanContext::getInstance() -> getSwapChain() -> swapChainImageExtent;

    // Create Vertex, Index and Uniforms Buffer;
    objBuffers.createVertexIndexUniformsBuffers(modelType);

    // CreateDescriptorsetLayout
    descriptor.createDescriptorLayoutSetPoolAndAllocate(swapChainImageCount);
    descriptor.populateDescriptorSets(swapChainImageCount, objBuffers.uniformBuffers
    );
}

// CreateGraphicsPipeline
gPipeline.createGraphicsPipelineLayoutAndPipeline(
    swapChainImageExtent,
    descriptor.descriptorsetLayout,
    VulkanContext::getInstance() -> getRenderpass() -> renderPass);

position = _position;
```

```
    scale = _scale;
}
```

We get the number of swap buffer images and their extents. Then we create the vertex index and uniform buffers, create and populate the descriptor set layout and sets, and then create the graphics pipeline itself. Finally, we set the position and scale of the current object. Then we add the `updateUniformBuffer` function. To get access to the swapchain and renderpass, we will make some changes to the `VulkanContext` class:

```
void ObjectRenderer::updateUniformBuffer(Camera camera) {

    UniformBufferObject ubo = {};

    glm::mat4 scaleMatrix = glm::mat4(1.0f);
    glm::mat4 rotMatrix = glm::mat4(1.0f);
    glm::mat4 transMatrix = glm::mat4(1.0f);

    scaleMatrix = glm::scale(glm::mat4(1.0f), scale);
    transMatrix = glm::translate(glm::mat4(1.0f), position);

    ubo.model = transMatrix * rotMatrix * scaleMatrix;

    ubo.view = camera.viewMatrix

    ubo.proj = camera.getprojectionMatrix

    ubo.proj[1][1] *= -1; // invert Y, in Opengl it is inverted to begin
    with

    void* data;
    vkMapMemory(VulkanContext::getInstance() -> getDevice() -> logicalDevice,
    objBuffers.uniformBuffersMemory, 0, sizeof(ubo), 0, &data);

    memcpy(data, &ubo, sizeof(ubo));

    vkUnmapMemory(VulkanContext::getInstance() -> getDevice() -> logicalDevice,
    objBuffers.uniformBuffersMemory);

}
```

Here, we create a new `UniformBufferObject` struct called `ubo`. Initialize the translation, rotation, and scale matrix, and assign the values for the scale and rotation matrix. After multiplying the scale, rotation, and translation matrices together, assign the result to the model matrix. From the camera class, we assign the view and projection matrix to `ubo.view` and `ubo.proj`. Then we have to invert the `y` axis in the projection space because, in OpenGL, the `y` axis is already inverted. We now copy the updated `ubo` to the uniform buffer memory.

Next is the `draw` function:

```
void ObjectRenderer::draw() {  
  
    VkCommandBuffer cBuffer =  
        VulkanContext::getInstance()->getCurrentCommandBuffer();  
  
    // Bind the pipeline  
    vkCmdBindPipeline(cBuffer, VK_PIPELINE_BIND_POINT_GRAPHICS,  
        gPipeline.graphicsPipeline);  
  
    // Bind vertex buffer to command buffer  
    VkBuffer vertexBuffers[] = { objBuffers.vertexBuffer };  
    VkDeviceSize offsets[] = { 0 };  
  
    vkCmdBindVertexBuffers(cBuffer,  
        0, // first binding index  
        1, // binding count  
        vertexBuffers,  
        offsets);  
  
    // Bind index buffer to the command buffer  
    vkCmdBindIndexBuffer(cBuffer,  
        objBuffers.indexBuffer,  
        0,  
        VK_INDEX_TYPE_UINT32);  
  
    // Bind uniform buffer using descriptorSets  
    vkCmdBindDescriptorSets(cBuffer,  
        VK_PIPELINE_BIND_POINT_GRAPHICS,  
        gPipeline.pipelineLayout,  
        0,  
        1,  
        &descriptor.descriptorSet, 0, nullptr);  
  
    vkCmdDrawIndexed(cBuffer,  
        static_cast<uint32_t>(objBuffers.indices.size()), // no of indices  
        1, // instance count -- just the 1  
        0, // first index -- start at 0th index
```

```
    0, // vertex offset -- any offsets to add
    0); // first instance -- since no instancing, is set to 0
}
```

Before we actually make the draw call, we have to bind the graphics pipeline and pass in the vertex, index, and descriptor using the command buffer. To do this, we get the current command buffer and pass the commands through it. We will make changes to the `VulkanContext` to get access to it as well.

We make the draw call using `vkCmdDrawIndexed`, in which we pass in the current command buffer, the index size, the instance count, the start of the index (which is 0), the vertex offset (which is again 0), and the location of the first index (which is 0). Then we add the `destroy` function, which just calls the `destroy` function of the pipeline, descriptor, and object buffer:

```
void ObjectRenderer::destroy() {
    gPipeline.destroy();
    descriptor.destroy();
    objBuffers.destroy();
}
```

Changes to the `VulkanContext` class

To get access to the `SwapChain`, `RenderPass`, and current command buffer, we will add the following functions to the `VulkanContext.h` file under the `VulkanContext` class in the public section:

```
void drawBegin();
void drawEnd();
void cleanup();

SwapChain* getSwapChain();
Renderpass* getRenderpass();
VkCommandBuffer getCurrentCommandBuffer();
```

And in the `VulkanContext.cpp` file, add the implementation for accessing the values:

```
SwapChain * VulkanContext::getSwapChain() {
    return swapChain;
}
```

```
Renderpass * VulkanContext::getRenderpass() {
    return renderPass;
}

VkCommandBuffer VulkanContext::getCurrentCommandBuffer() {
    return currentCommandBuffer;
}
```

Camera class

We will create a basic camera class so that we can set the camera's position and set the view and projection matrix. This class will be very similar to the camera class created for OpenGL project. The `camera.h` file is as follows:

```
#pragma once

#define GLM_FORCE_RADIANS
#include <glm\glm.hpp>
#include <glm\gtc\matrix_transform.hpp>

class Camera
{
public:
    void init(float FOV, float width, float height, float nearplane, float farPlane);
    void setCameraPosition(glm::vec3 position);
    glm::mat4 getViewMatrix();
    glm::mat4 getProjectionMatrix();

private:
    glm::mat4 projectionMatrix;
    glm::mat4 viewMatrix;
    glm::vec3 cameraPos;

};
```

It has an `init` function, which takes the `FOV`, width, and height of the viewport, and the near and the far plane to construct the projection matrix. We have a `setCameraPosition` function, which sets the location of the camera and two getter functions to get the camera view and projection matrix. In the private section, we have three local variables; two are for storing the projection and the view matrix, and the third is a `vec3` for storing the camera's position.

The `Camera.cpp` file is as follows:

```
#include "Camera.h"

void Camera::init(float FOV, float width, float height, float nearplane,
float farPlane) {

    cameraPos = glm::vec3(0.0f, 0.0f, 4.0f);
    glm::vec3 cameraFront = glm::vec3(0.0f, 0.0f, 0.0f);
    glm::vec3 cameraUp = glm::vec3(0.0f, 1.0f, 0.0f);

    viewMatrix = glm::mat4(1.0f);
    projectionMatrix = glm::mat4(1.0f);

    projectionMatrix = glm::perspective(FOV, width / height, nearplane,
farPlane);
    viewMatrix = glm::lookAt(cameraPos, cameraFront, cameraUp);

}

glm::mat4 Camera::getViewMatrix(){

    return viewMatrix;

}
glm::mat4 Camera::getprojectionMatrix(){

    return projectionMatrix;
}

void Camera::setCameraPosition(glm::vec3 position){

    cameraPos = position;
}
```

In the `init` function, we set the view and projection matrix, and we add two getter functions and add the `set camera position` function.

Drawing the object

Now that we have completed the prerequisites, let's draw a triangle:

1. In `source.cpp`, include `Camera.h` and `ObjectRenderer.h`:

```
#define GLFW_INCLUDE_VULKAN
#include<GLFW/glfw3.h>

#include "VulkanContext.h"

#include "Camera.h"
#include "ObjectRenderer.h"
```

2. In the main function, after initializing `VulkanContext`, create a new camera and an object to render, as follows:

```
int main() {

    glfwInit();

    glfwWindowHint(GLFW_CLIENT_API, GLFW_NO_API);
    glfwWindowHint(GLFW_RESIZABLE, GLFW_FALSE);

    GLFWwindow* window = glfwCreateWindow(1280, 720, "HELLO VULKAN",
                                         nullptr, nullptr);

    VulkanContext::getInstance()->initVulkan(window);

    Camera camera;
    camera.init(45.0f, 1280.0f, 720.0f, 0.1f, 10000.0f);
    camera.setCameraPosition(glm::vec3(0.0f, 0.0f, 4.0f));

    ObjectRenderer object;
    object.createObjectRenderer(MeshType::kTriangle,
                               glm::vec3(0.0f, 0.0f, 0.0f),
                               glm::vec3(0.5f));
```

3. In the while loop, update the objects buffer and call the `object.draw` function:

```
while (!glfwWindowShouldClose(window)) {

    VulkanContext::getInstance()->drawBegin();

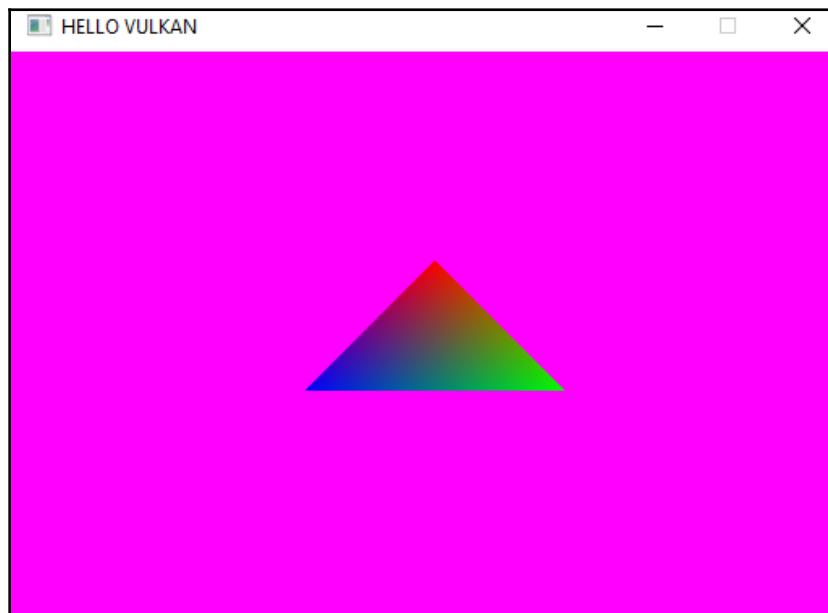
    object.updateUniformBuffer(camera);
    object.draw();
```

```
    VulkanContext::getInstance() -> drawEnd();  
  
    glfwPollEvents();  
}
```

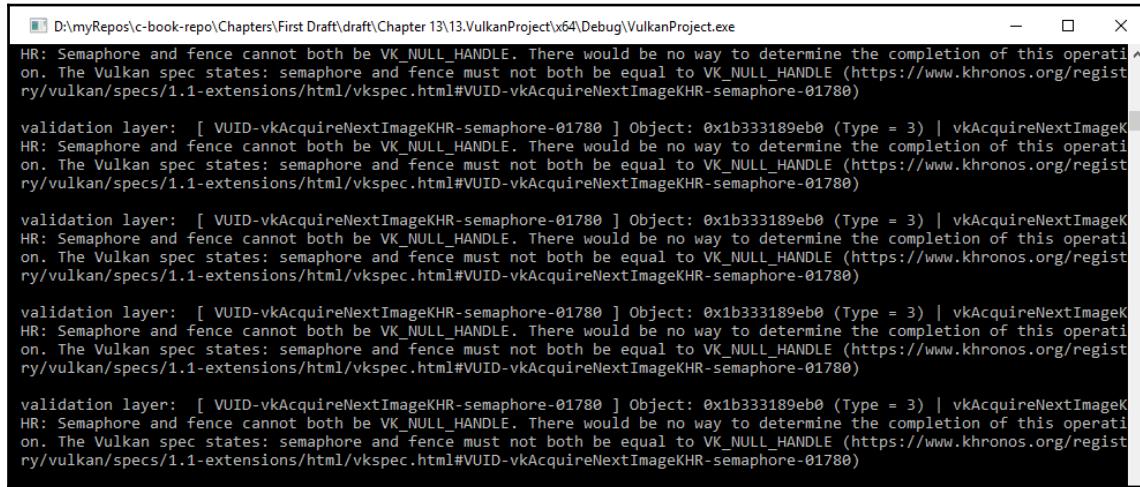
4. When the program is done, call the `object.destroy` function:

```
object.destroy();  
  
VulkanContext::getInstance() -> cleanup();  
  
glfwDestroyWindow(window);  
glfwTerminate();  
  
return 0;  
}
```

5. Run the application, and see the glorious triangle, as follows:



Woohoo! Finally, we have our triangle. Well, we are still not quite done yet. Remember this annoying validation layer error we are getting:



D:\myRepos\c-book-repo\Chapters\First Draft\draft\Chapter 13\13.VulkanProject\x64\Debug\VulkanProject.exe

```
HR: Semaphore and fence cannot both be VK_NULL_HANDLE. There would be no way to determine the completion of this operation. The Vulkan spec states: semaphore and fence must not both be equal to VK_NULL_HANDLE (https://www.khronos.org/registry/vulkan/specs/1.1-extensions/html/vkspec.html#VUID-vkAcquireNextImageKHR-semaphore-01780)
validation layer: [ VUID-vkAcquireNextImageKHR-semaphore-01780 ] Object: 0x1b333189eb0 (Type = 3) | vkAcquireNextImageKHR
HR: Semaphore and fence cannot both be VK_NULL_HANDLE. There would be no way to determine the completion of this operation. The Vulkan spec states: semaphore and fence must not both be equal to VK_NULL_HANDLE (https://www.khronos.org/registry/vulkan/specs/1.1-extensions/html/vkspec.html#VUID-vkAcquireNextImageKHR-semaphore-01780)
validation layer: [ VUID-vkAcquireNextImageKHR-semaphore-01780 ] Object: 0x1b333189eb0 (Type = 3) | vkAcquireNextImageKHR
HR: Semaphore and fence cannot both be VK_NULL_HANDLE. There would be no way to determine the completion of this operation. The Vulkan spec states: semaphore and fence must not both be equal to VK_NULL_HANDLE (https://www.khronos.org/registry/vulkan/specs/1.1-extensions/html/vkspec.html#VUID-vkAcquireNextImageKHR-semaphore-01780)
validation layer: [ VUID-vkAcquireNextImageKHR-semaphore-01780 ] Object: 0x1b333189eb0 (Type = 3) | vkAcquireNextImageKHR
HR: Semaphore and fence cannot both be VK_NULL_HANDLE. There would be no way to determine the completion of this operation. The Vulkan spec states: semaphore and fence must not both be equal to VK_NULL_HANDLE (https://www.khronos.org/registry/vulkan/specs/1.1-extensions/html/vkspec.html#VUID-vkAcquireNextImageKHR-semaphore-01780)
validation layer: [ VUID-vkAcquireNextImageKHR-semaphore-01780 ] Object: 0x1b333189eb0 (Type = 3) | vkAcquireNextImageKHR
HR: Semaphore and fence cannot both be VK_NULL_HANDLE. There would be no way to determine the completion of this operation. The Vulkan spec states: semaphore and fence must not both be equal to VK_NULL_HANDLE (https://www.khronos.org/registry/vulkan/specs/1.1-extensions/html/vkspec.html#VUID-vkAcquireNextImageKHR-semaphore-01780)
```

It is about time to understand why we are getting this error and what it actually means. This leads to our final topic of this book, which is synchronization.

Synchronizing the object

The process of drawing is actually asynchronous, meaning that the GPU might have to wait until the CPU has finished its current job until it can be executed on it. For example, using the constant buffer, we send instructions to update the model view projection matrix in each frame to the GPU using the command buffer. Now, if the GPU doesn't wait for the CPU to get the uniform buffer for the current frame, then the object would not be rendered correctly.

To make sure that the GPU only executes when the CPU has done the work, we need to synchronize the CPU and GPU. This can be done using two synchronization objects:

- This is done using fences. Fences are synchronization objects that synchronize CPU and GPU operations.

- We have a second kind of synchronization object called semaphores. Semaphore objects synchronize GPU queues. In the current scene of one triangle that we are rendering, the graphics queue submits all the graphics commands and then the presentation queue takes the image and then presents it to the viewport. Obviously, even this needs to be synchronized; otherwise, we will see scenes that haven't been fully rendered.

There are also events and barriers, which are other kinds of synchronization objects that are used for synchronizing work within a command buffer, or a sequence of command buffers.

Since we haven't used any synchronization objects, the Vulkan validation layer is throwing errors and telling us that when we acquire an image from the swapchain, we need to either use a fence or a semaphore to synchronize it.

In `VulkanContext.h`, in the private section, we will add the synchronization objects to be created, as follows:

```
const int MAX_FRAMES_IN_FLIGHT = 2;
VkSemaphore imageAvailableSemaphore;
VkSemaphore renderFinishedSemaphore;
std::vector<VkFence> inFlightFences;
```

We have created two semaphores: one semaphore to signal when an image is available for us to render into, and another to signal when the rendering of the image has finished. We also created two fences to synchronize the two frames. In `VulkanContext.cpp`, under the `initVulkan` function, create the synchronization objects after the draw command buffer object:

```
drawComBuffer = new DrawCommandBuffer();
drawComBuffer->createCommandPoolAndBuffer(swapChain->swapChainImages.size());
;

// Synchronization

VkSemaphoreCreateInfo semaphoreInfo = {};
semaphoreInfo.sType = VK_STRUCTURE_TYPE_SEMAPHORE_CREATE_INFO;

vkCreateSemaphore(device->logicalDevice, &semaphoreInfo, nullptr,
&imageAvailableSemaphore);
vkCreateSemaphore(device->logicalDevice, &semaphoreInfo, nullptr,
&renderFinishedSemaphore);

inFlightFences.resize(MAX_FRAMES_IN_FLIGHT);
```

```
VkFenceCreateInfo fenceCreateInfo = {};
fenceCreateInfo.sType = VK_STRUCTURE_TYPE_FENCE_CREATE_INFO;
fenceCreateInfo.flags = VK_FENCE_CREATE_SIGNALED_BIT;

for (size_t i = 0; i < MAX_FRAMES_IN_FLIGHT; i++) {

    if (vkCreateFence(device->logicalDevice, &fenceCreateInfo,
        nullptr, &inFlightFences[i]) != VK_SUCCESS) {

        throw std::runtime_error(" failed to create synchronization
objects per frame !!");
    }
}
```

We have created the semaphore first using the `semaphoreCreateInfo` struct. We just have to set the struct type. Then we can create it using the `vkCreateSemaphore` function and pass in the logical device and the info struct.

Next, we create the fences. We resize the vector with the number of frames in flight, which is 2. Then we create the struct fence info struct and set the type of the struct. We also signal the fences so that they are signaled and ready to be rendered. Then we create the fences using `vkCreateFence` and pass in the logical device and the create fence info using a `for` loop. In the `DrawBegin` function, when we acquire the image, we pass in the `imageAvailable` semaphore to the function so that the semaphore will be signaled when the image is available for us to render into:

```
vkAcquireNextImageKHR(device->logicalDevice,
    swapChain->swapChain,
    std::numeric_limits<uint64_t>::max(),
    imageAvailableSemaphore, // is signaled
    VK_NULL_HANDLE,
    &imageIndex);
```

Once an image is available to render into, we wait for the fence to become signaled so that we can start writing our command buffers:

```
vkWaitForFences(device->logicalDevice, 1, &inFlightFences[imageIndex],
    VK_TRUE, std::numeric_limits<uint64_t>::max());
```

We wait for the fence by calling `vkWaitForFences` and pass in the logical device, the fence count (which is 1), and the fence itself, pass true to wait for all fences, and pass in a timeout, which is a huge number because we don't care about timeout. Once the fence is available, we set it to unsignaled by calling `vkResetFence`, and pass in the logical device, the fence count, and the fence:

```
vkResetFences(device->logicalDevice, 1, &inFlightFences[imageIndex]);
```

The reset of the `DrawBegin` function remains the same so that we can begin command buffer recording. Now, in the `DrawEnd` function, when it is time to submit the command buffer, we set the pipeline stage for `imageAvailableSemaphore` to wait on and set `imageAvailableSemaphore` to wait. We will set `renderFinishedSemaphore` to be signaled.

The submit info is changed accordingly:

```
// submit command buffer
VkSubmitInfo submitInfo = {};
submitInfo.sType = VK_STRUCTURE_TYPE_SUBMIT_INFO;
submitInfo.commandBufferCount = 1;
submitInfo.pCommandBuffers = &currentCommandBuffer;

// Wait for the stage that writes to color attachment
VkPipelineStageFlags waitStages[] = {
    VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT };
// Which stage of the pipeline to wait
submitInfo.pWaitDstStageMask = waitStages;

// Semaphore to wait on before submit command execution begins
submitInfo.waitSemaphoreCount = 1;
submitInfo.pWaitSemaphores = &imageAvailableSemaphore;

// Semaphore to be signaled when command buffers have completed
submitInfo.signalSemaphoreCount = 1;
submitInfo.pSignalSemaphores = &renderFinishedSemaphore;
```

The stage to wait on is `VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT` for `imageAvailableSemaphore` to go from unsignaled to signaled. This will be signaled when the color buffer is written to. We then set `renderFinishedSemaphore` to be signaled so that the image will be ready for presenting. Submit the command and pass in the fence to show that the submission is done:

```
vkQueueSubmit(device->graphicsQueue, 1, &submitInfo,
inFlightFences[imageIndex]);
```

Once submission is done, we can present the image. In the `presentInfo` struct, we set `renderFinishedSemaphore` to wait to go from an unsignaled state to a signaled state. We do so because when the semaphore is signaled, the image will be ready for presentation:

```
// Present frame
VkPresentInfoKHR presentInfo = {};
presentInfo.sType = VK_STRUCTURE_TYPE_PRESENT_INFO_KHR;

presentInfo.waitSemaphoreCount = 1;
presentInfo.pWaitSemaphores = &renderFinishedSemaphore;

presentInfo.swapchainCount = 1;
presentInfo.pSwapchains = &swapChain->swapChain;
presentInfo.pImageIndices = &imageIndex;

vkQueuePresentKHR(device->presentQueue, &presentInfo);
vkQueueWaitIdle(device->presentQueue);
```

In the `cleanup` function in `VulkanContext`, make sure to destroy the semaphores and fences, as follows:

```
void VulkanContext::cleanup() {

    vkDeviceWaitIdle(device->logicalDevice);

    vkDestroySemaphore(device->logicalDevice, renderFinishedSemaphore,
        nullptr);
    vkDestroySemaphore(device->logicalDevice, imageAvailableSemaphore,
        nullptr);

    // Fences and Semaphores
    for (size_t i = 0; i < MAX_FRAMES_IN_FLIGHT; i++) {

        vkDestroyFence(device->logicalDevice, inFlightFences[i], nullptr);
    }

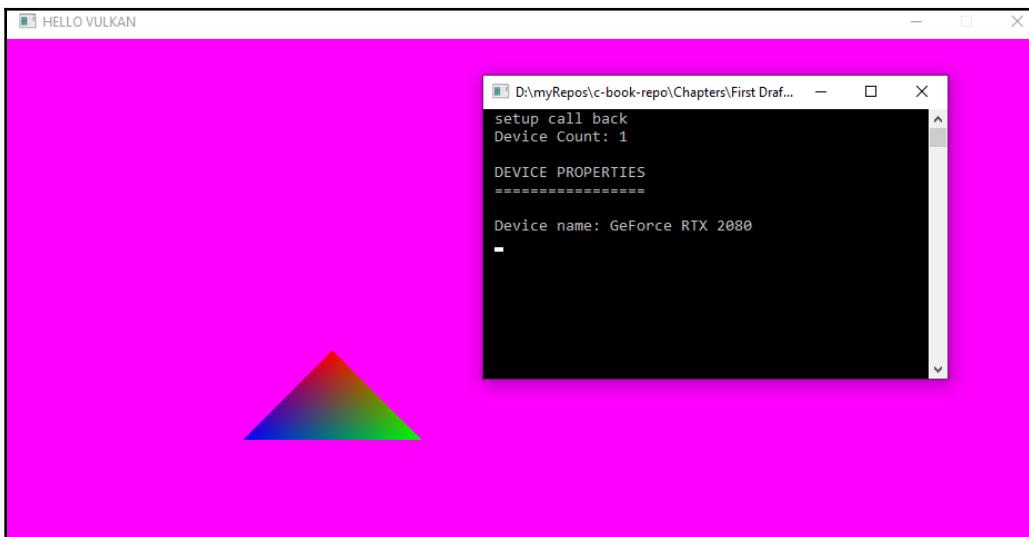
    drawComBuffer->destroy();
    renderTarget->destroy();
    renderPass->destroy();
    swapChain->destroy();

    device->destroy();

    valLayersAndExt->destroy(vInstance->vkInstance,
        isValidationLayersEnabled);
```

```
    vkDestroySurfaceKHR(vInstance->vkInstance, surface, nullptr);  
    vkDestroyInstance(vInstance->vkInstance, nullptr);  
  
}
```

Now build and run the application in debug mode to see that the validation layer has stopped complaining:



Now draw other objects as well, such as the quad, cube, and sphere, by changing the source .cpp, as follows:

```
#define GLFW_INCLUDE_VULKAN  
#include<GLFW/glfw3.h>  
#include "VulkanContext.h"  
#include "Camera.h"  
#include "ObjectRenderer.h"  
int main() {  
  
    glfwInit();  
  
    glfwWindowHint(GLFW_CLIENT_API, GLFW_NO_API);  
    glfwWindowHint(GLFW_RESIZABLE, GLFW_FALSE);  
  
    GLFWwindow* window = glfwCreateWindow(1280, 720, "HELLO VULKAN ", nullptr,  
    nullptr);  
    VulkanContext::getInstance()->initVulkan(window);  
  
    Camera camera;
```

```
camera.init(45.0f, 1280.0f, 720.0f, 0.1f, 10000.0f);
camera.setCameraPosition(glm::vec3(0.0f, 0.0f, 4.0f));
ObjectRenderer tri;
tri.createObjectRenderer(MeshType::kTriangle,
glm::vec3(-1.0f, 1.0f, 0.0f),
glm::vec3(0.5f));

ObjectRenderer quad;
quad.createObjectRenderer(MeshType::kQuad,
glm::vec3(1.0f, 1.0f, 0.0f),
glm::vec3(0.5f));

ObjectRenderer cube;
cube.createObjectRenderer(MeshType::kCube,
glm::vec3(-1.0f, -1.0f, 0.0f),
glm::vec3(0.5f));

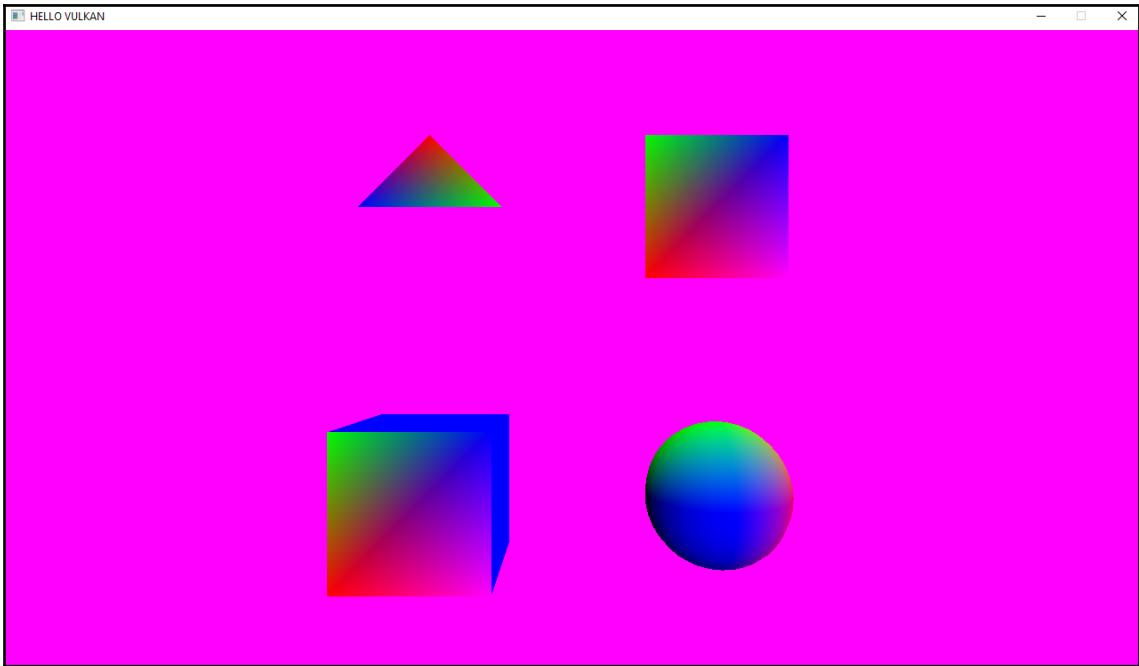
ObjectRenderer sphere;
sphere.createObjectRenderer(MeshType::kSphere,
glm::vec3(1.0f, -1.0f, 0.0f),
glm::vec3(0.5f));

while (!glfwWindowShouldClose(window)) {
    VulkanContext::getInstance() -> drawBegin();

    // updatetri.updateUniformBuffer(camera);
    quad.updateUniformBuffer(camera);
    cube.updateUniformBuffer(camera);
    sphere.updateUniformBuffer(camera);

    // draw command
    tri.draw();
    quad.draw();
    cube.draw();
    sphere.draw();
    VulkanContext::getInstance() -> drawEnd();
    glfwPollEvents();
}
tri.destroy();
quad.destroy();
cube.destroy();
sphere.destroy();
VulkanContext::getInstance() -> cleanup();
glfwDestroyWindow(window);
glfwTerminate();
return 0;
}
```

And this should be the final output, with all the objects rendered:



You can also add custom geometries that can be loaded from a file.

Also, now that you have different shapes to render, you can add physics and try to replicate the physics game made in OpenGL and port the game to use the Vulkan rendering API.

Furthermore, the code can be extended to include the depth buffer, add textures to the object, and more.

References

To learn more, I wholeheartedly recommend the Vulkan tutorial website, <https://vulkan-tutorial.com/>. The tutorial also covers how to add texture, depth buffer, model loading, and mipmaps. The code for the book is based on this tutorial, so it should be easy to follow and help you take the Vulkan code base in the book further:

The screenshot shows the Vulkan Tutorial website. On the left, there's a sidebar with a red header containing the title 'Vulkan Tutorial'. Below the header, the sidebar has a dark grey background with white text, listing navigation links: 'Introduction', 'Overview', 'Development environment', 'Drawing a triangle' (with a right-pointing arrow), and 'Vertex buffers'. The main content area has a light grey header with the title 'Introduction'. Below the header is a list of three items: 'About', 'E-book', and 'Tutorial structure', each with a small red dot next to it. The main content below the header is titled 'About' and contains a paragraph of text explaining the purpose and benefits of the Vulkan API.

Introduction

- [About](#)
- [E-book](#)
- [Tutorial structure](#)

About

This tutorial will teach you the basics of using the [Vulkan](#) graphics and compute API. Vulkan is a new API by the [Khronos group](#) (known for OpenGL) that provides a much better abstraction of modern graphics cards. This new interface allows you to better describe what your application intends to do, which can lead to better performance and less surprising driver behavior compared to existing APIs like [OpenGL](#) and [Direct3D](#). The ideas behind Vulkan are similar to those of [Direct3D 12](#) and [Metal](#), but Vulkan has the advantage of being fully cross-platform and allows you to develop for Windows, Linux and Android at the same time.

I would also recommend reading through the blog at <https://www.fasterthan.life/blog>, as it goes through the journey of porting the Doom 3 OpenGL code to Vulkan. In this book, we let Vulkan allocate and deallocate the resources. This blog goes into the details of how memory management is done in Vulkan. Also, the source code for the Doom 3 Vulkan renderer is available at <https://github.com/DustinHLand/vkDOOM3>, so it is fun to see the code in practice:

Vulkan DOOM 3 port based on DOOM 3 BFG Edition

vulkan doom3 doom 3d-graphics hardware-acceleration

102 commits 7 branches 5 releases 6 contributors GPL-3.0

Branch: master New pull request Find File Clone or download

DustinHLand Fix GLS_CULL_BITS ... Latest commit abe98fd on Jul 3, 2018

base/renderprogs	VK: Fix heavy glass deformation.	2 years ago
doomclassic	Remove OpenGL	2 years ago
neo	Fix GLS_CULL_BITS	10 months ago
.gitignore	Setup .gitignore	2 years ago
LICENSE.txt	Add GPL LICENSE file.	2 years ago
README.md	Update README.md	2 years ago

Summary

So this is the final summary for the book. In this book, we journeyed from creating a basic game in SFML, which uses OpenGL for rendering, to showing how a rendering API fits into the whole scheme when making a game.

We then created a complete physics-based game from the ground up, using our own mini game engine. Apart from just drawing objects using the high-level OpenGL graphics API, we also added bullet physics to take care of game physics and contact detection between game objects. We also added some text rendering to make the score visible to the player, and we also learned about basic lighting to do lighting calculations for our small scene in order to make the scene a little more interesting.

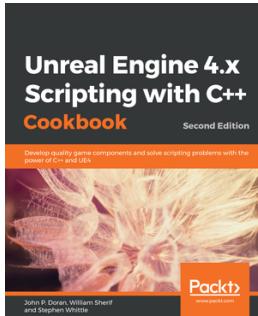
Finally, we moved on to the Vulkan rendering API, which is a low-level graphics library. In comparison to OpenGL, using which we were able to make a small game by the end of the third chapter, in Vulkan, at the end of four chapters, we were able to render a basic geometry. But with Vulkan, we have complete access to the GPU, which gives us more freedom to tailor the engine based on the game's requirements.

If you have come this far, then congratulations! And I hope you enjoyed going through the book and will continue with and expand on the SFML, OpenGL, and Vulkan projects.

Best wishes.

Other Books You May Enjoy

If you enjoyed this book, you may be interested in these other books by Packt:

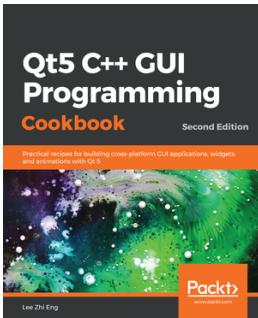


Unreal Engine 4.x Scripting with C++ Cookbook - Second Edition

John P. Doran

ISBN: 9781789809503

- Create C++ classes and structs that integrate well with UE4 and the Blueprints editor
- Discover how to work with various APIs that Unreal Engine already contains
- Utilize advanced concepts such as events, delegates, and interfaces in your UE4 projects
- Build user interfaces using Canvas and UMG through C++
- Extend the Unreal Editor by creating custom windows and editors
- Implement AI tasks and services using C++, Blackboard, and Behavior Trees
- Write C++ code with networking in mind and replicate properties and functions



Qt5 C++ GUI Programming Cookbook - Second Edition

Lee Zhi Eng

ISBN: 9781789803822

- Animate GUI elements using Qt5's built-in animation system
- Draw shapes and 2D images using Qt5's powerful rendering system
- Implement an industry-standard OpenGL library in your project
- Build a mobile app that supports touch events and exports it onto devices
- Parse and extract data from an XML file and present it on your GUI
- Interact with web content by calling JavaScript functions from C++
- Access MySQL and SQLite databases to retrieve data and display it on your GUI

Leave a review - let other readers know what you think

Please share your thoughts on this book with others by leaving a review on the site that you bought it from. If you purchased the book from Amazon, please leave us an honest review on this book's Amazon page. This is vital so that other potential readers can see and use your unbiased opinion to make purchasing decisions, we can understand what our customers think about our products, and our authors can see your feedback on the title that they have worked with Packt to create. It will only take a few minutes of your time, but is valuable to other potential customers, our authors, and Packt. Thank you!

Index

3

3D coordinate system
 about 54
 points 55, 56, 57

A

Ambient 253
Architecture Review Board (ARB) 355
arrays 34, 36, 38
audio
 adding 154, 155, 156

B

Bullet Physics
 adding 213, 214, 215, 216, 217

C

C++
 arrays 34, 36, 38
 classes 46, 47
 concepts 8
 data types 14
 enums 44
 functions 31, 33
 inheritance 48, 51
 pointers 39, 42
 program basics 9, 11, 12, 13
 scope of variables 34
 statements 22
 structs 43
camera class
 about 376
 creating 174, 176
class
 creating 119
classes 46, 47

clear screen

 creating 325, 327, 328, 330

ClearScreen

 creating 168, 171

collision detection 141, 142, 143

collision

 checking 232, 234

CommandBuffer

 creating 319, 321, 323

cross product 64

D

data types, C++

 operators 18, 20, 22

 strings 17

 variables 14, 15, 17

Descriptor class

 creating 348, 349, 350, 352, 353

Diffuse 253

dot product 63

dynamic link libraries (DLLs) 95

E

Element Buffer Object (EBO) 182

enemy class

 creating 128, 129

Enemy header file

 adding 130, 131, 132, 133, 134, 135, 226, 228

 moving 229, 230, 231, 232

enums 44

F

field of view (FOV) 175

First In, First Out (FIFO) 306

Framebuffers 315, 316, 319

function overloading 33

functions 31, 33

G

game objects

building 198

Gameloop

about 144

adding 236, 238

finishing 145, 146, 147

scoring, adding 145, 146, 147

GLM OpenGL mathematics 72, 73, 74

graphics pipeline class

color blend state create info 368

depth and stencil create info 368

dynamic state info 368

graphics pipeline create info 370

input assembly create info 366

MultiSample state create info 367

preparing 360, 362, 364

rasterization state create info 367

shader stage create info 365

vertex input state create info 366

viewport state create info 369

H

hero class

creating 121, 122, 123, 125, 126, 127

I

identity matrix 70

inheritance 50

iteration 25

J

jump statements

break statement 27

continue statement 27

K

keyboard controls

adding 234, 235, 236

keyboard input 113

L

Left Hand Coordinate System 54

Light Renderer class 180, 184, 186, 187, 189

lighting

adding 253, 254, 255, 256, 257, 258, 261

logical device

creating 283, 284, 285, 286, 287, 288, 289, 291, 292, 293, 294, 296, 297, 298, 299, 300

M

matrices

about 66, 67

addition 67

identity matrix 70, 71

inverse 72

multiplication 67, 68, 69, 70

subtraction 67

transpose 71

Mesh class

creating 172, 173, 174

updating, for Vulkan 333, 334

MeshRenderer class

creating 199, 200, 201, 202, 203

O

object renderer class

about 371, 372, 374, 375

object resources

creating 332

object

drawing 378, 380

synchronizing 380, 384, 385, 387

ObjectBuffers class

creating 335, 336, 337, 338, 339, 340, 341,

343, 345, 346, 347

OpenGL project

Camera class, creating 174, 176

ClearScreen, creating 168, 171

creating 165, 166, 168

Light Renderer class 180, 182, 184, 186, 187, 189

Mesh class, creating 172, 173

object, creating 193, 195, 196, 197

object, drawing 189, 191, 193

ShaderLoader class 176, 178, 179
window, creating 168, 171
windows, creating 171

OpenGL
about 164
data types 74

operators
about 18
arithmetic operators 19
comparison operators 22
decrement operators 21
increment operators 20
logical operators 22

P

physical device
selecting 283, 284, 285, 286, 287, 288, 290, 291, 292, 293, 294, 296, 297, 298, 299, 300
pipeline 83
player animations
adding 157, 158, 160, 161
player movement
handling 114, 115, 116
pointers 39, 42

R

render pipeline
about 82, 83
framebuffer 90
vertex shader 86
vertex specification 84
render targets
using 314, 315, 316
Renderpass
beginning 324
creating 310, 311, 312, 313, 314
ending 325
required objects, Bullet Physics
btBroadPhaseInterface 216
btCollisionDispatcher 216
btDefaultCollisionConfiguration 216
btSequentialImpulseConstraintSolver 216
Right Hand Coordinate System 54
rigid bodies
adding 217, 218, 219, 220, 221, 222, 223

RigidBody name
adding 226
rocket class
creating 136, 137
rockets, adding 138, 139

S

scalar product 63
scope of variables 34
scoring
adding 236, 238
ShaderLoader class
implementing 176, 178, 179
shapes
drawing 101, 102, 103, 104, 105, 106
Simple and Fast Media Library (SFML)
about 92, 164
download link 94
downloading 94, 95, 96, 97, 98
modules 93
overview 93
Video Studio, configuring 95, 96, 97, 98
Visual Studio, configuring 94
space transformations

about 75
global space 77
local space 75
object space 75
projection space 79, 80, 81
screen space 82
view space 77, 78, 79
world space 76

Specular 253
SPIR-V shader binary
creating 354, 355, 356, 358

sprite
adding 107, 108, 110, 111
statements
about 22
conditional statements 22, 24
iteration 25, 27
jump statements 27
switch statement 29
strings 17
structs 43

SwapChain
about 301
creating 303, 305, 306, 307, 308, 309, 310
switch statement 29

T

text
adding 148, 149, 150, 151, 152, 153, 154
rendering 238, 239, 240, 241, 242, 243, 244,
246, 248, 250, 251, 252
TextureLoader class
creating 204, 206, 207, 208, 209, 210, 212,
213

V

variables 14, 17
vectors
about 57, 58
cross product 64, 65
dot product 63
unit vector 62, 63
vector magnitude 61, 62
vector operations 58, 59, 60, 61
Vertex Array Object (VAO) 182
Vertex Buffer Object (VBO) 182

Visual Studio 2017 Community version
URL 8
Visual Studio
configuring 265, 266, 267, 268, 269
VkBuffers 335
VkDeviceMemory 335
Vulkan Context class 278, 279, 281
Vulkan extensions 269, 270, 272, 273, 275
Vulkan instances 276, 278
Vulkan objects
drawing 359
Vulkan tutorial website
URL 388
Vulkan validation layers 269, 270, 272, 273, 275

Vulkan
about 264, 265
Mesh class, updating for 333, 334
VulkanContext class
modifying 375

W

window surface
creating 282
window
creating 98, 100, 168, 171