

Beginning C++ Game Programming

Second Edition

Learn to program with C++ by building fun games



John Horton

Packt>

www.packt.com

Beginning C++ Game Programming *Second Edition*

Learn to program with C++ by building fun games

John Horton



BIRMINGHAM - MUMBAI

Beginning C++ Game Programming

Second Edition

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: Pavan Ramchandani

Acquisition Editor: Larissa Pinto

Content Development Editor: Akhil Nair

Senior Editor: Hayden Edwards

Technical Editor: Sachin Sunilkumar

Copy Editor: Safis Editing

Project Coordinator: Manthan Patel

Proofreader: Safis Editing

Indexer: Manju Arasan

Production Designer: Arvindkumar Gupta

First published: September 2016

Second Edition: September 2019

Production reference: 1250919

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK.

ISBN 978-1-83864-857-2

www.packt.com

For Jo, Jack and James.

- John Horton



`packt.com`

Subscribe to our online digital library for full access to over 7,000 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
- Learn better with Skill Plans built especially for you
- Get a free eBook or video every month
- Fully searchable for easy access to vital information
- Copy and paste, print, and bookmark content

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

John Horton is a programming and gaming enthusiast based in the UK. He has a passion for writing apps, games, books, and blog articles. He is the founder of Game Code School.

About the reviewer

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 the book, *Learning LibGDX Game Development*.

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 | xv |
| Chapter 1: C++, SFML, Visual Studio, and Starting the First Game | 1 |
| The games we will build | 1 |
| Timber!!! | 2 |
| Pong | 2 |
| Zombie Arena | 3 |
| Thomas was late | 4 |
| Space Invaders ++ | 4 |
| Meet C++ | 5 |
| Microsoft Visual Studio | 6 |
| SFML | 6 |
| Setting up the development environment | 7 |
| What about Mac and Linux? | 7 |
| Installing Visual Studio 2019 Community edition | 8 |
| Setting up SFML | 10 |
| Creating a new project | 12 |
| Configuring the project properties | 16 |
| Planning Timber!!! | 18 |
| The project assets | 22 |
| Outsourcing the assets | 22 |
| Making your own sound FX | 22 |
| Adding the assets to the project | 23 |
| Exploring the assets | 23 |
| Understanding screen and internal coordinates | 24 |
| Getting started with coding the game | 27 |
| Making code clearer with comments | 27 |
| The main function | 28 |

| | |
|---|-----------|
| Presentation and syntax | 28 |
| Returning values from a function | 29 |
| Running the game | 30 |
| Opening a window using SFML | 30 |
| #including SFML features | 31 |
| OOP, classes, and objects | 32 |
| Using namespace sf | 33 |
| SFML VideoMode and RenderWindow | 34 |
| Running the game | 35 |
| The main game loop | 35 |
| While loops | 37 |
| C-style code comments | 37 |
| Input, update, draw, repeat | 38 |
| Detecting a key press | 38 |
| Clearing and drawing the scene | 39 |
| Running the game | 39 |
| Drawing the game's background | 39 |
| Preparing the Sprite using a Texture | 40 |
| Double buffering the background sprite | 42 |
| Running the game | 43 |
| Handling errors | 43 |
| Configuration errors | 44 |
| Compile errors | 44 |
| Link errors | 44 |
| Bugs | 45 |
| Summary | 45 |
| FAQ | 45 |
| Chapter 2: Variables, Operators, and Decisions – Animating Sprites | 47 |
| C++ variables | 47 |
| Types of variables | 48 |
| User-defined types | 49 |
| Declaring and initializing variables | 49 |
| Declaring variables | 49 |
| Initializing variables | 50 |
| Declaring and initializing in one step | 50 |
| Constants | 50 |
| Declaring and initializing user-defined types | 51 |
| Manipulating variables | 52 |
| C++ arithmetic and assignment operators | 52 |
| Getting things done with expressions | 53 |

| | |
|--|------------|
| Adding clouds, a tree, and a buzzing bee | 56 |
| Preparing the tree | 56 |
| Preparing the bee | 57 |
| Preparing the clouds | 58 |
| Drawing the tree, the bee, and the clouds | 60 |
| Random numbers | 62 |
| Generating random numbers in C++ | 62 |
| Making decisions with if and else | 63 |
| Logical operators | 63 |
| C++ if and else | 65 |
| If they come over the bridge, shoot them! | 65 |
| Shoot them ... or else do this instead | 65 |
| Reader challenge | 67 |
| Timing | 68 |
| The frame rate problem | 68 |
| The SFML frame rate solution | 69 |
| Moving the clouds and the bee | 71 |
| Giving life to the bee | 71 |
| Blowing the clouds | 75 |
| Summary | 79 |
| FAQ | 80 |
| Chapter 3: C++ Strings and SFML Time – Player Input and HUD | 81 |
| Pausing and restarting the game | 81 |
| C++ Strings | 84 |
| Declaring Strings | 84 |
| Assigning a value to a String | 85 |
| Manipulating Strings | 85 |
| SFML's Text and Font classes | 86 |
| Implementing the HUD | 87 |
| Adding a time-bar | 93 |
| Summary | 100 |
| FAQ | 100 |
| Chapter 4: Loops, Arrays, Switches, Enumerations, and Functions – Implementing Game Mechanics | 101 |
| Loops | 102 |
| while loops | 102 |
| Breaking out of a while loop | 104 |
| for loops | 105 |
| Arrays | 106 |
| Declaring an array | 107 |

| | |
|---|------------|
| Initializing the elements of an array | 107 |
| Quickly initializing the elements of an array | 108 |
| What do these arrays really do for our games? | 108 |
| Making decisions with switch | 109 |
| Class enumerations | 111 |
| Getting started with functions | 113 |
| Function return types | 114 |
| Function names | 117 |
| Function parameters | 118 |
| The function body | 118 |
| Function prototypes | 119 |
| Organizing functions | 120 |
| Function gotcha! | 120 |
| More on functions | 121 |
| An absolute final word on functions – for now | 121 |
| Growing the branches | 122 |
| Preparing the branches | 123 |
| Updating the branch sprites each frame | 124 |
| Drawing the branches | 126 |
| Moving the branches | 127 |
| Summary | 129 |
| FAQ | 130 |
| Chapter 5: Collisions, Sound, and End Conditions – | |
| Making the Game Playable | 131 |
| Preparing the player (and other sprites) | 131 |
| Drawing the player and other sprites | 133 |
| Handling the player's input | 135 |
| Handling setting up a new game | 136 |
| Detecting the player chopping | 137 |
| Detecting a key being released | 141 |
| Animating the chopped logs and the axe | 142 |
| Handling death | 145 |
| Simple sound FX | 147 |
| How SFML sound works | 147 |
| When to play the sounds | 148 |
| Adding the sound code | 148 |
| Improving the game and the code | 151 |
| Summary | 153 |
| FAQ | 153 |

| | |
|---|------------|
| Chapter 6: Object-Oriented Programming – Starting the Pong Game | 155 |
| OOP | 155 |
| Encapsulation | 156 |
| Polymorphism | 157 |
| Inheritance | 157 |
| Why use OOP? | 157 |
| What exactly is a class? | 158 |
| The theory of a Pong Bat | 159 |
| The class variable and function declarations | 159 |
| The class function definitions | 161 |
| Using an instance of a class | 163 |
| Creating the Pong project | 164 |
| Coding the Bat class | 166 |
| Coding Bat.h | 166 |
| Constructor functions | 168 |
| Continuing with the Bat.h explanation | 168 |
| Coding Bat.cpp | 169 |
| Using the Bat class and coding the main function | 172 |
| Summary | 177 |
| FAQ | 177 |
| Chapter 7: Dynamic Collision Detection and Physics – Finishing the Pong Game | 179 |
| Coding the Ball class | 179 |
| Using the Ball class | 183 |
| Collision detection and scoring | 185 |
| Running the game | 188 |
| Summary | 188 |
| FAQ | 188 |
| Chapter 8: SFML Views – Starting the Zombie Shooter Game | 191 |
| Planning and starting the Zombie Arena game | 192 |
| Creating a new project | 193 |
| The project assets | 195 |
| Exploring the assets | 196 |
| Adding the assets to the project | 197 |
| OOP and the Zombie Arena project | 197 |
| Building the player – the first class | 198 |
| Coding the Player class header file | 199 |
| Coding the Player class function definitions | 206 |

| | |
|--|------------|
| Controlling the game camera with SFML View | 215 |
| Starting the Zombie Arena game engine | 218 |
| Managing the code files | 222 |
| Starting to code the main game loop | 224 |
| Summary | 233 |
| FAQ | 234 |
| Chapter 9: C++ References, Sprite Sheets, and Vertex Arrays | 235 |
| C++ references | 236 |
| References summary | 239 |
| SFML vertex arrays and sprite sheets | 240 |
| What is a sprite sheet? | 240 |
| What is a vertex array? | 241 |
| Building a background from tiles | 242 |
| Building a vertex array | 242 |
| Using the vertex array to draw | 244 |
| Creating a randomly generated scrolling background | 245 |
| Using the background | 251 |
| Summary | 254 |
| FAQ | 255 |
| Chapter 10: Pointers, the Standard Template Library, and Texture Management | 257 |
| Learning about Pointers | 258 |
| Pointer syntax | 259 |
| Declaring a pointer | 260 |
| Initializing a pointer | 261 |
| Reinitializing pointers | 262 |
| Dereferencing a pointer | 262 |
| Pointers are versatile and powerful | 264 |
| Dynamically allocated memory | 264 |
| Passing a pointer to a function | 266 |
| Declaring and using a pointer to an object | 267 |
| Pointers and arrays | 268 |
| Summary of pointers | 269 |
| The Standard Template Library | 269 |
| What is a map? | 270 |
| Declaring a map | 271 |
| Adding data to a Map | 271 |
| Finding data in a map | 271 |
| Removing data from a map | 272 |
| Checking the size of a map | 272 |
| Checking for keys in a map | 272 |

| | |
|--|------------|
| Looping/iterating through the key-value pairs of a map | 273 |
| The auto keyword | 274 |
| STL summary | 274 |
| The TextureHolder class | 274 |
| Coding the TextureHolder header file | 275 |
| Coding the TextureHolder function definitions | 276 |
| What have we achieved with TextureHolder? | 278 |
| Building a horde of zombies | 278 |
| Coding the Zombie.h file | 279 |
| Coding the Zombie.cpp file | 282 |
| Using the Zombie class to create a horde | 287 |
| Bringing the horde to life (back to life) | 291 |
| Using the TextureHolder class for all textures | 297 |
| Changing the way the background gets its textures | 297 |
| Changing the way the Player gets its texture | 298 |
| Summary | 299 |
| FAQ | 299 |
| Chapter 11: Collision Detection, Pickups, and Bullets | 301 |
| Coding the Bullet class | 302 |
| Coding the Bullet header file | 302 |
| Coding the Bullet source file | 305 |
| Making the bullets fly | 310 |
| Including the Bullet class | 310 |
| Control variables and the bullet array | 311 |
| Reloading the gun | 311 |
| Shooting a bullet | 314 |
| Updating the bullets each frame | 315 |
| Drawing the bullets each frame | 316 |
| Giving the player a crosshair | 317 |
| Coding a class for pickups | 321 |
| Coding the Pickup header file | 321 |
| Coding the Pickup class function definitions | 325 |
| Using the Pickup class | 330 |
| Detecting collisions | 334 |
| Has a zombie been shot? | 335 |
| Has the player been touched by a zombie? | 338 |
| Has the player touched a pickup? | 339 |
| Summary | 340 |
| FAQ | 340 |

| | |
|---|------------|
| Chapter 12: Layering Views and Implementing the HUD | 341 |
| Adding all the Text and HUD objects | 341 |
| Updating the HUD | 345 |
| Drawing the HUD, home, and level-up screens | 348 |
| Summary | 351 |
| FAQ | 352 |
| Chapter 13: Sound Effects, File I/O, and Finishing the Game | 353 |
| Saving and loading the high score | 353 |
| Preparing sound effects | 355 |
| Leveling up | 357 |
| Restarting the game | 360 |
| Playing the rest of the sounds | 360 |
| Adding sound effects while the player is reloading | 361 |
| Making a shooting sound | 361 |
| Playing a sound when the player is hit | 362 |
| Playing a sound when getting a pickup | 363 |
| Making a splat sound when a zombie is shot | 364 |
| Summary | 366 |
| FAQ | 366 |
| Chapter 14: Abstraction and Code Management – Making Better Use of OOP | 367 |
| The Thomas Was Late game | 368 |
| Features of Thomas Was Late | 368 |
| Creating the project | 372 |
| The project's assets | 373 |
| Game level designs | 373 |
| GLSL shaders | 374 |
| The graphical assets close up | 374 |
| The sound assets close up | 374 |
| Adding the assets to the project | 375 |
| Structuring the Thomas Was Late code | 376 |
| Building the game engine | 378 |
| Reusing the TextureHolder class | 378 |
| Coding Engine.h | 381 |
| Coding Engine.cpp | 385 |
| Coding the Engine class constructor definition | 385 |
| Coding the run function definition | 387 |
| Coding the input function definition | 388 |
| Coding the update function definition | 390 |
| Coding the draw function definition | 391 |
| The Engine class so far | 393 |

| | |
|--|------------|
| Coding the main function | 394 |
| Summary | 396 |
| FAQ | 396 |
| Chapter 15: Advanced OOP – Inheritance and Polymorphism | 397 |
| Inheritance | 397 |
| Extending a class | 398 |
| Polymorphism | 400 |
| Abstract classes – virtual and pure virtual functions | 401 |
| Building the PlayableCharacter class | 403 |
| Coding PlayableCharacter.h | 404 |
| Coding PlayableCharacter.cpp | 409 |
| Building the Thomas and Bob classes | 414 |
| Coding Thomas.h | 415 |
| Coding Thomas.cpp | 415 |
| Coding Bob.h | 417 |
| Coding Bob.cpp | 418 |
| Updating the game engine to use Thomas and Bob | 420 |
| Updating Engine.h to add an instance of Bob and Thomas | 420 |
| Updating the input function to control Thomas and Bob | 421 |
| Updating the update function to spawn and update the PlayableCharacter instances | 422 |
| Spawning Thomas and Bob | 423 |
| Updating Thomas and Bob each frame | 424 |
| Drawing Bob and Thomas | 427 |
| Summary | 431 |
| FAQ | 431 |
| Chapter 16: Building Playable Levels and Collision Detection | 433 |
| Designing some levels | 434 |
| Building the LevelManager class | 438 |
| Coding LevelManager.h | 439 |
| Coding the LevelManager.cpp file | 441 |
| Coding the loadLevel function | 448 |
| Updating the engine | 451 |
| Collision detection | 455 |
| Coding the detectCollisions function | 455 |
| More collision detection | 462 |
| Summary | 463 |
| Chapter 17: Sound Spatialization and the HUD | 465 |
| What is spatialization? | 465 |
| Emitters, attenuation, and listeners | 466 |

| | |
|--|------------|
| Handling spatialization using SFML | 466 |
| Building the SoundManager class | 469 |
| Coding SoundManager.h | 469 |
| Coding the SoundManager.cpp file | 471 |
| Coding the constructor | 471 |
| Coding the playFire function | 473 |
| Coding the rest of the SoundManager functions | 474 |
| Adding SoundManager to the game engine | 475 |
| Populating the sound emitters | 476 |
| Coding the populateEmitters function | 477 |
| Playing sounds | 479 |
| Implementing the HUD class | 482 |
| Coding HUD.h | 482 |
| Coding the HUD.cpp file | 483 |
| Using the HUD class | 486 |
| Summary | 490 |
| Chapter 18: Particle Systems and Shaders | 491 |
| Building a particle system | 491 |
| Coding the Particle class | 493 |
| Coding Particle.h | 493 |
| Coding the Particle.cpp file | 494 |
| Coding the ParticleSystem class | 495 |
| Exploring SFML's Drawable class and OOP | 495 |
| An alternative to inheriting from Drawable | 497 |
| Coding ParticleSystem.h | 498 |
| Coding the ParticleSystem.cpp file | 500 |
| Using the ParticleSystem object | 503 |
| Adding a ParticleSystem object to the Engine class | 503 |
| Initializing ParticleSystem | 504 |
| Updating the particle system each frame | 506 |
| Starting the particle system | 506 |
| Drawing the particle system | 508 |
| OpenGL, Shaders, and GLSL | 511 |
| The programmable pipeline and shaders | 512 |
| Coding a fragment shader | 513 |
| Coding a vertex shader | 514 |
| Adding shaders to the engine class | 515 |
| Loading the shaders | 515 |
| Updating and drawing the shader | 516 |
| Summary | 519 |

| | |
|---|------------|
| Chapter 19: Game Programming Design Patterns – | |
| Starting the Space Invaders ++ Game | 521 |
| Space Invaders ++ | 522 |
| Why Space Invaders ++? | 524 |
| Design patterns | 525 |
| Screen, InputHandler, UIPanel, and Button | 526 |
| Entity-Component pattern | 528 |
| Why lots of diverse object types are hard to manage | 528 |
| Using a generic GameObject for better code structure | 529 |
| Prefer composition over inheritance | 530 |
| Factory pattern | 532 |
| C++ smart pointers | 534 |
| Shared pointers | 534 |
| Unique pointers | 536 |
| Casting smart pointers | 537 |
| C++ assertions | 538 |
| Creating the Space Invaders ++ project | 539 |
| Organizing code files with filters | 540 |
| Adding a DevelopState file | 541 |
| Coding SpaceInvaders ++.cpp | 541 |
| Coding the GameEngine class | 542 |
| Coding the SoundEngine class | 545 |
| Coding the ScreenManager class | 547 |
| Coding the BitmapStore class | 550 |
| Coding the ScreenManagerRemoteControl class | 552 |
| Where are we now? | 553 |
| Coding the Screen class and its dependents | 554 |
| Coding the Button class | 554 |
| Coding the UIPanel class | 556 |
| Coding the InputHandler class | 560 |
| Coding the Screen class | 565 |
| Adding the WorldState.h file | 568 |
| Coding the derived classes for the select screen | 569 |
| Coding the SelectScreen class | 569 |
| Coding the SelectInputHandler class | 571 |
| Coding the SelectUIPanel class | 573 |
| Coding the derived classes for the game screen | 576 |
| Coding the GameScreen class | 576 |
| Coding the GameInputHandler class | 580 |
| Coding the GameUIPanel class | 581 |

| | |
|---|------------|
| Coding the GameOverInputHandler class | 583 |
| Coding the GameOverUIPanel class | 584 |
| Running the game | 586 |
| Summary | 588 |
| Chapter 20: Game Objects and Components | 589 |
| Preparing to code the components | 590 |
| Coding the Component base class | 590 |
| Coding the collider components | 591 |
| Coding the ColliderComponent class | 592 |
| Coding the RectColliderComponent class | 593 |
| Coding the graphics components | 596 |
| Coding the GraphicsComponent class | 596 |
| Coding the StandardGraphicsComponent class | 598 |
| Coding the TransformComponent class | 600 |
| Coding update components | 603 |
| Coding the UpdateComponent class | 603 |
| Coding the BulletUpdateComponent class | 604 |
| Coding the InvaderUpdateComponent class | 609 |
| Explaining the update function | 614 |
| Explaining the dropDownAndReverse function | 615 |
| Explaining the isMovingRight function | 616 |
| Explaining the initializeBulletSpawner function | 616 |
| Coding the PlayerUpdateComponent class | 616 |
| Coding the GameObject class | 622 |
| Explaining the GameObject class | 628 |
| Explaining the update function | 628 |
| Explaining the draw function | 629 |
| Explaining the getGraphicsComponent function | 630 |
| Explaining the getTransformComponent function | 630 |
| Explaining the addComponent function | 630 |
| Explaining the getter and setter functions | 632 |
| Explaining the start function | 633 |
| Explaining the GetComponentByTypeAndSpecificType function | 633 |
| Explaining the getEncompassingRectCollider function | 634 |
| Explaining the getEncompassingRectColliderTag function | 635 |
| Explaining the getFirstUpdateComponent function | 635 |
| Explaining the final getter functions | 635 |
| Summary | 636 |
| Chapter 21: File I/O and the Game Object Factory | 637 |
| The structure of the file I/O and factory classes | 638 |
| Describing an object in the world | 640 |
| Coding the GameObjectBlueprint class | 642 |
| Coding the ObjectTags class | 646 |

| | |
|---|------------|
| Coding the BlueprintObjectParser class | 648 |
| Coding the PlayModeObjectLoader class | 652 |
| Coding the GameObjectFactoryPlayMode class | 654 |
| Coding the GameObjectSharer class | 658 |
| Coding the LevelManager class | 659 |
| Updating the ScreenManager and ScreenManagerRemoteControl classes | 663 |
| Where are we now? | 664 |
| Summary | 666 |
| Chapter 22: Using Game Objects and Building a Game | 667 |
| Spawning bullets | 668 |
| Coding the BulletSpawner class | 668 |
| Updating GameScreen.h | 669 |
| Handling the player's input | 671 |
| Using a gamepad | 675 |
| Coding the PhysicsEnginePlayMode class | 678 |
| Making the game | 687 |
| Understanding the flow of execution and debugging | 692 |
| Reusing the code to make a different game and building a design mode | 695 |
| Summary | 698 |
| Chapter 23: Before You Go... | 699 |
| Thanks! | 700 |
| Other Books You May Enjoy | 701 |
| Index | 705 |

Preface

This book is all about offering you a fun introduction to the world of game programming, C++, and the OpenGL-powered SFML using five fun, fully playable games of increasing difficulty and advancing features. These games are an addictive, frantic two-button tapper, a Pong game, a multilevel zombie survival shooter, a split-screen multiplayer puzzle platformer, and a shooter game.

With this improved and extended second edition, we will start with the very basics of programming, such as variables, loops, and conditions, and you will become more skillful with each game as you move through the key C++ topics, such as **object-oriented programming (OOP)**, C++ pointers, and an introduction to the **Standard Template Library (STL)**. While building these games, you will also learn exciting game programming concepts, such as particle effects, directional sound (spatialization), OpenGL programmable Shaders, how to spawn thousands of objects, and more.

Who this book is for

This book is perfect for you if any of the following describes you: You have no C++ programming knowledge whatsoever, or need a beginner level refresher course, if you want to learn to build games or just use games as an engaging way to learn C++, if you have aspirations to publish a game one day, perhaps on Steam, or if you just want to have loads of fun and impress friends with your creations.

What this book covers

Chapter 1, C++, SFML, Visual Studio, and Starting the First Game, represents quite a hefty first chapter, but we will learn absolutely everything we need in order to have the first part of our first game up and running. Here is what we will do: Find out about the games we will build, discover C++, find out about Microsoft Visual C++, explore SFML and its relationship with C++, set up the development environment, plan and prepare for the first game project, Timber!!!, write the first C++ code in the book, and make a runnable game that draws a background.

Chapter 2, Variables, Operators, and Decisions – Animating Sprites, covers quite a bit more drawing on screen and, in order to achieve this, we will need to learn some of the basics of C++. We will learn how to use variables to remember and manipulate values, and we will also begin to add more graphics to the game. As the chapter progresses, we will see how we can manipulate these values to animate the graphics. These values are known as variables.

Chapter 3, C++ Strings and SFML Time – Player Input and HUD, continues with the Timber!!! game. We will spend half the chapter learning how to manipulate text and display it on the screen, and the other half looking at timing and how a visual time bar can inform the player and create a sense of urgency in the game. We will cover the following: Pausing and restarting the game, C++ Strings, SFML Text and SFML Font classes, adding an HUD to Timber!!!, and adding a time bar to Timber!!!.

Chapter 4, Loops, Arrays, Switches, Enumerations, and Functions – Implementing Game Mechanics, probably has more C++ information than any other chapter in the book. It is packed with fundamental concepts that will improve our understanding enormously. It will also begin to shed light on some of the murky areas we have been skipping over a little bit, such as functions and the game loop. Once we have explored a whole list of C++ language necessities, we will then use everything we know to make the main game mechanics – the tree branches – move. By the end of this chapter, we will be ready for the final phase and the completion of Timber!!!. This is what we will explore in this chapter: Loops, arrays, making decisions with switches, enumerations, getting started with functions, and creating and moving the tree branches.

Chapter 5, Collisions, Sound, and End Conditions – Making the Game Playable, constitutes the final phase of the first project. By the end of this chapter, you will have your first completed game. Once you have Timber!!! up and running, be sure to read the final section of this chapter as it will suggest ways to make the game better. In this chapter, we will cover the following topics: Adding the remainder of the sprites, handling the player input, animating the flying log, handling death, adding sound effects, adding features, and improving Timber!!!.

Chapter 6, Object-Oriented Programming – Starting the Pong Game, contains quite a large amount of theory, but the theory will give us the knowledge to start using OOP to powerful effect. Furthermore, we will not waste any time in putting that theory to good use coding the next project, a Pong game. We get to look behind the scenes at how we can create new types that we use as objects by coding a class. We will first look at a simplified Pong scenario to learn some class basics, and then we will start again and code a Pong game for real using the principles we have learned.

Chapter 7, Dynamic Collision Detection and Physics – Finishing the Pong Game, explains how to code our second class. We will see that although the ball is obviously quite different from the bat, we will use the exact same techniques to encapsulate the appearance and functionality of a ball inside a `Ball` class, as we did with the bat and the `Bat` class. We will then add the finishing touches to the Pong game by coding some dynamic collision detection and score keeping. This may sound complicated, but, as we are coming to expect, SFML will make things much easier than they otherwise would be.

Chapter 8, SFML Views – Starting the Zombie Shooter Game, explains how this project makes even more use of OOP, and to a powerful effect. We will also be exploring the SFML `View` class. This versatile class will enable us to easily divide our game up into layers for different aspects of the game. In the Zombie Shooter project, we will have a layer for the HUD and a layer for the main game. This will be necessary because, as the game world expands each time the player clears a wave of zombies, eventually, the game world will be bigger than the screen and will need to scroll. The use of the `View` class will prevent the text from the HUD from scrolling with the background. In the next project, we will take things even further and create a co-op split-screen game with the SFML `View` class doing most of the hard work. This is what we will do in this chapter: Plan the Zombie Arena game, code the `Player` class, learn about the SFML `View` class, and build the Zombie Arena game engine, putting the player class to work.

Chapter 9, C++ References, Sprite Sheets, and Vertex Arrays, explores C++ references, which allow us to work on variables and objects that are otherwise out of scope. In addition, references will help us to avoid having to pass large objects between functions, which is a slow process. It is a slow process because each time we do this, a copy of the variable or object must be made. Armed with this new knowledge about references, we will look at the SFML `VertexArray` class, which allows us to build up a large image that can be very quickly and efficiently drawn to the screen using multiple parts in a single image file. By the end of the chapter, we will have a scalable, random, scrolling background, using references and a `VertexArray` object.

Chapter 10, Pointers, the Standard Template Library, and Texture Management, first covers the fundamental C++ topic of pointers. Pointers are variables that hold a memory address. Typically, a pointer will hold the memory address of another variable. This sounds a bit like a reference, but we will see how they are much more powerful, and we will use a pointer to handle an ever-expanding horde of zombies. We will also learn about the STL, which is a collection of classes that allow us to quickly and easily implement common data management techniques. Once we understand the basics of the STL, we will be able to use that newly acquired knowledge to manage all the textures from the game, because if we have 1,000 zombies, we don't really want to load a copy of a zombie graphic into the GPU for each and every one. We will also dig a little deeper into OOP and use a static function, which is a function of a class that can be called without an instance of the class. At the same time, we will see how we can design a class to ensure that only one instance can ever exist. This is ideal when we need to guarantee that different parts of our code will use the same data.

Chapter 11, Collision Detection, Pickups, and Bullets, explains how we have implemented the main visual aspects of our game so far. We have a controllable character running around in an arena full of zombies that chase him. The problem is that they don't interact with one another. A zombie can wonder right through the player without leaving a scratch. We need to detect collisions between the zombies and the player. If the zombies are going to be able to injure and eventually kill the player, it is only fair that we give the player some bullets for his gun. We will then need to make sure that the bullets can hit and kill the zombies. At the same time, if we are writing collision detection code for bullets, zombies, and the player, it would be a good time to add a class for health and ammo pickups as well.

Chapter 12, Layering Views and Implementing the HUD, is the chapter where we will get to see the real value of SFML Views. We will add a large array of SFML `Text` objects and manipulate them, as we did before in the Timber project and the Pong project. What is new is that we will draw the HUD using a second View instance. This way, the HUD will stay neatly positioned over the top of the main game action, regardless of what the background, player, zombies, and other game objects are doing.

Chapter 13, Sound Effects, File I/O, and Finishing the Game, demonstrates how we can easily manipulate files stored on the hard drive using the C++ standard library, and we will also add sound effects. Of course, we know how to add sound effects, but we will discuss exactly where in the code the calls to the play function will go. We will also tie up a few loose ends to make the game complete. In this chapter, we will do the following: Save and load the hi-score using file input and file output, add sound effects to allow the player to level up, and create never-ending multiple waves.

Chapter 14, Abstraction and Code Management – Making Better Use of OOP, focuses on getting the Thomas Was Alone project started, especially exploring how the code will be structured to make better use of OOP. Here are the details of the topics that will be covered in this chapter: The final project, Thomas Was Late, is introduced, including the gameplay features and project assets, and a detailed discussion is provided of how we will improve the structure of the code compared to previous projects, code the Thomas Was Late game engine, and implement the split-screen functionality.

Chapter 15, Advanced OOP – Inheritance and Polymorphism, extends our knowledge of OOP further by looking at the slightly more advanced concepts of inheritance and polymorphism. We will then be able to use this new knowledge to implement the star characters of our game, Thomas and Bob. Here is what we will cover in this chapter: Learn how to extend and modify a class using inheritance, treat an object of a class as if it is more than one type of class by using polymorphism, learn about abstract classes and how designing classes that are never instantiated can actually be useful, build an abstract `PlayableCharacter` class, put inheritance to work with the Thomas and Bob classes, and add Thomas and Bob to the game project.

Chapter 16, Building Playable Levels and Collision Detection, will probably prove to be one of the most satisfying chapters of this project. The reason for this is that by the end of it, we will have a playable game. Although there will still be features to implement (sound, particle effects, HUD, and shader effects), Bob and Thomas will be able to run, jump, and explore the world. Furthermore, you will be able to create your very own level designs of any size or complexity by simply making platforms and obstacles in a text file. We will achieve all this by covering these topics: Exploring how to design levels in a text file, building a `LevelManager` class that will load levels from a text file, convert them into data that our game can use and keep track of the level details, such as spawn position, current level, and allowed time limit, update the game engine to use `LevelManager`, and code a polymorphic function to handle the collision detection for both Bob and Thomas.

Chapter 17, Sound Spatialization and the HUD, adds all the sound effects and the HUD. We have done this in both previous projects, but we will do things a bit differently this time. We will explore the concept of sound spatialization and how SFML makes this otherwise complicated concept nice and easy. In addition, we will build an HUD class to encapsulate our code that draws information to the screen. We will complete the tasks in the following order: What spatialization is, how SFML handles spatialization, building a `SoundManager` class, deploying emitters, using the `SoundManager` class, and building and then using an HUD class.

Chapter 18, Particle Systems and Shaders, examines what a particle system is and then proceeds to code one into our game. We will scratch the surface of the topic of OpenGL shaders and see how writing code in another language **OpenGL Shading Language (GLSL)**, which can be run directly on the graphics card, can lead to smooth graphical effects that might otherwise be impossible. As usual, we will also use our new skills and knowledge to enhance the current project.

Chapter 19, Game Programming Design Patterns – Starting the Space Invaders ++ Game, introduces the final project. As you have come to expect by now, this project will take a significant step forward in learning new C++ techniques. The next four chapters will look at topics including smart pointers, C++ assertions, using a gamepad controller, debugging using Visual Studio, casting pointers of a base class to become pointers of a specific derived class, debugging, and taking a first look at design patterns. The author is surmising that if you are going to make deep, large-scale games in C++, then design patterns are going to be a big part of your learning agenda in the months and years ahead. In order to introduce this vital topic, I have chosen a relatively simple, but fun, game to serve as an example. Let's find out a bit more about the Space Invaders ++ game, and then we can move on to the topic of design patterns and why we need them. In this hefty chapter, we will cover the following topics: Finding out about Space Invaders ++ and why it has been chosen for the final project, learning what design patterns are and why they matter to game developers, studying the design patterns in the Space Invaders ++ project that will be used over the next four chapters, getting started on the Space Invaders ++ project, and coding numerous classes to begin to flesh out the game.

Chapter 20, Game Objects and Components, covers all the coding related to the Entity-Component pattern we discussed at the beginning of the previous chapter. This means we will code the base component class that all the other components will be derived from. We will also put to good use our new knowledge of smart pointers so that we don't have to concern ourselves with keeping track of the memory we allocate for these components. We will also code the `GameObject` class in this chapter. Here is a list of the sections in this chapter: Preparing to code the components, coding the component base class, coding the collider components, coding the graphics components, coding the update components, and coding the `GameObject` class.

Chapter 21, File I/O and the Game Object Factory, explains how a `GameObject` gets into the `m_GameObjects` vector used in the game. We will see how we can describe individual objects and an entire level in a text file. We will write code to interpret the text and then load up values into a class that will be a blueprint for a game object. We will code a class called `LevelManager` that oversees the whole process, starting from the initial request to load a level sent from an `InputHandler` via the `ScreenManager`, right through to the factory pattern class that assembles a game object from components and delivers it to the `LevelManager` class neatly packed away in the `m_GameObjects` vector.

Chapter 22, Using Game Objects and Building a Game, constitutes the final stage of the Space Invaders ++ project. We will learn how to receive input from a gamepad using SFML to do all the hard work, and we will also code a class that will handle communication between the invaders and the `GameScreen` class, as well as the player and the `GameScreen` class. The class will allow the player and the invaders to spawn bullets, but the exact same technique could be used for any kind of communication that you need between different parts of your own game, so it is useful to know. The final part of the game (as usual) will be collision detection and the logic of the game itself. Once Space Invaders ++ is up and running, we will learn how to use the Visual Studio debugger, which will be invaluable when you are designing your own logic because it allows you to step through your code a line at a time and see the value of variables. It is also a useful tool for studying the flow of execution of the patterns we have assembled over the course of this project.

Chapter 23, Before You Go..., brings our journey to an end. When you first opened this big doorstop of a book, the back page probably seemed like a long way off. But it wasn't too tough, I hope? The point is that you are here now and hopefully, you have a good insight into how to build games using C++. It might surprise you to hear that even after all these hundreds of pages, we have only dipped our toes into C++. Even the topics we did cover could be covered in more depth and there are numerous, some quite significant, topics that we haven't even mentioned. With this in mind, let's take a look at what might be next.

To get the most out of this book

The following requirements need to be satisfied:

- Windows 7 Service Pack 1, Windows 8, or Windows 10
- 1.6 GHz or faster processor
- 1 GB of RAM (for x86) or 2 GB of RAM (for x64)
- 15 GB of available hard disk space

- 5400 RPM hard disk drive
- DirectX 9-capable video card that runs at 1024 x 768 or higher display resolution

All the software used in this book is free. Obtaining and installing the software is covered step by step within the book. The book uses Visual Studio for Windows throughout, but experienced Linux and Mac users will probably have no trouble running the code and following the instructions using their favorite programming environment.

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.packtpub.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 <http://www.packt.com>.
2. Select the **Support** tab.
3. Click on **Code Downloads**.
4. Enter the name of the book in the **Search** box and follow the on screen 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/Beginning-Cpp-Game-Programming-Second-Edition>. 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://static.packt-cdn.com/downloads/9781838648572_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. For example: "My main project directory is D:\VS Projects\Timber."

A block of code is set as follows:

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

Bold: Indicates a new term, an important word, or words that you see on screen. For example, words in menus or dialog boxes appear in the text like this. Here is an example: "Click on the **Create a new project** button."



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.packtpub.com/support/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

C++, SFML, Visual Studio, and Starting the First Game

Welcome to *Beginning C++ Game Programming*. I will not waste any time in getting you started on your journey to writing great games for the PC using C++ and the OpenGL powered SFML.

This is quite a hefty first chapter, but we will learn absolutely everything we need so that we have the first part of our first game up and running. Here is what we will do in this chapter:

- Find out about the games we will build
- Meet C++
- Find out about Microsoft Visual C++
- Explore SFML and its relationship with C++
- Setting up the development environment
- Plan and prepare for the first game project, Timber!!!
- Write the first C++ code of this book and make a runnable game that draws a background

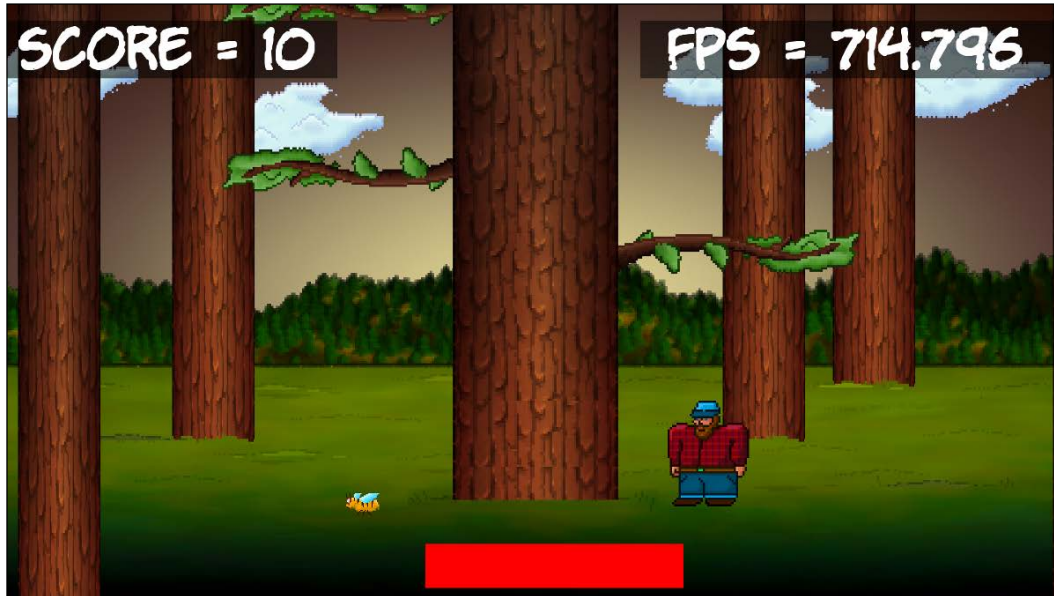
The games we will build

This journey will be smooth as we will learn about the fundamentals of the super-fast C++ language one step at a time, and then put this new knowledge to use by adding cool features to the five games we are going to build.

The following are our five projects for this book.

Timber!!!

The first game is an addictive, fast-paced clone of the hugely successful Timberman, which can be found at <http://store.steampowered.com/app/398710/>. Our game, Timber!!!, will introduce us to all the basics of C++ while we build a genuinely playable game. Here is what our version of the game will look like when we are done and we have added a few last-minute enhancements:



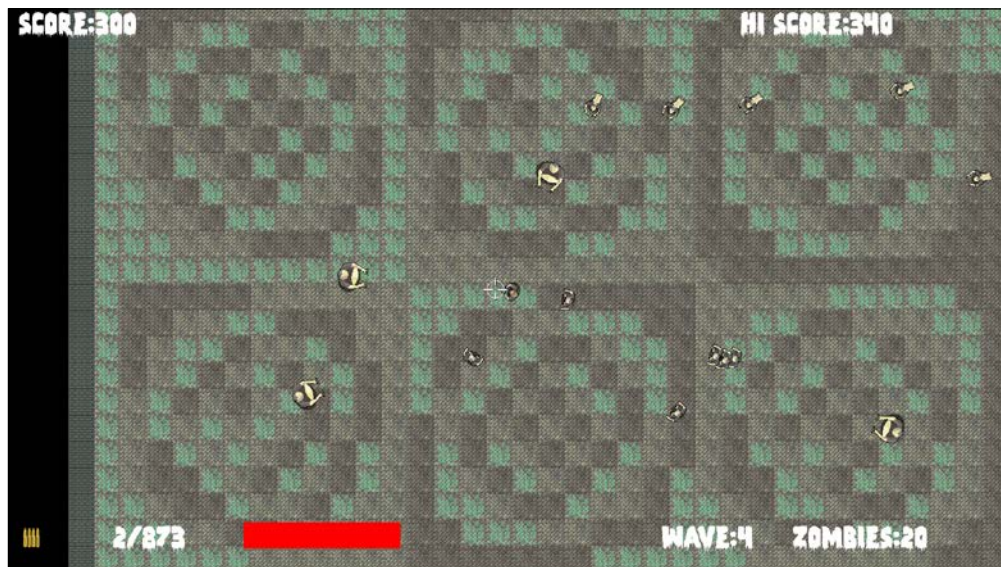
Pong

Pong was one of the first video games to be made, and you can find out about its history here: <https://en.wikipedia.org/wiki/Pong>. It is an excellent example of how the basics of game object animation and dynamic collision detection work. We will build this simple retro game to explore the concept of classes and object-oriented programming. The player will use the bat at the bottom of the screen and hit the ball back to the top of the screen:



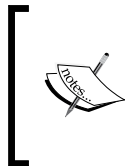
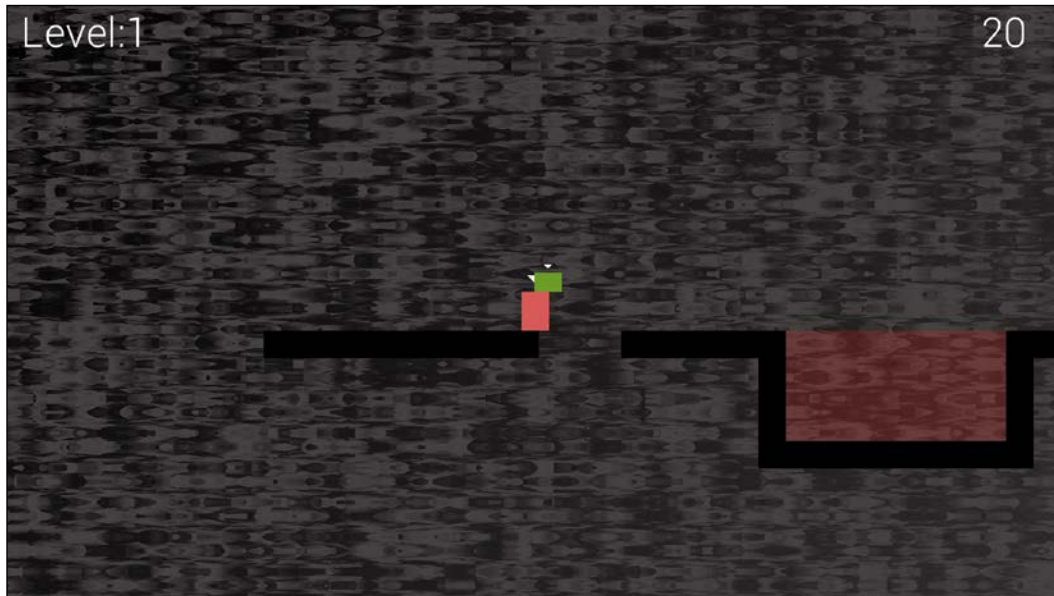
Zombie Arena

Next, we will build a frantic, zombie survival shooter, not unlike the Steam hit *Over 9,000 Zombies!*, which you can find out more about at <http://store.steampowered.com/app/273500/>. The player will have a machine gun and must fight off ever-growing waves of zombies. All this will take place in a randomly generated, scrolling world. To achieve this, we will learn about how object-oriented programming allows us to have a large **code base** (lots of code) that is easy to write and maintain. Expect exciting features such as hundreds of enemies, rapid-fire weaponry, pickups, and a character that can be "leveled up" after each wave:



Thomas was late

The fourth game will be a stylish and challenging single-player and co-op puzzle platformer. It is based on the very popular game *Thomas was Alone* (<http://store.steampowered.com/app/220780/>). Expect to learn about cool topics such as particle effects, OpenGL Shaders, and split-screen cooperative multiplayer:



If you want to play any of the games now, you can do so from the download bundle in the Runnable Games folder. Just double-click on the appropriate .exe file. Note that, in this folder, you can run either the completed games or any game in its partially completed state from any chapter.

Space Invaders ++

The final game will be a Space Invaders clone. In some ways, the game itself is not what is important about this project. The project will be used to learn about game programming patterns. As will become abundantly clear as this book progresses, our code keeps getting longer and more complicated. Each project will introduce one or more techniques for coping with this, but the complexity and length of our code will keep coming back to challenge us, despite these techniques.

The Space Invaders project (called Space Invaders++) will show us ways in which we can radically reorganize our game code also that we can take control of and properly manage our code once and for all. This will leave you with all the knowledge you need to plan and build deep, complex, and innovative games, without ending up in a tangle of code.

The game will also introduce concepts such as screens, input handlers, and entity-component systems. It will also allow us to learn how to let the player use a gamepad instead of the keyboard and introduce the C++ concepts of smart pointers, casts, assertions, breakpoint debugging, and teach us the most important lesson from the whole book: how to build your own unique games:



Let's get started by introducing C++, Visual Studio, and SFML!

Meet C++

Now that we know what games we will be building, let's get started by introducing C++, Visual Studio, and SFML. One question you might have is, *why use the C++ language at all?* C++ is fast – very fast. What makes this true is the fact that the code that we write is directly translated into machine-executable instructions. These instructions are what make the game. The executable game is contained within a .exe file, which the player can simply double-click to run.

There are a few steps in the process of changing our code into an executable file. First, the **preprocessor** looks to see if any *other code* needs to be included within our own code and adds it. Next, all the code is **compiled** into **object files** by the **compiler** program. Finally, a third program, called the **linker**, joins all the object files into the executable file for our game.

In addition, C++ is well established at the same time as being extremely up to date. C++ is an **object-oriented programming (OOP)** language, which means we can write and organize our code using well-tested conventions that make our games efficient and manageable. The benefits as well as the necessity of this will reveal themselves as we progress through this book.

Most of this *other code* that I referred to, as you might be able to guess, is SFML, and we will find out more about SFML in just a minute. The preprocessor, compiler, and linker programs I have just mentioned are all part of the Visual Studio **integrated development environment (IDE)**.

Microsoft Visual Studio

Visual Studio hides away the complexity of preprocessing, compiling, and linking. It wraps it all up into the press of a button. In addition to this, it provides a slick user interface for us to type our code into and manage what will become a large selection of code files and other project assets as well.

While there are advanced versions of Visual Studio that cost hundreds of dollars, we will be able to build all five of our games in the free "**Express 2019 for Community**" version. This is the latest free version of Visual Studio.

SFML

SFML is the **Simple Fast Media Library**. It is not the only C++ library for games and multimedia. It is possible to make an argument to use other libraries, but SFML seems to come through for me every time. Firstly, it is written using object-oriented C++. The benefits of object-oriented C++ are numerous, and you will experience them as you progress through this book.

SFML is also easy to get started with and is therefore a good choice if you are a beginner, yet at the same time it has the potential to build the highest-quality 2D games if you are a professional. So, a beginner can get started using SFML and not worry about having to start again with a new language/library as their experience grows.

Perhaps the biggest benefit is that most modern C++ programming uses OOP. Every C++ beginner's guide I have ever read uses and teaches OOP. OOP is the future (and the now) of coding in almost all languages, in fact. So why, if you're learning C++ from the beginning, would you want to do it any other way?

SFML has a module (code) for just about anything you would ever want to do in a 2D game. SFML works using OpenGL, which can also make 3D games. OpenGL is the de facto free-to-use graphics library for games when you want it to run on more than one platform. When you use SFML, you are automatically using OpenGL.

SFML allows you to create the following:

- 2D graphics and animations, including scrolling game worlds.
- Sound effects and music playback, including high-quality directional sound.
- Input handling with a keyboard, mouse, and gamepad.
- Online multiplayer features.
- The same code can be compiled and linked on all major desktop operating systems, and mobile as well!.

Extensive research has not uncovered any more suitable ways to build 2D games for PC, even for expert developers and especially if you are a beginner and want to learn C++ in a fun gaming environment.

In the sections that follow, we will set up the development environment, beginning with a discussion on what to do if you are using Mac or Linux operating systems.

Setting up the development environment

Now that you know a bit more about how we will be making games, it is time to set up a development environment so we can get coding.

What about Mac and Linux?

The games that we will be making can be built to run on Windows, Mac, and Linux! The code we use will be identical for each platform. However, each version does need to be compiled and linked on the platform for which it is intended, and Visual Studio will not be able to help us with Mac and Linux.

It would be unfair to say, especially for complete beginners, that this book is entirely suited for Mac and Linux users. Although, I guess, if you are an enthusiastic Mac or Linux user and you are comfortable with your operating system, you will likely succeed. Most of the extra challenges you will encounter will be in the initial setup of the development environment, SFML, and the first project.

To this end, I can highly recommend the following tutorials, which will hopefully replace the next 10 pages (approximately), up to the *Planning Timber!!!* section, when this book will become relevant to all operating systems.

For Linux, read this to replace the next few sections: <https://www.sfml-dev.org/tutorials/2.5/start-linux.php>.

On Mac, read this tutorial to get started: <https://www.sfml-dev.org/tutorials/2.5/start-osx.php>.

Installing Visual Studio 2019 Community edition

To start creating a game, we need to install Visual Studio 2019. Installing Visual Studio can be almost as simple as downloading a file and clicking a few buttons. I will walk you through the installation process a step at a time.



Note that, over the years, Microsoft is likely to change the name, appearance, and download page that's used to obtain Visual Studio. They are likely to change the layout of the user interface and make the instructions that follow out of date. However, the settings that we configure for each project are fundamental to C++ and SFML, so careful interpretation of the instructions that follow in this chapter will likely be possible, even if Microsoft does something radical to Visual Studio. Anyway, at the time of writing, Visual Studio 2019 has been released for just two weeks, so hopefully this chapter will be up to date for a while. If something significant does happen, then I will add an up-to-date tutorial on <http://gamecodeschool.com> as soon as I find out about it.

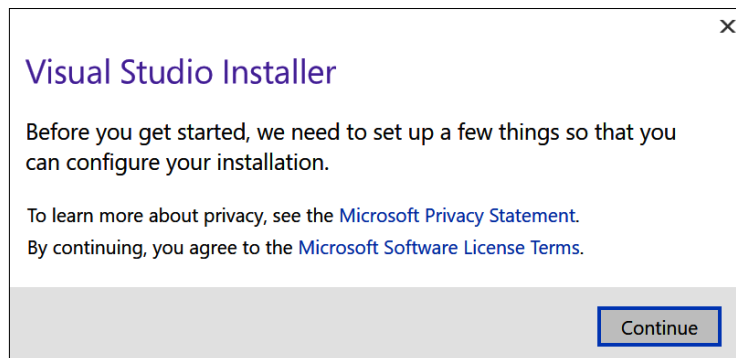
Let's get started with installing Visual Studio:

1. The first thing you need is a Microsoft account and login details. If you have a Hotmail or MSN email address, then you already have one. If not, you can sign up for a free one here: <https://login.live.com/>.
2. The next step is to visit <https://visualstudio.microsoft.com/vs/> and find the download link for **Community 2019**. This is what it looks like at the time of writing:

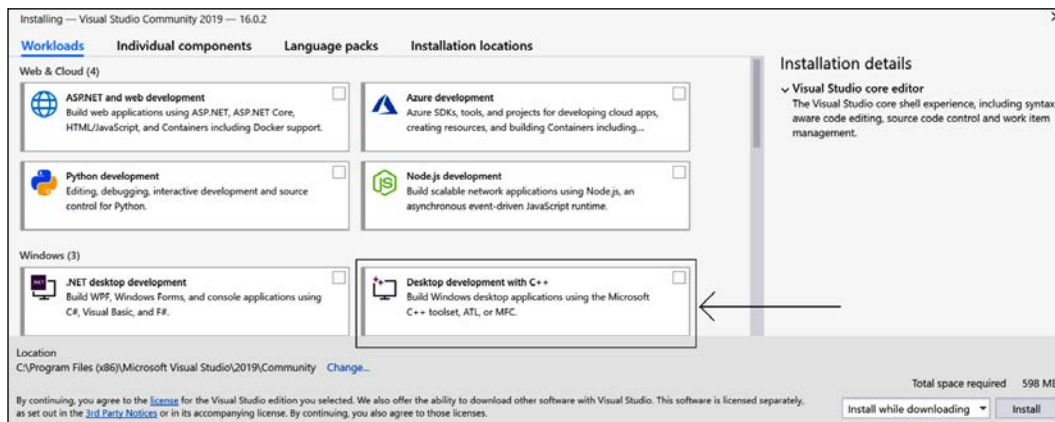


3. Save the file to your computer.

- When the download completes, run the download by double-clicking on it. My file, at the time of writing, was called `vs_community__33910147.1551368984.exe`. Yours will be different based on the current version of Visual Studio.
- After giving permission for Visual Studio to make changes to your computer, you will be greeted with the following window. Click **Continue**:



- Wait for the installer program to download some files and set up the next stage of the installation. Shortly, you will be presented with the following window:



- If you want to choose a new location to install Visual Studio, locate the **Change** option and configure the install location. The simplest thing to do is leave the file at the default location chosen by Visual Studio. When you are ready, locate the **Desktop development with C++** option and select it.

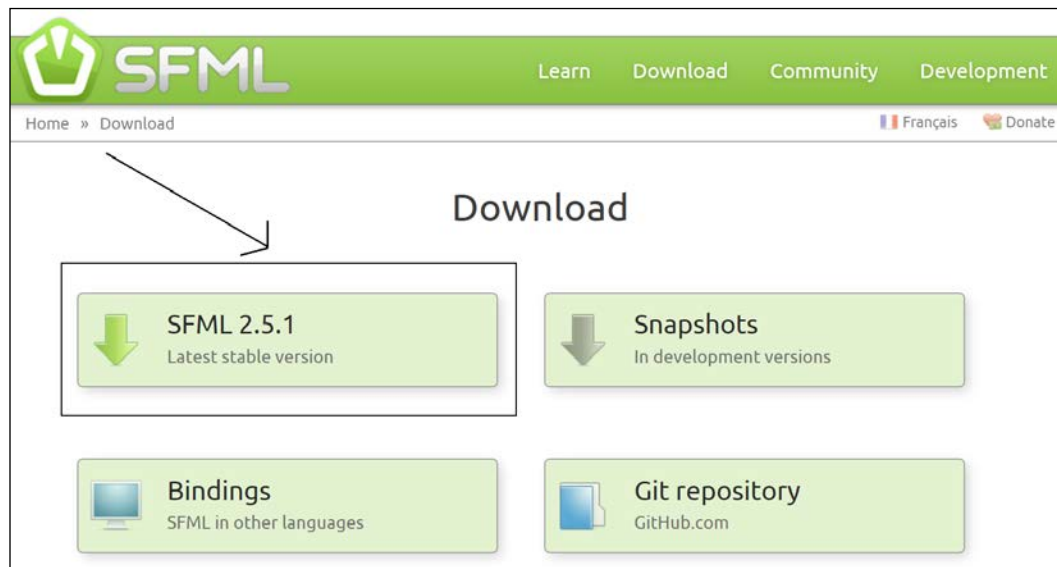
8. Next, click the **Install** button. Grab some refreshments as this step might take a while.
9. When the process completes, you can close all open windows, including any that prompt you to start a new project, as we are not ready to start coding until we have installed SFML.

Now, we are ready to turn our attention to SFML.

Setting up SFML

This short tutorial will guide you through downloading the SFML files that allow us to include the functionality contained in the library in our projects. In addition, we will see how we can use the SFML **DLL** files that will enable our compiled object code to run alongside SFML. To set up SFML, follow these steps:

1. Visit this link on the SFML website: <http://www.sfml-dev.org/download.php>. Click on the button that says **Latest stable version**, as shown here:



- By the time you read this book, the latest version will almost certainly have changed. This won't matter as long as you do the next step just right. We want to download the **32-bit version of Visual C++ 2017**. This might sound counter-intuitive because we have just installed Visual Studio 2019 and you probably (most commonly) have a 64-bit PC. The reason we chose to download the 32-bit version is that Visual C++ 2017 is part of Visual Studio 2019 (Visual Studio does more than C++) and we will be building games in 32-bit so that they can run on *both* 32- and 64-bit machines. Click the **Download** button that's shown in the following screenshot:

Download SFML 2.5.1

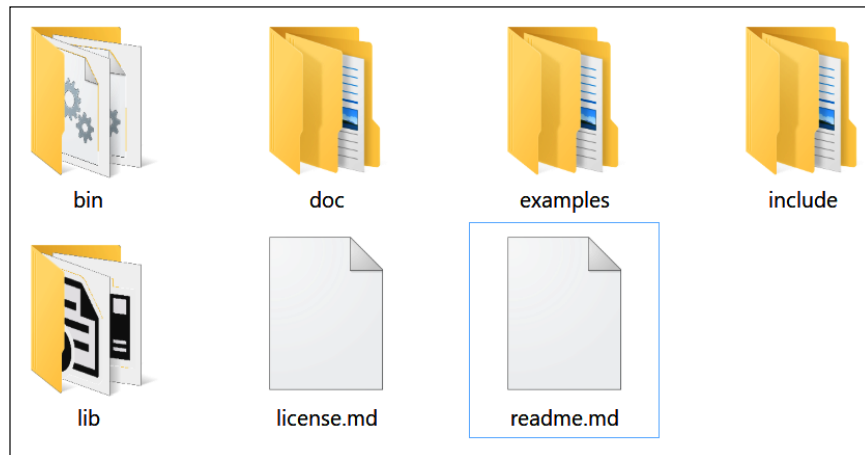
On Windows, choosing 32 or 64-bit libraries should be based on which platform you want to compile for, not which OS you have. Indeed, you can perfectly compile and run a 32-bit program on a 64-bit Windows. So you'll most likely want to target 32-bit platforms, to have the largest possible audience. Choose 64-bit packages only if you have good reasons.

The compiler versions have to match 100%!
 Here are links to the specific MinGW compiler versions used to build the provided packages:
 TDM 5.1.0 (32-bit), MinGW Builds 7.3.0 (32-bit), MinGW Builds 7.3.0 (64-bit)

| | | | |
|--|------------------------------------|--------------------------------|------------------------------------|
| Visual C++ 15 (2017) - 32-bit | Download 16.3 MB | Visual C++ 15 (2017) - 64-bit | Download 18.0 MB |
| Visual C++ 14 (2015) - 32-bit | Download 18.0 MB | Visual C++ 14 (2015) - 64-bit | Download 19.9 MB |
| Visual C++ 12 (2013) - 32-bit | Download 18.3 MB | Visual C++ 12 (2013) - 64-bit | Download 20.3 MB |
| GCC 5.1.0 TDM (SJLJ) - Code::Blocks - 32-bit | Download 14.1 MB | | |
| GCC 7.3.0 MinGW (DW2) - 32-bit | Download 15.5 MB | GCC 7.3.0 MinGW (SEH) - 64-bit | Download 16.5 MB |

- When the download completes, create a folder at the root of the same drive where you installed Visual Studio and name it `SFML`. Also, create another folder at the root of the drive where you installed Visual Studio and call it `VS Projects`.

4. Finally, unzip the SFML download. Do this on your desktop. When unzipping is complete, you can delete the .zip folder. You will be left with a single folder on your desktop. Its name will reflect the version of SFML that you downloaded. Mine is called `SFML-2.5.1-windows-vc15-32-bit`. Your filename will likely reflect a more recent version. Double-click this folder to see its contents, then double-click again into the next folder (mine is called `SFML-2.5.1`). The following screenshot shows what my `SFML-2.5.1` folder's content looks like. Yours should look the same:



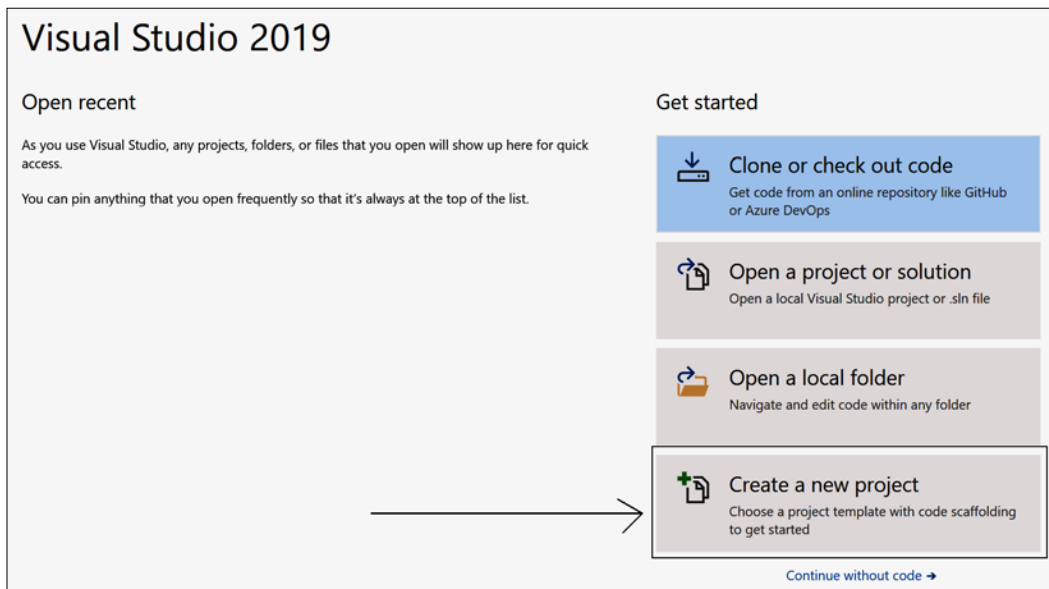
5. Copy the entire contents of this folder and paste all the files and folders into the SFML folder that you created in *Step 3*. For the rest of this book, I will refer to this folder simply as "your SFML folder".

Now, we are ready to start using C++ and SFML in Visual Studio.

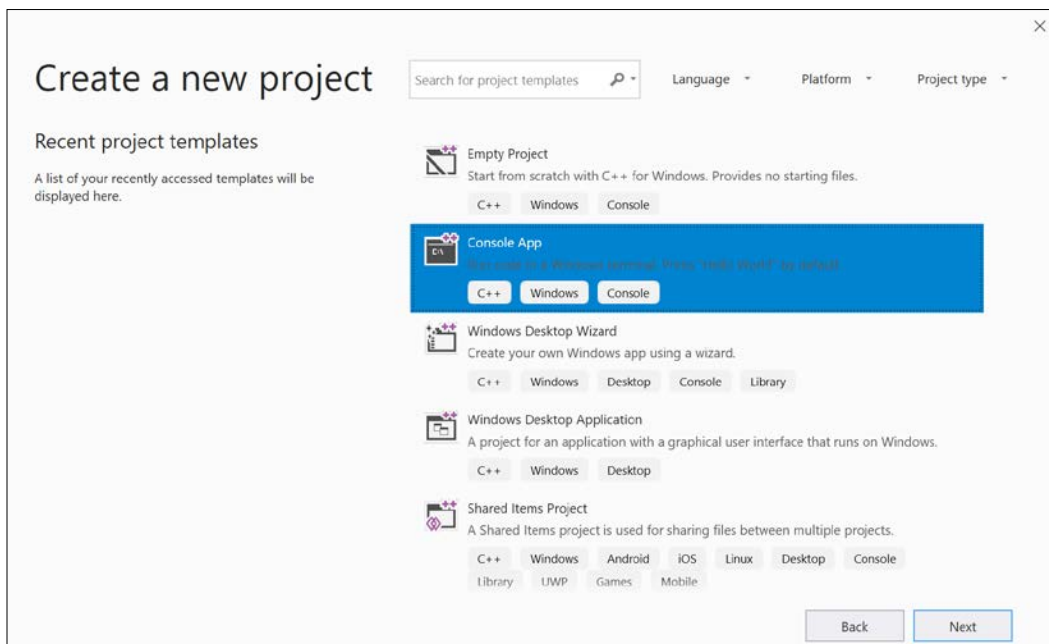
Creating a new project

As setting up a project is a fiddly process, we will go through it step by step so that we can start getting used to it:

1. Start Visual Studio in the same way you start any app: by clicking on its icon. The default installation options will have placed a **Visual Studio 2019** icon in the Windows start menu. You will see the following window:



2. Click on the **Create a new project** button, as highlighted in the preceding screenshot. You will see the **Create a new project** window, as shown in the following screenshot:



3. In the **Create a new project** window, we need to choose the type of project we will be creating. We will be creating a console app, so select **Console App**, as highlighted in the preceding screenshot, and click the **Next** button. You will then see the **Configure your new project** window. This following screenshot shows the **Configure your new project** window after the next three steps have been completed:

Configure your new project

Console App C++ Windows Console

Project name

Timber

Location

D:\VS Projects\ ...

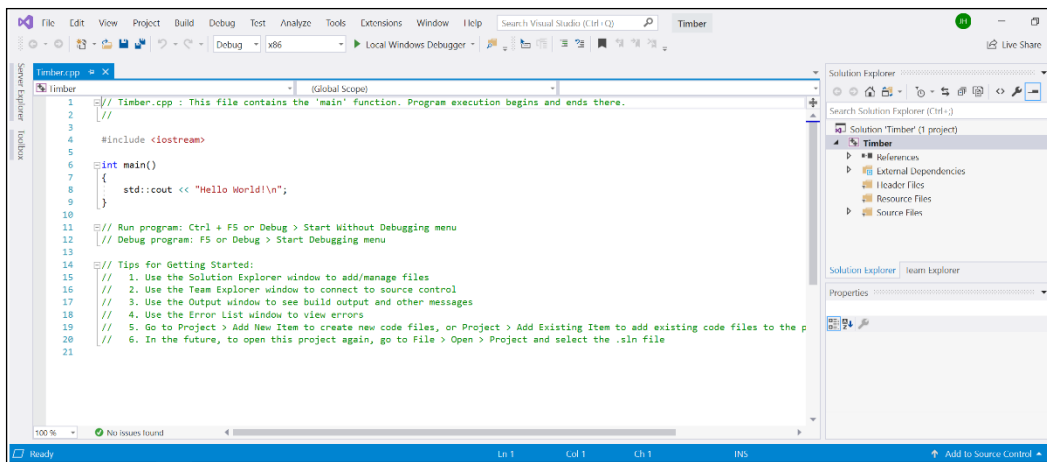
Solution name ⓘ

Timber

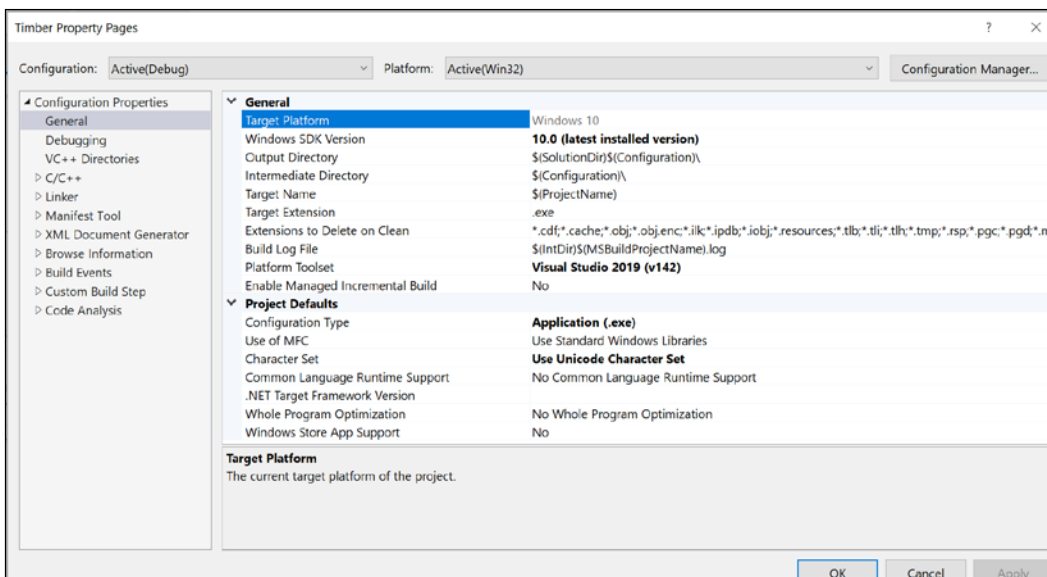
☒ Place solution and project in the same directory

Back Create

4. In the **Configure your new project** window, type `Timber` in the **Project name** field. Note that this causes Visual Studio to automatically configure the **Solution name** field to the same name.
5. In the **Location** field, browse to the `VS Projects` folder that we created in the previous tutorial. This will be the location that all our project files will be kept.
6. Check the option to **Place solution and project in the same directory**.
7. Note that the preceding screenshot shows what the window looks like when the previous three steps have been completed. When you have completed these steps, click **Create**. The project will be generated, including some C++ code. This following screenshot shows where we will be working throughout this book:



8. We will now configure the project to use the SFML files that we put in the SFML folder. From the main menu, select **Project | Timber properties....** You will see the following window:

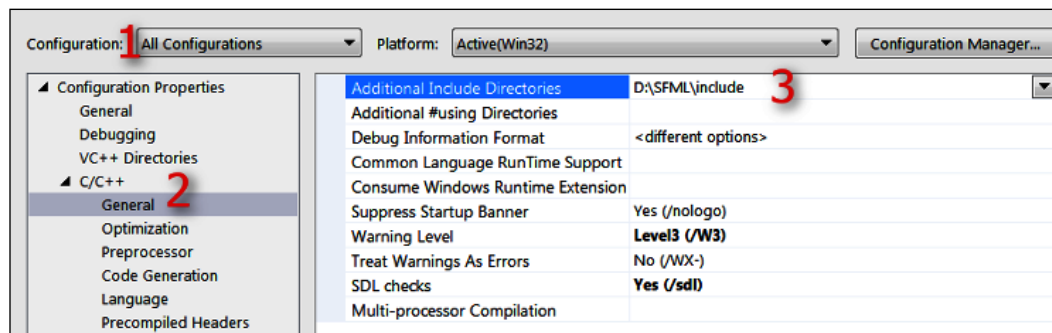


In the preceding screenshot, the **OK**, **Cancel**, and **Apply** buttons are not fully formed. This is likely a glitch with Visual Studio not handling my screen resolution correctly. Yours will hopefully be fully formed. Whether your buttons appear like mine do or not, continuing with the tutorial will be the same.

Next, we will begin to configure the project properties. As these steps are quite intricate, I will cover them in a new list of steps.

Configuring the project properties

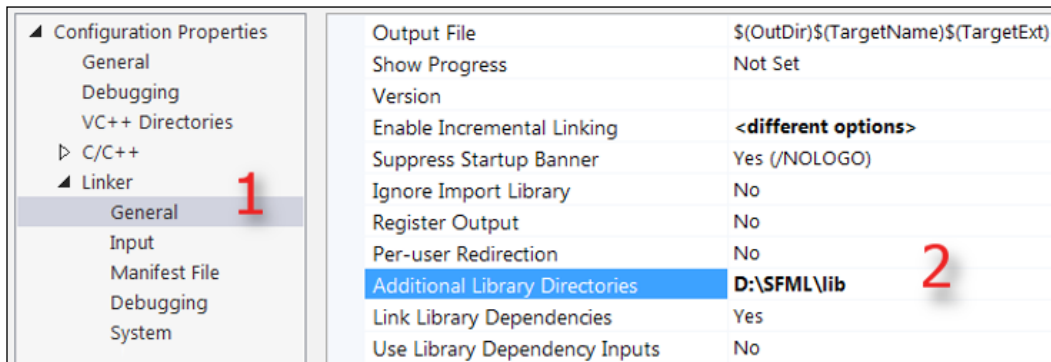
At this stage, you should have the **Timber Property Pages** window open, as shown in the preceding screenshot at the end of the previous section. Now, we will begin to configure some properties while using the following annotated screenshot for guidance:



We will add some fairly intricate and important project settings in this section. This is the laborious part, but we will only need to do this once per project. What we need to do is tell Visual Studio where to find a special type of code file from SFML. The special type of file I am referring to is a **header file**. Header files are the files that define the format of the SFML code so that when we use the SFML code, the compiler knows how to handle it. Note that the header files are distinct from the main source code files and they are contained in files with the `.hpp` file extension. All this will become clearer when we eventually start adding our own header files in the second project. In addition, we need to tell Visual Studio where it can find the SFML library files. In the **Timber Property Pages** window, perform the following three steps, which are numbered in the preceding screenshot:

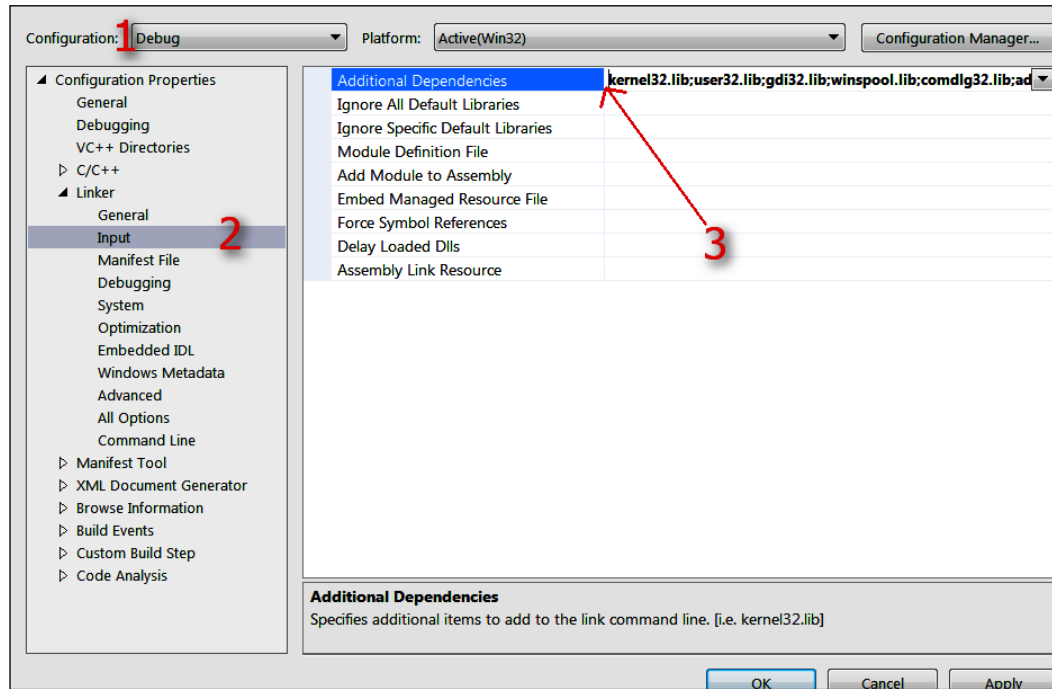
1. First (1), select **All Configurations** from the **Configuration:** drop down.
2. Second (2), select **C/C++** then **General** from the left-hand menu.
3. Third (3), locate the **Additional Include Directories** edit box and type the drive letter where your SFML folder is located, followed by `\SFML\include`. The full path to type, if you located your SFML folder on your D drive, is as shown in the preceding screenshot; that is, `D:\SFML\include`. Vary your path if you installed SFML on a different drive.

4. Click **Apply** to save your configurations so far.
5. Now, still in the same window, perform these steps, which refer to the following annotated screenshot. First (1), select **Linker** and then **General**.
6. Now, find the **Additional Library Directories** edit box (2) and type the drive letter where your SFML folder is, followed by \SFML\lib. So, the full path to type if you located your SFML folder on your D drive is, as shown in the following screenshot, D:\SFML\lib. Vary your path if you installed SFML to a different drive:



7. Click **Apply** to save your configurations so far.
8. Finally, for this stage, still in the same window, perform these steps, which refer to the following annotated screenshot. Switch the **Configuration:** drop down (1) to **Debug** as we will be running and testing our games in debug mode.
9. Select **Linker** and then **Input** (2).
10. Find the **Additional Dependencies** edit box (3) and click into it at the far-left-hand side. Now, copy and paste/type the following: sfml-graphics-d.lib;sfml-window-d.lib;sfml-system-d.lib;sfml-network-d.lib;sfml-audio-d.lib; at the indicated place. Be extra careful to place the cursor exactly in the right place and not to overwrite any of the text that is already there.

11. Click **OK**:




12. Click **Apply** and then **OK**.

Phew; that's it! We have successfully configured Visual Studio and can move on to planning the Timber!!! project.

Planning Timber!!!

Whenever you make a game, it is always best to start with a pencil and paper. If you don't know exactly how your game is going to work on the screen, how can you possibly make it work in code?

[ At this point, if you haven't already, I suggest you go and watch a video of Timberman in action so that you can see what we are aiming for. If you feel your budget can stretch to it, then grab a copy and give it a play. It is often on sale for under \$1 on Steam: <http://store.steampowered.com/app/398710/>.]

The features and objects of a game that define the gameplay are known as the **mechanics**. The basic mechanics of the game are as follows:

- Time is always running out.
- You can get more time by chopping the tree.
- Chopping the tree causes the branches to fall.
- The player must avoid the falling branches.
- Repeat until time runs out or the player is squished.

Expecting you to plan the C++ code at this stage is obviously a bit silly. This is, of course, the first chapter of a C++ beginner's guide. We can, however, take a look at all the assets we will use and an overview of what we will need to make our C++ code do.

Take a look at this annotated screenshot of the game:



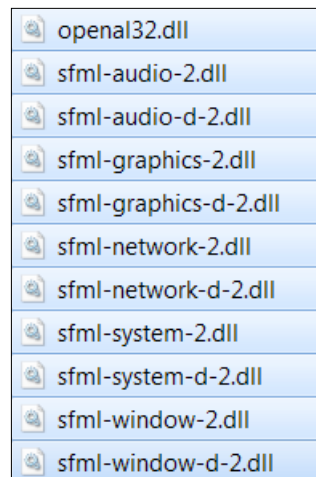
You can see that we have the following features:

- **The player's score:** Each time the player chops a log, they will get one point. They can chop a log with either the left or the right arrow (cursor) key.
- **Player character:** Each time the player chops, they will move to/stay on the same side of the tree relative to the cursor key they use. Therefore, the player must be careful which side they choose to chop on.
- When the player chops, a simple axe graphic will appear in the player character's hands.
- **Shrinking time-bar:** Each time the player chops, a small amount of time will be added to the ever-shrinking time-bar.
- **The lethal branches:** The faster the player chops, the more time they will get, but also the faster the branches will move down the tree and therefore the more likely they are to get squished. The branches spawn randomly at the top of the tree and move down with each chop.
- When the player gets squished – and they will get squished quite regularly – a gravestone graphic will appear.
- **The chopped log:** When the player chops, a chopped log graphic will whiz off, away from the player.
- **Just for decoration:** There are three floating clouds that will drift at random heights and speeds, as well as a bee that does nothing but fly around.
- **The background:** All this takes place on a pretty background.

So, in a nutshell, the player must frantically chop to gain points and avoid running out of time. As a slightly perverse, but fun consequence, the faster they chop, the more likely their squishy demise is.

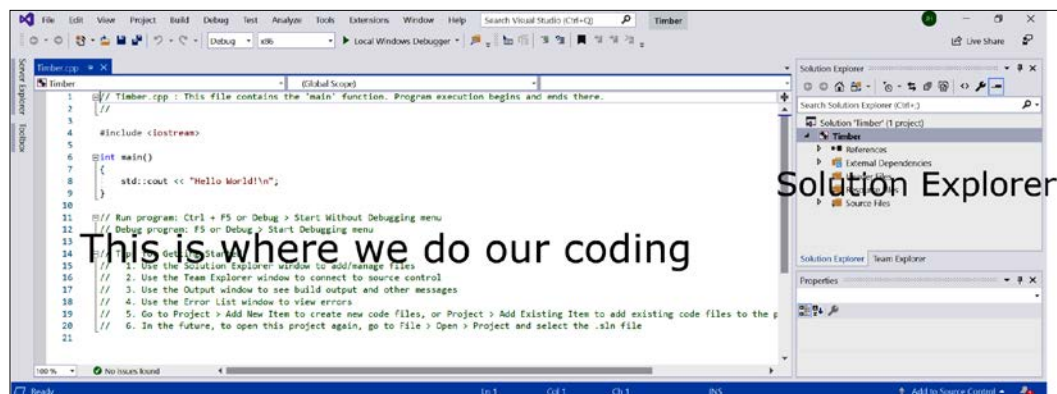
We now know what the game looks like, how it is played, and the motivation behind the game mechanics. Now, we can go ahead and start building it. Follow these steps:

1. Now, we need to copy the SFML .dll files into the main project directory. My main project directory is `D:\VS Projects\Timber`. It was created by Visual Studio in the previous tutorial. If you put your `VS Projects` folder somewhere else, then perform this step there instead. The files we need to copy into the project folder are located in your `SFML\bin` folder. Open a window for each of the two locations and highlight all the files in the `SFML\bin` folder, as shown in the following screenshot:



- Now, copy and paste the highlighted files into the project folder, that is, `D:\VS Projects\Timber`.

The project is now set up and ready to go. You will be able to see the following screen. I have annotated this screenshot so you can start familiarizing yourself with Visual Studio. We will revisit all these areas, and others, soon:



Your layout might look slightly different to what's shown in the preceding screenshot because the windows of Visual Studio, like most applications, are customizable. Take the time to locate the **Solution Explorer** window on the right and adjust it to make its content nice and clear, like it is in the previous screenshot.

We will be back here soon to start coding. But first, we will explore the project assets we will be using.

The project assets

Assets are anything you need to make your game. In our case, these assets include the following:

- A font for the writing on the screen
- Sound effects for different actions, such as chopping, dying, and running out of time
- Graphics for the character, background, branches, and other game objects

All the graphics and sounds that are required for this game are included in the download bundle for this book. They can be found in the `Chapter 1/graphics` and `Chapter 1/sound` folders as appropriate.

The font that is required has not been supplied. This is because I wanted to avoid any possible ambiguity regarding the license. This will not cause a problem, though, as I will show you exactly where and how to choose and download fonts for yourself.

Although I will provide either the assets themselves or information on where to get them, you might like to create or acquire them for yourself.

Outsourcing the assets

There are a number of websites that allow you to contract artists, sound engineers, and even programmers. One of the biggest is Upwork (www.upwork.com). You can join this site for free and post your jobs. You will need to write a clear explanation of your requirements, as well as state how much you are prepared to pay. Then, you will probably get a good selection of contractors bidding to do the work. Be aware, however, that there are a lot of unqualified contractors whose work might be disappointing, but if you choose carefully, you will likely find a competent, enthusiastic, and great-value person or company to do the job.

Making your own sound FX

Sound effects can be downloaded for free from sites such as Freesound (www.freesound.org), but often the licence won't allow you to use them if you are selling your game. Another option is to use an open source software called BFXR from www.bfxr.net, which can help you generate lots of different sound effects that are yours to keep and do with as you like.

Adding the assets to the project

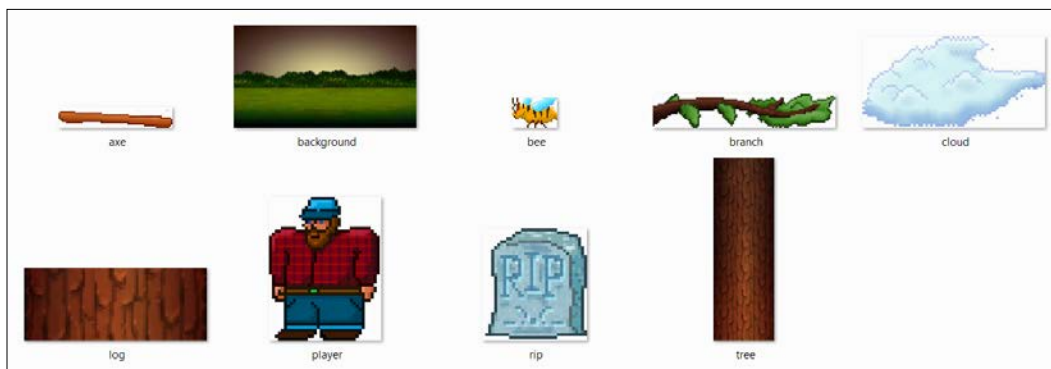
Once you have decided which assets you will use, it is time to add them to the project. The following instructions will assume you are using all the assets that are supplied in this book's download bundle. Where you are using your own, simply replace the appropriate sound or graphic file with your own, using exactly the same filename:

1. Browse to the project folder, that is, `D:\VS Projects\Timber`.
2. Create three new folders within this folder and name them `graphics`, `sound`, and `fonts`.
3. From the download bundle, copy the entire contents of Chapter 1/`graphics` into the `D:\VS Projects\Timber\graphics` folder.
4. From the download bundle, copy the entire contents of Chapter 1/`sound` into the `D:\VS Projects\Timber\sound` folder.
5. Now, visit http://www.1001freefonts.com/komika_poster.font in your web browser and download the **Komika Poster** font.
6. Extract the contents of the zipped download and add the `KOMIKAP_.ttf` file to the `D:\VS Projects\Timber\fonts` folder.

Let's take a look at these assets – especially the graphics – so that we can visualize what is happening when we use them in our C++ code.

Exploring the assets

The graphical assets make up the parts of the scene that is our Timber!!! game. If you take a look at the graphical assets, it should be clear where in our game they will be used:



The sound files are all in `.wav` format. These files contain the sound effects that we will play at certain events throughout the game. They were all generated using BFXR and are as follows:

- `chop.wav`: A sound that is a bit like an axe (a retro axe) chopping a tree
- `death.wav`: A sound a bit like a retro "losing" sound
- `out_of_time.wav`: A sound that plays when the player loses by running out of time, as opposed to being squashed

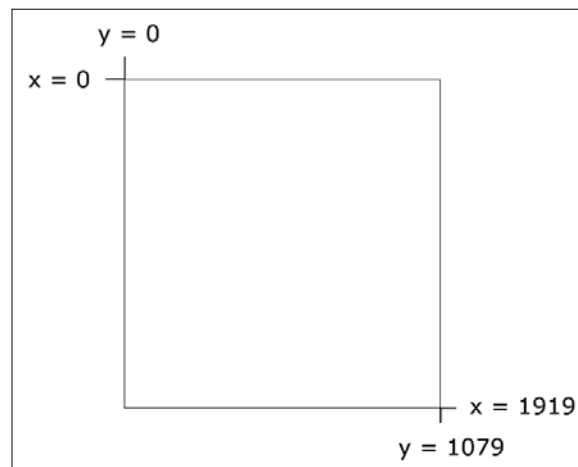
We have seen all the assets, including the graphics, so now we will have a short discussion related to the resolution of the screen and how we position the graphics on it.

Understanding screen and internal coordinates

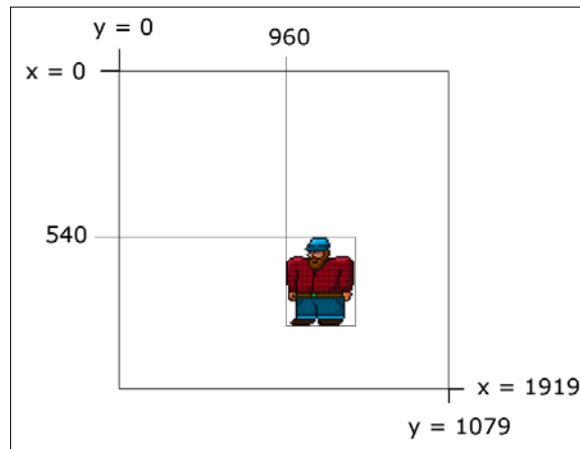
Before we move on to the actual C++ coding, let's talk a little about coordinates. All the images that we see on our monitors are made out of pixels. Pixels are little tiny dots of light that combine to make the images we see.

There are many different resolutions of monitor but, as an example, consider that a fairly typical gaming monitor might have 1,920 pixels horizontally and 1,080 pixels vertically.

The pixels are numbered, starting from the top left of the screen. As you can see from the following diagram, our 1,920 × 1,080 example is numbered from 0 through to 1,919 on the horizontal (x) axis and 0 through 1,079 on the vertical (y) axis:



A specific and exact screen location can therefore be identified by an x and y coordinate. We create our games by drawing the game objects such as the background, characters, bullets, and text to specific locations on the screen. These locations are identified by the coordinates of the pixels. Take a look at the following hypothetical example of how we might draw at the approximately central coordinates of the screen. In the case of a 1,920 x 1080 screen, this would be at the 960, 540 position:

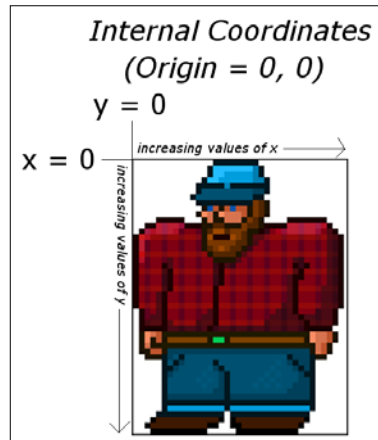


In addition to the screen coordinates, our game objects will each have their own similar coordinate system as well. Like the screen coordinate system, their **internal** or **local** coordinates start at 0,0 in the top left-hand corner.

In the previous image, we can see that 0,0 of the character is drawn at 960, 540 of the screen.

A visual, 2D game object, such as a character or perhaps a zombie, is called a **Sprite**. A sprite is typically made from an image file. All sprites have what is known as an **origin**.

If we draw a sprite to a specific location on the screen, it is the origin that will be located at this specific location. The 0,0 coordinates of the sprite are its origin. The following image demonstrates this:



Therefore, in the image showing the character drawn to the screen, although we drew the image at the central position (960, 540), it appears off to the right and down a bit.

This is important to know as it will help us understand the coordinates we use to draw all the graphics.



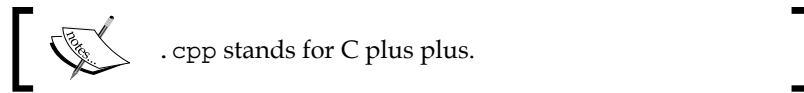
Note that, in the real world, gamers have a huge variety of screen resolutions, and our games will need to work with as many of them as possible. In the third project, we will see how we can make our games dynamically adapt to almost any resolution. In this first project, we will need to assume that the screen resolution is 1,920 x 1,080. If your screen resolution is higher, this will be fine. Don't worry if your screen is lower than this as I have provided a separate set of code for each chapter for the Timber!!! game. The code files are nearly identical apart from adding and swapping a few lines of code near the beginning. If you have a lower-resolution screen, then simply follow the code in this book, which assumes that you have a 1,920 x 1,080 resolution. When it comes to trying out the game, you can copy and paste the code files from the low res folder in the first five chapters as appropriate. In fact, once the extra lines have been added from this first chapter, all the rest of the code will be identical, regardless of your screen resolution. I have supplied the low-resolution code for each chapter, just as a convenience. How the few lines of code work their magic (scale the screen) will be discussed in the third project. The alternative code will work on resolutions as low as 960 x 540 and so should be OK on almost any PC or laptop.

Now, we can write our first piece of C++ code and see it in action.

Getting started with coding the game

Open up Visual Studio if it isn't already open. Open up the Timber!!! project by left-clicking it from the **Recent** list on the main Visual Studio window.

Find the **Solution Explorer** window on the right-hand side. Locate the `Timber.cpp` file under the **Source Files** folder.



Delete the entire contents of the code window and add the following code so that you have the same code yourself. You can do so in the same way that you would with any text editor or word processor; you could even copy and paste it if you prefer. After you have made the edits, we can talk about it:

```
// This is where our game starts from
int main()
{
    return 0;
}
```

This simple C++ program is a good place to start. Let's go through it line by line.

Making code clearer with comments

The first line of code is as follows:

```
// This is where our game starts from
```

Any line of code that starts with two forward slashes (`//`) is a comment and is ignored by the compiler. As such, this line of code does nothing. It is used to leave in any information that we might find useful when we come back to the code at a later date. The comment ends at the end of the line, so anything on the next line is not part of the comment. There is another type of comment called a **multi-line** or **c-style** comment, which can be used to leave comments that take up more than a single line. We will see some of them later in this chapter. Throughout this book, I will leave hundreds of comments to help add context and further explain the code.

The main function

The next line we see in our code is as follows:

```
int main()
```

int is what is known as a **type**. C++ has many types and they represent different types of data. An **int** is an **integer** or whole number. Hold that thought and we will come back to it in a minute.

The `main()` part is the name of the section of code that follows. The section of code is marked out between the opening curly brace (`{`) and the next closing curly brace (`}`).

So, everything in between these curly braces `{ ... }` is a part of `main`. We call a section of code like this a **function**.

Every C++ program has a `main` function and it is the place where the **execution** (running) of the entire program will start. As we progress through this book, eventually, our games will have many code files. However, there will only ever be one `main` function, and no matter what code we write, our game will always begin execution from the first line of code that's inside the opening curly brace of the `main` function.

For now, don't worry about the strange brackets that follow the function name `()`. We will discuss them further in *Chapter 4, Loops, Arrays, Switches, Enumerations, and Functions – Implementing Game Mechanics*, when we get to see functions in a whole new and more interesting light.

Let's look closely at the one single line of code within our `main` function.

Presentation and syntax

Take a look at the entirety of our `main` function again:

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

We can see that, inside `Main`, there is just one single line of code, `return 0;`. Before we move on to find out what this line of code does, let's look at how it is presented. This is useful because it can help us prepare to write code that is easy to read and distinguished from other parts of our code.

First, notice that `return 0;` is indented to the right by one tab. This clearly marks it out as being internal to the `main` function. As our code grows in length, we will see that indenting our code and leaving white space will be essential to maintaining readability.

Next, notice the punctuation on the end of the line. A semicolon (`;`) tells the compiler that it is the end of the instruction and that whatever follows it is a new instruction. We call an instruction that's been terminated by a semicolon a **statement**.

Note that the compiler doesn't care whether you leave a new line or even a space between the semicolon and the next statement. However, not starting a new line for each statement will lead to desperately hard-to-read code, and missing the semicolon altogether will result in a **syntax error** and the game will not compile or run.

A section of code together, often denoted by its indentation with the rest of the section, is called a **block**.

Now that you're comfortable with the idea of the `main` function, indenting your code to keep it tidy, and putting a semicolon on the end of each statement, we can move on to finding out exactly what the `return 0;` statement actually does.

Returning values from a function

Actually, `return 0;` does almost nothing in the context of our game. The concept, however, is an important one. When we use the `return` keyword, either on its own or followed by a value, it is an instruction for the program execution to jump/move back to the code that got the function started in the first place.

Often, the code that got the function started will be yet another function somewhere else in our code. In this case, however, it is the operating system that started the `main` function. So, when `return 0;` is executed, the `main` function exits and the entire program ends.

Since we have a `0` after the `return` keyword, that value is also sent to the operating system. We could change the value of `0` to something else and that value would be sent back instead.

We say that the code that starts a function **calls** the function and that the function **returns** the value.

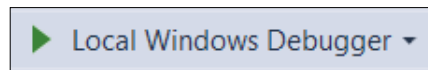
You don't need to fully grasp all this function information just yet. It is just useful to introduce it here. There's one last thing on functions that I will cover before we move on. Remember the `int` from `int main()`? This tells the compiler that the type of value that's returned from `main` must be an `int` (integer/whole number). We can return any value that qualifies as an `int`; perhaps 0, 1, 999, 6,358, and so on. If we try and return something that isn't an `int`, perhaps 12.76, then the code won't compile, and the game won't run.

Functions can return a big selection of different types, including types that we invent for ourselves! That type, however, must be made known to the compiler in the way we have just seen.

This little bit of background information on functions will make things smoother as we progress.

Running the game

You can even run the game at this point. Do so by clicking the **Local Windows Debugger** button in the quick-launch bar of Visual Studio. Alternatively, you can use the `F5` shortcut key:



You will just get a black screen. If the black screen doesn't automatically close itself, you can tap any key to close it. This window is the C++ console, and we can use this to debug our game. We don't need to do this now. What is happening is that our program is starting, executing from the first line of `main`, which is `return 0;`, and then immediately exiting back to the operating system.

We now have the simplest program possible coded and running. We will now add some more code to open a window that the game will eventually appear in.

Opening a window using SFML

Now, let's add some more code. The code that follows will open a window using SFML that *Timber!!!* will eventually run in. The window will be 1,920 pixels wide by 1,080 pixels high, and will be full screen (no border or title).

Enter the new code that is highlighted here to the existing code and then we will examine it. As you type (or copy and paste), try and work out what is going on:

```
// Include important libraries here
#include <SFML/Graphics.hpp>

// Make code easier to type with "using namespace"
using namespace sf;

// This is where our game starts from
int main()
{
    // Create a video mode object
    VideoMode vm(1920, 1080);

    // Create and open a window for the game
    RenderWindow window(vm, "Timber!!!", Style::Fullscreen);

    return 0;
}
```

#including SFML features

The first thing we will notice in our new code is the `#include` directive.

The `#include` **directive** tells Visual Studio to *include*, or add, the contents of another file before compiling. The effect of this is that some other code, which we have not written ourselves, will be a part of our program when we run it. The process of adding code from other files into our code is called **preprocessing** and perhaps unsurprisingly is performed by something called a **preprocessor**. The `.hpp` file extension means it is a **header** file.

Therefore, `#include <SFML/Graphics.hpp>` tells the preprocessor to include the contents of the `Graphics.hpp` file that is contained within the folder named `SFML`. It is the same folder that we created while setting up the project.

This line adds code from the aforementioned file, which gives us access to some of the features of SFML. Exactly how it achieves this will become clearer when we start writing our own separate code files and using `#include` to use them.

The main files that we will be including throughout this book are the SFML header files that give us access to all the cool game-coding features. We will also use `#include` to access the **C++ Standard Library** header files. These header files give us access to core features of the C++ language itself.

What matters for now is that we have a whole bunch of new functionalities that have been provided by SFML available to use if we add that single line of code.

The next new line is using `namespace sf;`. We will come back to what this line does soon.

OOP, classes, and objects

We will fully discuss OOP, classes, and objects as we proceed through this book. What follows is a brief introduction so that we can understand what is happening.

We already know that OOP stands for object-oriented programming. OOP is a programming paradigm, that is, a *way* of coding. OOP is generally accepted throughout the world of programming, in almost every language, as the best, if not the only, professional way to write code.

OOP introduces a lot of coding concepts, but fundamental to them all are **classes** and **objects**. When we write code, whenever possible, we want to write code that is reusable, maintainable, and secure. The way we do this is by structuring our code as a class. We will learn how to do this in *Chapter 6, Object-Oriented Programming – Starting the Pong Game*.

All we need to know about classes for now is that once we have coded our class, we don't just execute that code as part of our game; instead, we create usable objects *from* the class.

For example, if we wanted 100 zombie NPCs (**non-player characters**), we could carefully design and code a class called `Zombie` and then, from that single class, create as many zombie objects as we like. Each and every zombie object would have the same functionality and internal data types, but each and every zombie object would be a separate and distinct entity.

To take the hypothetical zombie example further but without showing any code for the `Zombie` class, we might create a new object based on the `Zombie` class, like this:

```
Zombie z1;
```

The `z1` object is now a fully coded and functioning `Zombie` object. We could then do this:

```
Zombie z2;  
Zombie z3;  
Zombie z4;  
Zombie z5;
```

We now have five separate `Zombie` **instances**, but they are all based on one carefully coded class. Let's take things one step further before we get back to the code we have just written. Our zombies can contain both behavior (defined by functions) as well as data, which might represent things such as the zombie's health, speed, location, or direction of travel. As an example, we could code our `Zombie` class to enable us to use our `Zombie` objects, perhaps like this:

```
z1.attack(player);
z2.growl();
z3.headExplode();
```



Note again that all this zombie code is hypothetical for the moment. Don't type this code into Visual Studio – it will just produce a bunch of errors.

We would design our class so that we can use the data and behaviors in the most appropriate manner to suit our game's objectives. For example, we could design our class so that we can assign values for the data for each zombie object at the time we create it.

Let's say we need to assign a unique name and speed in meters per second at the time we create each zombie. Careful coding of the `Zombie` class could enable us to write code like this:

```
// Dave was a 100 metre Olympic champion before infection
// He moves at 10 metres per second
Zombie z1("Dave", 10);

// Gill had both of her legs eaten before she was infected
// She drags along at .01 metres per second
Zombie z2("Gill", .01);
```

The point is that classes are almost infinitely flexible, and once we have coded the class, we can go about using them by creating an object/instance *of* them. It is through classes and the objects that we create from them that we will harness the power of SFML. And yes, we will also write our own classes, including a `Zombie` class.

Let's get back to the real code we just wrote.

Using namespace sf

Before we move on and look more closely at `VideoMode` and `RenderWindow`, which as you have probably guessed are classes provided by SFML, we will learn what the `using namespace sf;` line of code does.

When we create a class, we do so in a **namespace**. We do this to distinguish our classes from those that others have written. Consider the `VideoMode` class. It is entirely possible that, in an environment such as Windows, somebody has already written a class called `VideoMode`. By using a namespace, we and the SFML programmers can make sure that the names of classes never clash.

The full way of using the `VideoMode` class is like this:

```
sf::VideoMode...
```

using namespace sf; enables us to omit the `sf::` prefix from everywhere in our code. Without it, there would be over 100 instances of `sf::` in this simple game alone. It also makes our code more readable, as well as shorter.

SFML VideoMode and RenderWindow

Inside the `main` function, we now have two new comments and two new lines of actual code. The first line of actual code is this:

```
VideoMode vm(1920, 1080);
```

This code creates an object called `vm` from the class called `VideoMode` and sets up two internal values of 1920 and 1080. These values represent the resolution of the player's screen.

The next new line of code is as follows:

```
RenderWindow window(vm, "Timber!!!", Style::Fullscreen);
```

In the previous line of code, we are creating a new object called `window` from the SFML-provided class called `RenderWindow`. Furthermore, we are setting up some values inside our window object.

Firstly, the `vm` object is used to initialize part of `window`. At first, this might seem confusing. Remember, however, that a class can be as varied and flexible as its creator wants to make it. And yes, some classes can contain other instances of other classes.



It is not necessary to fully understand how this works at this point, as long as you appreciate the concept. We code a class and then make useable objects from that class – a bit like an architect might draw a blueprint. You certainly can't move all your furniture, kids, and dog into the blueprint, but you could build a house (or many houses) from the blueprint. In this analogy, a class is like a blueprint and an object is like a house.

Next, we use the "Timber!!!" value to give the window a name. Then, we use the predefined `Style::Fullscreen` value to make our window object fullscreen.



`Style::Fullscreen` is a value that's defined in SFML. It is useful because we don't need to remember the integer number the internal code uses to represent a full screen. The coding term for this type of value is constant. Constants and their close C++ relatives, **variables**, are covered in the next chapter.

Let's take a look at our window object in action.

Running the game

You can run the game again at this point. You will see a bigger black screen flash on and then disappear. This is the 1,920 x 1,080 fullscreen window that we just coded. Unfortunately, what is still happening is that our program is starting, executing from the first line of `main`, creating the cool new game window, then coming to `return 0;` and immediately exiting back to the operating system.

Next, we will add some code that will form the basic structure of every game in this book. This is known as the game loop.

The main game loop

We need a way to stay in the program until the player wants to quit. At the same time, we should clearly mark out where the different parts of our code will go as we progress with Timber!!!. Furthermore, if we are going to stop our game from exiting, we had better provide a way for the player to exit when they are ready; otherwise, the game will go on forever!

Add the following highlighted code to the existing code and then we will go through it and discuss it all:

```
int main()
{
    // Create a video mode object
    VideoMode vm(1920, 1080);

    // Create and open a window for the game
    RenderWindow window(vm, "Timber!!!", Style::Fullscreen);

    while (window.isOpen())
```

```
{

    /*
    *****
    Handle the players input
    *****
    */

    if (Keyboard::isKeyPressed(Keyboard::Escape))
    {
        window.close();
    }

    /*
    *****
    Update the scene
    *****
    */

    /*
    *****
    Draw the scene
    *****
    */

    // Clear everything from the last frame
    window.clear();

    // Draw our game scene here

    // Show everything we just drew
    window.display();

}

return 0;
}
```

While loops

The very first thing we saw in the new code is as follows:

```
while (window.isOpen())
{
```

The very last thing we saw in the new code is a closing `}`. We have created a **while** loop. Everything between the opening `()` and closing `()` brackets of the `while` loop will continue to execute, over and over, potentially forever.

Look closely between the parentheses `(...)` of the `while` loop, as shown here:

```
while (window.isOpen())
```

The full explanation of this code will have to wait until we discuss loops and conditions in *Chapter 4, Loops, Arrays, Switches, Enumerations, and Functions – Implementing Game Mechanics*. What is important for now is that when the `window` object is set to closed, the execution of the code will break out of the `while` loop and move on to the next statement. Exactly how a window is closed is covered soon.

The next statement is, of course, `return 0;`, which ends our game.

We now know that our `while` loop will whiz round and round, repeatedly executing the code within it, until our window object is set to closed.

C-style code comments

Just inside the `while` loop, we can see what, at first glance, might look a bit like ASCII art:

```
/*
*****
Handle the player's input
*****
*/
```



ASCII art is a niche but fun way of creating images with computer text. You can read more about it here: https://en.wikipedia.org/wiki/ASCII_art.

The previous code is simply another type of comment. This type of comment is known as a C-style comment. The comment begins with `(/*` and ends with `*/`. Anything in between is just for information and is not compiled. I have used this slightly elaborate text to make it absolutely clear what we will be doing in each part of the code file. And of course, you can now work out that any code that follows will be related to handling the player's input.

Skip over a few lines of code and you will see that we have another C-style comment, announcing that in that part of the code, we will be updating the scene.

If you jump to the next C-style comment, it will be clear where we will be drawing all the graphics.

Input, update, draw, repeat

Although this first project uses the simplest possible version of a game loop, every game will need these phases in the code. Let's go over the steps:

1. Get the player's input (if any).
2. Update the scene based on things such as artificial intelligence, physics, or the player's input.
3. Draw the current scene.
4. Repeat these steps at a fast-enough rate to create a smooth, animated game world.

Now, let's look at the code that actually does something within the game loop.

Detecting a key press

Firstly, within the section that's identifiable by the comment with the `Handle the player's input` text, we have the following code:

```
if (Keyboard::isKeyPressed(Keyboard::Escape))
{
    window.close();
}
```

This code checks whether the *Esc* key is currently being pressed. If it is, the highlighted code uses the `window` object to close itself. Now, the next time the `while` loop begins, it will see that the `window` object is closed and jump to the code immediately after the closing curly brace of the `while` loop and the game will exit. We will discuss `if` statements more fully in *Chapter 2, Variables, Operators, and Decisions – Animating Sprites*.

Clearing and drawing the scene

Currently, there is no code in the `Update the scene` section, so let's move on to the `Draw the scene` section.

The first thing we will do is rub out the previous frame of animation using the following code:

```
window.clear();
```

What we would do now is draw every object from the game. However, we don't have any game objects.

The next line of code is as follows:

```
window.display();
```

When we draw all the game objects, we are drawing them to a hidden surface ready to be displayed. The `window.display()` code flips from the previously displayed surface to the newly updated (previously hidden) one. This way, the player will never see the drawing process as the surface has all the sprites added to it. It also guarantees that the scene will be complete before it is flipped. This prevents a graphical glitch known as **tearing**. This process is called **double buffering**.

Also note that all this drawing and clearing functionality is performed using our `window` object, which was created from the SFML `RenderWindow` class.

Running the game

Run the game and you will get a blank, full screen window that remains open until you press the *Esc* key.

That is good progress. At this stage, we have an executing program that opens a window and loops around, waiting for the player to press the *Esc* key to exit. Now, we are able to move on to drawing the background image of the game.

Drawing the game's background

Now, we will get to see some graphics in our game. What we need to do is create a sprite. The first one we will create will be the game background. We can then draw it in between clearing the window and displaying/flipping it.

Preparing the Sprite using a Texture

The SFML `RenderWindow` class allowed us to create our window object, which basically took care of all the functionality that our game's window needs.

We will now look at two more SFML classes that will take care of drawing sprites to the screen. One of these classes, perhaps unsurprisingly, is called `Sprite`. The other class is called `Texture`. A texture is a graphic stored in memory, on the **graphics processing unit (GPU)**.

An object that's made from the `Sprite` class needs an object made from the `Texture` class in order to display itself as an image. Add the following highlighted code. Try and work out what is going on as well. Then, we will go through it, a line at a time:

```
int main()
{
    // Create a video mode object
    VideoMode vm(1920, 1080);

    // Create and open a window for the game
    RenderWindow window(vm, "Timber!!!", Style::Fullscreen);

    // Create a texture to hold a graphic on the GPU
    Texture textureBackground;

    // Load a graphic into the texture
    textureBackground.loadFromFile("graphics/background.png");

    // Create a sprite
    Sprite spriteBackground;

    // Attach the texture to the sprite
    spriteBackground.setTexture(textureBackground);

    // Set the spriteBackground to cover the screen
    spriteBackground.setPosition(0,0);

    while (window.isOpen())
    {
```

First, we create an object called `textureBackground` from the SFML `Texture` class:

```
Texture textureBackground;
```

Once this is done, we can use the `textureBackground` object to load a graphic from our graphics folder into `textureBackground`, like this:

```
textureBackground.loadFromFile("graphics/background.png");
```



We only need to specify `graphics/background` as the path is relative to the Visual Studio **working directory** where we created the folder and added the image.

Next, we create an object called `spriteBackground` from the SFML `Sprite` class with this code:

```
Sprite spriteBackground;
```

Then, we can associate the `Texture` object (`backgroundTexture`) with the `Sprite` object (`backgroundSprite`), like this:

```
spriteBackground.setTexture(textureBackground);
```

Finally, we can position the `spriteBackground` object in the window object at the `0, 0` coordinates:

```
spriteBackground.setPosition(0, 0);
```

Since the `background.png` graphic in the `graphics` folder is 1,920 pixels wide by 1,080 pixels high, it will neatly fill the entire screen. Just note that this previous line of code doesn't actually show the sprite. It just sets its position, ready for when it is shown.

The `backgroundSprite` object can now be used to display the background graphic. Of course, you are almost certainly wondering why we had to do things in such a convoluted way. The reason is because of the way that graphics cards and OpenGL work.

Textures take up graphics memory, and this memory is a finite resource. Furthermore, the process of loading a graphic into the GPU's memory is very slow – not so slow that you can watch it happen or that you will see your PC noticeably slow down while it is happening, but slow enough that you can't do it every frame of the game loop. So, it is useful to disassociate the actual texture (`textureBackground`) from any code that we will manipulate during the game loop.

As you will see when we start to move our graphics, we will do so using the `sprite`. Any objects that are made from the `Texture` class will sit happily on the GPU, just waiting for an associated `Sprite` object to tell it where to show itself. In later projects, we will also reuse the same `Texture` object with multiple different `Sprite` objects, which makes efficient use of GPU memory.

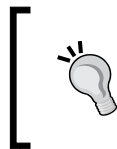
In summary, we can state the following:

- Textures are very slow to load onto the GPU.
- Textures are very fast to access once they are on the GPU.
- We associate a `Sprite` object with a texture.
- We manipulate the position and orientation of `Sprite` objects (usually in the `Update the scene` section).

We draw the `Sprite` object, which, in turn, displays the `Texture` object that is associated with it (usually in the `Draw the scene` section). So, all we need to do now is use our double buffering system, which is provided by our `window` object, to draw our new `Sprite` object (`spriteBackground`), and we should get to see our game in action.

Double buffering the background sprite

Finally, we need to draw that sprite and its associated texture in the appropriate place in the game loop.



Note that when I present code that is all from the same block, I don't add the indentations because it lessens the instances of line wraps in the text of the book. The indenting is implied. Check out the code file in the download bundle to see full use of indenting.

Add the following highlighted code:

```
/*
*****
Draw the scene
*****
*/

// Clear everything from the last run frame
window.clear();

// Draw our game scene here
window.draw(spriteBackground);

// Show everything we just drew
window.display();
```

The new line of code simply uses the `window` object to draw the `spriteBackground` object, in between clearing the display and showing the newly drawn scene.

We now know what a sprite is, and that we can associate a texture with it and then position it on the screen and finally draw it. The game is ready to be run again so that we can see the results of this code.

Running the game

If we run the program now, we will see the first signs that we have a real game in progress:



It's not going to get Indie Game of the Year on Steam in its current state, but we are on the way at least!

Let's look at some of the things that might go wrong in this chapter and as we proceed through this book.

Handling errors

There will always be problems and errors in every project you make. This is guaranteed! The tougher the problem, the more satisfying it is when you solve it. When, after hours of struggling, a new game feature finally bursts into life, it can cause a genuine high. Without this struggle, it would somehow be less worthwhile.

At some point in this book, there will probably be some struggle. Remain calm, be confident that you will overcome it, and then get to work.

Remember that, whatever your problem, it is very likely you are **not** the first person in the world to have ever had this same problem. Think of a concise sentence that describes your problem or error and then type it into Google. You will be surprised how quickly, precisely, and often, someone else will have already solved your problem for you.

Having said that, here are a few pointers (pun intended; see *Chapter 10, Pointers, the Standard Template Library, and Texture Management*) to get you started in case you are struggling with making this first chapter work.

Configuration errors

The most likely cause of problems in this chapter will be **configuration errors**. As you probably noticed during the process of setting up Visual Studio, SFML and the project itself, there's an awful lot of filenames, folders, and settings that need to be just right. Just one wrong setting could cause one of a number of errors, whose text don't make it clear exactly what is wrong.

If you can't get the empty project with the black screen working, it might be easier to start again. Make sure all the filenames and folders are appropriate for your specific setup and then get the simplest part of the code running. This is the part where the screen flashes black and then closes. If you can get to this stage, then configuration is probably not the issue.

Compile errors

Compile errors are probably the most common error we will experience going forward. Check that your code is identical to mine, especially semicolons on the ends of lines and subtle changes in upper and lower case for class and object names. If all else fails, open the code files in the download bundle and copy and paste it in. While it is always possible that a code typo made it into this book, the code files were made from actual working projects – they definitely work!

Link errors

Link errors are most likely caused by missing SFML .dll files. Did you copy all of them into the project folder?

Bugs

Bugs are what happen when your code works, but not as you expect it to. Debugging can actually be fun. The more bugs you squash, the better your game and the more satisfying your day's work will be. The trick to solving bugs is to find them early! To do this, I recommend running and playing your game every time you implement something new. The sooner you find the bug, the more likely the cause will be fresh in your mind. In this book, we will run the code to see the results at every possible stage.

Summary

This was quite a challenging chapter and perhaps a little bit mean to be the first one. It is true that configuring an IDE to use a C++ library can be a bit awkward and long. Also, the concepts of classes and objects are well known to be slightly awkward for people who are new to coding.

Now that we are at this stage, however, we can totally focus on C++, SFML, and games. As we progress with this book, we will learn more and more C++, as well as implement increasingly interesting game features. As we do so, we will take a further look at things such as functions, classes, and objects to help demystify them a little more.

We have achieved plenty in this chapter, including outlining a basic C++ program with the main function, constructing a simple game loop that listens for player input and draws a sprite (along with its associated texture) to the screen.

In the next chapter, we will learn about all the C++ we need to draw some more sprites and animate them.

FAQ

Here are some questions that might be on your mind:

Q) I am struggling with the content that's been presented so far. Am I cut out for programming?

A) Setting up a development environment and getting your head around OOP as a concept is probably the toughest thing you will do in this book. As long as your game is functioning (drawing the background), you are ready to proceed with the next chapter.

Q) All this talk of OOP, classes, and objects is too much and kind of spoiling the whole learning experience.

A) Don't worry. We will keep returning to OOP, classes, and objects constantly. In *Chapter 6, Object-Oriented Programming – Starting the Pong Game*, we will really begin getting to grips with the whole OOP thing. All you need to understand for now is that SFML have written a whole load of useful classes and that we get to use this code by creating usable objects from those classes.

Q) I really don't get this function stuff.

A) It doesn't matter; we will be returning to it again constantly and will learn about functions more thoroughly. You just need to know that, when a function is called, its code is executed, and when it is done (reaches a `return` statement), the program jumps back to the code that called it.

2

Variables, Operators, and Decisions – Animating Sprites

In this chapter, we will do quite a bit more drawing on the screen and, to achieve this, we will need to learn about some of the basics of C++. We will learn how to use variables to remember and manipulate values, and we will begin to add more graphics to the game. As this chapter progresses, we will find out how we can manipulate these values to animate the graphics. These values are known as variables.

Here is what is in store:

- Learning all about C++ variables
- Seeing how to manipulate the values stored in variables
- Adding a static tree graphic, ready for the player to chop away at
- Drawing and animating a bee and three clouds

C++ variables

Variables are the way that our C++ games store and manipulate values/data. If we want to know how much health the player has, we need a variable. Perhaps you want to know how many zombies are left in the current wave. That is a variable as well. If you need to remember the name of the player who got a high score, you guessed it—we need a variable for that. Is the game over or still playing? Yes, that's a variable too.

Variables are named identifiers for locations in the memory of the PC. The memory of the PC is where computer programs are stored as they are being executed. So, we might name a variable `numberOfZombies` and that variable could refer to a place in memory that stores a value to represent the number of zombies that are left in the current wave.

The way that computer systems address locations in memory is complex. Programming languages use variables to give us a human-friendly way to manage our data in that memory.

The small amount we have just mentioned about variables implies that there must be different **types** of variable.

Types of variables

There is a wide variety of C++ variable types (see the next tip about variables in a couple of pages). It would easily be possible to spend an entire chapter discussing them. What follows is a table of the most commonly used types of variable in this book. Then, in the next section, we will look at how to use each of these variable types:

| Type | Examples of values | Explanation |
|--------|--|---|
| int | -42, 0, 1, 9826, and so on. | Integer whole numbers |
| float | -1.26f, 5.8999996f, 10128.3f | Floating-point values with precision up to seven digits |
| double | 925.83920655234, 1859876.94872535 | Floating-point values with precision up to 15 digits |
| char | a, b, c, 1, 2, 3 (a total of 128 symbols, including ?, ~, and #) | Any symbol from the ASCII table |
| bool | true or false | bool stands for Boolean and can be only true or false |
| String | Hello Everyone! I am a String. | Any text value from a single letter or digit up to perhaps an entire book |

The compiler must be told what type of variable it is so that it can allocate the right amount of memory for it. It is good practice to use the best and most appropriate type for each variable you use. In practice, however, you will often get away with promoting a variable. Perhaps you need a floating-point number with just five significant digits? The compiler won't complain if you store it as a `double`. However, if you tried to store a `float` or a `double` in an `int`, it will **change/cast** the value to fit the `int`. As we progress through this book, I will make it plain what the best variable type to use in each case is, and we will even see a few instances where we deliberately convert/cast between variable types.

A few extra details worth noticing in the preceding table include the `f` postfix next to all of the `float` values. This `f` postfix tells the compiler that the value is a `float` type, not `double`. A floating-point value without the `f` prefix is assumed to be `double`. See the next tip about variables for more about this.

As we mentioned previously, there are many more types. If you want to find out more about types, see the next tip about variables.

User-defined types

User-defined types are way more advanced than the types we have just seen. When we talk about user-defined types in C++, we are usually talking about classes. We briefly talked about classes and their related objects in the previous chapter. We would write code in a separate file, sometimes two. We are then able to declare, initialize, and use them. We will leave how we define/create our own types until *Chapter 6, Object-Oriented Programming – Starting the Pong Game*.

Declaring and initializing variables

So far, we know that variables are for storing the data/values that our games need in order to work. For example, a variable would represent the number of lives a player has or the player's name. We also know that there is a wide selection of different types of values that these variables can represent, such as `int`, `float`, `bool`, and so on. Of course, what we haven't seen yet is how we would actually go about using a variable.

There are two stages when it comes to creating and preparing a new variable. These stages are called **declaration** and **initialization**.

Declaring variables

We can declare variables in C++ like this:

```
// What is the player's score?
int playerScore;

// What is the player's first initial
char playerInitial;

// What is the value of pi
float valuePi;

// Is the player alive or dead?
bool isAlive;
```

Once we have written the code to declare a variable, it exists and is ready to be used in our code. However, we will usually want to give the variable an appropriate value, which is where initialization comes in.

Initializing variables

Now that we have declared the variables with meaningful names, we can initialize those same variables with appropriate values, like this:

```
playerScore = 0;
playerInitial = 'J';
valuePi = 3.141f;
isAlive = true;
```

At this point, the variable exists and holds a specific value. Soon, we will see how we can change, test, and respond to these values. Next, we will see that we can combine declaring and initializing into one step.

Declaring and initializing in one step

When it suits us, we can combine the declaration and initialization steps into one. Sometimes, we know what value a variable must start the program with, and declaring and initializing in one step is appropriate. Often, we won't, and we will first declare the variable and then initialize it later in the program, like so:

```
int playerScore = 0;
char playerInitial = 'J';
float valuePi = 3.141f;
bool isAlive = true;
```



Variables tip

As promised, here is the tip on variables. If you want to see a complete list of C++ types, then check out this web page: http://www.tutorialspoint.com/cplusplus/cpp_data_types.htm. If you want a deeper discussion on float, double, and the f postfix, then read this: <http://www.cplusplus.com/forum/beginner/24483/>. Finally, if you want to know the ins and outs of the ASCII character codes, then there is some more information here: <http://www.cplusplus.com/doc/ascii/>. Note that these links are for the extra curious reader and we have already discussed enough in order to proceed.

Constants

Sometimes, we need to make sure that a value can never be changed. To achieve this, we can declare and initialize a **constant** using the `const` keyword:

```
const float PI = 3.141f;
const int PLANETS_IN_SOLAR_SYSTEM = 8;
const int NUMBER_OF_ENEMIES = 2000;
```

It is convention to declare constants in all uppercase. The values of the preceding constants can never be altered. We will see some constants in action in *Chapter 4, Loops, Arrays, Switches, Enumerations, and Functions – Implementing Game Mechanics*.

Declaring and initializing user-defined types

We have already seen examples of how we can declare and initialize some SFML defined types. It is because of the way that we can create/define these types (classes) so flexibly that the way we declare and initialize them is also so varied. Here are a couple of reminders for declaring and initializing user-defined types from the previous chapter.

Create an object of the `VideoMode` type called `vm` and initialize it with two `int` values, 1920 and 1080:

```
// Create a video mode object
VideoMode vm(1920, 1080);
```

Create an object of the `Texture` type called `textureBackground`, but don't do any initialization:

```
// Create a texture to hold a graphic on the GPU
Texture textureBackground;
```

Note that it is possible (in fact, very likely) that even though we are not suggesting any specific values with which to initialize `textureBackground`, some setup of variables may take place internally. Whether or not an object needs/has the option of giving initialization values at this point is entirely dependent on how the class is coded and is almost infinitely flexible. This further suggests that, when we get to write our own classes, there will be some complexity. Fortunately, this also means we will have significant power to design our types/classes to be just what we need to make our games! Add this huge flexibility to the power of the SFML designed classes and the potential for our games is almost limitless.

We will see a few more user created types/classes provided by SFML in this chapter too, and loads more throughout this book.

We have now seen that a variable is a named location in the computer's memory and that a variable can be a simple integer through to a more powerful object. Now that we know we can initialize these variables, we will look at how we can manipulate the values they hold.

Manipulating variables

At this point, we know exactly what variables are, the main types they can be, and how to declare and initialize them. We still can't do that much with them, however. We need to manipulate our variables; add them; take them away; and multiply, divide, and test them.

First, we will deal with how we can manipulate them and then we will look at how and why we test them.

C++ arithmetic and assignment operators

In order to manipulate variables, C++ has a range of **arithmetic operators** and **assignment operators**. Fortunately, most arithmetic and assignment operators are quite intuitive to use and those that aren't are quite easy to explain. To get us started, let's look at a table of arithmetic operators, followed by a table of assignment operators, all of which we will regularly use throughout this book:

| Arithmetic operator | Explanation |
|---------------------|---|
| + | The addition operator can be used to add together the values of two variables or values. |
| - | The subtraction operator can be used to take away the value of one variable or value from another variable or value. |
| * | The multiplication operator can multiply the value of variables and values. |
| / | The division operator can divide the value of variables and values. |
| % | The Modulo operator divides a value or variable by another value or variable to find the remainder of the operation. |

And now for the assignment operators:

| Assignment operators | Explanation |
|----------------------|---|
| = | We have already seen this one. It is the assignment operator. We use it to initialize/set a variable's value. |
| += | Add the value on the right-hand side to the variable on the left. |
| -= | Takes away the value on the right-hand side from the variable on the left. |
| *= | This multiplies the value on the right-hand side by the variable on the left. |
| /= | This divides the value on the right-hand side by the variable on the left. |

| Assignment operators | Explanation |
|----------------------|---|
| ++ | An increment operator adds one to a variable. |
| -- | A decrement operator takes away one from a variable. |



Technically, all of these operators, except for =, --, and ++, are called **compound assignment operators** because they comprise more than one operator.

Now that we have seen a good range of arithmetic and assignment operators, we can actually look at how we can manipulate our variables by combining operators, variables, and values to form **expressions**.

Getting things done with expressions

Expressions are the result of combining variables, operators, and values. Using expressions, we can arrive at a result. Furthermore, as we will soon see, we can use an expression in a test. These tests can be used to decide what our code should do next. First, let's look at some simple expressions we might see in our game code. Here is one example of a simple expression:

```
// Player gets a new high score
hiScore = score;
```

In the preceding code, the value held in the `score` variable is used to change the value in the `hiScore` variable. The two variables now hold the same value, but note that they are still separate and distinct variables (places in memory). This would likely be just what we need when the player beats a high score. Here is another example:

```
// Set the score to 100
score = 100;
```

Let's take a look at the addition operator, which used in conjunction with the assignment operator:

```
// Add to the score when an alien is shot
score = aliensShot + wavesCleared;
```

In the preceding code, the values held by `aliensShot` and `wavesCleared` are added together using the addition operator and then the result of the addition is assigned to the `score` variable. Now, let's take a look at the following code:

```
// Add 100 to whatever the score currently is
score = score + 100;
```

Note that it is perfectly acceptable to use the same variable on both sides of an operator. In the preceding code, 100 is added to the value held by the `score` variable and then this new value is then assigned back into `score`.

Look at the subtraction operator in conjunction with the assignment operator. The following code subtracts the value on the right-hand side of the subtraction operator from the value on the left. It is usually used in conjunction with the assignment operator, perhaps like so:

```
// Uh oh lost a life
lives = lives - 1;
```

It can also be used like this:

```
// How many aliens left at end of game
aliensRemaining = aliensTotal - aliensDestroyed;
```

Next, we will see how we might use the division operator. The following code divides the number on the left by the number on the right. Again, it is usually used with the assignment operator, like this:

```
// Make the remaining hit points lower based on swordLevel
hitPoints = hitPoints / swordLevel;
```

It can also be used like this:

```
// Give something, but not everything, back for recycling a block
recycledValueOfBlock = originalValue / .9f;
```

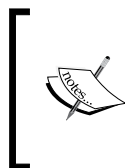
Obviously, in the previous example, the `recycledValueOfBlock` variable will need to be of the `float` type to accurately store the answer to a calculation like that.

Perhaps unsurprisingly, we could use the multiplication operator like this:

```
// answer is equal to 100, of course
answer = 10 * 10;
```

It can also be used like this:

```
// biggerAnswer = 1000, of course
biggerAnswer = 10 * 10 * 10;
```



As a side note, have you ever wondered how C++ got its name? C++ is an extension of the C language. Its inventor, **Bjarne Stroustrup**, originally called it "C with classes", but the name evolved. If you are interested, you can read the story of C++ at <http://www.cplusplus.com/info/history/>.

Now, let's look at the increment operator in action. This is a neat way to add 1 to the value of one of our game's variables.

Take a look at the following code:

```
// Add one to myVariable
myVariable = myVariable + 1;
```

The preceding code gives the same result as the following code:

```
// Much neater and quicker
myVariable ++;
```

The decrement operator, --, is, you guessed it, a quick way to subtract 1 from something, like so:

```
playerHealth = playerHealth -1;
```

This is the same as doing the following:

```
playerHealth --;
```

Let's look at a few more operators in action and then we can get back to building the Timber!!! game. The addition, subtraction, multiplication, and division operators each have a related operator that combines their primary function (adding, subtracting, and so on) with assignment. They allow us to use more concise code when we want to perform the primary function of the operator, followed by assignment. Have a look at the four examples (one for each operator) that follow:

```
someVariable = 10;

// Multiply the variable by 10 and put the answer
// back in the variable
someVariable *= 10;
// someVariable now equals 100

// Divide someVariable by 5 put the answer back
// into the variable
someVariable /= 5;
// someVariable now equals 20

// Add 3 to someVariable and put the answer back
// into the variable
someVariable += 3;
// someVariable now equals 23

// Take 25 from someVariable and put the answer back
```

```
// into the variable
someVariable -= 25;
// someVariable now equals -2
```

In the preceding four examples, we can see that the `*=`, `/=`, `+=`, and `-=` operators can be used to shorten the syntax when we want to use one of the four arithmetic operators followed by an assignment. We will do this quite a bit throughout this book.

It's time to add some more sprites to our game.

Adding clouds, a tree, and a buzzing bee

In this section, we will add clouds, a tree, and a buzzing bee to our Timber!!! game. First, we will add a tree. This is going to be easy. The reason for this is because the tree doesn't move. We will use the same procedure that we used in the previous chapter when we drew the background. The bee and the clouds will also be easy to draw in their starting positions, but we will need to combine what we have just learned about manipulating variables with some new C++ topics to make them move.

Preparing the tree

Let's get ready to draw the tree! Add the following highlighted code. Notice the unhighlighted code, which is the code we have already written. This should help you to identify that the new code should be typed immediately after we set the position of the background but before the start of the main game loop. We will provide a recap regarding what is going on in the new code after we have added it:

```
int main()
{

    // Create a video mode object
    VideoMode vm(1920, 1080);

    // Create and open a window for the game

    RenderWindow window(vm, "Timber!!!", Style::Fullscreen);

    // Create a texture to hold a graphic on the GPU
    Texture textureBackground;

    // Load a graphic into the texture
    textureBackground.loadFromFile("graphics/background.png");

    // Create a sprite
```

```
Sprite spriteBackground;

// Attach the texture to the sprite
spriteBackground.setTexture(textureBackground);

// Set the spriteBackground to cover the screen
spriteBackground.setPosition(0, 0);

// Make a tree sprite
Texture textureTree;
textureTree.loadFromFile("graphics/tree.png");
Sprite spriteTree;
spriteTree.setTexture(textureTree);
spriteTree.setPosition(810, 0);

while (window.isOpen())
{
```

This is what the following five lines of code (excluding the comment) do:

1. First, we create an object of the `Texture` type called `textureTree`.
2. Next, we load a graphic into the texture from the `tree.png` graphics file.
3. Then, we declare an object of the `Sprite` type called `spriteTree`
4. After, we associate `textureTree` with `spriteTree`. Whenever we draw `spriteTree`, it will show the `textureTree` texture, which is a neat tree graphic.
5. Finally, we set the position of the tree using the coordinates `810` on the *x* axis and `0` on the *y* axis.

The tree sprite is ready to draw, along with the tree texture. Let's move on to the bee object, which is handled in an almost identical manner.

Preparing the bee

Preparing the bee sprite is very similar but not identical to preparing the tree sprite. The difference between the following code and the tree code is small but important. Since the bee needs to move, we also declare two bee-related variables. Add the following highlighted code and see whether you can work out how we might use the `beeActive` and `beeSpeed` variables:

```
// Make a tree sprite
Texture textureTree;
textureTree.loadFromFile("graphics/tree.png");
```



```
Sprite spriteTree;
spriteTree.setTexture(textureTree);
spriteTree.setPosition(810, 0);

// Prepare the bee
Texture textureBee;
textureBee.loadFromFile("graphics/bee.png");
Sprite spriteBee;
spriteBee.setTexture(textureBee);
spriteBee.setPosition(0, 800);

// Is the bee currently moving?
bool beeActive = false;

// How fast can the bee fly
float beeSpeed = 0.0f;

while (window.isOpen())
{
```

We create a bee in the same way we created a background and a tree. We use a `Texture`, a `Sprite`, and associate the two. Note that, in the previous bee code, there's some new code we haven't seen before. There is a `bool` variable for determining whether the bee is active. Remember that a `bool` variable can be either `true` or `false`. We initialize `beeActive` to `false` for now.

Next, we declare a new `float` variable called `beeSpeed`. This will hold the speed in pixels per second that our bee will fly across the screen at.

Soon, we will see how we use these two new variables to move the bee. Before we do, let's set up some clouds in an almost identical manner.

Preparing the clouds

Add the following highlighted code. Study the new code and try and work out what it will do:

```
// Prepare the bee
Texture textureBee;
textureBee.loadFromFile("graphics/bee.png");
Sprite spriteBee;
spriteBee.setTexture(textureBee);
spriteBee.setPosition(0, 800);

// Is the bee currently moving?
```

```
bool beeActive = false;

// How fast can the bee fly
float beeSpeed = 0.0f;

// make 3 cloud sprites from 1 texture
Texture textureCloud;

// Load 1 new texture
textureCloud.loadFromFile("graphics/cloud.png");

// 3 New sprites with the same texture
Sprite spriteCloud1;
Sprite spriteCloud2;
Sprite spriteCloud3;
spriteCloud1.setTexture(textureCloud);
spriteCloud2.setTexture(textureCloud);
spriteCloud3.setTexture(textureCloud);

// Position the clouds on the left of the screen
// at different heights
spriteCloud1.setPosition(0, 0);
spriteCloud2.setPosition(0, 250);
spriteCloud3.setPosition(0, 500);

// Are the clouds currently on screen?
bool cloud1Active = false;
bool cloud2Active = false;
bool cloud3Active = false;

// How fast is each cloud?
float cloud1Speed = 0.0f;
float cloud2Speed = 0.0f;
float cloud3Speed = 0.0f;

while (window.isOpen())
{
```

The only thing about the code we have just added that might seem a little odd is that we have only one object of the `Texture` type. It is completely normal for multiple `Sprite` objects to share a texture. Once a `Texture` is stored in GPU memory, it can be associated with a `Sprite` object very quickly. It is only the initial loading of the graphic in the `loadFromFile` code that is a relatively slow operation. Of course, if we wanted three different shaped clouds, then we would need three textures.

Apart from the minor texture issue, the code we have just added is nothing new compared to the bee. The only difference is that there are three cloud sprites, three `bool` variables to determine whether each cloud is active, and three `float` variables to hold the speed for each cloud.

At this stage, all of the sprites and variables have been prepared. We can now move on to drawing them.

Drawing the tree, the bee, and the clouds

Finally, we can draw them all to the screen by adding the following highlighted code in the drawing section:

```
/*
*****
Draw the scene
*****
*/

// Clear everything from the last run frame
window.clear();

// Draw our game scene here
window.draw(spriteBackground);

// Draw the clouds
window.draw(spriteCloud1);
window.draw(spriteCloud2);
window.draw(spriteCloud3);

// Draw the tree
window.draw(spriteTree);

// Draw the insect
window.draw(spriteBee);

// Show everything we just drew
window.display();
```

Drawing the three clouds, the bee, and the tree is done in the same way that the background was drawn. Note, however, the order in which we draw the different objects to the screen. We must draw all of the graphics after the background, or they will be covered, and we must draw the clouds before the tree, or they will look a bit odd drifting in front of the tree. The bee would look OK either in front or behind the tree. I opted to draw the bee in front of the tree so that it can try and distract our lumberjack, a bit like a real bee might.

Run Timber!!! and gaze in awe at the tree, three clouds, and a bee that... don't do anything! They look like they are lining up for a race; a race where the bee has to go backward:



Using what we know about operators, we could try and move the graphics we have just added, but there's a problem. The problem is that real clouds and bees move in a non-uniform manner. They don't have a set speed or location, with these elements determined by factors such as wind speed or how much of a hurry the bee might be in. To the casual observer, the path they take and their speed appear to be *random*.

Random numbers

Random numbers are useful for lots of reasons in games – perhaps determining what card the player is dealt or how much damage within a certain range is subtracted from an enemy's health. We will now learn how to generate random numbers to determine the starting location and speed of the bee and the clouds.

Generating random numbers in C++

To generate random numbers, we will need to use some more C++ functions – two more, to be precise. Don't add any code to the game yet. Let's just look at the syntax and the steps that are required with some hypothetical code.

Computers can't genuinely pick random numbers. They can only use **algorithms/calculations** to pick a number that *appears* to be random. So that this algorithm doesn't constantly return the same value, we must **seed** the random number generator. The seed can be any integer number, although it must be a different seed each time you require a unique random number. Look at the following code, which seeds the random number generator:

```
// Seed the random number generator with the time
srand((int)time(0));
```

The preceding code gets the time from the PC using the `time` function, that is, `time(0)`. The call to the `time` function is enclosed as the value to be sent to the `srand` function. The result of this is that the current time is used as the seed.

The previous code is made to look a little more complicated because of the slightly unusual looking `(int)` syntax. What this does is convert/cast the value that's returned from `time` into an `int`. This is required by the `srand` function in this situation.



The term that's used to describe a conversion from one type to another is **cast**.

So, in summary, the previous line of code does the following:

- Gets the time using `time`
- Converts it into an `int`
- Sends this resulting value to `srand`, which seeds the random number generator

The time is, of course, always changing. This makes the `time` function a great way to seed the random number generator. However, think about what might happen if we seed the random number generator more than once and in such quick succession that `time` returns the same value. We will see and solve this problem when we animate our clouds.

At this stage, we can create the random number, between a range, and save it to a variable for later use, like so:

```
// Get the random number & save it to a variable called number
int number = (rand() % 100);
```

Notice the odd-looking way we assign a value to `number`. By using the Modulo operator (%) and the value of 100, we are asking for the remainder, after dividing the number returned from `rand`, by 100. When you divide by 100, the highest number you can possibly have as a remainder is 99. The lowest number possible is 0. Therefore, the previous code will generate a number between 0 and 99 inclusive. This knowledge will be useful for generating a random speed and starting location for our bees and clouds.

But before we can implement our random bees and clouds, we will need to learn how to make decisions in C++.

Making decisions with `if` and `else`

The C++ `if` and `else` keywords are what allow us to make decisions. We have already seen `if` in action in the previous chapter when we detected whether the player had pressed the *Esc* key each frame:

```
if (Keyboard::isKeyPressed(Keyboard::Escape))
{
    window.close();
}
```

So far, we have seen how we can use arithmetic and assignment operators to create expressions. Now, we will look at some new operators.

Logical operators

Logical operators are going to help us to make decisions by building expressions that can be tested for a value of either true or false. At first, this might seem like quite a narrow choice and insufficient for the kind of choices that might be needed in an advanced PC game. Once we dig a little deeper, we will see that we can make all of the required decisions we will need with just a few of the logical operators.

Here is a table of the most useful logical operators. Look at them and the associated examples, and then we will see how we can put them to use:

| Logical operator | Name and example |
|------------------|--|
| == | The comparison operator tests for equality and is either true or false. An expression such as <code>(10 == 9)</code> , for example, is false. 10 is obviously not equal to 9. |
| ! | This is the logical NOT operator . The <code>(! (2 + 2 == 5))</code> expression is true because <code>2 + 2</code> is NOT 5. |
| != | This is another comparison operator but is different to the <code>==</code> comparison operator. This tests if something is NOT equal. For example, the <code>(10 != 9)</code> expression is true. 10 is not equal to 9. |
| > | This is another comparison operator – there are a few more as well. This tests whether something is greater than something else. The <code>(10 > 9)</code> expression is true. |
| < | You guessed it. This tests for values that are less than. The <code>(10 < 9)</code> expression is false. |
| >= | This operator tests for whether one value is greater than or equal to the other and if either is true, the result is true. For example, the <code>(10 >= 9)</code> expression is true. The <code>(10 >= 10)</code> expression is also true. |
| <= | Like the previous operator, this one tests for two conditions, but this time less than or equal to. The <code>(10 <= 9)</code> expression is false. The <code>(10 <= 10)</code> expression is true. |
| && | This operator is known as logical AND . It tests two or more separate parts of an expression and both parts must be true for the result to be true. Logical AND is usually used in conjunction with the other operators to build more complex tests. The <code>((10 > 9) && (10 < 11))</code> expression is true because both parts are true, so the expression is true. The <code>((10 > 9) && (10 < 9))</code> expression is false because only one part of the expression is true and the other is false. |
| | This operator is called logical OR and it is just like logical AND except that at least one of two or more parts of an expression need to be true for the expression to be true. Let's look at the previous example we used but switch <code>&&</code> for <code> </code> . The <code>((10 > 9) (10 < 9))</code> expression is now true because one part of the expression is true. |

Let's take a look at the C++ `if` and `else` keywords, which will allow us to put all of these logical operators to good use.

C++ if and else

Let's make the previous examples less abstract. Meet the C++ **if** keyword. We will use **if** and a few operators along with a small story to demonstrate their use. Next follows a made-up military situation that will hopefully be less abstract than the previous examples.

If they come over the bridge, shoot them!

The captain is dying and, knowing that his remaining subordinates are not very experienced, he decides to write a C++ program to convey his last orders for after he has died. The troops must hold one side of a bridge while waiting for reinforcements.

The first command the captain wants to make sure his troops understand is this:

"If they come over the bridge, shoot them!"

So, how do we simulate this situation in C++? We need a `bool` variable, `isComingOverBridge`. The following bit of code assumes that the `isComingOverBridge` variable has been declared and initialized to either `true` or `false`.

We can then use **if** like this:

```
if (isComingOverBridge)
{
    // Shoot them
}
```

If the `isComingOverBridge` variable is equal to `true`, the code inside the opening and closing curly braces `{ . . }` will run. If not, the program continues after the **if** block and without running the code within it.

Shoot them ... or else do this instead

The captain also wants to tell his troops to stay put if the enemy is not coming over the bridge.

Now, we can introduce another C++ keyword, **else**. When we want to explicitly do something when the **if** does **not** evaluate to `true`, we can use **else**.

For example, to tell the troops to stay put if the enemy is not coming over the bridge, we could write the following code:

```
if(isComingOverBridge)
{
    // Shoot them
}

else
{
    // Hold position
}
```

The captain then realized that the problem wasn't as simple as he first thought. What if the enemy comes over the bridge, but has too many troops? His squad would be overrun and slaughtered. So, he came up with the following code (we'll use some variables as well this time):

```
bool isComingOverBridge;
int enemyTroops;
int friendlyTroops;

// Initialize the previous variables, one way or another

// Now the if
if(isComingOverBridge && friendlyTroops > enemyTroops)
{
    // shoot them
}

else if(isComingOverBridge && friendlyTroops < enemyTroops)
{
    // blow the bridge
}

else
{
    // Hold position
}
```

The preceding code has three possible paths of execution. First, if the enemy is coming over the bridge and the friendly troops are greater in number:

```
if(isComingOverBridge && friendlyTroops > enemyTroops)
```

The second occurs if the enemy troops are coming over the bridge but outnumber the friendly troops:

```
else if (isComingOveBridge && friendlyTroops < enemyTroops)
```

Then, the third and final possible outcome, which will execute if neither of the others is `true`, is captured by the final `else`, without an `if` condition.

Reader challenge

Can you spot a flaw with the preceding code? One that might leave a bunch of inexperienced troops in complete disarray? The possibility of the enemy troops and friendly troops being exactly equal in number has not been handled explicitly and would therefore be handled by the final `else`. The final `else` is meant for when there are no enemy troops. I guess any self-respecting captain would expect his troops to fight in this situation. He could change the first `if` statement to accommodate this possibility, like so:

```
if (isComingOverBridge && friendlyTroops >= enemyTroops)
```

Finally, the captain's last concern was that if the enemy came over the bridge waving the white flag of surrender and were promptly slaughtered, then his men would end up as war criminals. The C++ code that was needed was obvious. Using the `wavingWhiteFlag` Boolean variable, he wrote this test:

```
if (wavingWhiteFlag)
{
    // Take prisoners
}
```

But where to put this code was less clear. In the end, the captain opted for the following nested solution and changed the test for `wavingWhiteFlag` to logical NOT, like this:

```
if (!wavingWhiteFlag)
{
    // not surrendering so check everything else
    if (isComingOverTheBridge && friendlyTroops >= enemyTroops)
    {
        // shoot them
    }

    else if (isComingOverTheBridge && friendlyTroops < enemyTroops)
    {
```

```
        // blow the bridge
    }

}

else
{
    // this is the else for our first if
    // Take prisoners
}

// Holding position
```

This demonstrates that we can nest `if` and `else` statements inside one another to create quite deep and detailed decisions.

We could go on making more and more complicated decisions with `if` and `else` but what we have seen is more than enough as an introduction. It is probably worth pointing out, that often, there is more than one way to arrive at a solution to a problem. The *right* way will usually be the way that solves the problem in the clearest and simplest manner.

We are getting closer to having all of the C++ knowledge we need to be able to animate our clouds and bee. We have one final animation issue to discuss and then we can get back to the game.

Timing

Before we can move the bee and the clouds, we need to consider timing. As we already know, the main game loop executes repeatedly until the player presses the *Escape* key.

We have also learned that C++ and SFML are exceptionally fast. In fact, my aging laptop executes a simple game loop (like the current one) at around five thousand times per second.

The frame rate problem

Let's consider the speed of the bee. For the purpose of discussion, we could pretend that we are going to move it at 200 pixels per second. On a screen that is 1,920 pixels wide, it would take, very approximately, 10 seconds to cross the entire width, because 10 x 200 is 2,000 (near enough to 1,920).

Furthermore, we know that we can position any of our sprites with `setPosition(..., ...)`. We just need to put the x and the y coordinates in the parentheses.

In addition to setting the position of a sprite, we can also get the current position of a sprite. To get the horizontal x coordinate of the bee for example, we would use the following code:

```
int currentPosition = spriteBee.getPosition().x;
```

The current x coordinate of the bee is now stored in `currentPosition`. To move the bee to the right, we need to add the appropriate fraction of 200 (our intended speed) divided by 5,000 (the approximate frames per second on my laptop) to `currentPosition`, like this:

```
currentPosition += 200/5000;
```

Now, we would use `setPosition` to move our bee. It would smoothly move from left to right by 200 divided by 5,000 pixels each frame. But there are two big problems with this approach.

Frame rate is the number of times per second that our game loop is processed. That is, the number of times that we handle the players input, update the game objects, and draw them to the screen. We will expand on and discuss matters of frame rate now and throughout the rest of this book.

The frame rate on my laptop might not always be constant. The bee might look like it is intermittently "boosting" its way across the screen.

And of course, we want a wider audience for our game than just my laptop! Every PC's frame rate will vary, at least slightly. If you have an old PC, the bee will appear to be weighed down with lead, and if you have the latest gaming rig, it will probably be something of a blurry turbo bee.

Fortunately, this problem is the same for every game, and SFML has provided a solution. The easiest way to understand this solution is to implement it.

The SFML frame rate solution

We will now measure and use the frame rate to control our game. To get started with implementing this, add the following code just before the main game loop:

```
// How fast is each cloud?
float cloud1Speed = 0;
float cloud2Speed = 0;
```

```
float cloud3Speed = 0;

// Variables to control time itself
Clock clock;

while (window.isOpen())
{
```

In the previous code, we declare an object of the `Clock` type and we name it `clock`. The class name starts with a capital letter and the object name (which we will actually use) starts with a lowercase letter. The object name is arbitrary, but `clock` seems like an appropriate name for, well, a clock. We will add some more time-related variables here soon as well.

Now, in the update section of our game code, add the following highlighted code:

```
/*
*****
Update the scene
*****
*/

// Measure time
Time dt = clock.restart();

/*
*****
Draw the scene
*****
*/
```

The `clock.restart()` function, as you might expect, restarts the clock. We want to restart the clock every frame so that we can time how long each and every frame takes. In addition, however, it returns the amount of time that has elapsed since the last time we restarted the clock.

As a result of this, in the previous code, we are declaring an object of the `Time` type called `dt` and using it to store the value returned by the `clock.restart()` function.

Now, we have a `Time` object called `dt` that holds the amount of time that elapsed since the last time we updated the scene and restarted the clock. Maybe you can see where this is going? We will be using the elapsed time each frame to control how far we move the bee and the clouds.

Let's add some more code to the game and use everything we have learned so far about manipulating variables, generating random numbers, the `if` keyword, and the `else` keyword. Then, we will see how we can overcome the framerate problem with a `Clock` object and `dt`.



`dt` stands for **delta time**, which is the time between two updates.

Moving the clouds and the bee

Let's use the elapsed time since the last frame to breathe life into the bee and the clouds. This will solve the problem of having a consistent frame rate across different PCs.

Giving life to the bee

The first thing we want to do is set up the bee at a certain height and a certain speed. We only want to do this when the bee is inactive. Due to this, we will wrap the following code in an `if` block. Examine and add the following highlighted code, and then we will discuss it:

```
/*
*****
Update the scene
*****
*/

// Measure time
Time dt = clock.restart();

// Setup the bee
if (!beeActive)
{
    // How fast is the bee
    srand((int)time(0));
    beeSpeed = (rand() % 200) + 200;

    // How high is the bee
    srand((int)time(0) * 10);
    float height = (rand() % 500) + 500;
    spriteBee.setPosition(2000, height);
}
```

```
        beeActive = true;

    }

    /*
    *****
    Draw the scene
    *****
    */
```

Now, if the bee is not active, just like it won't be when the game first starts, `if(!beeActive)` will be true and the preceding code will do the following things, in this order:

- Seed the random number generator.
- Get a random number between 200 and 399 and assign the result to `beeSpeed`.
- Seed the random number generator again.
- Get a random number between 500 and 999 and assign the result to a new float variable called `height`.
- Set the position of the bee to 2000 on the x axis (just off-screen to the right) and to whatever `height` equals on the y axis.
- Set `beeActive` to true.



Note that the `height` variable is the first variable we have ever declared inside the game loop. Furthermore, because it was declared inside an `if` block, it is actually "invisible" outside of the `if` block. This is fine for our use because once we have set the height of the bee, we don't need it anymore. This phenomenon that affects variables is called **scope**. We will explore this more fully in *Chapter 4, Loops, Arrays, Switches, Enumerations, and Functions – Implementing Game Mechanics*.

If we run the game, nothing will happen to the bee yet, but now the bee is active, we can write some code that runs when `beeActive` is true.

Add the following highlighted code which, as you can see, executes whenever `beeActive` is true. This is because it follows with `else` after the `if(!beeActive)` block:

```
// Set up the bee
if (!beeActive)
{

    // How fast is the bee
```

```

        srand((int)time(0) );
        beeSpeed = (rand() % 200) + 200;

        // How high is the bee
        srand((int)time(0) * 10);
        float height = (rand() % 1350) + 500;
        spriteBee.setPosition(2000, height);
        beeActive = true;
    }
    else
    // Move the bee
    {

        spriteBee.setPosition(
            spriteBee.getPosition().x -
                (beeSpeed * dt.asSeconds()),
            spriteBee.getPosition().y);

        // Has the bee reached the left-hand edge of the screen?
        if (spriteBee.getPosition().x < -100)
        {
            // Set it up ready to be a whole new bee next frame
            beeActive = false;
        }
    }

    /*
    *****
    Draw the scene
    *****
    */

```

In the else block, the following things happen.

The bee's position is changed using the following criteria. The `setPosition` function uses the `getPosition` function to get the current horizontal coordinate of the bee. It then subtracts `beeSpeed * dt.asSeconds()` from that coordinate.

The `beeSpeed` variable value is many pixels per second and was randomly assigned in the previous `if` block. The value of `dt.asSeconds()` will be a fraction of 1 that represents how long the previous frame of animation took.

Let's assume that the bee's current horizontal coordinate is **1000**. Now, suppose a basic PC loops at 5,000 frames per second. This would mean that `dt.asSeconds` would be **0.0002**. Now, let's also suppose that `beeSpeed` was set to the maximum **399** pixels per second. With this information, we can say that the code that determines the value that `setPosition` uses for the horizontal coordinate is as follows:

```
1000 - 0.0002 x 399
```

Therefore, the new position on the horizontal axis for the bee would be 999.9202. We can see that the bee is very, very smoothly drifting to the left, at well under a pixel per frame. If the frame rate fluctuates, then the formula will produce a new value to suit. If we run the same code on a PC that only achieves 100 frames per second or a PC that achieves a million frames per second, the bee will move at the same speed.

The `setPosition` function uses `getPosition().y` to keep the bee in exactly the same vertical coordinate throughout this cycle of being active.

The final part of the code in the `else` block we just added is as follows:

```
// Has the bee reached the right hand edge of the screen?
if (spriteBee.getPosition().x < -100)
{
    // Set it up ready to be a whole new bee next frame
    beeActive = false;
}
```

This code tests, in each and every frame (when `beeActive` is `true`), whether the bee has disappeared off of the left-hand side of the screen. If the `getPosition` function returns less than -100, it will certainly be out of view of the player. When this occurs, `beeActive` is set to `false` and, on the next frame, a "new" bee will be set flying at a new random height and a new random speed.

Try running the game and watch our bee dutifully fly from right to left and then come back to the right again at a new height and speed. It's almost like a new bee every time.



Of course, a real bee would stick around for ages and pester you while you're trying to concentrate on chopping the tree. We will make some smarter game characters in later projects.

Now, we will get the clouds moving in a very similar way.

Blowing the clouds

The first thing we want to do is set up the first cloud at a certain height and a certain speed. We only want to do this when the cloud is inactive. Consequently, we will wrap the code that follows in an `if` block. Examine and add the following highlighted code, just after the code we added for the bee, and then we will discuss it. It is almost identical to the code we used on the bee:

```

else
// Move the bee
{

    spriteBee.setPosition(
        spriteBee.getPosition().x -
            (beeSpeed * dt.asSeconds()),
        spriteBee.getPosition().y);

    // Has the bee reached the right hand edge of the screen?
    if (spriteBee.getPosition().x < -100)
    {
        // Set it up ready to be a whole new bee next frame
        beeActive = false;
    }
}

// Manage the clouds
// Cloud 1
if (!cloud1Active)
{
    // How fast is the cloud
    srand((int)time(0) * 10);
    cloud1Speed = (rand() % 200);

    // How high is the cloud
    srand((int)time(0) * 10);
    float height = (rand() % 150);
    spriteCloud1.setPosition(-200, height);
    cloud1Active = true;
}

/*
*****
Draw the scene
*****
*/

```

The only difference between the code we have just added and the bee-related code is that we work on a different sprite and use different ranges for our random numbers. Also, we multiply by ten (`* 10`) to the result returned by `time(0)` so that we are always guaranteed to get a different seed for each of the clouds. When we code the other cloud movement next, you will see that we use `* 20` and `* 30`, respectively.

Now, we can act when the cloud is active. We will do so in the `else` block. Like the `if` block, the code is identical to that of the bee-related code, except that all of the code works on the cloud and not the bee:

```
// Manage the clouds
if (!cloud1Active)
{
    // How fast is the cloud
    srand((int)time(0) * 10);
    cloud1Speed = (rand() % 200);

    // How high is the cloud
    srand((int)time(0) * 10);
    float height = (rand() % 150);
    spriteCloud1.setPosition(-200, height);
    cloud1Active = true;
}
else
{

    spriteCloud1.setPosition(
        spriteCloud1.getPosition().x +
        (cloud1Speed * dt.asSeconds()),
        spriteCloud1.getPosition().y);

    // Has the cloud reached the right hand edge of the screen?
    if (spriteCloud1.getPosition().x > 1920)
    {
        // Set it up ready to be a whole new cloud next frame
        cloud1Active = false;
    }
}

/*
*****
Draw the scene
*****
*/
```

Now that we know what to do, we can duplicate the same code for the second and third clouds. Add the following highlighted code, which handles the second and third clouds, immediately after the code for the first cloud:

```
...

// Cloud 2
if (!cloud2Active)
{

    // How fast is the cloud
    srand((int)time(0) * 20);
    cloud2Speed = (rand() % 200);

    // How high is the cloud
    srand((int)time(0) * 20);
    float height = (rand() % 300) - 150;
    spriteCloud2.setPosition(-200, height);
    cloud2Active = true;

}
else
{

    spriteCloud2.setPosition(
        spriteCloud2.getPosition().x +
        (cloud2Speed * dt.asSeconds()),
        spriteCloud2.getPosition().y);

    // Has the cloud reached the right hand edge of the screen?
    if (spriteCloud2.getPosition().x > 1920)
    {
        // Set it up ready to be a whole new cloud next frame
        cloud2Active = false;
    }
}

if (!cloud3Active)
{
    // How fast is the cloud
    srand((int)time(0) * 30);
    cloud3Speed = (rand() % 200);
```

```
// How high is the cloud
srand((int)time(0) * 30);
float height = (rand() % 450) - 150;
spriteCloud3.setPosition(-200, height);
cloud3Active = true;
}
else
{

    spriteCloud3.setPosition(
        spriteCloud3.getPosition().x +
        (cloud3Speed * dt.asSeconds()),
        spriteCloud3.getPosition().y);

    // Has the cloud reached the right hand edge of the screen?
    if (spriteCloud3.getPosition().x > 1920)
    {
        // Set it up ready to be a whole new cloud next frame
        cloud3Active = false;
    }
}

/*
*****
Draw the scene
*****
*/
```

Now, you can run the game and the clouds will randomly and continuously drift across the screen. The bee will also buzz from right to left before respawning once more back on the right. The following screenshot shows what we've achieved in this chapter:



Does all of this cloud and bee handling seem a little bit repetitious? We will see how we could save lots of typing and make our code more readable as, in C++, there are ways of handling multiple instances of the same type of variable or object. One such way is called **arrays**, and we will learn about them in *Chapter 4, Loops, Arrays, Switches, Enumerations, and Functions – Implementing Game Mechanics*. At the end of this project, once we have learned about arrays, we will discuss how we can improve our cloud code.

Take a look at a few frequently asked questions related to the topics in this chapter.

Summary

In this chapter, we learned that a variable is a named storage location in memory in which we can keep values of a specific type. The types include `int`, `float`, `double`, `bool`, `String`, and `char`.

We can declare and initialize all of the variables we need to store the data for our game. Once we have our variables, we can manipulate them using the arithmetic and assignment operators, as well as use them in tests with the logical operators. Used in conjunction with the `if` and `else` keywords, we can branch our code depending on the current situation in the game.

Using all of this new knowledge, we animated some clouds and a bee. In the next chapter, we will use these skills some more to add a **Heads Up Display (HUD)** and add more input options for the player, as well as represent time visually using a time-bar.

FAQ

Q) Why do we set the bee to inactive when it gets to -100? Why not just zero since zero is the left-hand side of the window?

A) The bee graphic is 60 pixels wide and its origin is at the top left pixel. As a result, when the bee is drawn with its origin at x equals zero, the entire bee graphic is still on screen for the player to see. By waiting until it is at -100, we can be sure it is out of the player's view.

Q) How do I know how fast my game loop is?

A) If you have a modern NVIDIA graphics card, you might be able to already by configuring your GeForce Experience overlay to show the frame rate. To measure this explicitly using our own code, however, we will need to learn a few more things. We will add the ability to measure and display the current frame rate in *Chapter 5, Collisions, Sound, and End Conditions – Making the Game Playable*.

3

C++ Strings and SFML Time – Player Input and HUD

In this chapter, we will continue with the Timber!! game. We will spend around half of this chapter learning how to manipulate text and display it on the screen, and the other half looking at timing and how a visual time-bar can inform the player about the remaining time and create a sense of urgency in the game.

We will cover the following topics:

- Pausing and restarting the game
- C++ Strings
- SFML text and SFML font classes
- Adding a HUD to Timber!!!
- Adding a time-bar to Timber!!!

Pausing and restarting the game

As we work on this game over the next three chapters, the code will obviously get longer and longer. So, now seems like a good time to think ahead and add a little bit more structure to our code. We will add this structure so that we can pause and restart the game.

We will add code so that, when the game is run for the first time, it will be in a paused state. The player will then be able to press the *Enter* key to start the game. Then, the game will run until either the player gets squished or runs out of time. At this point, the game will pause and wait for the player to press *Enter* so that they can restart the game.

Let's step through setting this up a bit at a time.

First, declare a new `bool` variable called `paused` outside the main game loop and initialize it to `true`:

```
// Variables to control time itself
Clock clock;

// Track whether the game is running
bool paused = true;

while (window.isOpen())
{

    /*
    *****
    Handle the players input
    *****
    */
```

Now, whenever the game is run, we have a `paused` variable that will be `true`.

Next, we will add another `if` statement where the expression will check whether the *Enter* key is currently being pressed. If it is being pressed, it sets `paused` to `false`. Add the following highlighted code just after our other keyboard-handling code:

```
/*
*****
Handle the players input
*****
*/

if (Keyboard::isKeyPressed(Keyboard::Escape))
{
    window.close();
}

// Start the game
if (Keyboard::isKeyPressed(Keyboard::Return))
{
    paused = false;
}
```

```

/*
*****
Update the scene
*****
*/

```

Now, we have a `bool` called `paused` that starts off `true` but changes to `false` when the player presses the *Enter* key. At this point, we must make our game loop respond appropriately, based on whatever the current value of `paused` might be.

This is how we will proceed. We will wrap the entire update part of the code, including the code we wrote in the last chapter for moving the bee and clouds, in an `if` statement.

In the following code, note that the `if` block will only execute when `paused` is not equal to `true`. Put another way, the game won't move/update when it is paused.

This is exactly what we want. Look carefully at the place where we added the new `if` statement and its corresponding opening and closing curly braces `{ ... }`. If they are put in the wrong place, things will not work as expected.

Add the following highlighted code to wrap the updated part of the code, paying close attention to the context that follows. I have added `...` on a few lines to represent hidden code. Obviously, `...` is not real code and should not be added to the game. You can identify where to place the new code (highlighted) at the start and the end by the unhighlighted code surrounding it:

```

/*
*****
Update the scene
*****
*/

if (!paused)
{

    // Measure time
        ...
    ...

    // Has the cloud reached the right hand edge of the screen?
    if (spriteCloud3.getPosition().x > 1920)
    {

```

```
        // Set it up ready to be a whole new cloud next frame
        cloud3Active = false;
    }

} // End if(!paused)

/*
*****
Draw the scene
*****
*/
```

Note that, when you place the closing curly brace of the new `if` block, Visual Studio neatly adjusts all the indenting to keep the code tidy.

Now, you can run the game and everything will be static until you press the *Enter* key. It is now possible to go about adding features to our game. We just need to remember that, when the player dies or runs out of time, we need to set `paused` to `true`.

In the previous chapter, we took our first look at C++ Strings. We need to learn a bit more about them so that we can implement the player's HUD.

C++ Strings

In the previous chapter, we briefly mentioned Strings and we learned that a String can hold alphanumeric data – anything from a single character to a whole book. We didn't look at declaring, initializing, or manipulating Strings, so let's do that now.

Declaring Strings

Declaring a String variable is simple. It is the same process that we used for other variables in the previous chapter: we state the type, followed by the name:

```
String levelName;
String playerName;
```

Once we have declared a String, we can assign a value to it.

Assigning a value to a String

To assign a value to a String, just like regular variables, we simply put the name, followed by the assignment operator, and then the value:

```
levelName = "DastardlyCave";
playerName = "John Carmack";
```

Note that the values need to be enclosed in quotation marks. Just like regular variables, we can also declare and assign values in a single line:

```
String score = "Score = 0";
String message = "GAME OVER!!";
```

In the next section, we will see how we can change the values of our String variables.

Manipulating Strings

We can use the `#include <sstream>` directive to give us some extra manipulation options for our Strings. The `sstream` class allows us to "add" some Strings together. When we add Strings together, this is known as **concatenation**:

```
String part1 = "Hello ";
String part2 = "World";

sstream ss;
ss<< part1 << part2;

// ss now holds "Hello World"
```

In addition to this, by using `sstream` objects a String variable can even be concatenated with a variable of a different type. The following code starts to reveal how Strings might be useful to us:

```
String scoreText = "Score = ";
int score = 0;

// Later in the code
score ++;

sstream ss;
ss<<scoreText<< score;
// ss now holds "Score = 1"
```

In the preceding code, `ss` is used to join the content of `scoreText` with the value from `score`. Note that although `score` holds an `int` value, the final value held by `ss` is still a `String` that holds an equivalent value; in this case, "1".



The `<<` operator is one of the bitwise operators. C++, however, allows you to write your own classes and override what a specific operator does within the context of your class. The `sstream` class has done this to make the `<<` operator work the way it does. The complexity is hidden in the class. We can use its functionality without worrying about how it works. If you are feeling adventurous, you can read about operator overloading at http://www.tutorialspoint.com/cplusplus/cpp_overloading.htm. You don't need any more information in order to continue with the project.

Now that we know the basics of C++ Strings and how we can use `sstream`, we will look at how we can use some SFML classes to display them on the screen.

SFML's Text and Font classes

Let's talk about the `Text` and `Font` classes using some hypothetical code before we go ahead and add the code to our game.

The first step in being able to draw text on the screen is to have a font. In *Chapter 1, C++, SFML, Visual Studio, and Starting the First Game*, we added a font file to the project folder. Now, we can load the font into an SFML `Font` object, so that it's ready to use.

The code to do so looks like the following:

```
Font font;  
font.loadFromFile("myfont.ttf");
```

In the preceding code, we first declare the `Font` object and then load an actual font file. Note that `myfont.ttf` is a hypothetical font and that we could use any font in the project folder.

Once we have loaded a font, we need an SFML `Text` object:

```
Text myText;
```

Now, we can configure our `Text` object. This includes the size, the color, the position on-screen, the `String` that holds the message, and of course the act of associating it with our font object:

```
// Assign the actual message
myText.setString("Press Enter to start!");

// assign a size
myText.setCharacterSize(75);

// Choose a color
myText.setFillColor(Color::White);

// Set the font to our Text object
myText.setFont(font);
```

Now that we can create and manipulate `String` values as well as assign, declare, and initialize SFML `Text` objects, we can move on to the next section, where we will add a HUD to Timber!!!

Implementing the HUD

Now, we know enough about `Strings`, SFML `Text`, and SFML `Font` to go about implementing the HUD. **HUD** stands for **Heads Up Display**. It can be as simple as the score and text messages on the screen or it can include more complex elements such as a time-bar, mini-map, or compass that represents the direction that the player character is facing.

To get started with the HUD, we need to add another `#include` directive to the top of the code file to add access to the `sstream` class. As we already know, the `sstream` class adds some really useful functionality for combining `Strings` and other variable types into a `String`.

Add the following line of highlighted code:

```
#include <sstream>
#include <SFML/Graphics.hpp>

using namespace sf;

int main()
{
```

Next, we will set up our SFML `Text` objects: one to hold a message that we will vary to suit the state of the game and one that will hold the score and will need to be regularly updated.

The code declares the `Text` and `Font` objects, loads the font, assigns the font to the `Text` objects, and then adds the `String` messages, color, and size. This should look familiar from our discussion in the previous section. In addition, we added a new `int` variable called `score` that we can manipulate so that it holds the player's score.



Remember that, if you chose a different font from `KOMIKAP_.ttf`, back in *Chapter 1, C++, SFML, Visual Studio, and Starting the First Game*, you will need to change that part of the code to match the `.ttf` file that you have in the `Visual Studio Stuff/Projects/Timber/fonts` folder.

By adding the following highlighted code, we will be ready to move on to updating the HUD:

```
// Track whether the game is running
bool paused = true;

// Draw some text
int score = 0;

Text messageText;
Text scoreText;

// We need to choose a font
Font font;
font.loadFromFile("fonts/KOMIKAP_.ttf");

// Set the font to our message
messageText.setFont(font);
scoreText.setFont(font);

// Assign the actual message
messageText.setString("Press Enter to start!");
scoreText.setString("Score = 0");

// Make it really big
messageText.setCharacterSize(75);
scoreText.setCharacterSize(100);

// Choose a color
```

```

messageText.setFillColor(Color::White);
scoreText.setFillColor(Color::White);

while (window.isOpen())
{

    /*
    *****
    Handle the players input
    *****
    */

```

In the preceding code we have achieved the following:

- Declared a variable to hold the score
- Declared some SFML Text and Font objects
- Initialized the Font object by loading a font from a file
- Initialized the Text objects using the font and some Strings
- Set the size and color of the Text objects using the `setCharacterSize` and `setFillColor` functions

The following snippet of code might look a little convoluted, even complex. It is, however, straightforward when you break it down a bit. Examine and add the new highlighted code. We will go through it after this:

```

// Choose a color
messageText.setFillColor(Color::White);
scoreText.setFillColor(Color::White);

// Position the text
FloatRect textRect = messageText.getLocalBounds();

messageText.setOrigin(textRect.left +
    textRect.width / 2.0f,
    textRect.top +
    textRect.height / 2.0f);

messageText.setPosition(1920 / 2.0f, 1080 / 2.0f);

scoreText.setPosition(20, 20);

while (window.isOpen())
{

```



```
/*
*****
Handle the players input
*****
*/
```

We have two objects of the `Text` type that we will display on the screen. We want to position `scoreText` to the top left with a little bit of padding. This is not a challenge; we simply use `scoreText.setPosition(20, 20)`, which positions it at the top left with 20 pixels of horizontal and vertical padding.

Positioning `messageText`, however, is not so easy. We want to position it in the exact midpoint of the screen. Initially, this might not seem like a problem, but then we have to remember that the origin of everything we draw is at the top left-hand corner. So, if we simply divide the screen width and height by two and use the results in `messageText.setPosition(...)`, then the top left of the text will be in the center of the screen and it will spread out untidily to the right.

The following is the code under discussion again for convenience:

```
// Position the text
FloatRect textRect = messageText.getLocalBounds();

messageText.setOrigin(textRect.left +
    textRect.width / 2.0f,
    textRect.top +
    textRect.height / 2.0f);
```

What the code does is set the *center* of `messageText` to the center of the screen. The rather complex-looking bit of code that we are reviewing repositions the origin of `messageText` to the center of itself.

In the preceding code, we first declare a new object of the `FloatRect` type called `textRect`. A `FloatRect` object, as its name suggests, holds a rectangle with floating-point coordinates.

The code then uses the `messageText.getLocalBounds` function to initialize `textRect` with the coordinates of the rectangle that wraps `messageText`.

The next line of code, which is spread over four lines as it is quite long, uses the `messageText.setOrigin` function to change the origin (the point that is used to draw at) to the center of `textRect`. Of course, `textRect` holds a rectangle that matches the coordinates that wrap `messageText`. Then, this following line of code executes:

```
messageText.setPosition(1920 / 2.0f, 1080 / 2.0f);
```

Now, `messageText` will be neatly positioned in the exact center of the screen. We will use this code each time we change the text of `messageText` because changing the message changes the size of `messageText`, so its origin will need recalculating.

Next, we declare an object of the `stringstream` type called `ss`. Note that we use the full name, including the namespace, that is, `std::stringstream`. We could avoid this syntax by adding `using namespace std` to the top of our code file. We aren't going to here, though, because we use it infrequently. Take a look at the following code and add it to the game; then, we can go through it in more detail. Since we only want this code to execute when the game is not paused, be sure to add it with the other code, inside the `if(!paused)` block, as follows:

```
else
{

    spriteCloud3.setPosition(
        spriteCloud3.getPosition().x +
        (cloud3Speed * dt.asSeconds()),
        spriteCloud3.getPosition().y);

    // Has the cloud reached the right hand edge of the screen?
    if (spriteCloud3.getPosition().x > 1920)
    {
        // Set it up ready to be a whole new cloud next frame
        cloud3Active = false;
    }
}

// Update the score text
std::stringstream ss;
ss<< "Score = " << score;
scoreText.setString(ss.str());

} // End if(!paused)

/*
*****
Draw the scene
*****
*/
```

We use `ss` and the special functionality provided by the `<<` operator, which concatenates variables into a `stringstream`. Here, `ss << "Score = " << score` has the effect of creating a `String` with `"Score = "`. Whatever the value of `score` is, is concatenated together. For example, when the game first starts, `score` is equal to zero, so `ss` will hold the `"Score = 0"` value. If `score` ever changes, `ss` will adapt each frame.

The following line of code simply sets the `String` contained in `ss` to `scoreText`:

```
scoreText.setString(ss.str());
```

It is now ready to be drawn to the screen.

This following code draws both `Text` objects (`scoreText` and `messageText`), but the code that draws `messageText` is wrapped in an `if` statement. This `if` statement causes `messageText` to only be drawn when the game is paused.

Add the following highlighted code:

```
// Now draw the insect
window.draw(spriteBee);

// Draw the score
window.draw(scoreText);

if (paused)
{
    // Draw our message
    window.draw(messageText);
}

// Show everything we just drew
window.display();
```

We can now run the game and see our HUD being drawn on the screen. You will see the **Score = 0** and **PRESS ENTER TO START** messages. The latter will disappear when you press *Enter*:



If you want to see the score updating, add a temporary line of code, `score ++;`, anywhere in the `while (window.isOpen)` loop. If you add this temporary line, you will see the score go up fast, very fast!



If you added the temporary code, that is, `score ++;`, be sure to delete it before continuing.

Adding a time-bar

Since time is a crucial mechanism in the game, it is necessary to keep the player aware of it. They need to know if their allotted six seconds are about to run out. It will give them a sense of urgency as the end of the game draws near and a sense of accomplishment if they perform well enough to maintain or increase their remaining time.

Drawing the number of seconds remaining on the screen is not easy to read (when concentrating on the branches), nor is it a particularly interesting way to achieve the objective.

What we need is a time-bar. Our time-bar will be a simple red rectangle that's prominently displayed on the screen. It will start off nice and wide, but rapidly shrink as time runs out. When the player's remaining time reaches zero, the time-bar will be gone completely.

At the same time as adding the time-bar, we will add the necessary code to keep track of the player's remaining time, and respond when it runs out. Let's go through this step by step.

Find the `Clock clock;` declaration from earlier and add the highlighted code just after it, as follows:

```
// Variables to control time itself
Clock clock;

// Time bar
RectangleShape timeBar;
float timeBarStartWidth = 400;
float timeBarHeight = 80;
timeBar.setSize(Vector2f(timeBarStartWidth, timeBarHeight));
timeBar.setFillColor(Color::Red);
timeBar.setPosition((1920 / 2) - timeBarStartWidth / 2, 980);

Time gameTimeTotal;
float timeRemaining = 6.0f;
float timeBarWidthPerSecond = timeBarStartWidth / timeRemaining;

// Track whether the game is running
bool paused = true;
```

First, we declare an object of the `RectangleShape` type and call it `timeBar`. `RectangleShape` is an SFML class that is perfect for drawing simple rectangles.

Next, we will add a few float variables, `timeBarStartWidth` and `timeBarHeight`. We initialize them to 400 and 80, respectively. These variables will help us keep track of the size we need to draw `timeBar` at each frame.

Next, we set the size of `timeBar` using the `timeBar.setSize` function. We don't just pass in our two new float variables. First, we create a new object of the `Vector2f` type. What is different here, however, is that we don't give the new object a name. Instead, we simply initialize it with our two float variables and pass it straight in to the `setSize` function.



Vector2f is a class that holds two float variables. It also has some other functionality that will be introduced throughout this book.

After that, we color `timeBar` red by using the `setFillColor` function.

The last thing we do to `timeBar` in the previous code is set its position. The vertical coordinate is completely straightforward but the way we set the horizontal coordinate is slightly convoluted. Here is the calculation again:

```
(1920 / 2) - timeBarStartWidth / 2
```

First, the code divides 1920 by 2. Then, it divides `timeBarStartWidth` by 2. Finally, it subtracts the latter from the former.

The result makes `timeBar` sit neatly and centrally, in a horizontal fashion, on the screen.

The final three lines of code that we are talking about declare a new `Time` object called `gameTimeTotal`, a new float called `timeRemaining` that is initialized to 6, and a curious-sounding float named `timeBarWidthPerSecond`, which we will discuss next.

The `timeBarWidthPerSecond` variable is initialized with `timeBarStartWidth` divided by `timeRemaining`. The result is exactly the amount of pixels that `timeBar` needs to shrink by each second of the game. This will be useful when we resize `timeBar` in each frame.

Obviously, we need to reset the time remaining each time the player starts a new game. The logical place to do this is when the *Enter* key is pressed. We can also set score back to zero at the same time. Let's do that now by adding the following highlighted code

```
// Start the game
if (Keyboard::isKeyPressed(Keyboard::Return))
{
    paused = false;

    // Reset the time and the score
    score = 0;
    timeRemaining = 6;
}
```

Now, we must reduce each frame by the amount of time remaining and resize `timeBar` accordingly. Add the following highlighted code to the update section, as shown here:

```
/*
*****
Update the scene
*****
*/
if (!paused)
{
    // Measure time
    Time dt = clock.restart();

    // Subtract from the amount of time remaining
    timeRemaining -= dt.asSeconds();
    // size up the time bar
    timeBar.setSize(Vector2f(timeBarWidthPerSecond *
        timeRemaining, timeBarHeight));

    // Set up the bee
    if (!beeActive)
    {

        // How fast is the bee
        srand((int)time(0) * 10);
        beeSpeed = (rand() % 200) + 200;

        // How high is the bee
        srand((int)time(0) * 10);
        float height = (rand() % 1350) + 500;
        spriteBee.setPosition(2000, height);
        beeActive = true;

    }
    else
        // Move the bee
}
```

First, we subtracted the amount of time the player has left by however long the previous frame took to execute with the following code:

```
timeRemaining -= dt.asSeconds();
```

Then, we adjusted the size of `timeBar` with the following code:

```
timeBar.setSize(Vector2f(timeBarWidthPerSecond *
    timeRemaining, timeBarHeight));
```

The `x` value of `Vector2F` is initialized with `timeBarWidthPerSecond` when multiplied by `timeRemaining`. This produces exactly the right width, relative to how long the player has left. The height remains the same and `timeBarHeight` is used without any manipulation.

And of course, we must detect when the time has run out. For now, we will simply detect that time has run out, pause the game, and change the text of `messageText`. Later, we will do more work here. Add the following highlighted code right after the previous code we added. We will look at it in more detail later:

```
// Measure time
Time dt = clock.restart();

// Subtract from the amount of time remaining
timeRemaining -= dt.asSeconds();

// resize up the time bar
timeBar.setSize(Vector2f(timeBarWidthPerSecond *
    timeRemaining, timeBarHeight));

if (timeRemaining<= 0.0f) {

    // Pause the game
    paused = true;

    // Change the message shown to the player
    messageText.setString("Out of time!!");

    //Reposition the text based on its new size
    FloatRect textRect = messageText.getLocalBounds();
    messageText.setOrigin(textRect.left +
        textRect.width / 2.0f,
        textRect.top +
        textRect.height / 2.0f);

    messageText.setPosition(1920 / 2.0f, 1080 / 2.0f);

}

// Set up the bee
```



```
if (!beeActive)
{

    // How fast is the bee
    srand((int)time(0) * 10);
    beeSpeed = (rand() % 200) + 200;

    // How high is the bee
    srand((int)time(0) * 10);
    float height = (rand() % 1350) + 500;
    spriteBee.setPosition(2000, height);
    beeActive = true;

}
else
    // Move the bee
```

Let's step through the previous code:

1. First, we test whether time has run out with `if(timeRemaining<= 0.0f)`.
2. Then, we set `paused` to `true`, so this will be the last time the update part of our code is executed (until the player presses *Enter* again).
3. Then, we change the message of `messageText`, calculate its new center to be set as its origin, and position it in the center of the screen.

Finally, for this part of the code, we need to draw `timeBar`. There is nothing new in this code that we haven't seen many times before. Just note that we draw `timeBar` after the tree so that it is not partially obscured. Add the following highlighted code to draw the time-bar:

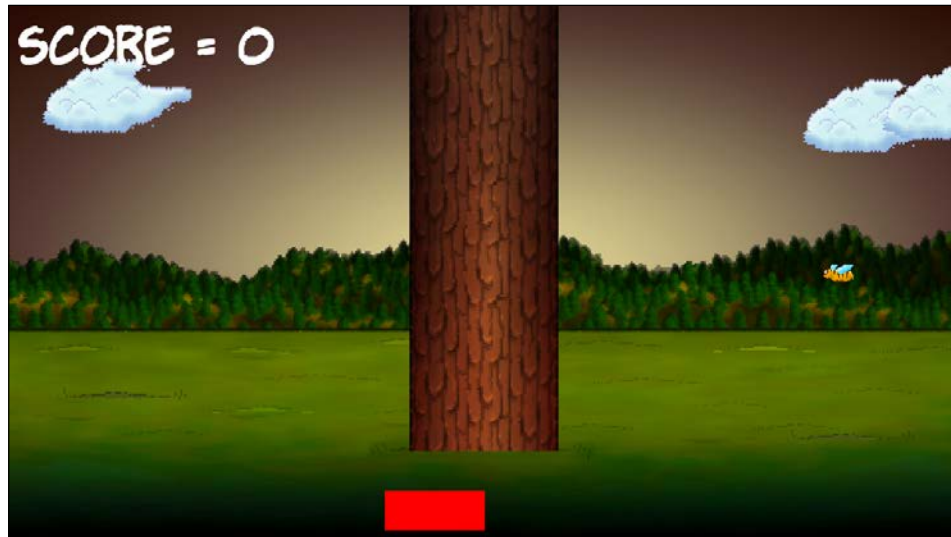
```
// Draw the score
window.draw(scoreText);

// Draw the timebar
window.draw(timeBar);

if (paused)
{
    // Draw our message
    window.draw(messageText);
}

// Show everything we just drew
window.display();
```

Now, you can run the game, press *Enter* to start it, and watch the time-bar smoothly disappear down to nothing:



The game then pauses and the **OUT OF TIME!!** message will appear in the center of the screen:



You can, of course, press *Enter* to start the game again and watch it run from the beginning.

Summary

In this chapter, we learned about `Strings`, `SFML Text`, and `SFML Font`. Between them, they allowed us to draw text to the screen, which provided the player with a HUD. We also used `sstream`, which allows us to concatenate `Strings` and other variables to display the score.

We also explored the `SFML RectangleShape` class, which does exactly what its name suggests. We used an object of the `RectangleShape` type and some carefully planned variables to draw a time-bar that visually displays to the player how much time they have left. Once we implement chopping and moving branches that can squash the player, the time-bar will provide visual feedback that will create tension and urgency.

In the next chapter, we are going to learn about a whole range of new C++ features, including loops, arrays, switching, enumerations, and functions. This will allow us to move tree branches, keep track of their locations, and squash the player.

FAQ

Q) I can foresee that positioning sprites by their top-left corner could sometimes be inconvenient. Is there an alternative?

A) Fortunately, you can choose what point of a sprite is used as the positioning/origin pixel, just like we did with `messageText`, using the `setOrigin` function.

Q) The code is getting rather long and I am struggling to keep track of where everything is. How can we fix this?

A) Yes, I agree. In the next chapter, we will look at the first of a few ways we can organize our code and make it more readable. We will look at this when we learn about writing C++ functions. In addition, we will learn about a new way we can handle multiple objects/variables of the same type (like the clouds) when we learn about C++ arrays.

4

Loops, Arrays, Switches, Enumerations, and Functions – Implementing Game Mechanics

Switches C++ information in it than any other chapter in this book. It is packed with fundamental concepts that will move our understanding on enormously. It will also begin to shed light on some of the murky areas we have been skipping over a little bit, such as functions and the game loop.

Once we have explored a whole list of C++ language necessities, we will then use everything we know to make the main game mechanic—the tree branches—move. By the end of this chapter, we will be ready for the final phase and the completion of Timber!!!.

In this chapter, we will cover the following topics:

- Loops
- Arrays
- Making decisions with `switch`
- Enumerations
- Getting started with functions
- Creating and moving the tree branches

Loops

In programming, we often need to do the same thing more than once. The obvious example that we have seen so far is the game loop. With all the code stripped out, our game loop looks like this:

```
while (window.isOpen())
{

}
```

There are a few different types of loops, and we will look at the most commonly used ones here. The correct term for this type of loop is a **while** loop.

while loops

The `while` loop is quite straightforward. Think back to the `if` statements and their expressions that evaluated to either `true` or `false`. We can use the exact same combination of operators and variables in the conditional expressions of our `while` loops.

Like `if` statements, if the expression is `true`, the code executes. The difference with a `while` loop, however, is that the C++ code within it will repeatedly execute until the condition is `false`. Take a look at the following code.

```
int numberOfZombies = 100;

while(numberOfZombies > 0)
{
    // Player kills a zombie
    numberOfZombies--;

    // numberOfZombies decreases each pass through the loop
}

// numberOfZombies is no longer greater than 0
```

Let's go over what's happening in the previous code. Outside of the `while` loop, `int numberOfZombies` is declared and initialized to `100`. Then, the `while` loop begins. Its conditional expression is `numberOfZombies > 0`. Consequently, the `while` loop will continue looping through the code in its body until the condition evaluates to `false`. This means that the preceding code will execute 100 times.

On the first pass through the loop, `numberOfZombies` equals 100, then 99, then 98, and so on. But once `numberOfZombies` is equal to zero, it is, of course, no longer *greater* than zero. Then, the code will break out of the `while` loop and continue to run, after the closing curly brace.

Just like an `if` statement, it is possible that the `while` loop will not execute even once. Take a look at the following code:

```
int availableCoins = 10;

while(availableCoins > 10)
{
    // more code here.
    // Won't run unless availableCoins is greater than 10
}
```

The preceding code inside the `while` loop will not execute because the condition is false.

Note that there is no limit to the complexity of the expression or the amount of code that can go in the loop body. Consider the following hypothetical variation of our game loop:

```
int playerLives = 3;
int alienShips = 10;

while(playerLives !=0 && alienShips !=0 )
{
    // Handle input
    // Update the scene
    // Draw the scene
}

// continue here when either playerLives or alienShips equals 0
```

The previous `while` loop would continue to execute until either `playerLives` or `alienShips` was equal to zero. As soon as one of those conditions occurred, the expression would evaluate to `false` and the program would continue to execute from the first line of code after the `while` loop.

It is worth noting that once the body of the loop has been entered, it will always complete at least once, even if the expression evaluates to false partway through, as it is not tested again until the code tries to start another pass. Let's take a look at an example of this:

```
int x = 1;

while(x > 0)
{
    x--;
    // x is now 0 so the condition is false
    // But this line still runs
    // and this one
    // and me!
}

// Now I'm done!
```

The previous loop body will execute once. We can also set up a `while` loop that will run forever, and unsurprisingly is called an **infinite loop**. Here is an example:

```
int y = 0;

while(true)
{
    y++; // Bigger... Bigger...
}
```

If you find the preceding loop confusing, just think of it literally. A loop executes when its condition is `true`. Well, `true` is always `true`, and will therefore keep executing.

Breaking out of a while loop

We might use an infinite loop so that we can decide when to exit the loop from within its body rather than in the expression. We would do this by using the **break** keyword when we are ready to leave the loop body, perhaps like this:

```
int z = 0;

while(true)
{
    z++; // Bigger... Bigger...
    break; // No you're not

    // Code doesn't reach here
}
```

In the preceding code, the code inside the loop will execute once, upto and including the `break` statement, and then execution will continue after the closing curly brace of the `while` loop.

As you may have been able to guess, we can combine any of the C++ decision-making tools such as `if`, `else`, and another that we will learn about shortly, known as `switch`, within our `while` loops and other loop types as well. Consider the following example:

```
int x = 0;
int max = 10;

while(true)
{
    x++; // Bigger... Bigger...

    if(x == max){
        break;
    } // No you're not

    // code reaches here only until max = 10
}
```

In the preceding code, the `if` condition decides if and when the `break` statement is executed. In this case, the code will keep looping until `max` reaches 10.

We could go on for a long time looking at the various permutations of C++ `while` loops, but, at some point, we want to get back to making games. So, let's move on to another type of loop: the `for` loop.

for loops

The `for` loop has a slightly more complicated syntax than the `while` loop because it takes three parts to set one up. Take a look at the following code first. We will break it apart after:

```
for(int x = 0; x < 100; x ++){
    // Something that needs to happen 100 times goes here
}
```

Here is what all the parts of the `for` loop condition do:

```
for(declaration and initialization; condition; change before each
iteration)
```


To clarify this further, here is a table to explain each of the three key parts, as they appear in the previous `for` loop example:

| Part | Description |
|---|--|
| Declaration and initialization | We create a new <code>int</code> variable, <code>i</code> , and initialize it to 0 |
| Condition | Just like the other loops, it refers to the condition that must be <code>true</code> for the loop to execute |
| Change after each pass through the loop | In the example, <code>x ++</code> means that 1 is added/incremented to <code>x</code> on each pass |

We can vary `for` loops so that they do many more things. Here is another simple example that counts down from 10:

```
for(int i = 10; i > 0; i--)
{
    // countdown
}


// blast off
```

The `for` loop takes control of initialization, condition evaluation, and the control variable. We will use `for` loops in our game, later in this chapter.

Now, we can move on to the topic of C++ arrays, which help us store large amounts of related data.

Arrays

If a variable is a box in which we can store a value of a specific type, such as `int`, `float`, or `char`, then we can think of an array as a row of boxes. The rows of boxes can be of almost any size and type, including objects made from classes. However, all the boxes must be of the same type.

 The limitation of having to use the same type in each box can be circumvented to an extent once we learn some more advanced C++ in the penultimate project.

This array sounds like it could have been useful for our clouds in *Chapter 2, Variables, Operators, and Decisions – Animating Sprites*. So, how do we go about creating and using an array?

Declaring an array

We can declare an array of `int` type variables like this:

```
int someInts[10];
```

Now, we have an array called `someInts` that can store ten `int` values. Currently, however, it is empty.

Initializing the elements of an array

To add values to the elements of an array, we can use the type of syntax we are already familiar with while introducing some new syntax, known as **array notation**. In the following code, we store the value of 99 in the first **element** of the array:

```
someInts[0] = 99;
```

In order to store a value of 999 in the second element, we need to use the following code:

```
someInts[1] = 999;
```

We can store a value of 3 in the last element like this:

```
someInts[9] = 3;
```

Note that the elements of an array always start at zero and go up to the size of the array minus one. Similar to ordinary variables, we can manipulate the values stored in an array. The only difference is that we would use the array notation to do so because although our array has a name — `someInts` — the individual elements do not.

In the following code, we add the first and second elements together and store the answer in the third:

```
someInts[2] = someInts[0] + someInts[1];
```

Arrays can also interact seamlessly with regular variables, for example:

```
int a = 9999;  
someInts[4] = a;
```

There are more ways we can initialize arrays, so let's look at one way now.

Quickly initializing the elements of an array

We can quickly add values to the elements as follows. This example uses a float array:

```
float myFloatingPointArray[3] {3.14f, 1.63f, 99.0f};
```

Now, the 3.14, 1.63, and 99.0 values are stored in the first, second, and third positions, respectively. Remember that, when using an array notation to access these values, we would use [0], [1], and [2].

There are other ways to initialize the elements of an array. This slightly abstract example shows using a for loop to put the values 0 through 9 into the uselessArray array:

```
for(int i = 0; i < 10; i++)
{
    uselessArray[i] = i;
}
```

The preceding code assumes that uselessArray had previously been initialized to hold at least 10 int variables.

But why do we need arrays?

What do these arrays really do for our games?

We can use arrays anywhere a regular variable can be used – perhaps in an expression like this:

```
// someArray[4] is declared and initialized to 9999


for(int i = 0; i < someArray[4]; i++)
{
    // Loop executes 9999 times
}
```

One of the biggest benefits of arrays in game code was hinted at at the start of this section. Arrays can hold objects (instances of classes). Let's imagine that we have a Zombie class and we want to store a whole bunch of them. We can do so like this:

```
Zombie horde [5] {zombie1, zombie2, zombie3}; // etc...
```

The `horde` array now holds a load of instances of the `Zombie` class. Each one is a separate, living (kind of), breathing, self-determining `Zombie` object. We could then loop through the `horde` array, each of which passes through the game loop, move the zombies, and check if their heads have met with an axe or if they have managed to catch the player.

Arrays, had we known about them at the time, would have been perfect for handling our clouds in *Chapter 2, Variables, Operators, and Decisions – Animating Sprites*. We could have had as many clouds as we wanted and written less code than we did for our three measly clouds.

 To check out this improved cloud code in full and in action, look at the enhanced version of *Timber!!!* (code and playable game) in the download bundle. Alternatively, you can try to implement the clouds using arrays yourself before looking at the code.

The best way to get a feel for all of this array stuff is to see it in action. We will do this when we implement our tree branches.

For now, we will leave our cloud code as it is so that we can get back to adding features to the game as soon as possible. But first, let's do a bit more C++ decision-making with **switch**.

Making decisions with switch

We have already looked at `if`, which allows us to decide whether to execute a block of code based upon the result of its expression. But sometimes, a decision in C++ can be made in other ways that are better.

When we must make a decision based on a clear list of possible outcomes that don't involve complex combinations or wide ranges of values, then `switch` is usually the way to go. We can start a `switch` decision as follows:

```
switch(expression)
{
    // More code here
}
```

In the previous example, `expression` could be an actual expression or just a variable. Then, within the curly braces, we can make decisions based on the result of the expression or value of the variable. We do this with the `case` and `break` keywords:

```
case x:
    //code for x
    break;

case y:
    //code for y
    break;
```

As you can see, each `case` states a possible result and each `break` denotes the end of that case and the point that the execution leaves the `switch` block.

Optionally, we can also use the `default` keyword without a value to run some code in case none of the `case` statements evaluate to `true`, as follows:

```
default: // Look no value
    // Do something here if no other case statements are true
    break;
```

As a final and less abstract example for `switch`, consider a retro text adventure where the player enters a letter such as "n", "e", "s", or "w" to move North, East, South, or West. A `switch` block could be used to handle each possible input from the player:

```
// get input from user in a char called command

switch(command){

    case 'n':
        // Handle move here
        break;

    case 'e':
        // Handle move here
        break;

    case 's':
        // Handle move here
        break;

    case 'w':
        // Handle move here
```

```
        break;

    // more possible cases

    default:
        // Ask the player to try again
        break;
}
```

The best way of understanding all we have seen regarding `switch` is by putting it into action, along with all the other new concepts we are learning about.

Next, we will learn about another C++ concept we need to understand before we write some more code. Let's look at class enumerations.

Class enumerations

An **enumeration** is a list of all the possible values in a logical collection. C++ enumerations are a great way of, well, enumerating things. For example, if our game uses variables that can only be in a specific range of values and if those values could logically form a collection or a set, then enumerations are probably appropriate to use. They will make your code clearer and less error-prone.

To declare a class enumeration in C++, we can use these two keywords, `enum class`, together, followed by the name of the enumeration, followed by the values the enumeration can contain, enclosed in a pair of curly braces `{ ... }`.

As an example, examine the following enumeration declaration. Note that it is convention to declare the possible values from the enumeration in uppercase:

```
enum class zombieTypes {
    REGULAR, RUNNER,
    CRAWLER, SPITTER, BLOATER
};
```

Note that, at this point, we have not declared any instances of `zombieType`, just the type itself. If that sounds odd, think about it like this. SFML created the `Sprite`, `RectangleShape`, and `RenderWindow` classes, but to use any of those classes, we had to declare an object/instance of the class.

At this point, we have created a new type called `zombieTypes`, but we have no instances of it. So, let's do that now:

```
zombieType jeremy = zombieTypes::CRAWLER;
zombieType anna = zombieTypes::SPITTER;
zombieType diane = zombieTypes::BLOATER;

/*
    Zombies are fictional creatures and any resemblance
    to real people is entirely coincidental
*/
```

Next is a sneak preview of the type of code we will soon be adding to Timber!!!. We will want to track which side of the tree a branch or the player is on, so we will declare an enumeration called `side`, like this:

```
enum class side { LEFT, RIGHT, NONE };
```

We could position the player on the left like this:

```
// The player starts on the left
side playerSide = side::LEFT;
```

We could make the fourth level (arrays start from zero) of an array of branch positions have no branch at all, like this:

```
branchPositions[3] = side::NONE;
```

We can use enumerations in expressions as well:

```
if(branchPositions[5] == playerSide)
{
    // The lowest branch is the same side as the player
    // SQUISHED!!
}
```

The preceding code tests whether the branch in position [5] element of the array is on the same side as the player.

We will look at one more vital C++ topic, that is, functions, and then we will get back to coding the game. When we want to compartmentalize some code that does one specific thing, we can use a function.

Getting started with functions

What exactly are C++ functions? A function is a collection of variables, expressions, and **control flow statements** (loops and branches). In fact, any of the code we have learned about in this book so far can be used in a function. The first part of a function that we write is called the **signature**. Here is an example function signature:

```
void shootLasers(int power, int direction);
```

If we add an opening and closing pair of curly braces `{ ... }` along with some code that the function performs, we will have a complete function, that is, a **definition**:

```
void shootLasers(int power, int direction)
{
    // ZAPP!
}
```

We could then use our new function from another part of our code, perhaps like this:

```
// Attack the player
shootLasers(50, 180) // Run the code in the function
// I'm back again - code continues here after the function ends
```

When we use a function, we say that we **call** it. At the point where we call `shootLasers`, our program's execution branches to the code contained within that function. The function will run until it reaches the end or is told to `return`. Then, the code will continue running from the first line after the function call. We have already been using the functions that SFML provides. What is different here is that we will learn to write and call our own functions.

Here is another example of a function, complete with the code to make the function return to the code that called it:

```
int addAToB(int a, int b)
{
    int answer = a + b;
    return answer;
}
```

The call so that we can use the preceding function may look like this:

```
int myAnswer = addAToB(2, 4);
```

Obviously, we don't need to write functions to add two variables together, but this example helps us look into the workings of functions. First, we pass in the values 2 and 4. In the function signature, the value 2 is assigned to `int a`, and the value 4 is assigned to `int b`.

Within the function body, the `a` and `b` variables are added together and used to initialize the new variable, `int answer`. The `return answer;` line does just that. It returns the value stored in `answer` to the calling code, causing `myAnswer` to be initialized with the value 6.

Notice that each of the function signatures in the preceding examples vary a little. The reason for this is that the C++ function signature is quite flexible, allowing us to build exactly the functions we require.

Exactly how the function signature defines how the function must be called and if/how the function must return a value deserves further discussion. Let's give each part of that signature a name so that we can break it into parts and learn about them.

Here is a function signature with its parts described by their formal/technical term:

return type | **name of function** | **(parameters)**

Here are a few examples that we can use for each of those parts:

- **Return-type:** `void`, `bool`, `float`, `int`, and so on, or any C++ type or expression
- **Name of function:** `shootLasers`, `addAToB`, and so on
- **Parameters:** `(int number, bool hitDetected)`, `(int x, int y)`, `(float a, float b)`

Now, let's look at each part in turn, starting with the return type.

Function return types

The return type, as its name suggests, is the type of the value that will be returned from the function to the calling code:

```
int addAToB(int a, int b){  
  
    int answer = a + b;  
    return answer;  
  
}
```

In our slightly dull but useful `addAToB` example that we looked at previously, the return type in the signature is `int`. The `addAToB` function sends back and returns a value that will fit in an `int` variable to the code that called it. The return type can be any C++ type we have seen so far or one of the ones we haven't seen yet.

A function does not have to return a value at all, however. In this case, the signature must use the `void` keyword as the return type. When the `void` keyword is used, the function body must not attempt to return a value as this will cause an error. It can, however, use the `return` keyword without a value. Here are some combinations of the return type and use of the `return` keyword that are valid:

```
void doWhatever() {  
  
    // our code  
    // I'm done going back to calling code here  
    // no return is necessary  
  
}
```

Another possibility is as follows:

```
void doSomethingCool() {  
  
    // our code  
  
    // I can do this if I don't try and use a value  
    return;  
  
}
```

The following code is yet more examples of possible functions. Be sure to read the comments as well as the code:

```
void doYetAnotherThing() {  
    // some code  
  
    if(someCondition) {  
  
        // if someCondition is true returning to calling code  
        // before the end of the function body  
        return;  
    }  
  
    // More code that might or might not get executed  
  
    return;  
  
    // As I'm at the bottom of the function body  
    // and the return type is void, I'm  
    // really not necessary but I suppose I make it
```

```
        // clear that the function is over.
    }

    bool detectCollision(Ship a, Ship b){

        // Detect if collision has occurred
        if(collision)
        {
            // Bam!!!
            return true;
        }
        else
        {
            // Missed
            return false;
        }
    }
}
```

The last function example in the preceding code, which is for `detectCollision`, is a glimpse into the near future of our C++ code and demonstrates that we can also pass in user-defined types known as objects into functions so that we can perform calculations on them.

We could call each of the functions, in turn, like this:

```
// OK time to call some functions
doWhatever();
doSomethingCool();
doYetAnotherThing();

if (detectCollision(millenniumFalcon, lukesXWing))
{
    // The jedi are doomed!
    // But there is always Leia.
    // Unless she was on the Falcon?
}
else
{
    // Live to fight another day
}

// Continue with code from here
```

Don't worry about the odd-looking syntax regarding the `detectCollision` function; we will see real code like this soon. Simply, we are using the return value (`true` or `false`) as the expression directly in an `if` statement.

Function names

The function name that we use when we design our own function can be almost anything at all. But it is best to use words, usually verbs, that clearly explain what the function will do. For example, take a look at the following function:

```
void functionaroonieboonie(int blibbityblob, float floppyfloatything)
{
    //code here
}
```

The preceding function is perfectly legal and will work, but the following function names are much clearer:

```
void doSomeVerySpecificTask()
{
    //code here
}

int getMySpaceShipHealth()
{
    //code here
}

void startNewGame()
{
    //code here
}
```

Using clear and descriptive function names such as in the preceding three examples is good practice, but, as we saw from the `functionaroonieboonie` function, this is not a rule that the compiler enforces. Next, we will take a closer look at how we share some values with a function.

Function parameters

We know that a function can return a result to the calling code. But what if we need to share some data values from the calling code with the function? **Parameters** allow us to share values with the function. We have already seen examples of parameters while looking at return types. We will look at the same example but a little more closely:

```
int addAToB(int a, int b)
{
    int answer = a + b;
    return answer;
}
```

Here, the parameters are `int a` and `int b`. Notice that, in the first line of the function body, we use `a + b` as if they are already declared and initialized variables. Well, that's because they are. The parameters in the function signature is their declaration, and the code that calls the function initializes them.



Important jargon note

Note that we are referring to the variables in the function signature brackets (`int a, int b`) as parameters. When we pass values into the function from the calling code, these values are called arguments. When the arguments arrive, they are used by the parameters to initialize real, usable variables, like:

```
int returnedAnswer = addAToB(10, 5);
```

Also, as we have partly seen in previous examples, we don't have to just use `int` in our parameters. We can use any C++ type. We can also use as many parameters as is necessary to solve our problem, but it is good practice to keep the parameter list as short and therefore as manageable as possible.

As we will see in future chapters, we have left a few of the cooler uses of functions out of this introductory tutorial so that we can learn about related C++ concepts before we take the topic of functions further.

The function body

The body is the part we have been kind of avoiding and has comments such as the following:

```
// code here
// some code
```

Actually, we already know exactly what to do here! Any C++ code we have learned about so far will work in the body of a function.

Function prototypes

So far, we have seen how to code a function and we have seen how to call one as well. There is one more thing we need to do, however, to make them work. All functions must have a **prototype**. A prototype is what makes the compiler aware of our function, and without a prototype the entire game will fail to compile. Fortunately, prototypes are straightforward.

We can simply repeat the function's signature, followed by a semicolon. The caveat is that the prototype must appear *before* any attempt to call or define the function. So, the absolute most simple example of a fully usable function in action is as follows. Look carefully at the comments and the location in the code that the different parts of the function appear in:

```
// The prototype
// Notice the semicolon on the end
int addAToB(int a, int b);

int main()
{
    // Call the function
    // Store the result in answer
    int answer = addAToB(2,2);

    // Called before the definition
    // but that's OK because of the prototype

    // Exit main
    return 0;

} // End of main

// The function definition
int addAToB(int a, int b)
{
    return a + b;
}
```

What the previous code demonstrates is the following:

- The prototype is before the `main` function.
- The call to use the function is as we might expect, inside the `main` function.
- The definition is after/outside the `main` function.



Note that we can omit the function prototype and go straight to the definition when the definition occurs before the function is used. As our code becomes longer and spread across multiple files, however, this will almost never happen. We will use separate prototypes and definitions all the time.

Let's see how we can keep our functions organized.

Organizing functions

It's well worth pointing out that if we have multiple functions, especially if they are fairly long, our `.cpp` file will quickly become unwieldy. This defeats part of the objective that functions are intended for. The solution that we will see in the next project is that we can add all our function prototypes to our very own header file (`.hpp` or `.h`). Then, we can code all our functions in another `.cpp` file and simply add another `#include...` directive in our main `.cpp` file. This way, we can use any number of functions without adding any of their code (prototype or definition) to our main code file.

Function gotcha!

Another point that we should discuss about functions is **scope**. If we declare a variable in a function, either directly or in one of the parameters, that variable is not usable/visible outside of that function. Furthermore, any variables declared inside other functions cannot be seen/used inside the function.

The way that we should share values between function code and calling code is through the parameters/arguments and the return value.

When a variable is not available because it is from another function, it is said to be out of scope. When it is available and usable, it is said to be in scope.



Variables declared within any block in C++ are only in scope within that block! This includes loops and `if` blocks as well. A variable that's declared at the top of `main` is in scope anywhere in `main`, a variable that's declared in the game loop is only in scope within the game loop, and so on. A variable that's declared within a function or other block is called a **local** variable. The more code we write, the more this will make sense. Every time we come across an issue in our code regarding scope, I will discuss it to make things clear. There will be one such issue coming up in the next section. There are also some more C++ staples that blow this issue wide open. They are called **references** and **pointers**, and we will learn about them in *Chapter 9, C++ References, Sprite Sheets, and Vertex Arrays* and *Chapter 10, Pointers, the Standard Template Library, and Texture Management* respectively.

More on functions

There is even more we could learn about functions, but we know enough about them already to implement the next part of our game. And don't worry if all the technical terms such as parameters, signatures, and definitions have not completely sunk in yet. These concepts will become clearer when we start to use them.

An absolute final word on functions – for now

It has probably not escaped your attention that we have been calling functions, especially the SFML functions, by appending the name of an object and a period before the function name, like this:

```
spriteBee.setPosition...
window.draw...
// etc
```

And yet, our entire discussion of functions saw us calling functions without any objects. We can write functions as part of a class or simply as a standalone function. When we write a function as part of a class, we need an object of that class to call the function, but when we have a standalone function, we don't.

We will write a standalone function in a minute and we will write classes with functions starting from *Chapter 6, Object-Oriented Programming – Starting the Pong Game*. Everything we know so far about functions is relevant in both cases.

Now, we can get back to coding the branches in the Timber!!! game.

Growing the branches

Next, as I have been promising for the last 20 pages, we will use all the new C++ techniques we've learned about to draw and move some branches on our tree.

Add the following code outside of the main function. Just to be absolutely clear, I mean *before* the code for `int main()`:

```
#include <sstream>
#include <SFML/Graphics.hpp>

using namespace sf;

// Function declaration
void updateBranches(int seed);

const int NUM_BRANCHES = 6;
Sprite branches[NUM_BRANCHES];

// Where is the player/branch?
// Left or Right
enum class side { LEFT, RIGHT, NONE };
side branchPositions[NUM_BRANCHES];

int main()
```

We just achieved quite a few things with that new code:

- First, we wrote a function prototype for a function called `updateBranches`. We can see that it does not return a value (`void`) and that it takes an `int` argument called `seed`. We will write the function definition soon, and we will then see exactly what it does.
- Next, we declare an `int` constant called `NUM_BRANCHES` and initialize it to 6. There will be six moving branches on the tree, and we will soon see how `NUM_BRANCHES` is useful to us.
- Following this, we declare an array of `Sprite` objects called `branches` that can hold six `Sprite` instances.
- After that, we declare a new enumeration called `side` with three possible values: `LEFT`, `RIGHT`, and `NONE`. This will be used to describe the position of individual branches, as well as the player, in a few places throughout our code.

- Finally, in the preceding code, we initialize an array of `side` types with a size of `NUM_BRANCHES` (6). To be clear about what this achieves, we will have an array called `branchPositions` with six values in it. Each of these values is of the `side` type and can be either `LEFT`, `RIGHT`, or `NONE`.



Of course, what you really want to know is why the constant, two arrays, and the enumeration were declared *outside* of the main function. By declaring them above `main`, they now have **global scope**. To describe this in another way, the constant, two arrays, and the enumeration have scope for the entire game. This means we can access and use them all anywhere in the main function and the `updateBranches` function. Note that it is good practice to make all the variables as local to where they are actually used as possible. It might seem useful to make everything global, but this leads to hard-to-read and error-prone code.

Preparing the branches

Now, we will prepare our six `Sprite` objects and load them into the branches array. Add the following highlighted code just before our game loop:

```
// Position the text
FloatRect textRect = messageText.getLocalBounds();
messageText.setOrigin(textRect.left +
    textRect.width / 2.0f,
    textRect.top +
    textRect.height / 2.0f);

messageText.setPosition(1920 / 2.0f, 1080 / 2.0f);

scoreText.setPosition(20, 20);

// Prepare 6 branches
Texture textureBranch;
textureBranch.loadFromFile("graphics/branch.png");

// Set the texture for each branch sprite
for (int i = 0; i < NUM_BRANCHES; i++) {
    branches[i].setTexture(textureBranch);
    branches[i].setPosition(-2000, -2000);

    // Set the sprite's origin to dead centre
```

```
        // We can then spin it round without changing its position
        branches[i].setOrigin(220, 20);
    }

    while (window.isOpen())
```

In the preceding code, we are doing the following:

1. First, we declare an SFML `Texture` object and load the `branch.png` graphic into it.
2. Next, we create a `for` loop that sets `i` to zero and increments `i` by one on each pass through the loop until `i` is no longer less than `NUM_BRANCHES`. This is exactly right because `NUM_BRANCHES` is 6 and the `branches` array has positions 0 through 5.
3. Inside the `for` loop, we set the `Texture` for each `Sprite` in the `branches` array with `setTexture` and then hide it off-screen with `setPosition`.
4. Finally, we set the origin (the point that is used to locate the sprite when it is drawn) with `setOrigin`, to the center of the sprite. Soon, we will be rotating these sprites. Having the origin in the center means they will spin nicely around, without moving the sprite out of position.

Now that we have prepared all the branches, we can write some code to update them all each frame.

Updating the branch sprites each frame

In the following code, we will set the position of all the sprites in the `branches` array, based upon their position in the array and the value of `side` in the corresponding `branchPositions` array. Add the following highlighted code and try to understand it first before we go through it in detail:

```
// Update the score text
std::stringstream ss;
ss << "Score: " << score;
scoreText.setString(ss.str());

// update the branch sprites
for (int i = 0; i < NUM_BRANCHES; i++)
{

    float height = i * 150;
```

```

        if (branchPositions[i] == side::LEFT)
        {
            // Move the sprite to the left side
            branches[i].setPosition(610, height);

            // Flip the sprite round the other way
            branches[i].setRotation(180);
        }
        else if (branchPositions[i] == side::RIGHT)
        {
            // Move the sprite to the right side
            branches[i].setPosition(1330, height);

            // Set the sprite rotation to normal
            branches[i].setRotation(0);
        }
        else
        {
            // Hide the branch
            branches[i].setPosition(3000, height);
        }
    }

} // End if(!paused)

/*
*****
Draw the scene
*****

```

The code we just added is one big `for` loop that sets `i` to zero and increments `i` by one each time through the loop and keeps going until `i` is no longer less than 6.

Inside the `for` loop, a new `float` variable called `height` is set to `i * 150`. This means that the first branch will have a height of 0, the second a height of 150, and the sixth a height of 750.

Next, we have a structure of `if` and `else` blocks. Take a look at the structure with the code stripped out:

```

    if()
    {
    }

```

```
    else if ()
    {
    }
    else
    {
    }
}
```

The first `if` statement uses the `branchPositions` array to see whether the current branch should be on the left. If it is, it sets the corresponding `Sprite` from the `branches` array to a position on the screen, appropriate for the left (610 pixels) and whatever the current `height` is. It then flips the `Sprite` by 180 degrees because the `branch.png` graphic "hangs" to the right by default.

Note that `else if` only executes if the branch is not on the left. It uses the same method to see if it is on the right. If it is, then the branch is drawn on the right (1,330 pixels). Then, the `sprite rotation` is set to zero degrees, just in case it had previously been at 180 degrees. If the `x` coordinate seems a little bit strange, just remember that we set the origin for the branch sprites to their center.

The final `else` statement correctly assumes that the current `branchPosition` must be `NONE` and hides the branch off-screen at 3,000 pixels.

At this point, our branches are in position and ready to be drawn.

Drawing the branches

Here, we will use another `for` loop to step through the entire `branches` array from 0 to 5 and draw each branch sprite. Add the following highlighted code:

```
// Draw the clouds
window.draw(spriteCloud1);
window.draw(spriteCloud2);
window.draw(spriteCloud3);

// Draw the branches
for (int i = 0; i < NUM_BRANCHES; i++) {
    window.draw(branches[i]);
}

// Draw the tree
window.draw(spriteTree);
```

Of course, we still haven't written the function that moves all the branches. Once we have written that function, we will also need to work out when and how to call it. Let's solve the first problem and write the function.

Moving the branches

We have already added the function prototype, above the main function. Now, we can code the actual definition of the function that will move all the branches down by one position each time it is called. We will code this function in two parts so that we can easily examine what is happening.

Add the first part of the `updateBranches` function *after* the closing curly brace of the main function:

```
// Function definition
void updateBranches(int seed)
{
    // Move all the branches down one place
    for (int j = NUM_BRANCHES-1; j > 0; j--) {
        branchPositions[j] = branchPositions[j - 1];
    }
}
```

In this first part of the function, we simply move all the branches down one position, one at a time, starting with the sixth branch. This is achieved by making the `for` loop count from 5 through to 0. Note that `branchPositions[j] = branchPositions[j - 1];` makes the actual move.

The other thing to note with this previous code is that after we have moved the branch in position 4 to position 5, then the branch in position 3 to position 4, and so on, we will need to add a new branch at position 0, which is the top of the tree.

Now, we can spawn a new branch at the top of the tree. Add the following highlighted code, and then we will talk about it:

```
// Function definition
void updateBranches(int seed)
{
    // Move all the branches down one place
    for (int j = NUM_BRANCHES-1; j > 0; j--) {
        branchPositions[j] = branchPositions[j - 1];
    }

    // Spawn a new branch at position 0
    // LEFT, RIGHT or NONE
    srand((int)time(0)+seed);
    int r = (rand() % 5);

    switch (r) {
    case 0:
```

```
        branchPositions[0] = side::LEFT;
        break;

    case 1:
        branchPositions[0] = side::RIGHT;
        break;

    default:
        branchPositions[0] = side::NONE;
        break;
}

}
```

In the final part of the `updateBranches` function, we use the integer `seed` variable that gets passed in with the function call. We do this to guarantee that the random number seed is always different. We will see how we arrived at this value in the next chapter.

Next, we generate a random number between zero and four and store the result in the `int` variable called `r`. Now, we `switch`, using `r` as the expression.

The `case` statements mean that if `r` is equal to zero, then we add a new branch to the left-hand side, at the top of the tree. If `r` is equal to 1, then the branch goes to the right. If `r` is anything else, (2, 3, or 4), then `default` ensures no branch will be added at the top. This balance of left, right, and none makes the tree seem realistic and the game work quite well. You could easily change the code to make the branches more frequent or less so.

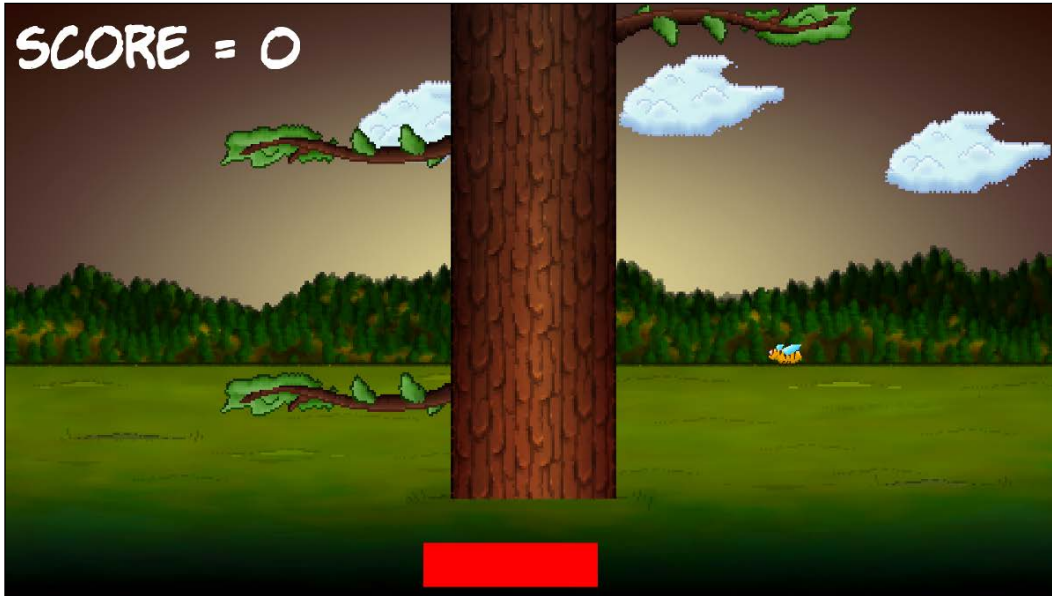
Even after all this code for our branches, we still can't see a single one of them in the game. This is because we have more work to do before we can call the `updateBranches` function.

If you want to see a branch now, you can add some temporary code and call the function five times with a unique seed each time, just before the game loop:

```
updateBranches(1);
updateBranches(2);
updateBranches(3);
updateBranches(4);
updateBranches(5);

while (window.isOpen())
{
```

You can now see the branches in place. But if the branches are to actually move, we will need to call `updateBranches` on a regular basis:



Don't forget to remove the temporary code before moving on.



Now, we can turn our attention to the player as well, as calling the `updateBranches` function for real. We will do so in the next chapter.

Summary

Although not quite the longest, this was probably the chapter where we've covered the most C++ so far. We looked at the different types of loops we can use, such as `for` and `while` loops. We then studied arrays that we can use them to handle large amounts of variables and objects without breaking a sweat. We also learned about enumerations and `switch`. Probably the biggest concept in this chapter was functions, which allow us to organize and abstract our game's code. We will be looking more deeply at functions in a few more places in this book.

Now that we have a fully "working" tree, we can finish the game off, which we will do in the next and final chapter for this project.

FAQ

Q) You mentioned there were a few more types of C++ loops. Where can I find out about them?

A) Yes, take a look at this tutorial and explanation for the `do while` loops:
http://www.tutorialspoint.com/cplusplus/cpp_do_while_loop.htm.

Q) Can I assume I am now an expert on arrays?

A) Like many of the topics in this book, there is always more to learn. You know enough about arrays to proceed, but if you're hungry for more, take a look at this fuller arrays tutorial: <http://www.cplusplus.com/doc/tutorial/arrays/>.

Q) Can I assume that I am an expert on functions?

A) Like many of the topics in this book, there is always more to learn. You know enough about functions to proceed, but if want to know even more, take a look at this tutorial: <http://www.cplusplus.com/doc/tutorial/functions/>.

5

Collisions, Sound, and End Conditions – Making the Game Playable

This is the final phase of the first project. By the end of this chapter, you will have your first completed game. Once you have Timber!!! up and running, be sure to read the last section of this chapter as it will suggest ways to make the game better.

In this chapter, we will cover the following topics:

- Adding the rest of the sprites
- Handling the player input
- Animating the flying log
- Handling death
- Adding sound effects
- Adding features and improving Timber!!!

Preparing the player (and other sprites)

Let's add the code for the player's sprite as well as a few more sprites and textures at the same time. The following, quite large, block of code also adds a gravestone sprite for when the player gets squashed, an axe sprite to chop with, and a log sprite that can whiz away each time the player chops.

Notice that, after the `spritePlayer` object, we declare a side variable, `playerSide`, to keep track of where the player is currently standing. Furthermore, we add some extra variables for the `spriteLog` object, including `logSpeedX`, `logSpeedY`, and `logActive`, to store how fast the log will move and whether it is currently moving. The `spriteAxe` also has two related `float` constant variables to remember where the ideal pixel position is on both the left and the right.

Add the following block of code just before the `while(window.isOpen())` code, like we have done so often before. Note that all of the code in the following block is new, not just the highlighted code. I haven't provided any extra context for this block of code as the `while(window.isOpen())` should be easy to identify. The highlighted code is the code we have just discussed.

Add the entirety of the following code, just before the `while(window.isOpen())` line, and make a mental note of the highlighted lines we briefly discussed. It will make the rest of this chapter's code easier to understand:

```
// Prepare the player
Texture texturePlayer;
texturePlayer.loadFromFile("graphics/player.png");
Sprite spritePlayer;
spritePlayer.setTexture(texturePlayer);
spritePlayer.setPosition(580, 720);

// The player starts on the left
side playerSide = side::LEFT;

// Prepare the gravestone
Texture textureRIP;
textureRIP.loadFromFile("graphics/rip.png");
Sprite spriteRIP;
spriteRIP.setTexture(textureRIP);
spriteRIP.setPosition(600, 860);

// Prepare the axe
Texture textureAxe;
textureAxe.loadFromFile("graphics/axe.png");
Sprite spriteAxe;
spriteAxe.setTexture(textureAxe);
spriteAxe.setPosition(700, 830);

// Line the axe up with the tree
const float AXE_POSITION_LEFT = 700;
const float AXE_POSITION_RIGHT = 1075;
```

```
// Prepare the flying log
Texture textureLog;
textureLog.loadFromFile("graphics/log.png");
Sprite spriteLog;
spriteLog.setTexture(textureLog);
spriteLog.setPosition(810, 720);

// Some other useful log related variables
bool logActive = false;
float logSpeedX = 1000;
float logSpeedY = -1500;
```

In the preceding code, we added quite a few new variables. They are hard to explain in full until we get to where we actually use them, but here is an overview of what they will be used for. There is a variable of the `side` enumeration type called `playerSide` that is initialized to `left`. This will track which side of the tree the player is on.

There are two `const float` values that determine the horizontal position the axe will be drawn at, depending on whether the player is on the left-or right-hand side of the tree.

There are also three variables to help to keep control of the log as it is chopped and flies off of the tree, `bool` to determine whether the log is in motion (`logActive`) and two `float` values to hold the horizontal and vertical speeds of the log.

Now, we can draw all of our new sprites.

Drawing the player and other sprites

Before we add the code to move the player and use all of our new sprites, let's draw them. We are doing it this way so that as we add code to update/change/move them, we will be able to see what is happening.

Add the following highlighted code to draw the four new sprites:

```
// Draw the tree
window.draw(spriteTree);

// Draw the player
window.draw(spritePlayer);

// Draw the axe
```

```
window.draw(spriteAxe);

// Draw the flying log
window.draw(spriteLog);

// Draw the gravestone
window.draw(spriteRIP);

// Draw the bee
window.draw(spriteBee);
```

The preceding code passes our four new sprites, one after the other, to the draw function.

Run the game and you will see our new sprites in the scene:



We are really close to a working game now. The next task is to write some code to allow the player to control what happens.

Handling the player's input

A few different things depend on the movement of the player, as follows:

- When to show the axe
- When to begin animating the log
- When to move all of the branches down

Therefore, it makes sense to set up keyboard handling for the player who's chopping. Once this is done, we can put all of the features we just mentioned into the same part of the code.

Let's think for a moment about how we detect keyboard presses. Each frame, we test whether a particular keyboard key is currently being held down. If it is, we take action. If the *Esc* key is being held down, we quit the game, and if the *Enter* key is being held down, we restart the game. So far, this has been sufficient for our needs.

There is, however, a problem with this approach when we try and handle the chopping of the tree. The problem has always been there; it just didn't matter until now. Depending on how powerful your PC is, the game loop could be executing thousands of times per second. Each and every pass through the game loop that a key is held down, it is detected, and the related code will execute.

So, actually, every time you press *Enter* to restart the game, you are most likely restarting it well in excess of a hundred times. This is because even the briefest of presses will last a significant fraction of a second. You can verify this by running the game and holding down the *Enter* key. Note that the time-bar doesn't move. This is because the game is being restarted over and over again, hundreds or even thousands of times a second.

If we don't use a different approach for the player chopping, then just one attempted chop will bring the entire tree down in a mere fraction of a second. We need to be a bit more sophisticated. What we will do is allow the player to chop, and then when they do so, disable the code that detects a key press. We will then detect when the player removes their finger from a key and then reenables the detection of key presses. Here are the steps laid out clearly:

1. Wait for the player to use the left or right arrow keys to chop a log.
2. When the player chops, disable key press detection.
3. Wait for the player to remove their finger from a key.
4. Reenable chop detection.
5. Repeat from step 1.

This might sound complicated but, with SFML's help, this will be straightforward. Let's implement this now, one step at a time.

Add the following highlighted line of code, which declares a `bool` variable called `acceptInput`, which will be used to determine when to listen for chops and when to ignore them:

```
float logSpeedX = 1000;
float logSpeedY = -1500;

// Control the player input
bool acceptInput = false;

while (window.isOpen())
{
```

Now that we have our Boolean set up, we can move on to the next step.

Handling setting up a new game

So that we're ready to handle chops, add the following highlighted code to the `if` block that starts a new game:

```
/*
*****
Handle the players input
*****
*/

if (Keyboard::isKeyPressed(Keyboard::Escape))
{
    window.close();
}

// Start the game
if (Keyboard::isKeyPressed(Keyboard::Return))
{
    paused = false;

    // Reset the time and the score
    score = 0;
    timeRemaining = 6;

    // Make all the branches disappear -
    // starting in the second position
```

```

    for (int i = 1; i < NUM_BRANCHES; i++)
    {
        branchPositions[i] = side::NONE;
    }

    // Make sure the gravestone is hidden
    spriteRIP.setPosition(675, 2000);

    // Move the player into position
    spritePlayer.setPosition(580, 720);

    acceptInput = true;

}

/*
*****
Update the scene
*****
*/

```

In the previous code, we are using a `for` loop to prepare the tree with no branches. This is fair to the player because, if the game started with a branch right above their head, it would be considered unsporting. Then, we simply move the gravestone off of the screen and the player into their starting location on the left. The last thing the preceding code does is set `acceptInput` to `true`.

We are now ready to receive chopping key presses.

Detecting the player chopping

Now, we can handle the left and right cursor key presses. Add this simple `if` block, which only executes when `acceptInput` is `true`:

```

// Start the game
if (Keyboard::isKeyPressed(Keyboard::Return))
{
    paused = false;

    // Reset the time and the score
    score = 0;
    timeRemaining = 5;

    // Make all the branches disappear

```



```
    for (int i = 1; i < NUM_BRANCHES; i++)
    {
        branchPositions[i] = side::NONE;
    }

    // Make sure the gravestone is hidden
    spriteRIP.setPosition(675, 2000);

    // Move the player into position
    spritePlayer.setPosition(675, 660);

    acceptInput = true;
}

// Wrap the player controls to
// Make sure we are accepting input
if (acceptInput)
{
    // More code here next...
}

/*
*****
Update the scene
*****
*/
```

Now, inside the `if` block that we just coded, add the following highlighted code to handle what happens when the player presses the right cursor key on the keyboard:

```
    // Wrap the player controls to
    // Make sure we are accepting input
    if (acceptInput)
    {
        // More code here next...

        // First handle pressing the right cursor key
        if (Keyboard::isKeyPressed(Keyboard::Right))
        {
            // Make sure the player is on the right
            playerSide = side::RIGHT;
        }
    }
}
```

```

        score ++;

        // Add to the amount of time remaining
        timeRemaining += (2 / score) + .15;

        spriteAxe.setPosition(AXE_POSITION_RIGHT,
                               spriteAxe.getPosition().y);

        spritePlayer.setPosition(1200, 720);

        // Update the branches
        updateBranches(score);

        // Set the log flying to the left
        spriteLog.setPosition(810, 720);
        logSpeedX = -5000;
        logActive = true;

        acceptInput = false;
    }

    // Handle the left cursor key
}

```

Quite a bit is happening in that preceding code, so let's go through it:

- First, we detect whether the player has chopped on the right-hand side of the tree. If they have, then we set `playerSide` to `side::RIGHT`. We will respond to the value of `playerSide` later in the code. Then, we add one to the score with `score ++`.
- The next line of code is slightly mysterious, but all that is happening is we are adding to the amount of time remaining. We are rewarding the player for taking action. The problem for the player, however, is that the higher the score gets, the less additional time is added on. You can play with this formula to make the game easier or harder.
- Then, the axe is moved into its right-hand-side position with `spriteAxe.setPosition` and the player sprite is moved into its right-hand-position as well.
- Next, we call `updateBranches` to move all the branches down one place and spawn a new random branch (or space) at the top of the tree.

- Then, `spriteLog` is moved into its starting position, camouflaged against the tree, and its `speedX` variable is set to a negative number so that it whizzes off to the left. Also, `logActive` is set to `true` so that the log moving code that we will write soon animates the log each frame.
- Finally, `acceptInput` is set to `false`. At this point, no more chops can be made by the player. We have solved the problem of the presses being detected too frequently, and we will see how we can reenable chopping soon.

Now, still inside the `if (acceptInput)` block that we just coded, add the following highlighted code to handle what happens when the player presses the left cursor key on the keyboard:

```
// Handle the left cursor key

if (Keyboard::isKeyPressed(Keyboard::Left))
{
    // Make sure the player is on the left
    playerSide = side::LEFT;

    score++;

    // Add to the amount of time remaining
    timeRemaining += (2 / score) + .15;

    spriteAxe.setPosition(AXE_POSITION_LEFT,
        spriteAxe.getPosition().y);

    spritePlayer.setPosition(580, 720);

    // update the branches
    updateBranches(score);

    // set the log flying
    spriteLog.setPosition(810, 720);
    logSpeedX = 5000;
    logActive = true;

    acceptInput = false;
}
}
```

The previous code is just the same as the code that handles the right-hand-side chop, except that the sprites are positioned differently and the `logSpeedX` variable is set to a positive value so that the log whizzes to the right.

Now, we can code what happens when a keyboard key is released.

Detecting a key being released

To make the preceding code work beyond the first chop, we need to detect when the player releases a key and then set `acceptInput` back to `true`.

This is slightly different to the key handling we have seen so far. SFML has two different ways of detecting keyboard input from the player. We have already seen the first way when we handled the *Enter* key, and it is dynamic and instantaneous, which is exactly what we need to respond immediately to a key press.

The following code uses the method of detecting when a key is released. Enter the following highlighted code at the top of the `Handle the players input` section and then we will go through it:

```
/*
*****
Handle the players input
*****
*/

Event event;

while (window.pollEvent(event))
{
    if (event.type == Event::KeyReleased && !paused)
    {
        // Listen for key presses again
        acceptInput = true;

        // hide the axe
        spriteAxe.setPosition(2000,
                               spriteAxe.getPosition().y);
    }
}

if (Keyboard::isKeyPressed(Keyboard::Escape))
{
    window.close();
}
```

In the preceding code, we declare an object of the `Event` type called `event`. Then, we call the `window.pollEvent` function, passing in our new object, `event`. The `pollEvent` function puts data into the `event` object that describes an operating system event. This could be a key press, key release, mouse movement, mouse click, game controller action, or something that happened to the window itself (resized, moved, and so on).

The reason that we wrap our code in a `while` loop is because there might be many events stored in a queue. The `window.pollEvent` function will load them, one at a time, into `event`. With each pass through the loop, we will see whether we are interested in the current event and respond if we are. When `window.pollEvent` returns `false`, that means there are no more events in the queue and the `while` loop will exit.

This `if` condition (`event.type == Event::KeyReleased && !paused`) executes when both a key has been released and the game is not paused.

Inside the `if` block, we set `acceptInput` back to `true` and hide the axe sprite off screen.

You can now run the game and gaze in awe at the moving tree, swinging axe, and animated player. It won't, however, squash the player, and the log doesn't move yet when chopped.

Let's move on to making the log move.

Animating the chopped logs and the axe

When the player chops, `logActive` is set to `true`, so we can wrap some code in a block that only executes when `logActive` is `true`. Furthermore, each chop sets `logSpeedX` to either a positive or negative number, so the log is ready to start flying away from the tree in the correct direction.

Add the following highlighted code right after where we update the branch sprites:

```
// update the branch sprites
for (int i = 0; i < NUM_BRANCHES; i++)
{
    float height = i * 150;

    if (branchPositions[i] == side::LEFT)
    {
        // Move the sprite to the left side
    }
}
```

```
        branches[i].setPosition(610, height);

        // Flip the sprite round the other way
        branches[i].setRotation(180);
    }
    else if (branchPositions[i] == side::RIGHT)
    {
        // Move the sprite to the right side
        branches[i].setPosition(1330, height);

        // Flip the sprite round the other way
        branches[i].setRotation(0);
    }
    else
    {
        // Hide the branch
        branches[i].setPosition(3000, height);
    }
}

// Handle a flying log
if (logActive)
{
    spriteLog.setPosition(
        spriteLog.getPosition().x +
        (logSpeedX * dt.asSeconds()),

        spriteLog.getPosition().y +
        (logSpeedY * dt.asSeconds()));

    // Has the log reached the right hand edge?
    if (spriteLog.getPosition().x < -100 ||
        spriteLog.getPosition().x > 2000)
    {
        // Set it up ready to be a whole new log next frame
        logActive = false;
        spriteLog.setPosition(810, 720);
    }
}
```

```
    } // End if(!paused)

    /*
    *****
    Draw the scene
    *****
    */
```

The code sets the position of the sprite by getting its current horizontal and vertical location with `getPosition` and then adding to it using `logSpeedX` and `logSpeedY`, respectively, multiplied by `dt.asSeconds`.

After the log sprite has been moved each frame, the code uses an `if` block to see whether the sprite has disappeared out of view on either the left or the right. If it has, the log is moved back to its starting point, ready for the next chop.

If you run the game now, you will be able to see the log flying off to the appropriate side of the screen:



Now, let's move on to a more sensitive subject.

Handling death

Every game must end badly with either the player running out of time (which we have already handled) or getting squashed by a branch.

Detecting the player getting squashed is really simple. All we want to know is: does the last branch in the `branchPositions` array equal `playerSide`? If it does, the player is dead.

Add the following highlighted code that detects and executes when the player is squashed by a branch. We will talk about it later:

```
// Handle a flying log
if (logActive)
{

    spriteLog.setPosition(
        spriteLog.getPosition().x +
        (logSpeedX * dt.asSeconds()),

    spriteLog.getPosition().y +
        (logSpeedY * dt.asSeconds()));

    // Has the log reached the right-hand edge?
    if (spriteLog.getPosition().x < -100 ||
        spriteLog.getPosition().x > 2000)
    {
        // Set it up ready to be a whole new cloud next frame
        logActive = false;
        spriteLog.setPosition(800, 600);
    }
}

// has the player been squished by a branch?
if (branchPositions[5] == playerSide)
{
    // death
    paused = true;
    acceptInput = false;

    // Draw the gravestone
    spriteRIP.setPosition(525, 760);

    // hide the player
    spritePlayer.setPosition(2000, 660);
```



```
        // Change the text of the message
        messageText.setString("SQUISHED!!");

        // Center it on the screen
        FloatRect textRect = messageText.getLocalBounds();

        messageText.setOrigin(textRect.left +
                               textRect.width / 2.0f,
                               textRect.top + textRect.height / 2.0f);

        messageText.setPosition(1920 / 2.0f,
                                1080 / 2.0f);

    }

} // End if(!paused)

/*
*****
Draw the scene
*****
*/
```

The first thing the preceding code does, after the player's demise, is set `paused` to `true`. Now, the loop will complete this frame and won't run the update part of the loop again until a new game is started by the player.

Then, we move the gravestone into position, near where the player was standing, and hide the player sprite off screen.

We set the `String` of `messageText` to "Squished!!" and then use the usual technique to center it on the screen.

You can now run the game and play it for real. The following screenshot shows the player's final score and their gravestone, as well as the **SQUISHED** message:



There is just one more problem to deal with. Is it just me, or is it a little bit quiet?

Simple sound FX

In this section, we will add three sounds. Each sound will be played on a particular game event, that is, a simple thud sound whenever the player chops, a gloomy losing sound when the player runs out of time, and a retro crushing sound when the player is squashed to death.

How SFML sound works

SFML plays sound effects using two different classes. The first class is the `SoundBuffer` class. This is the class that holds the actual audio data from the sound file. It is `SoundBuffer` that is responsible for loading the .wav files into the PC's RAM in a format that can be played without any further decoding work.

When we write code for the sound effects in a minute, we will see that, once we have a `SoundBuffer` object with our sound stored in it, we will then create another object of the `Sound` type. We can then associate this `Sound` object with a `SoundBuffer` object. Then, at the appropriate moment in our code, we will be able to call the `play` function of the appropriate `Sound` object.

When to play the sounds

As we will see very soon, the C++ code to load and play sounds is really simple. What we need to consider, however, is *when* we call the `play` function, where in our code will we put the function calls to `play`? Let's see:

- The chop sound can be called from the key presses of the left and right cursor keys.
- The death sound can be played from the `if` block that detects that a tree has mangled the player.
- The out of time sound can be played from the `if` block which detects whether `timeRemaining` is less than zero.

Now, we can write our sound code.

Adding the sound code

First, we will add another `#include` directive to make the SFML sound-related classes available. Add the following highlighted code:

```
#include <sstream>
#include <SFML/Graphics.hpp>
#include <SFML/Audio.hpp>

using namespace sf;
```

Now, we will declare three different `SoundBuffer` objects, load three different sound files into them, and associate three different objects of the `Sound` type with the related objects of the `SoundBuffer` type. Add the following highlighted code:

```
// Control the player input
bool acceptInput = false;

// Prepare the sounds
// The player chopping sound
SoundBuffer chopBuffer;
chopBuffer.loadFromFile("sound/chop.wav");
Sound chop;
chop.setBuffer(chopBuffer);

// The player has met his end under a branch
SoundBuffer deathBuffer;
deathBuffer.loadFromFile("sound/death.wav");
Sound death;
```

```
death.setBuffer(deathBuffer);

// Out of time
SoundBuffer ootBuffer;
ootBuffer.loadFromFile("sound/out_of_time.wav");
Sound outOfTime;
outOfTime.setBuffer(ootBuffer);

while (window.isOpen())
{
```

Now, we can play our first sound effect. Add the following single line of code to the `if` block, which detects that the player has pressed the right cursor key:

```
// Wrap the player controls to
// Make sure we are accepting input
if (acceptInput)
{
    // More code here next...

    // First handle pressing the right cursor key
    if (Keyboard::isKeyPressed(Keyboard::Right))
    {
        // Make sure the player is on the right
        playerSide = side::RIGHT;

        score++;

        timeRemaining += (2 / score) + .15;

        spriteAxe.setPosition(AXE_POSITION_RIGHT,
                               spriteAxe.getPosition().y);


        spritePlayer.setPosition(1120, 660);

        // update the branches
        updateBranches(score);

        // set the log flying to the left
        spriteLog.setPosition(800, 600);
        logSpeedX = -5000;
        logActive = true;
```

```
        acceptInput = false;

        // Play a chop sound
        chop.play();
    }
```

 Add exactly the same code at the end of the next block of code that starts with `if (Keyboard::isKeyPressed(Keyboard::Left))` to make a chopping sound when the player chops on the left-hand side of the tree.

Find the code that deals with the player running out of time and add the following highlighted code to play the out of time-related sound effect:

```
if (timeRemaining <= 0.f) {
    // Pause the game
    paused = true;

    // Change the message shown to the player
    messageText.setString("Out of time!!");

    //Reposition the text based on its new size
    FloatRect textRect = messageText.getLocalBounds();
    messageText.setOrigin(textRect.left +
        textRect.width / 2.0f,
        textRect.top +
        textRect.height / 2.0f);

    messageText.setPosition(1920 / 2.0f, 1080 / 2.0f);

    // Play the out of time sound
    outOfTime.play();
}
```

Finally, to play the death sound when the player is squished, add the following highlighted code to the `if` block, which executes when the bottom branch is on the same side as the player:

```
// has the player been squished by a branch?
if (branchPositions[5] == playerSide)
{
    // death
    paused = true;
```

```
        acceptInput = false;

        // Draw the gravestone
        spriteRIP.setPosition(675, 660);

        // hide the player
        spritePlayer.setPosition(2000, 660);

        messageText.setString("SQUISHED!!");
        FloatRect textRect = messageText.getLocalBounds();

        messageText.setOrigin(textRect.left +
                               textRect.width / 2.0f,
                               textRect.top + textRect.height / 2.0f);

        messageText.setPosition(1920 / 2.0f, 1080 / 2.0f);

        // Play the death sound
        death.play();
    }
```

That's it! We have finished the first game. Let's discuss some possible enhancements before we move on to the second project.

Improving the game and the code

Take a look at these suggested enhancements for the Timber!!! project. You can see the enhancements in action in the `Runnable` folder of the download bundle:

- **Speed up the code:** There is a part of our code that is slowing down our game. It doesn't matter for this simple game, but we can speed things up by putting the `sstream` code in a block that only executes occasionally. After all, we don't need to update the score thousands of times a second!
- **Debugging console:** Let's add some more text so that we can see the current frame rate. Like the score, we don't need to update this too often. Once every hundred frames will do.
- **Add more trees to the background:** Simply add some more tree sprites and draw them in whatever position looks good (some nearer the camera and some further away).

- **Improve the visibility of the HUD text:** We can draw simple `RectangleShape` objects behind the score and the FPS counter. Black with a bit of transparency will look quite good.
- **Make the cloud code more efficient:** As we alluded to a few times already, we can use our knowledge of arrays to make the cloud code a lot shorter.

Take a look at the game in action with extra trees, clouds, and a transparent background for the text:



To see the code for these enhancements, take a look in the `Timber Enhanced Version` folder of the download bundle.

Summary

In this chapter, we added the finishing touches and graphics to the Timber!!! game. If, prior to this book, you had never coded a single line of C++, then you can give yourself a big pat on the back. In just five modest chapters, you have gone from zero knowledge to a working game.

However, we will not be congratulating ourselves for too long because, in the next chapter, we will move straight on to some slightly more hardcore C++. While the next game, a simple Pong game, in some ways is simpler than Timber!!, learning about writing our own classes will prepare us for building more complicated and fuller-featured games.

FAQ

Q) I admit that the arrays solution for the clouds was more efficient. But do we really need three separate arrays – one for active, one for speed, and one for the sprite itself?

A) If we look at the properties/variables that various objects have, for example, `Sprite` objects, we will see they are numerous. Sprites have position, color, size, rotation, and more as well. But it would be just perfect if they had `active`, `speed`, and perhaps some more. The problem is that the coders at SFML can't possibly predict all of the ways that we will want to use their `Sprite` class. Fortunately, we can make our own classes. We could make a class called `Cloud` that has a Boolean for `active` and `int` for `speed`. We can even give our `Cloud` class an SFML `Sprite` object. We could then simplify our cloud code even further. We will look at designing our own classes in the next chapter.

6

Object-Oriented Programming – Starting the Pong Game

In this chapter, there's quite a large amount of theory, but the theory will give us the knowledge that we need to start using **object-oriented programming (OOP)** with some expertise. Furthermore, we will not waste any time in putting that theory to good use as we will use it to code the next project, a Pong game. We will get to look behind the scenes at how we can create new types that we can use as objects by coding a class. First, we will look at a simplified Pong scenario so that we can learn about some class basics, and then we will start again and code a Pong game for real using the principles we have learned.

In this chapter, we will cover the following topics:

- Learn about OOP and classes using a hypothetical `Bat` class
- Start working on the Pong game and code a real class to represent the player's bat

OOP

Object-oriented programming is a programming paradigm that we could consider to be almost the standard way to code. It is true there are non-OOP ways to code and there are even some non-OOP game coding languages/libraries. However, since we are starting from scratch, there is no reason to do things in any other way.


OOP will do the following:

- Make our code easier to manage, change, or update
- Make our code quicker and more reliable to write
- Make it possible to easily use other people's code (like we have with SFML)


We have already seen the third benefit in action. Let's discuss exactly what OOP is.

OOP is a way of programming that involves breaking our requirements down into chunks that are more manageable than the whole. Each chunk is self-contained yet potentially reusable by other programs, while working together as a whole with the other chunks. These chunks are what we have been referring to as objects.

When we plan and code an object, we do so with a **class**.

[ A class can be thought of as the blueprint of an object.]

We implement an object *of* a class. This is called an **instance** of a class. Think about a house blueprint. You can't live in it, but you can build a house from it. You build an instance of the house. Often, when we design classes for our games, we write them to represent real-world *things*. In the next project, we will write classes for a bat that the player controls and a ball that the player can bounce around the screen with the bat. However, OOP is more than this.

[ OOP is a *way* of doing things, a methodology that defines best practices.]

The three core principles of OOP are **encapsulation**, **polymorphism**, and **inheritance**. This might sound complex but, taken a step at a time, this is reasonably straightforward.

Encapsulation

Encapsulation means keeping the internal workings of your code safe from interference from the code that uses it. You can achieve this by allowing only the variables and functions you choose to be accessed. This means your code can always be updated, extended, or improved without affecting the programs that use it, provided the exposed parts are still accessed in the same way.

As an example, with proper encapsulation, it wouldn't matter whether the SFML team needed to update the way their `Sprite` class works. If the function signatures remain the same, they don't have to worry about what goes on inside. The code that we wrote before the update will still work after the update.

Polymorphism

Polymorphism allows us to write code that is less dependent on the *types* we are trying to manipulate. This will make our code clearer and more efficient. Polymorphism means *different forms*. If the objects that we code can be more than one type of thing, then we can take advantage of this. Polymorphism might sound a little bit like black magic at this point. We will use polymorphism in the fourth project, which we will start in *Chapter 14, Abstraction and Code Management – Making Better Use of OOP*. Everything will become clearer.

Inheritance

Just like it sounds, **inheritance** means we can harness all the features and benefits of other peoples' classes, including encapsulation and polymorphism, while further refining their code specifically to our situation. We will use inheritance for the first time at the same time as we use polymorphism.

Why use OOP?

When written properly, OOP allows you to add new features without worrying about how they interact with existing features. When you do have to change a class, its self-contained (encapsulated) nature means less or perhaps even zero consequences for other parts of the program.

You can use other people's code (like the SFML classes) without knowing or perhaps even caring for how it works inside.

OOP and, by extension, SFML, allows you to write games that use complicated concepts such as multiple cameras, multiplayer, OpenGL, directional sound, and more besides – all of this without breaking a sweat.

You can create multiple, similar, yet different versions of a class without starting the class from scratch by using inheritance.

You can still use the functions intended for the original type of object with your new object because of polymorphism.

All this makes sense really. And as we know, C++ was designed from the start with all this OOP in mind.



The ultimate key to success with OOP and making games (or any other type of app), other than the determination to succeed, is planning and design. It is not so much just "knowing" all the C++, SFML, and OOP topics that will help you to write great code, but rather applying all that knowledge to write code that is well-structured/designed. The code in this book is presented in an order and manner that's appropriate for learning about the various C++ topics in a gaming context. The art and science of structuring your code is called **design patterns**. As your code gets longer and more complex, effective use of design patterns will become more important. The good news is that we don't need to invent these design patterns ourselves. We will need to learn about them as our projects get more complex. As our projects become more complex, our design patterns will evolve too.

In this project, we will learn about and use basic classes and encapsulation. As this book progresses, we will get a bit more daring and use inheritance, polymorphism, and other OOP-related C++ features too.

What exactly is a class?

A class is a bunch of code that can contains functions, variables, loops, and all the other C++ syntax we have already learned about. Each new class will be declared in its own `.h` code file with the same name as the class, while its functions will be defined in their own `.cpp` file.

Once we have written a class, we can use it to make as many objects from it as we want. Remember, the class is the blueprint, and we make objects based on the blueprint. The house isn't the blueprint, just like the object isn't the class. It is an object made *from* the class.



You can think of an object as a variable and the class as a type.

Of course, with all this talk of OOP and classes, we haven't actually seen any code. Let's fix that now.

The theory of a Pong Bat

What follows is a hypothetical discussion of how we might use OOP to get started with the Pong project by coding a Bat class. Don't add any code to the project just yet as what follows is over-simplified in order to explain the theory. Later in this chapter, we will code it for real. When we get to coding the class for real, it will actually be quite different, but the principles we will learn about here will prepare us for success.

We will begin by exploring variables and functions as part of a class.

The class variable and function declarations

A bat that bounces a ball would be an excellent first candidate for a class.



If you don't know what Pong is, then take a look at this link:
<https://en.wikipedia.org/wiki/Pong>.

Let's take a look at a hypothetical Bat .h file:

```
class Bat
{
    private:

        // Length of the pong bat
        int m_Length = 100;

        // Height of the pong bat
        int m_Height = 10;

        // Location on x axis
        int m_XPosition;

        // Location on y axis
        int m_YPosition;

    public:

        void moveRight();
        void moveLeft();
};
```

At first glance, the code might appear a little complex, but when it has been explained, we will see there are very few concepts we haven't already covered.

The first thing to notice is that a new class is declared using the `class` keyword, followed by the name of the class and that the entire declaration is enclosed in curly braces, followed by a closing semicolon:

```
class Bat
{
    ...
    ...

};
```

Now, let's take a look at the variable declarations and their names:

```
// Length of the pong bat
int m_Length = 100;

// Height of the pong bat
int m_Height = 10;

// Location on x axis
int m_XPosition;

// Location on y axis
int m_YPosition;
```

All the names are prefixed with `m_`. This `m_` prefix is not compulsory, but it is a good convention. Variables that are declared as part of the class are called **member variables**. Prefixing with an `m_` makes it plain when we are dealing with a member variable. When we write functions for our classes, we will start to see local (non-member) variables and parameters as well. The `m_` convention will then prove itself useful.

Also, notice that all the variables are in a section of the code headed with the `private:` keyword. Scan your eyes over the previous code and note that the body of the class code is separated into two sections:

```
private:
    // more code here

public:
    // More code here
```

The `public` and `private` keywords control the encapsulation of our class. Anything that is `private` cannot be accessed directly by the user of an instance/object of the class. If you are designing a class for others to use, you don't want them to be able to alter anything at will. Note that member variables do not have to be `private`, but good encapsulation is achieved by making them `private` whenever possible.

This means that our four member variables (`m_Length`, `m_Height`, `m_XPosition`, and `m_YPosition`) cannot be accessed directly by our game engine from the `main` function. They can only be accessed indirectly by the code of the class. This is encapsulation in action. For the `m_Length` and `m_Height` variables, this is fairly easy to accept as long as we don't need to change the size of the bat. The `m_XPosition` and `m_YPosition` member variables, however, need to be accessed, or how on earth will we move the bat?

This problem is solved in the `public:` section of the code, as follows:

```
void moveRight();  
void moveLeft();
```

The class provides two functions that are `public` and will be usable with an object of the `Bat` type. When we look at the definitions of these functions, we will see how exactly these functions manipulate the `private` variables.

In summary, we have a bunch of inaccessible (`private`) variables that cannot be used from the `main` function. This is good because encapsulation makes our code less error-prone and more maintainable. We then solve the problem of moving the bat by providing indirect access to the `m_XPosition` and `m_YPosition` variables by providing two `public` functions.

The code in the `main` function can call these functions using an instance of the class, but the code inside the functions control exactly how the variables are used.

Let's take a look at the function definitions.

The class function definitions

The function definitions we will write in this book will all go in a separate file to the class and function declarations. We will use files with the same name as the class and the `.cpp` file extension. So, for example, the following code would go in a file called `Bat.cpp`. Look at the following code, which has just one new concept:

```
#include "Bat.h"  
  
void Bat::moveRight()  
{
```

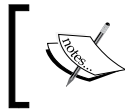


```
        // Move the bat a pixel to the right
        xPosition ++;
    }

    void Bat::moveLeft()
    {
        // Move the bat a pixel to the left
        xPosition --;
    }
```

The first thing to note is that we must use an include directive to include the class and function declarations from the `Bat.h` file.

The new concept we can see here is the use of the **scope resolution operator**, `::`. Since the functions belong to a class, we must write the signature part by prefixing the function name with the class name, as well as `::void Bat::moveLeft()` and `void Bat::moveRight`.



Actually, we have briefly seen the scope resolution operator before, that is, whenever we declare an object of a class, and we have not previously used `using namespace...`

Note that we could have put the function definitions and declarations in one file, like this:

```
class Bat
{
    private:

        // Length of the pong bat
        int m_Length = 100;

        // Length of the pong bat
        int m_Height = 10;

        // Location on x axis
        int m_XPosition;

        // Location on y axis
        int m_YPosition;

    public:

        void Bat::moveRight()
```

```
    {  
        // Move the bat a pixel to the right  
        xPosition ++;  
    }  
  
    void Bat::moveLeft()  
    {  
        // Move the bat a pixel to the left  
        xPosition --;  
    }  
  
};
```

However, when our classes get longer (as they will with our first Zombie Arena class), it is more organized to separate the function definitions into their own file. Furthermore, header files are considered "public", and are often used for documentation purposes if other people will be using the code that we write.

But how do we use a class once we have coded it?

Using an instance of a class

Despite all the code we have seen related to classes, we haven't actually used the class. We already know how to do this as we have used the SFML classes many times already.

First, we would create an instance of the `Bat` class, like this:

```
Bat bat;
```

The `bat` object has all the variables we declared in `Bat.h`. We just can't access them directly. We can, however, move our bat using its public functions, like this:

```
bat.moveLeft();
```

Or we can move it like this:

```
bat.moveRight();
```

Remember that `bat` *is a* `Bat`, and as such it has all the member variables and has all of the functions available to it.

Later, we may decide to make our Pong game multiplayer. In the `main` function, we could change the code so that the game has two bats, perhaps like this:

```
Bat bat;  
Bat bat2;
```

It is vitally important to realize that each of these instances of `Bat` are separate objects with their very own set of variables. There are more ways to initialize an instance of a class, and we will see an example of this when we code the `Bat` class for real, next.

Now, we can start the project for real.

Creating the Pong project

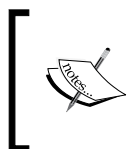
Since setting up a project is a fiddly process, we will go through it a step by step, like we did for the Timber!!! project. I won't show you the same screenshots that I did for the Timber!!! project, but the process is the same, so flip back to *Chapter 1, C++, SFML, Visual Studio, and Starting the First Game* if you want a reminder of the locations of the various project properties:

1. Start Visual Studio and click on the **Create New Project** button. Or, if you still have the Timber!!! project open, you can select **File | New project**.
2. In the window shown next, choose **Console app** and click the **Next** button. You will then see the **Configure your new project** window.
3. In the **Configure your new project** window, type `Pong` in the **Project name** field. Note that this causes Visual Studio to automatically configure the **Solution name** field so that it has the same name.
4. In the **Location** field, browse to the `vs_projects` folder that we created in Chapter 1. Like the Timber!!! project, this will be the location that all our project files will be kept.
5. Check the option to **Place solution and project in the same directory**.
6. When you have completed these steps, click **Create**. The project is generated by Visual Studio, including some C++ code in the `main.cpp` file, like it was previously.
7. We will now configure the project to use the SFML files that we put in the `SFML` folder. From the main menu, select **Project | Pong properties....**. At this stage, you should have the **Pong Property Pages** window open.
8. In the **Pong Property Pages** window, select **All Configurations** from the **Configuration:** drop-down.
9. Now, select **C/C++** and then **General** from the left-hand menu.
10. After, locate the **Additional Include Directories** edit box and type the drive letter where your SFML folder is located, followed by `\SFML\include`. The full path to type, if you located your SFML folder on your D drive, is `D:\SFML\include`. Change your path if you installed SFML on a different drive.

11. Click **Apply** to save your configurations so far.
12. Now, still in the same window, perform these steps. From the left-hand menu, select **Linker** and then **General**.
13. Now, find the **Additional Library Directories** edit box and type the drive letter where your SFML folder is, followed by `\SFML\lib`. So, the full path to type if you located your SFML folder on your D drive is `D:\SFML\lib`. Change your path if you installed SFML on a different drive.
14. Click **Apply** to save your configurations so far.
15. Next, still in the same window, perform these steps. Switch the **Configuration:** drop-down to **Debug** as we will be running and testing Pong in debug mode.
16. Select **Linker** and then **Input**.
17. Find the **Additional Dependencies** edit box and click into it on the far left-hand side. Now, copy and paste/type in the following: `sfml-graphics-d.lib;sfml-window-d.lib;sfml-system-d.lib;sfml-network-d.lib;sfml-audio-d.lib;`. Be extra careful to place the cursor exactly at the start of the edit box's current content so that you don't overwrite any of the text that is already there.
18. Click **OK**.
19. Click **Apply** and then **OK**.
20. Now, we need to copy the SFML .dll files into the main project directory. My main project directory is `D:\VS Projects\Pong`. It was created by Visual Studio in the previous steps. If you put your VS Projects folder somewhere else, then perform this step there instead. The files we need to copy into the project folder are located in our `SFML\bin` folder. Open a window for each of the two locations and highlight all the files in the `SFML\bin` folder.
21. Now, copy and paste the highlighted files into the project folder, that is, `D:\VS Projects\Pong`.

We now have the project properties configured and ready to go.

We will be displaying some text for a HUD (Heads Up Display) in this game that will show the player's score and remaining lives. For this, we need a font.



Download this free-for-personal-use font from <http://www.dafont.com/theme.php?cat=302> and unzip the download. Or feel free to use a font of your choice. You will just need to make some minor changes to the code when we load the font.

Create a new folder called `fonts` in the `VS Projects\Pong` folder and add the `DS-DIGIT.ttf` file into the `VS Projects\Pong\fonts` folder.

We are now ready to code our first C++ class.

Coding the Bat class

The simple Pong bat example was a good way of introducing the basics of classes. Classes can be simple and short, like the preceding `Bat` class, but they can also be longer and more complicated and contain other objects made from other classes.

When it comes to making games, there is a few vital things missing from the hypothetical `Bat` class. It might be fine for all these private member variables and public functions, but how will we draw anything? Our Pong bat needs a sprite, and in some games, they will also need a texture. Furthermore, we need a way to control the rate of animation of all our game objects, just like we did with the bee and the clouds in the previous project. We can include other objects in our class in exactly the same way that we included them in the `main.cpp` file. Let's code our `Bat` class for real so that we can see how we can solve all these issues.

Coding Bat.h

To get started, we will code the header file. Right-click on **Header Files** in the **Solution Explorer** window and select **ADD | New Item**. Next, choose the **Header File (.h)** option and name the new file `Bat.h`. Click the **Add** button. We are now ready to code the file.

Add the following code to `Bat.h`:

```
#pragma once
#include <SFML/Graphics.hpp>

using namespace sf;

class Bat
{
private:
    Vector2f m_Position;

    // A RectangleShape object
    RectangleShape m_Shape;

    float m_Speed = 1000.0f;
```

```
    bool m_MovingRight = false;
    bool m_MovingLeft = false;

public:
    Bat(float startX, float startY);

    FloatRect getPosition();

    RectangleShape getShape();

    void moveLeft();

    void moveRight();

    void stopLeft();

    void stopRight();

    void update(Time dt);

};
```

First, note the `#pragma once` declaration at the top of the file. This prevents the file from being processed by the compiler more than once. As our games get more complicated with perhaps dozens of classes, this will speed up compilation time.

Note the names of the member variables and the parameters and return types of the functions. We have a `Vector2f` called `m_Position`, which will hold the horizontal and vertical position of the player's bat. We also have an SFML `RectangleShape`, which will be the actual bat that appears on the screen.

There are two Boolean members that will track which direction, if any, the bat is currently moving in, and we have a `float` called `m_Speed` that tells us the number of pixels per second that the bat can move at when the player decides to move it left or right.

The next part of the code needs some explanation since we have a function called `Bat`; this is the exact same name as the class. This is called a constructor.

Constructor functions

When a class is coded, a special function is created by the compiler. We don't see this function in our code, but it is there. It is called a constructor. It is the function that would have been called if we used our hypothetical `Bat` class example.

When we need to write some code to prepare an object for use, often a good place to do this is in the constructor. When we want the constructor to do anything other than simply create an instance, we must replace the default (unseen) constructor provided by the compiler. This is what we will do with the `Bat` constructor function.

Notice that the `Bat` constructor takes two `float` parameters. This is perfect for initializing the position on the screen when we first create a `Bat` object. Also note that constructors have no return type, not even `void`.

We will soon use the constructor function, `Bat`, to put this game object into its starting position. Remember that this function is called at the time that an object of the `Bat` type is declared.

Continuing with the `Bat.h` explanation

Next is the `getPosition` function, which returns a `FloatRect`, the four points that define a rectangle. Then, we have `getShape`, which returns a `RectangleShape`. This will be used so that we can return to the main game loop, `m_Shape`, so that it can be drawn.

We also have the `moveLeft`, `moveRight`, `stopLeft`, and `stopRight` functions, which are for controlling if, when, and in which direction the bat will be in motion.

Finally, we have the `update` function, which takes a `Time` parameter. This function will be used to calculate how to move the bat each frame. As a bat and a ball will both move quite differently to each other, it makes sense to encapsulate the movement code inside the class. We will call the `update` function once each frame of the game from the `main` function.



You can probably guess that the `Ball` class will also have an `update` function.



Now, we can code `Bat.cpp`, which will implement all the definitions and use the member variables.

Coding Bat.cpp

Let's create the file, and then we can start discussing the code. Right-click the **Source Files** folder in the Solution Explorer window. Now, select **C++ File (.cpp)** and enter `Bat.cpp` in the **Name:** field. Click the **Add** button and our new file will be created for us.

We will divide the code for this file into two parts to make discussing it simpler.

First, code the `Bat` constructor function, as follows:

```
#include "Bat.h"

// This the constructor and it is called when we create an object
Bat::Bat(float startX, float startY)
{
    m_Position.x = startX;
    m_Position.y = startY;

    m_Shape.setSize(sf::Vector2f(50, 5));
    m_Shape.setPosition(m_Position);
}
```

In the preceding code, we can see that we include the `bat.h` file. This makes all the functions and variables that were declared previously in `bat.h` available to us.

We implement the constructor because we need to do some work to get the instance set up, and the default unseen empty constructor provided by the compiler is not sufficient. Remember that the constructor is the code that runs when we initialize an instance of `Bat`.

Notice that we use the `Bat::Bat` syntax as the function name to make it clear we are using the `Bat` function from the `Bat` class.

This constructor receives two `float` values, `startX` and `startY`. The next thing that happens is we assign these values to `m_Position.x` and `m_Position.y`. The `Vector2f` named `m_Position` now holds the values that were passed in, and because `m_Position` is a member variable, these values are accessible throughout the class. Note, however, that `m_Position` was declared as `private` and will not be accessible in our main function file—not directly, anyway. We will see how we can resolve this issue soon.

Finally, in the constructor, we initialize the `RectangleShape` called `m_Shape` by setting its size and position. This is different to how we coded the hypothetical `Bat` class in the *The theory of a Pong Bat* section. The SFML `Sprite` class has convenient variables for size and position that we can access using the `setSize` and `setPosition` functions, so we don't need the hypothetical `m_Length` and `m_Height` anymore.

Furthermore, note that we will need to vary how we initialize the `Bat` class (compared to the hypothetical `Bat` class) to suit our custom constructor.

We need to implement the remaining five functions of the `Bat` class. Add the following code to `Bat.cpp` after the constructor we just discussed:

```
FloatRect Bat::getPosition()
{
    return m_Shape.getGlobalBounds();
}

RectangleShape Bat::getShape()
{
    return m_Shape;
}

void Bat::moveLeft()
{
    m_MovingLeft = true;
}

void Bat::moveRight()
{
    m_MovingRight = true;
}

void Bat::stopLeft()
{
    m_MovingLeft = false;
}

void Bat::stopRight()
{
    m_MovingRight = false;
}

void Bat::update(Time dt)
{

```

```

        if (m_MovingLeft) {
            m_Position.x -= m_Speed * dt.asSeconds();
        }

        if (m_MovingRight) {
            m_Position.x += m_Speed * dt.asSeconds();
        }

        m_Shape.setPosition(m_Position);
    }

```

Let's go through the code we have just added.

First, we have the `getPosition` function. All it does is return a `FloatRect` to the code that called it. The `m_Shape.getGlobalBounds` line of code returns a `FloatRect` that is initialized with the coordinates of the four corners of the `RectangleShape`, that is, `m_Shape`. We will call this function from the `main` function when we are determining whether the ball has hit the bat.

Next, we have the `getShape` function. All this function does is pass a copy of `m_Shape` to the calling code. This is necessary so that we can draw the bat in the `main` function. When we code a public function with the sole purpose of passing back private data from a class, we call it a getter function.

Now, we can look at the `moveLeft`, `moveRight`, `stopLeft`, and `stopRight` functions. All they do is set the `m_MovingLeft` and `m_MovingRight` Boolean variables appropriately so that they keep track of the player's current intentions. Note, however, that they don't do anything to the `RectangleShape` instance or the `FloatRect` instance that determine the position. This is just what we need.

The last function in the `Bat` class is `update`. We will call this function once per frame of the game. The `update` function will grow in complexity as our game projects get more complicated. For now, all we need to do is tweak `m_Position`, depending on whether the player is moving left or right. Note that the formula that's used to do this tweak is the same one that we used for updating the bee and the clouds in the `Timber!!!` project. The code multiplies the speed by the delta time and then adds or subtracts it from the position. This causes the bat to move relative to how long the frame took to update. Next, the code sets the position of `m_Shape` with whatever the latest values held in `m_Position` happen to be.

Having an `update` function in our `Bat` class rather than the `main` function is encapsulation. Rather than updating the positions of all the game objects in the `main` function like we did in the `Timber!!!` project, each object will be responsible for updating themselves. As we will do next, however, we will call this `update` function from the `main` function.

Using the Bat class and coding the main function

Switch to the `main.cpp` file that was automatically generated when we created the project. Delete all its auto-generated code and add the code that follows.

Code the `Pong.cpp` file as follows:

```
#include "Bat.h"
#include <sstream>
#include <cstdlib>
#include <SFML/Graphics.hpp>

int main()
{
    // Create a video mode object
    VideoMode vm(1920, 1080);

    // Create and open a window for the game

    RenderWindow window(vm, "Pong", Style::Fullscreen);

    int score = 0;
    int lives = 3;

    // Create a bat at the bottom center of the screen
    Bat bat(1920 / 2, 1080 - 20);

    // We will add a ball in the next chapter

    // Create a Text object called HUD
    Text hud;

    // A cool retro-style font
    Font font;
    font.loadFromFile("fonts/DS-DIGI.ttf");

    // Set the font to our retro-style
    hud.setFont(font);
```

```
// Make it nice and big
hud.setCharacterSize(75);

// Choose a color
hud.setFillColor(Color::White);

hud.setPosition(20, 20);

// Here is our clock for timing everything
Clock clock;

while (window.isOpen())
{
    /*
    Handle the player input
    *****
    *****
    *****
    */

    /*
    Update the bat, the ball and the HUD
    *****
    *****
    *****
    */

    /*
    Draw the bat, the ball and the HUD
    *****
    *****
    *****
    */

}

return 0;
}
```

In the preceding code, the structure is similar to the one we used in the Timber!!! project. The first exception, however, is when we create an instance of the `Bat` class:

```
// Create a bat
Bat bat(1920 / 2, 1080 - 20);
```

The preceding code calls the constructor function to create a new instance of the `Bat` class. The code passes in the required arguments and allows the `Bat` class to initialize its position in the center of the screen near the bottom. This is the perfect position for our bat to start.

Also note that I have used comments to indicate where the rest of the code will eventually be placed. It is all within the game loop, just like it was in the Timber!!! project. Here is where the rest of the code will go again, just to remind you:

```
/*
    Handle the player input
    ...

/*
    Update the bat, the ball and the HUD
    ...

/*
    Draw the bat, the ball and the HUD
    ...
```

Next, add the code to the `Handle the player input` section, as follows:

```
Event event;
while (window.pollEvent(event))
{
    if (event.type == Event::Closed)
        // Quit the game when the window is closed
        window.close();
}

// Handle the player quitting
if (Keyboard::isKeyPressed(Keyboard::Escape))
{
    window.close();
}
```

```
// Handle the pressing and releasing of the arrow keys
if (Keyboard::isKeyPressed(Keyboard::Left))
{
    bat.moveLeft();
}
else
{
    bat.stopLeft();
}

if (Keyboard::isKeyPressed(Keyboard::Right))
{
    bat.moveRight();
}
else
{
    bat.stopRight();
}
```

The preceding code handles the player quitting the game by pressing the *Escape* key, exactly like it did in the Timber!!! project. Next, there are two `if - else` structures that handle the player moving the bat. Let's analyze the first of these two structures:

```
if (Keyboard::isKeyPressed(Keyboard::Left))
{
    bat.moveLeft();
}
else
{
    bat.stopLeft();
}
```

The preceding code will detect whether the player is holding down the left arrow cursor key on the keyboard. If they are, then the `moveLeft` function is called on the `Bat` instance. When this function is called, the `true` value is set to the `m_MovingLeft` private Boolean variable. If, however, the left arrow key is not being held down, then the `stopLeft` function is called and the `m_MovingLeft` is set to `false`.

The exact same process is then repeated in the next `if - else` block of code to handle the player pressing (or not pressing) the right arrow key.

Next, add the following code to the `Update` the bat the ball and the HUD section, as follows:

```
// Update the delta time
Time dt = clock.restart();
```

```
bat.update(dt);  
// Update the HUD text  
std::stringstream ss;  
ss << "Score:" << score << "  Lives:" << lives;  
hud.setString(ss.str());
```

In the preceding code, we use the exact same timing technique that we used for the Timber!!! project, only this time, we call `update` on the `Bat` instance and pass in the delta time. Remember that, when the `Bat` class receives the delta time, it will use the value to move the bat based on the previously received movement instructions from the player and the desired speed of the bat.

Next, add the following code to the `Draw` the bat, the ball and the HUD section, as follows:

```
window.clear();  
window.draw(hud);  
window.draw(bat.getShape());  
window.display();
```

In the preceding code, we clear the screen, draw the text for the HUD, and use the `bat.getShape` function to grab the `RectangleShape` instance from the `Bat` instance and draw it to the screen. Finally, we call `window.display`, just like we did in the previous project, to draw the bat in its current position.

At this stage, you can run the game and you will see the HUD and a bat. The bat can be moved smoothly left and right using the arrow/cursor keys:



Congratulations! That is the first class, all coded and deployed.

Summary

In this chapter, we discovered the basics of OOP, such as how to code and use a class, including making use of encapsulation to control how code outside of our classes can access the member variables, but only to the extent and in the manner that we want it to. This is just like SFML classes, which allow us to create and use `Sprite` and `Text` instances, but only in the way they were designed to be used.

Don't concern yourself too much if some of the details around OOP and classes are not entirely clear. The reason I say this is because we will spend the rest of this book coding classes and the more we use them, the clearer they will become.

Furthermore, we have a working bat and a HUD for our Pong game.

In the next chapter, we will code the `Ball` class and get it bouncing around the screen. We will then be able to add collision detection and finish the game.

FAQ

Q) I have learned other languages and OOP seems much simpler in C++. Is this a correct assessment?

A) This was an introduction to OOP and its basic fundamentals. There is more to it than this. We will learn about more OOP concepts and details throughout this book.

7

Dynamic Collision Detection and Physics – Finishing the Pong Game

In this chapter, we will code our second class. We will see that although the ball is obviously quite different from the bat, we will use the exact same techniques to encapsulate the appearance and functionality of a ball inside a `Ball` class, just like we did with the bat and the `Bat` class. We will then add the finishing touches to the Pong game by coding some dynamic collision detection and scorekeeping. This might sound complicated but as we are coming to expect, SFML will make things much easier than they otherwise would be.

We will cover the following topics in this chapter:

- Coding the `Ball` class
- Using the `Ball` class
- Collision detection and scoring
- Running the game

We will start by coding the class that represents the ball.

Coding the `Ball` class

To get started, we will code the header file. Right-click on **Header Files** in the Solution Explorer window and select **ADD | New Item**. Next, choose the **Header File (.h)** option and name the new file `Ball.h`. Click the **Add** button. Now, we are ready to code the file.

Add the following code to `Ball.h`:

```
#pragma once
#include <SFML/Graphics.hpp>

using namespace sf;

class Ball
{
private:
    Vector2f m_Position;
    RectangleShape m_Shape;

    float m_Speed = 300.0f;
    float m_DirectionX = .2f;
    float m_DirectionY = .2f;

public:
    Ball(float startX, float startY);

    FloatRect getPosition();

    RectangleShape getShape();

    float getXVelocity();

    void reboundSides();

    void reboundBatOrTop();

    void reboundBottom();

    void update(Time dt);

};
```

The first thing you will notice is the similarity in the member variables compared to the `Bat` class. There is a member variable for the position, appearance, and speed, just like there was for the player's bat, and they are the same types (`Vector2f`, `RectangleShape`, and `float`, respectively). They even have the same names (`m_Position`, `m_Shape`, and `m_Speed`, respectively). The difference between the member variables of this class is that the direction is handled with two `float` variables that will track horizontal and vertical movement. These are `m_DirectionX` and `m_DirectionY`.

Note that we will need to code eight functions to bring the ball to life. There is a constructor that has the same name as the class, which we will use to initialize a `Ball` instance. There are three functions with the same name and usage as the `Bat` class. They are `getPosition`, `getShape`, and `update`. The `getPosition` and `getShape` functions will share the location and the appearance of the ball with the `main` function, and the `update` function will be called from the `main` function to allow the `Ball` class to update its position once per frame.

The remaining functions control the direction the ball will travel in. The `reboundSides` function will be called from `main` when a collision is detected with either side of the screen, the `reboundBatOrTop` function will be called in response to the ball hitting the player's bat or the top of the screen, and the `reboundBottom` function will be called when the ball hits the bottom of the screen.

Of course, these are just the declarations, so let's write the C++ that actually does the work in the `Ball.cpp` file.

Let's create the file, and then we can start discussing the code. Right-click the **Source Files** folder in the Solution Explorer window. Now, select **C++ File (.cpp)** and enter `Ball.cpp` in the **Name:** field. Click the **Add** button and our new file will be created for us.

Add the following code to `Ball.cpp`:

```
#include "Ball.h"

// This the constructor function
Ball::Ball(float startX, float startY)
{
    m_Position.x = startX;
    m_Position.y = startY;

    m_Shape.setSize(sf::Vector2f(10, 10));
    m_Shape.setPosition(m_Position);
}
```

In the preceding code, we have added the required `include` directive for the `Ball` class header file. The constructor function with the same name as the class receives two `float` parameters, which are used to initialize the `m_Position` member's `Vector2f` instance. The `RectangleShape` instance is then sized with the `setSize` function and positioned with `setPosition`. The size that's being used is 10 pixels wide and 10 high; this is arbitrary but works well. The position that's being used is, of course, taken from the `m_Position` `Vector2f` instance.

Add the following code underneath the constructor in the `Ball.cpp` function:

```
FloatRect Ball::getPosition()
{
    return m_Shape.getGlobalBounds();
}

RectangleShape Ball::getShape()
{
    return m_Shape;
}

float Ball::getXVelocity()
{
    return m_DirectionX;
}
```

In the preceding code, we are coding the three getter functions of the `Ball` class. They each return something to the main function. The first, `getPosition`, uses the `getGlobalBounds` function on `m_Shape` to return a `FloatRect` instance. This will be used for collision detection.

The `getShape` function returns `m_Shape` so that it can be drawn each frame of the game loop. The `getXVelocity` function tells the main function which direction the ball is traveling in, and we will see very soon exactly how this is useful to us. Since we don't ever need to get the vertical velocity, there is no corresponding `getYVelocity` function, but it would be simple to add one if we did.

Add the following functions underneath the previous code we just added:

```
void Ball::reboundSides()
{
    m_DirectionX = -m_DirectionX;
}

void Ball::reboundBatOrTop()
{
    m_DirectionY = -m_DirectionY;
}

void Ball::reboundBottom()
{
    m_Position.y = 0;
    m_Position.x = 500;
    m_DirectionY = -m_DirectionY;
}
```

In the preceding code, the three functions whose names begin with `rebound...` handle what happens when the ball collides with various places. In the `reboundSides` function, `m_DirectionX` has its value inverted, which will have the effect of making a positive value negative and a negative value positive, thereby reversing (horizontally) the direction the ball is traveling in. `reboundBatOrTop` does exactly the same but with `m_DirectionY`, which has the effect of reversing the direction the ball is traveling in vertically. The `reboundBottom` function repositions the ball at the top center of the screen and sends it downward. This is just what we want after the player has missed a ball and it has hit the bottom of the screen.

Finally, for the `Ball` class, add the update function, as follows:

```
void Ball::update(Time dt)
{
    // Update the ball's position
    m_Position.y += m_DirectionY * m_Speed * dt.asSeconds();
    m_Position.x += m_DirectionX * m_Speed * dt.asSeconds();

    // Move the ball
    m_Shape.setPosition(m_Position);
}
```

In the preceding code, `m_Position.y` and `m_Position.x` are updated using the appropriate direction velocity, the speed, and the amount of time the current frame took to complete. The newly updated `m_Position` values are then used to change the position the `m_Shape` `RectangleShape` instance is positioned at.

The `Ball` class is done, so let's put it into action.

Using the Ball class

To put the ball into action, add the following code to make the `Ball` class available in the main function:

```
#include "Ball.h"
```

Add the following highlighted line of code to declare and initialize an instance of the `Ball` class using the constructor function that we have just coded:

```
// Create a bat
Bat bat(1920 / 2, 1080 - 20);

// Create a ball
```

```
Ball ball(1920 / 2, 0);

// Create a Text object called HUD
Text hud;
```

Add the following code positioned exactly as highlighted:

```
/*
Update the bat, the ball and the HUD
*****
*****
*****
*/
// Update the delta time
Time dt = clock.restart();
bat.update(dt);
ball.update(dt);
// Update the HUD text
std::stringstream ss;
ss << "Score:" << score << "    Lives:" << lives;
hud.setString(ss.str());
```

In the preceding code, we simply call `update` on the `ball` instance. The ball will be repositioned accordingly.

Add the following highlighted code to draw the ball on each frame of the game loop:

```
/*
Draw the bat, the ball and the HUD
*****
*****
*****
*/
window.clear();
window.draw(hud);
window.draw(bat.getShape());
window.draw(ball.getShape());
window.display();
```

At this stage, you could run the game and the ball would spawn at the top of the screen and begin its descent toward the bottom of the screen. It would, however, disappear off the bottom of the screen because we are not detecting any collisions yet. Let's fix that now.

Collision detection and scoring

Unlike in the Timber!!! game when we simply checked whether a branch in the lowest position was on the same side as the player's character, in this game, we will need to mathematically check for the intersection of the ball with the bat or the ball with any of the four sides of the screen.

Let's look at some hypothetical code that would achieve this so that we understand what we are doing. Then, we will turn to SFML to solve the problem for us.

The code for testing the intersection of two rectangles would look something like this. Don't use the following code. It is for demonstration purposes only:

```
if(objectA.getPosition().right > objectB.getPosition().left
    && objectA.getPosition().left < objectB.getPosition().right )
{
    // objectA is intersecting objectB on x axis
    // But they could be at different heights

    if(objectA.getPosition().top < objectB.getPosition().bottom
        && objectA.getPosition().bottom > objectB.getPosition().top )
    {
        // objectA is intersecting objectB on y axis as well
        // Collision detected
    }
}
```

We don't need to write this code; however, we will be using the SFML `intersects` function, which works on `FloatRect` objects. Think or look back to the `Bat` and `Ball` classes; they both had a `getPosition` function, which returned a `FloatRect` of the object's current location. We will see how we can use `getPosition`, along with `intersects`, to do all our collision detection.

Add the following highlighted code at the end of the update section of the main function:

```
/*
Update the bat, the ball and the HUD
*****
*****
*****
*/
// Update the delta time
Time dt = clock.restart();
bat.update(dt);
```



```
ball.update(dt);
// Update the HUD text
std::stringstream ss;
ss << "Score:" << score << "    Lives:" << lives;
hud.setString(ss.str());

// Handle ball hitting the bottom
if (ball.getPosition().top > window.getSize().y)
{
    // reverse the ball direction
    ball.reboundBottom();

    // Remove a life
    lives--;

    // Check for zero lives
    if (lives < 1) {
        // reset the score
        score = 0;
        // reset the lives
        lives = 3;
    }
}
```

In the preceding code, the first `if` condition checks whether the ball has hit the bottom of the screen:

```
if (ball.getPosition().top > window.getSize().y)
```

If the top of the ball is at a greater position than the height of the window, then the ball has disappeared off the bottom of the player's view. In response, the `ball.reboundBottom` function is called. Remember that, in this function, the ball is repositioned at the top of the screen. At this point, the player has lost a life, so the `lives` variable is decremented.

The second `if` condition checks whether the player has run out of lives (`lives < 1`). If this is the case, the score is reset to 0, the number of lives is reset to 3, and the game is restarted. In the next project, we will learn how to keep and display the player's highest score.

Add the following code underneath the previous code:

```
// Handle ball hitting top
if (ball.getPosition().top < 0)
{
    ball.reboundBatOrTop();

    // Add a point to the players score
    score++;
}
```

In the preceding code, we are detecting that the top of the ball hits the top of the screen. When this occurs, the player is awarded a point and `ball.reboundBatOrTop` is called, which reverses the vertical direction of travel and sends the ball back toward the bottom of the screen.

Add the following code underneath the previous code:

```
// Handle ball hitting sides
if (ball.getPosition().left < 0 ||
    ball.getPosition().left + ball.getPosition().width > window.
    getSize().x)
{
    ball.reboundSides();
}
```

In the preceding code, the `if` condition detects a collision with the left-hand side of the ball with the left-hand side of the screen or the right-hand side of the ball (`left + 10`) with the right-hand side of the screen. In either event, the `ball.reboundSides` function is called and the horizontal direction of travel is reversed.

Add the following code:

```
// Has the ball hit the bat?
if (ball.getPosition().intersects(bat.getPosition()))
{
    // Hit detected so reverse the ball and score a point
    ball.reboundBatOrTop();
}
```

In the preceding code, the `intersects` function is used to determine whether the ball has hit the bat. When this occurs, we use the same function that we used for a collision with the top of the screen to reverse the vertical direction of travel of the ball.

Running the game

You can now run the game and bounce the ball around the screen. The score will increase when you hit the ball with the bat and the lives will decrease when you miss it. When `lives` gets to 0, the score will reset, and the `lives` will go back up to 3, as follows:



Summary

Congratulations; that's the second game completed! We could have added more features to that game such as coop play, high scores, sound effects, and more, but I just wanted to use the simplest possible example to introduce classes and dynamic collision detection. Now that we have these topics in our game developer's arsenal, we can move on to a much more exciting project and yet more game development topics.

In the next chapter, we will plan the Zombie Arena game, learn about the SFML `View` class, which acts as a virtual camera into our game world, and code some more classes.

FAQ

Q) Isn't this game a little quiet?

A) I didn't add sound effects to this game because I wanted to keep the code as short as possible while using our first classes and learning to use the time to smoothly animate all the game objects. If you want to add sound effects, then all you need to do is add the `.wav` files to the project, use SFML to load the sounds, and play a sound effect in each of the collision events. We will do this in the next project.

Q) The game is too easy! How can I make the ball speed up a little?

A) There are lots of ways you can make the game more challenging. One simple way would be to add a line of code in the `Ball` class' `reboundBatOrTop` function that increases the speed. As an example, the following code would increase the speed of the ball by 10% each time the function is called:

```
// Speed up a little bit on each hit
m_Speed = m_Speed * 1.1f;
```

The ball would get quite fast quite quickly. You would then need to devise a way to reset the speed back to `300.0f` when the player has lost all their lives. You could create a new function in the `Ball` class, perhaps called `resetSpeed`, and call it from `main` when the code detects that the player has lost their last life.

8

SFML Views – Starting the Zombie Shooter Game

In this project, we will be making even more use of **OOP** and to a powerful effect. We will also be exploring the SFML `View` class. This versatile class will allow us to easily divide our game up into layers for different aspects of the game. In the Zombie Shooter project, we will have a layer for the HUD and a layer for the main game. This is necessary because as the game world expands each time the player clears a wave of zombies and, eventually, the game world will be bigger than the screen and will need to scroll. The use of the `View` class will prevent the text of the HUD from scrolling with the background. In the next project, we will take things even further and create a co-op split screen game with the SFML `View` class doing most of the hard work.

This is what we will do in this chapter:

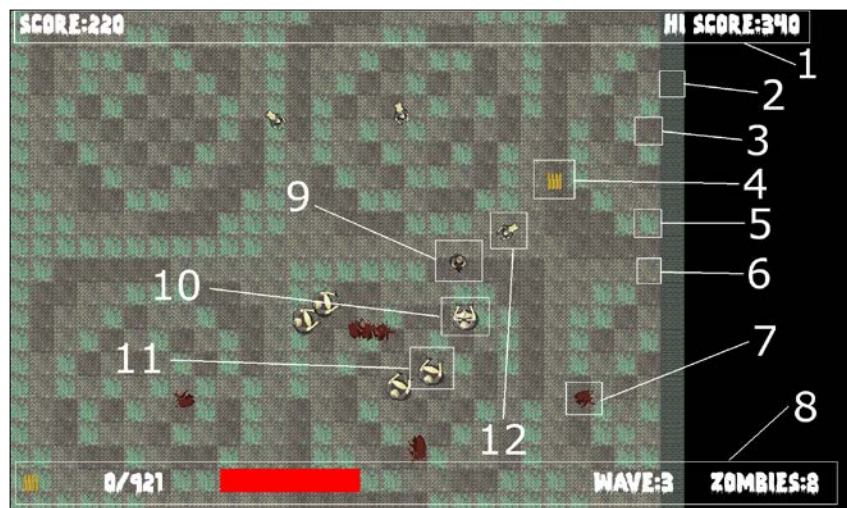
- Planning and starting the Zombie Arena game
- Coding the `Player` class
- Learning about the SFML `View` class
- Building the Zombie Arena game engine
- Putting the `Player` class to work

Planning and starting the Zombie Arena game

At this point, if you haven't already, I suggest you go and watch a video of *Over 9000 Zombies* (<http://store.steampowered.com/app/273500/>) and *Crimson Land* (<http://store.steampowered.com/app/262830/>). Our game will obviously not be as in-depth or advanced as either of these examples, but we will also have the same basic set of features and game mechanics, such as the following:

- A Heads Up Display (HUD) that shows details such as the score, high score, and bullets in clip, the number of bullets left, player health, and the number of zombies left to kill.
- The player will shoot zombies while frantically running away from them.
- Move around a scrolling world using the *WASD* keyboard keys while aiming the gun using the mouse.
- In-between each level, the player will choose a "level-up" that will affect the way the game needs to be played for the player to win.
- The player will need to collect "pick-ups" to restore health and ammunition.
- Each wave brings more zombies and a bigger arena to make it more challenging.

There will be three types of zombies to splatter. They will have different attributes, such as appearance, health, and speed. We will call them chasers, bloaters, and crawlers. Take a look at the following annotated screenshot of the game to see some of the features in action and the components and assets that make up the game:



Here is some more information about each of the numbered points:

1. The score and hi-score. These, along with the other parts of the HUD, will be drawn in a separate layer, known as a view, and represented by an instance of the `View` class. The hi-score will be saved and loaded to a file.
2. A texture that will build a wall around the arena. This texture is contained in a single graphic called a **sprite sheet**, along with the other background textures (points 3, 5, and 6).
3. The first of two mud textures from the sprite sheet.
4. This is an "ammo pick-up." When the player gets this, they will be given more ammunition. There is a "health pick-up" as well, from which the player will receive more health. These pick-ups can be chosen by the player to be upgraded in-between waves of zombies.
5. A grass texture, also from the sprite sheet.
6. The second mud texture from the sprite sheet.
7. A blood splat where there used to be a zombie.
8. The bottom part of the HUD. From left to right, there is an icon to represent ammo, the number of bullets in the clip, the number of spare bullets, a health bar, the current wave of zombies, and the number of zombies remaining for the current wave.
9. The player's character.
10. A crosshair, which the player aims with the mouse.
11. A slow-moving, but strong, "bloater" zombie.
12. A slightly faster-moving, but weaker, "crawler" zombie. There is also a "chaser zombie" that is very fast and weak. Unfortunately, I couldn't manage to get one in the screenshot before they were all killed.

So, we have a lot to do and new C++ skills to learn. Let's start by creating a new project.

Creating a new project

As creating a project is a relatively involved process, I will detail all the steps again. For even more detail and images, please refer to the *Setting up the Timber project* section in *Chapter 1, C++, SFML, Visual Studio, and Starting the First Game*.

As setting up a project is a fiddly process, we will go through it step by step, like we did for the Timber project. I won't show you the same images as I did for the Timber project, but the process is the same, so flip back to *Chapter 1, C++, SFML, Visual Studio, and Starting the First Game* if you want a reminder of the locations of the various project properties. Let's look at the following steps:

1. Start Visual Studio and click on the **Create New Project** button. If you have another project open, you can select **File | New project**.
2. In the window shown next, choose **Console app** and click on the **Next** button. You will then see the **Configure your new project** window.
3. In the **Configure your new project** window, type `Zombie Arena` in the **Project name** field.
4. In the **Location** field, browse to the `VS Projects` folder.
5. Check the option to **Place solution and project in the same directory**.
6. When you have completed the preceding steps, click on **Create**.
7. We will now configure the project to use the SFML files that we put in the SFML folder. From the main menu, select **Project | Zombie Arena properties....** At this stage, you should have the **Zombie Arena Property Pages** window open.
8. In the **Zombie Arena Property Pages** window, take the following steps. Select **All Configurations** from the **Configuration:** dropdown menu.
9. Now, select **C/C++** and then **General** from the left-hand menu.
10. Next, locate the **Additional Include Directories** edit box and type the drive letter where your SFML folder is located, followed by `\SFML\include`. The full path to type, if you located your SFML folder on your D drive, will be `D:\SFML\include`. Vary your path if you installed SFML on a different drive.
11. Click on **Apply** to save your configurations so far.
12. Now, still in the same window, perform these next steps. From the left-hand menu, select **Linker** and then **General**.
13. Now, find the **Additional Library Directories** edit box and type the drive letter where your SFML folder is, followed by `\SFML\lib`. So, the full path to type, if you located your SFML folder on your D drive, will be `D:\SFML\lib`. Change your path if you installed SFML to a different drive.
14. Click on **Apply** to save your configurations so far.
15. Next, still in the same window, perform these steps. Switch the **Configuration:** dropdown menu to **Debug** as we will be running and testing Pong in debug mode.
16. Select **Linker** and then **Input**.

17. Find the **Additional Dependencies** edit box and click on it in the far left-hand side. Now copy and paste/type the following: `sfml-graphics-d.lib;sfml-window-d.lib;sfml-system-d.lib;sfml-network-d.lib;sfml-audio-d.lib;`. Be extra careful to place the cursor exactly at the start of the edit box's current content so as not to overwrite any of the text that is already there.
18. Click on **OK**.
19. Click on **Apply** and then **OK**.

Now, you have configured the project properties and you are nearly ready to go. Next, we need to copy the SFML .dll files into the main project directory by following these steps:

1. My main project directory is `D:\VS Projects\Zombie Arena`. This folder was created by Visual Studio in the previous steps. If you put your Projects folder somewhere else, then perform this step in your directory. The files we need to copy into the project folder are in your `SFML\bin` folder. Open a window for each of the two locations and highlight all the .dll files.
2. Now, copy and paste the highlighted files into the project.

The project is now set up and ready to go. Next, we will explore and add the project assets.

The project assets

The assets in this project are more numerous and more diverse than the previous games. The assets include the following:

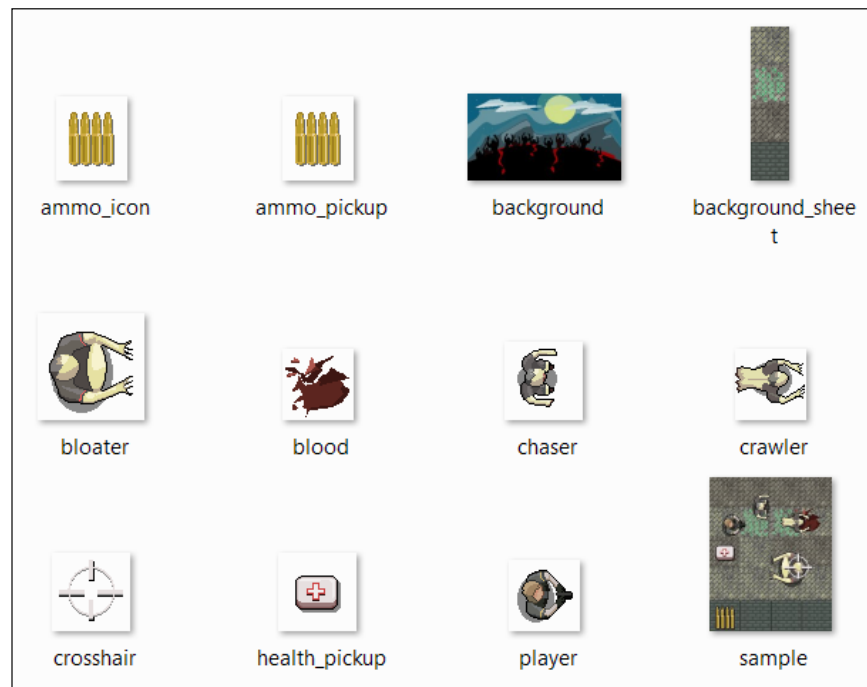
- A font for the text on the screen
- Sound effects for different actions such as shooting, reloading, or getting hit by a zombie
- Graphics for the character, zombies, and a sprite sheet for the various background textures

All the graphics and sound effects that are required for the game are included in the download bundle. They can be found in the `Chapter 8/graphics` and `Chapter 8/sound` folders, respectively.

The font that is required has not been supplied. This is done to avoid any possible ambiguity regarding the license. This will not cause a problem because the links for downloading the fonts and how and where to choose the font will be provided.

Exploring the assets

The graphical assets make up the parts of the scene of our Zombie Arena game. Look at the following graphical assets; it should be clear to you where the assets in the game will be used:



What might be less obvious, however, is the `background_sheet.png` file, which contains four different images. This is the sprite sheet we mentioned previously. We will see how we can save memory and increase the speed of our game using the sprite sheet in *Chapter 9, C++ References, Sprite Sheets, and Vertex Arrays*.

The sound files are all in `.wav` format. These are files that contain the sound effects that will be played when certain events are triggered. They are as follows:

- `hit.wav`: A sound that plays when a zombie comes into contact with the player.
- `pickup.wav`: A sound that plays when the player collides or steps on (collects) a health boost (pick-up).
- `powerup.wav`: A sound for when the player chooses an attribute to increase their strength (power-up) in-between each wave of zombies.

- `reload.wav`: A satisfying click to let the player know they have loaded a fresh clip of ammunition.
- `reload_failed.wav`: A less satisfying sound that indicates failing to load new bullets.
- `shoot.wav`: A shooting sound.
- `splat.wav`: A sound like a zombie being hit by a bullet.

Once you have decided which assets you will use, it is time to add them to the project.

Adding the assets to the project

The following instructions will assume you are using all the assets that were supplied in the book's download bundle. Where you are using your own assets, simply replace the appropriate sound or graphic file with your own, using the same filename. Let's take a look at the steps:

1. Browse to `D:\VS Projects\ZombieArena`.
2. Create three new folders within this folder and name them `graphics`, `sound`, and `fonts`.
3. From the download bundle, copy the entire contents of Chapter 8/`graphics` into the `D:\VS Projects\ZombieArena\graphics` folder.
4. From the download bundle, copy the entire contents of Chapter 6/`sound` into the `D:\VS Projects\ZombieArena\sound` folder.
5. Now, visit http://www.1001freefonts.com/zombie_control.font in your web browser and download the **Zombie Control** font.
6. Extract the contents of the zipped download and add the `zombiecontrol.ttf` file to the `D:\VS Projects\ZombieArena\fonts` folder.

Now, it's time to consider how OOP will help us with this project and then we can start writing the code for Zombie Arena.

OOP and the Zombie Arena project

The initial problem we are faced with is the complexity of the current project. Let's consider that there is just a single zombie; here is what we need to make it function in the game:

- Its horizontal and vertical position
- Its size

- The direction it is facing
- A different texture for each zombie type
- A Sprite
- A different speed for each zombie type
- A different health for each zombie type
- Keeping track of the type of each zombie
- Collision detection data
- Its intelligence (to chase the player), which is slightly different for each type of zombie
- An indication of whether the zombie is alive or dead

This suggests perhaps a dozen variables for just one zombie, and entire arrays of each of these variables will be required for managing a zombie horde. But what about all the bullets from the machine gun, the pick-ups, and the different level-ups? The simple Timber!!! and Pong games also started to get a bit unmanageable, and it is easy to speculate that this more complicated shooter will be many times worse!

Fortunately, we will put all the OOP skills we learned in the previous two chapters into action, as well as learn some new C++ techniques.

We will start our coding for this project with a class to represent the player.

Building the player – the first class

Let's think about what our `Player` class will need to do and what we require for it. The class will need to *know* how fast it can move, where in the game world it currently is, and how much health it has. As the `Player` class, in the player's eyes, is represented as a 2D graphical character, the class will need both a `Sprite` object and a `Texture` object.

Furthermore, although the reasons might not be obvious at this point, our `Player` class will also benefit from knowing a few details about the overall environment the game is running in. These details are screen resolution, the size of the tiles that make up an arena, and the overall size of the current arena.

As the `Player` class will be taking full responsibility for updating itself in each frame (like the bat and ball did), it will need to know the player's intentions at any given moment. For example, is the player currently holding down a keyboard direction key? Or is the player currently holding down multiple keyboard direction keys? Boolean variables are used to determine the status of the *W*, *A*, *S*, and *D* keys and will be essential.

It is clear that we are going to need quite a selection of variables in our new class. Having learned all we have about OOP, we will, of course, be making all of these variables private. This means that we must provide access, where appropriate, from the `main` function.

We will use a whole bunch of getter functions as well as some functions to set up our object. These functions are quite numerous. There are 21 functions in this class. At first, this might seem a little daunting, but we will go through them all and see that most of them simply set or get one of the private variables.

There are just a few in-depth functions: `update`, which will be called once each frame from the `main` function, and `spawn`, which will handle initializing some of the private variables each time the player is spawned. As we will see, however, there is nothing complicated and they will all be described in detail.

The best way to proceed is to code the header file. This will give us the opportunity to see all the private variables and examine all the function signatures.



Pay close attention to the return values and argument types, as this will make understanding the code in the function definitions much easier.

Coding the Player class header file

Start by right-clicking on **Header Files** in **Solution Explorer** and select **Add | New Item....** In the **Add New Item** window, highlight (by left-clicking on it) **Header File (.h)** and then, in the **Name** field, type `Player.h`. Finally, click on the **Add** button. We are now ready to code the header file for our first class.

Start coding the `Player` class by adding the declaration, including the opening and closing curly braces, followed by a semicolon:

```
#pragma once
#include <SFML/Graphics.hpp>

using namespace sf;

class Player
{

};
```

Now, let's add all our private member variables in the file. Based on what we have already discussed, see whether you can work out what each of them will do. We will go through them individually in a moment:

```
class Player
{
private:
    const float START_SPEED = 200;
    const float START_HEALTH = 100;

    // Where is the player
    Vector2f m_Position;

    // Of course, we will need a sprite
    Sprite m_Sprite;

    // And a texture
    // !!Watch this space - Interesting changes here soon!!
    Texture m_Texture;

    // What is the screen resolution
    Vector2f m_Resolution;

    // What size is the current arena
    IntRect m_Arena;

    // How big is each tile of the arena
    int m_TileSize;

    // Which direction(s) is the player currently moving in
    bool m_UpPressed;
    bool m_DownPressed;
    bool m_LeftPressed;
    bool m_RightPressed;

    // How much health has the player got?
    int m_Health;
    // What is the maximum health the player can have
    int m_MaxHealth;

    // When was the player last hit
    Time m_LastHit;

    // Speed in pixels per second
```

```
float m_Speed;

// All our public functions will come next

};
```

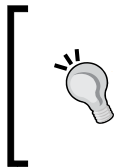
The previous code declares all our member variables. Some are regular variables, while some of them are objects. Notice that they are all under the `private:` section of the class and, therefore, are not directly accessible from outside the class.

Also, notice that we are using the naming convention of prefixing `m_` to all the names of the non-constant variables. The `m_` prefix will remind us, while coding the function definitions, that they are member variables, are distinct from the local variables we will create in some of the functions, and are also distinct from the function parameters.

All the variables that are used are straightforward, such as `m_Position`, `m_Texture`, and `m_Sprite`, which are for the current location, texture, and sprite of the player, respectively. In addition to this, each variable (or group of variables) is commented to make its usage plain.

However, why exactly they are needed, and the context they will be used in, might not be so obvious. For example, `m_LastHit`, which is an object of the `Time` type, is for recording the time that the player last received a hit from a zombie. It is not obvious *why* we might need this information, but we will go over this soon.

As we piece the rest of the game together, the context for each of the variables will become clearer. The important thing, for now, is to familiarize yourself with the names and data types to make following along with the rest of the project trouble-free.



You don't need to memorize the variable names and types as we will discuss all the code when they are used. You do, however, need to take your time to look over them and get more familiar with them. Furthermore, as we proceed, it might be worth referring to this header file if anything seems unclear.

Now, we can add a complete long list of functions. Add the following highlighted code and see whether you can work out what it all does. Pay close attention to the return types, parameters, and the name of each function. This is key to understanding the code we will write throughout the rest of this project. What do they tell us about each function? Add the following highlighted code and then we will examine it:

```
// All our public functions will come next
public:

    Player();

    void spawn(IntRect arena, Vector2f resolution, int tileSize);

    // Call this at the end of every game
    void resetPlayerStats();

    // Handle the player getting hit by a zombie
    bool hit(Time timeHit);

    // How long ago was the player last hit
    Time getLastHitTime();

    // Where is the player
    FloatRect getPosition();

    // Where is the center of the player
    Vector2f getCenter();

    // What angle is the player facing
    float getRotation();

    // Send a copy of the sprite to the main function
    Sprite getSprite();

    // The next four functions move the player
    void moveLeft();

    void moveRight();

    void moveUp();

    void moveDown();
```

```

    // Stop the player moving in a specific direction
    void stopLeft();

    void stopRight();

    void stopUp();

    void stopDown();

    // We will call this function once every frame
    void update(float elapsedTime, Vector2i mousePosition);

    // Give the player a speed boost
    void upgradeSpeed();

    // Give the player some health
    void upgradeHealth();

    // Increase the maximum amount of health the player can have
    void increaseHealthLevel(int amount);

    // How much health has the player currently got?
    int getHealth();
};

```

Firstly, note that all the functions are public. This means we can call all of these functions using an instance of the class from the `main` function with code like this:

```
player.getSprite();
```

Assuming `player` is a fully set up instance of the `Player` class, the previous code will return a copy of `m_Sprite`. Putting this code into a real context, we could, in the `main` function, write code like this:

```
window.draw(player.getSprite());
```

The previous code would draw the player graphic in its correct location, just as if the sprite was declared in the `main` function itself. This is what we did with the `Bat` class in the Pong project.

Before we move on to implement (that is, write the definitions) of these functions in a corresponding `.cpp` file, let's take a closer look at each of them in turn:

- `void spawn(IntRect arena, Vector2f resolution, int tileSize):` This function does what its name suggests. It will prepare the object ready for use, which includes putting it in its starting location (that is, spawning it). Notice that it doesn't return any data, but it does have three arguments. It receives an `IntRect` instance called `arena`, which will be the size and location of the current level; a `Vector2f` instance, which will contain the screen resolution; and an `int`, which will hold the size of a background tile.
- `void resetPlayerStats:` Once we give the player the ability to level up between waves, we will need to be able to take away/reset those abilities at the start of a new game.
- `Time getLastHitTime():` This function does just one thing – it returns the time when the player was last hit by a zombie. We will use this function when detecting collisions, and it will allow us to make sure the player isn't punished *too* frequently for making contact with a zombie.
- `FloatRect getPosition():` This function returns a `FloatRect` instance that describes the horizontal and vertical floating-point coordinates of the rectangle, which contains the player graphic. This is also useful for collision detection.
- `Vector2f getCenter():` This is slightly different to `getPosition` because it is a `Vector2f` type and contains just the *x* and *y* locations of the very center of the player graphic.
- `float getRotation():` The code in the `main` function will sometimes need to know, in degrees, which way the player is currently facing. 3 o'clock is 0 degrees and increases clockwise.
- `Sprite getSprite():` As we discussed previously, this function returns a copy of the sprite that represents the player.
- `void moveLeft(), ..Right(), ..Up(), ..Down():` These four functions have no return type or parameters. They will be called from the `main` function and the `Player` class will then be able to act when one or more of the *WASD* keys have been pressed.
- `void stopLeft(), ..Right(), ..Up(), ..Down():` These four functions have no return type or parameters. They will be called from the `main` function, and the `Player` class will then be able to act when one or more of the *WASD* keys have been released.

- `void update(float elapsedTime, Vector2i mousePosition)`: This will be the only long function of the entire class. It will be called once per frame from `main`. It will do everything necessary to make sure the `player` object's data is updated so that it's ready for collision detection and drawing. Notice that it returns no data but receives the amount of elapsed time since the last frame, along with a `Vector2i` instance, which will hold the horizontal and vertical screen location of the mouse pointer/crosshair.



Note that these are integer screen coordinates and are distinct from the floating-point world coordinates.

- `void upgradeSpeed()`: A function that can be called from the leveling up screen when the player chooses to make the player faster.
- `void upgradeHealth()`: Another function that can be called from the leveling up screen when the player chooses to make the player stronger (that is, have more health).
- `void increaseHealthLevel(int amount)`: A subtle but important difference regarding the previous function in that this one will increase the amount of health the player has, up to the maximum that's currently set. This function will be used when the player picks up a health pick-up.
- `int getHealth()`: With the level of health being as dynamic as it is, we need to be able to determine how much health the player has at any given moment. This function returns an `int`, which holds that value.

Like the variables, it should now be plain what each of the functions is for. Also the *why* and the precise context of using some of these functions will only reveal themselves as we progress with the project.



You don't need to memorize the function names, return types, or parameters as we will discuss the code when they are used. You do, however, need to take your time to look over them, along with the previous explanations, and get more familiar with them. Furthermore, as we proceed, it might be worth referring to this header file if anything seems unclear.

Now, we can move on to the meat of our functions: the definitions.

Coding the Player class function definitions

Finally, we can begin writing the code that does the work of our class.

Right-click on **Source Files** in **Solution Explorer** and select **Add | New Item....** In the **Add New Item** window, highlight (by *left-clicking* on) **C++ File (.cpp)** and then, in the **Name** field, type `Player.cpp`. Finally, click on the **Add** button.



From now on, I will simply ask you to create a new class or header file. So, commit the preceding step to memory or refer back here if you need a reminder.

We are now ready to code the `.cpp` file for our first class in this project.

Here are the necessary include directives, followed by the definition of the constructor. Remember, the constructor will be called when we first instantiate an object of the `Player` type. Add the following code to the `Player.cpp` file and then we can take a closer look at it:

```
#include "player.h"

Player::Player()
{
    m_Speed = START_SPEED;
    m_Health = START_HEALTH;
    m_MaxHealth = START_HEALTH;

    // Associate a texture with the sprite
    // !!Watch this space!!
    m_Texture.loadFromFile("graphics/player.png");
    m_Sprite.setTexture(m_Texture);

    // Set the origin of the sprite to the center,
    // for smooth rotation
    m_Sprite.setOrigin(25, 25);
}
```

In the constructor function, which, of course, has the same name as the class and no return type, we write code that begins to set up the `Player` object, ready for use.

To be clear; this code will run when we write the following code from the main function:

```
Player player;
```

Don't add the previous line of code just yet.

All we do in the constructor is initialize `m_Speed`, `m_Health`, and `m_MaxHealth` from their related constants. Then, we load the player graphic into `m_Texture`, associate `m_Texture` with `m_Sprite`, and set the origin of `m_Sprite` to the center, (25, 25).



Note the cryptic comment, `// !!Watch this space!!`, indicating that we will return to the loading of our texture and some important issues regarding it. We will eventually change how we deal with this texture once we have discovered a problem and learned a bit more C++. We will do so in *Chapter 10, Pointers, the Standard Template Library, and Texture Management*.

Next, we will code the `spawn` function. We will only ever create one instance of the `Player` class. We will, however, need to spawn it into the current level for each wave. This is what the `spawn` function will handle for us. Add the following code to the `Player.cpp` file and be sure to examine the details and read the comments:

```
void Player::spawn(IntRect arena,
                  Vector2f resolution,
                  int tileSize)
{
    // Place the player in the middle of the arena
    m_Position.x = arena.width / 2;
    m_Position.y = arena.height / 2;

    // Copy the details of the arena
    // to the player's m_Arena
    m_Arena.left = arena.left;
    m_Arena.width = arena.width;
    m_Arena.top = arena.top;
    m_Arena.height = arena.height;

    // Remember how big the tiles are in this arena
    m_TileSize = tileSize;

    // Store the resolution for future use
    m_Resolution.x = resolution.x;
    m_Resolution.y = resolution.y;
}
```

The preceding code starts off by initializing the `m_Position.x` and `m_Position.y` values to half the height and width of the passed in arena. This has the effect of moving the player to the center of the level, regardless of its size.

Next, we copy all the coordinates and dimensions of the passed in `arena` to the member object of the same type, `m_Arena`. The details of the size and coordinates of the current arena are used so frequently that it makes sense to do this. We can now use `m_Arena` for tasks such as making sure the player can't walk through walls. In addition to this, we copy the passed in `tileSize` instance to the member variable, `m_TileSize`, for the same purpose. We will see `m_Arena` and `m_TileSize` in action in the update function.

The final two lines from the preceding code copy the screen resolution from the `Vector2f`, `resolution`, which is a parameter of `spawn`, into `m_Resolution`, which is a member variable of `Player`. We now have access to these values inside the `Player` class.

Now, add the very straightforward code of the `resetPlayerStats` function:

```
void Player::resetPlayerStats()
{
    m_Speed = START_SPEED;
    m_Health = START_HEALTH;
    m_MaxHealth = START_HEALTH;
}
```

When the player dies, we will use this to reset any upgrades they might have used.

We will not write the code that calls the `resetPlayerStats` function until nearly completing the project, but it is there ready for when we need it.

In the next part of the code, we will add two more functions. They will handle what happens when the player is hit by a zombie. We will be able to call `player.hit()` and pass in the current game time. We will also be able to query the last time that the player was hit by calling `player.getLastHitTime()`. Exactly how these functions are useful will become apparent when we have some zombies.

Add the two new definitions to the `Player.cpp` file and then examine the C++ code a little more closely:

```
Time Player::getLastHitTime()
{
    return m_LastHit;
}

bool Player::hit(Time timeHit)
{
    if (timeHit.asMilliseconds()
        - m_LastHit.asMilliseconds() > 200)
    {
```

```

        m_LastHit = timeHit;
        m_Health -= 10;
        return true;
    }
    else
    {
        return false;
    }
}

```

The code for `getLastHitTime()` is very straightforward; it will return whatever value is stored in `m_LastHit`.

The `hit` function is a bit more in-depth and nuanced. First, the `if` statement checks to see whether the time that's passed in as a parameter is 200 milliseconds further ahead than the time stored in `m_LastHit`. If it is, `m_LastHit` is updated with the time passed in and `m_Health` has 10 deducted from its current value. The last line of code in this `if` statement is `return true`. Notice that the `else` clause simply returns `false` to the calling code.

The overall effect of this function is that health points will only be deducted from the player up to five times per second. Remember that our game loop might be running at thousands of iterations per second. In this scenario, without the restriction this function provides, a zombie would only need to be in contact with the player for one second and tens of thousands of health points would be deducted. The `hit` function controls and restricts this phenomenon. It also lets the calling code know whether a new hit has been registered (or not) by returning `true` or `false`.

This code implies that we will detect collisions between a zombie and the player in the main function. We will then call `player.hit()` to determine whether to deduct any health points.

Next, for the `Player` class, we will implement a bunch of getter functions. They allow us to keep the data neatly encapsulated in the `Player` class, at the same time as making their values available to the main function.

Add the following code, right after the previous block:

```

FloatRect Player::getPosition()
{
    return m_Sprite.getGlobalBounds();
}

Vector2f Player::getCenter()
{
    return m_Position;
}

```



```
}

float Player::getRotation()
{
    return m_Sprite.getRotation();
}

Sprite Player::getSprite()
{
    return m_Sprite;
}

int Player::getHealth()
{
    return m_Health;
}
```

The previous code is very straightforward. Each one of the previous five functions returns the value of one of our member variables. Look carefully at each of them and familiarize yourself with which function returns which value.

The next eight short functions enable the keyboard controls (which we will use from the main function) so that we can change the data contained in our object of the `Player` type. Add the following code to the `Player.cpp` file and then we will summarize how it all works:

```
void Player::moveLeft()
{
    m_LeftPressed = true;
}

void Player::moveRight()
{
    m_RightPressed = true;
}

void Player::moveUp()
{
    m_UpPressed = true;
}

void Player::moveDown()
{
    m_DownPressed = true;
}
```

```
void Player::stopLeft()
{
    m_LeftPressed = false;
}

void Player::stopRight()
{
    m_RightPressed = false;
}

void Player::stopUp()
{
    m_UpPressed = false;
}

void Player::stopDown()
{
    m_DownPressed = false;
}
```

The previous code has four functions (`moveLeft`, `moveRight`, `moveUp`, and `moveDown`), which set the related Boolean variables (`m_LeftPressed`, `m_RightPressed`, `m_UpPressed`, and `m_DownPressed`) to `true`. The other four functions (`stopLeft`, `stopRight`, `stopUp`, and `stopDown`) do the opposite and set the same Boolean variables to `false`. The instance of the `Player` class can now be kept informed of which of the *WASD* keys were pressed and which were not.

The following function is the one that does all the hard work. The `update` function will be called once in every single frame of our game loop. Add the following code, and then we will examine it in detail. If we followed along with the previous eight functions and we remember how we animated the clouds and bees for the *Timber!!!* project and the bat and ball for *Pong*, we will probably understand most of the following code:

```
void Player::update(float elapsedTime, Vector2i mousePosition)
{
    if (m_UpPressed)
    {
        m_Position.y -= m_Speed * elapsedTime;
    }

    if (m_DownPressed)
    {
        m_Position.y += m_Speed * elapsedTime;
    }
}
```

```
        if (m_RightPressed)
        {
            m_Position.x += m_Speed * elapsedTime;
        }

        if (m_LeftPressed)
        {
            m_Position.x -= m_Speed * elapsedTime;
        }

        m_Sprite.setPosition(m_Position);

        // Keep the player in the arena
        if (m_Position.x > m_Arena.width - m_TileSize)
        {
            m_Position.x = m_Arena.width - m_TileSize;
        }

        if (m_Position.x < m_Arena.left + m_TileSize)
        {
            m_Position.x = m_Arena.left + m_TileSize;
        }

        if (m_Position.y > m_Arena.height - m_TileSize)
        {
            m_Position.y = m_Arena.height - m_TileSize;
        }

        if (m_Position.y < m_Arena.top + m_TileSize)
        {
            m_Position.y = m_Arena.top + m_TileSize;
        }

        // Calculate the angle the player is facing
        float angle = (atan2(mousePosition.y - m_Resolution.y / 2,
            mousePosition.x - m_Resolution.x / 2)
            * 180) / 3.141;

        m_Sprite.setRotation(angle);
    }
```

The first portion of the previous code moves the player sprite. The four `if` statements check which of the movement-related Boolean variables (`m_LeftPressed`, `m_RightPressed`, `m_UpPressed`, or `m_DownPressed`) are true and changes `m_Position.x` and `m_Position.y` accordingly. The same formula, from the previous two projects, to calculate the amount to move is also used:

position (+ or -) speed * elapsed time.

After these four `if` statements, `m_Sprite.setPosition` is called and `m_Position` is passed in. The sprite has now been adjusted by exactly the right amount for that one frame.

The next four `if` statements check whether `m_Position.x` or `m_Position.y` is beyond any of the edges of the current arena. Remember that the confines of the current arena were stored in `m_Arena`, in the `spawn` function. Let's look at the first one of these four `if` statements in order to understand them all:

```
if (m_Position.x > m_Arena.width - m_TileSize)
{
    m_Position.x = m_Arena.width - m_TileSize;
}
```

The previous code tests to see whether `m_position.x` is greater than `m_Arena.width`, minus the size of a tile (`m_TileSize`). As we will see when we create the background graphics, this calculation will detect the player straying into the wall.

When the `if` statement is true, the `m_Arena.width - m_TileSize` calculation is used to initialize `m_Position.x`. This means that the center of the player graphic will never be able to stray past the left-hand edge of the right-hand wall.

The next three `if` statements, which follow the one we have just discussed, do the same thing but for the other three walls.

The last two lines in the preceding code calculate and set the angle that the player sprite is rotated to (that is, facing). This line of code might look a little complex, but it is simply using the position of the crosshair (`mousePosition.x` and `mousePosition.y`) and the center of the screen (`m_Resolution.x` and `m_Resolution.y`) in a tried-and-tested trigonometric function.

How `atan` uses these coordinates along with `Pi` (3.141) is quite complicated, and that is why it is wrapped up in a handy function for us.



If you want to explore trigonometric functions in more detail, you can do so here: <http://www.cplusplus.com/reference/cmath/>.

The last three functions we will add for the `Player` class make the player 20% faster, increase the player's health by 20%, and increase the player's health by the amount passed in, respectively.

Add the following code at the end of the `Player.cpp` file, and then we will take a closer look at it:

```
void Player::upgradeSpeed()
{
    // 20% speed upgrade
    m_Speed += (START_SPEED * .2);
}

void Player::upgradeHealth()
{
    // 20% max health upgrade
    m_MaxHealth += (START_HEALTH * .2);
}

void Player::increaseHealthLevel(int amount)
{
    m_Health += amount;

    // But not beyond the maximum
    if (m_Health > m_MaxHealth)
    {
        m_Health = m_MaxHealth;
    }
}
```

In the preceding code, the `upgradeSpeed()` and `upgradeHealth()` functions increase the value stored in `m_Speed` and `m_MaxHealth`, respectively. These values are increased by 20% by multiplying the starting values by `.2` and adding them to the current values. These functions will be called from the `main` function when the player is choosing what attributes of their character they wish to improve (that is, level up) between levels.

The `increaseHealthLevel()` function takes an `int` value from `main` in the `amount` parameter. This `int` value will be provided by a class called `Pickup`, which we will write in *Chapter 11, Collision Detection, Pickups, and Bullets*. The `m_Health` member variable is increased by the passed-in value. However, there is a catch for the player. The `if` statement checks whether `m_Health` has exceeded `m_MaxHealth` and, if it has, sets it to `m_MaxHealth`. This means the player cannot simply gain infinite health from pick-ups. Instead, they must carefully balance the upgrades they choose between levels.

Of course, our `Player` class can't do anything until we instantiate it and put it to work in our game loop. Before we do that, let's look at the concept of a game camera.

Controlling the game camera with SFML View

In my opinion, the SFML `View` class is one of the neatest classes. After finishing this book, when we make games without using a media/gaming library, we will really notice the absence of `View`.

The `View` class allows us to consider our game as taking place in its own world, with its own properties. What do I mean? Well, when we create a game, we are usually trying to create a virtual world. That virtual world rarely, if ever, is measured in pixels, and rarely, if ever, will that world be the same number of pixels as the player's monitor. We need a way to abstract the virtual world we are building so that it can be of whatever size or shape we like.

Another way to think of SFML `View` is as a camera through which the player views a part of our virtual world. Most games will have more than one camera/view of the world.

For example, consider a split screen game where two players can be in different parts of the world at the same time.

Or, consider a game where there is a small area of the screen that represents the entire game world, but at a very high level/zoomed out, like a mini map.

Even if our games are much simpler than the previous two examples and don't need split screens or mini maps, we will likely want to create a world that is bigger than the screen it is being played on. This is, of course, the case with *Zombie Arena*.

Additionally, if we are constantly moving the game camera around to show different parts of the virtual world (usually to track the player), what happens to the HUD? If we draw the score and other onscreen HUD information and then we scroll the world around to follow the player, the score would move relative to that camera.

The SFML `View` class easily enables all these of features and solves this problem with very straightforward code. The trick is to create an instance of `View` for every camera – perhaps a `View` instance for the mini map, a `View` instance for the scrolling game world, and then a `View` instance for the HUD.

The instances of `View` can be moved around, sized, and positioned as required. So, the main `View` instance following the game can track the player, the mini-map view can remain in a fixed, zoomed-out small corner of the screen, and the HUD can overlay the entire screen and never move, despite the fact the main `View` instance could go wherever the player goes.

Let's look at some code using a few instances of `View`.



This code is being used to introduce the `View` class.
Don't add this code to the Zombie Arena project.

Create and initialize a few instances of `View`:

```
// Create a view to fill a 1920 x 1080 monitor
View mainView(sf::FloatRect(0, 0, 1920, 1080));

// Create a view for the HUD
View hudView(sf::FloatRect(0, 0, 1920, 1080));
```

The previous code creates two `View` objects that fill a 1920 x 1080 monitor. Now, we can do some magic with `mainView` while leaving `hudView` completely alone:

```
// In the update part of the game
// There are lots of things you can do with a View

// Make the view centre around the player
mainView.setCenter(player.getCenter());

// Rotate the view 45 degrees
mainView.rotate(45)

// Note that hudView is totally unaffected by the previous code
```

When we manipulate the properties of a `View` instance, we do so like this. When we draw sprites, text, or other objects to a view, we must specifically **set** the view as the current view for the window:

```
// Set the current view
window.setView(mainView);
```

Now, we can draw everything we want into that view:

```
// Do all the drawing for this view
window.draw(playerSprite);
window.draw(otherGameObject);
// etc
```

The player might be at any coordinate whatsoever; it doesn't matter because `mainView` is centered around the graphic.


Now, we can draw the HUD into `hudView`. Note that just like we draw individual elements (background, game objects, text, and so on) in layers from back to front, we also draw views from back to front as well. Hence, a HUD is drawn after the main game scene:

```
// Switch to the hudView
window.setView(hudView);

// Do all the drawing for the HUD
window.draw(scoreText);
window.draw(healthBar);
// etc
```

Finally, we can draw/show the window and all its views for the current frame in the usual way:

```
window.display();
```

 If you want to take your understanding of SFML `View` further than is necessary for this project, including how to achieve split screens and mini maps, then the best guide on the web is on the official SFML website: <https://www.sfm1-dev.org/tutorials/2.5/graphics-view.php>.

Now that we have learned about `View`, we can start coding the *Zombie Arena* `main` function and use our first `View` instance for real. In *Chapter 12, Layering Views and Implementing the HUD*, we will introduce a second instance of `View` for the HUD and layer it over the top of the main `View` instance.

Starting the Zombie Arena game engine

In this game, we will need a slightly upgraded game engine in `main`. We will have an enumeration called `state`, which will track what the current state of the game is. Then, throughout `main`, we can wrap parts of our code so that different things happen in different states.

When we created the project, Visual Studio created a file for us called `ZombieArena.cpp`. This will be the file that contains our `main` function and the code that instantiates and controls all our classes.

We begin with the now-familiar `main` function and some include directives. Note the addition of an include directive for the `Player` class.

Add the following code to the `ZombieArena.cpp` file:

```
#include <SFML/Graphics.hpp>
#include "Player.h"

using namespace sf;

int main()
{

    return 0;
}
```

The previous code has nothing new in it except that the `#include "Player.h"` line means we can now use the `Player` class within our code.

Let's flesh out some more of our game engine. The following code does quite a lot. Be sure to read the comments when you add the code to get an idea of what is going on. We will then go through it in more detail.

Add the following highlighted code at the start of the `main` function:

```
int main()
{
    // The game will always be in one of four states
    enum class State { PAUSED, LEVELING_UP,
                      GAME_OVER, PLAYING };

    // Start with the GAME_OVER state
    State state = State::GAME_OVER;

    // Get the screen resolution and
```

```
// create an SFML window
Vector2f resolution;
resolution.x =
    VideoMode::getDesktopMode().width;

resolution.y =
    VideoMode::getDesktopMode().height;

RenderWindow window(
    VideoMode(resolution.x, resolution.y),
    "Zombie Arena", Style::Fullscreen);

// Create a an SFML View for the main action
View mainView(sf::FloatRect(0, 0,
    resolution.x, resolution.y));

// Here is our clock for timing everything
Clock clock;

// How long has the PLAYING state been active
Time gameTimeTotal;

// Where is the mouse in
// relation to world coordinates
Vector2f mouseWorldPosition;

// Where is the mouse in
// relation to screen coordinates
Vector2i mouseScreenPosition;

// Create an instance of the Player class
Player player;

// The boundaries of the arena
IntRect arena;

// The main game loop
while (window.isOpen())
{

}

return 0;
}
```

Let's run through each section of all the code that we entered. Just inside the main function, we have the following code:

```
// The game will always be in one of four states
enum class State { PAUSED, LEVELING_UP, GAME_OVER, PLAYING };

// Start with the GAME_OVER state
State state = State::GAME_OVER;
```

The previous code creates a new enumeration class called `State`. Then, the code creates an instance of the `State` class called `state`. The `state` enumeration can now be one of four values, as defined in the declaration. Those values are `PAUSED`, `LEVELING_UP`, `GAME_OVER`, and `PLAYING`. These four values will be just what we need for keeping track and responding to the different states that the game can be in at any given time. Note that it is not possible for `state` to hold more than one value at a time.

Immediately after, we added the following code:

```
// Get the screen resolution and create an SFML window
Vector2f resolution;
resolution.x = VideoMode::getDesktopMode().width;

resolution.y = VideoMode::getDesktopMode().height;

RenderWindow window(VideoMode(resolution.x, resolution.y),
    "Zombie Arena", Style::Fullscreen);
```

The previous code declares a `Vector2f` instance called `resolution`. We initialize the two member variables of `resolution` (`x` and `y`) by calling the `VideoMode::getDesktopMode` function for both width and height. The `resolution` object now holds the resolution of the monitor on which the game is running. The final line of code creates a new `RenderWindow` instance called `window` using the appropriate resolution.

The following code creates an SFML view object. The view is positioned (initially) at the exact coordinates of the pixels of the monitor. If we were to use this `View` to do some drawing in this current position, it would be the same as drawing to a window without a view. However, we will eventually start to move this view to focus on the parts of our game world that the player needs to see. Then, when we start to use a second `View` instance, which remains fixed (for the HUD), we will see how this `View` instance can track the action while the other remains static to display the HUD:

```
// Create a an SFML View for the main action
View mainView(sf::FloatRect(0, 0, resolution.x, resolution.y));
```

Next, we created a `Clock` instance to do our timing and a `Time` object called `gameTimeTotal` that will keep a running total of the game time that has elapsed. As the project progresses, we will also introduce more variables and objects to handle timing:

```
// Here is our clock for timing everything
Clock clock;
// How long has the PLAYING state been active
Time gameTimeTotal;
```

The following code declares two vectors: one holding two float variables, called `mouseWorldPosition`, and one holding two integers, called `mouseScreenPosition`. The mouse pointer is something of an anomaly because it exists in two different coordinate spaces. We could think of these as parallel universes if we like. Firstly, as the player moves around the world, we will need to keep track of where the crosshair is in that world. These will be floating-point coordinates and will be stored in `mouseWorldCoordinates`. Of course, the actual pixel coordinates of the monitor itself never change. They will always be 0,0 to horizontal resolution -1, vertical resolution -1. We will track the mouse pointer position that is relative to this coordinate space using the integers stored in `mouseScreenPosition`:

```
// Where is the mouse in relation to world coordinates
Vector2f mouseWorldPosition;
// Where is the mouse in relation to screen coordinates
Vector2i mouseScreenPosition;
```

Finally, we get to use our `Player` class. This line of code will cause the constructor function (`Player::Player`) to execute. Refer to `Player.cpp` if you want to refresh your memory about this function:

```
// Create an instance of the Player class
Player player;
```

This `IntRect` object will hold starting horizontal and vertical coordinates, as well as a width and a height. Once initialized, we will be able to access the size and location details of the current arena with code such as `arena.left`, `arena.top`, `arena.width`, and `arena.height`:

```
// The boundaries of the arena
IntRect arena;
```

The last part of the code that we added previously is, of course, our game loop:

```
// The main game loop
while (window.isOpen())
{

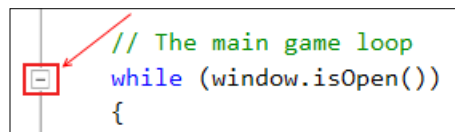
}
```

We have probably noticed that the code is getting quite long. We'll talk about this inconvenience in the following section.

Managing the code files

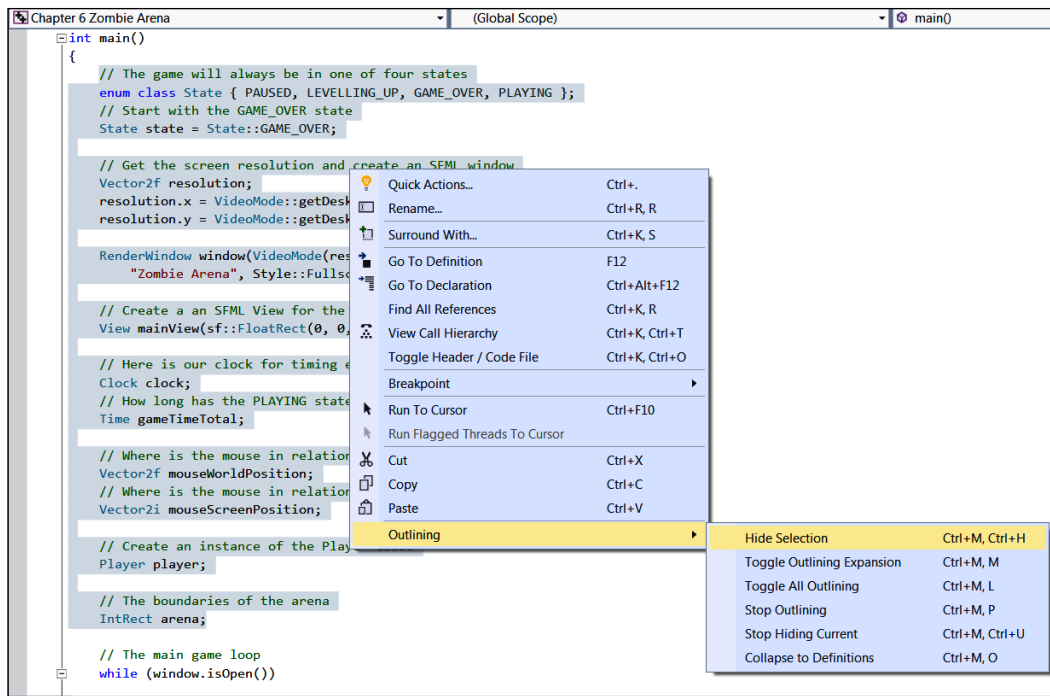
One of the advantages of abstraction using classes and functions is that the length (number of lines) of our code files can be reduced. Even though we will be using more than a dozen code files for this project, the length of the code in `ZombieArena.cpp` will still get a little unwieldy toward the end. In the final project, `Space Invaders++`, we will look at even more ways to abstract and manage our code.

For now, use this tip to keep things manageable. Notice that on the left-hand side of the code editor in Visual Studio, there are several **+** and **-** signs, one of which is shown in this diagram:

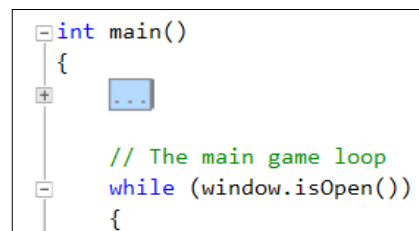


There will be one sign for each block (`if`, `while`, `for`, and so on) of the code. You can expand and collapse these blocks by clicking on the **+** and **-** signs. I recommend keeping all the code not currently under discussion collapsed. This will make things much clearer.

Furthermore, we can create our own collapsible blocks. I suggest making a collapsible block out of all the code before the start of the main game loop. To do so, highlight the code and then *right-click* and choose **Outlining | Hide Selection**, as shown in the following screenshot:



Now, you can click the - and + signs to expand and collapse the block. Each time we add code before the main game loop (and that will be quite often), you can expand the code, add the new lines, and then collapse it again. The following screenshot shows what the code looks like when it is collapsed:



This is much more manageable than it was before. Now, we can make a start with the main game loop.

Starting to code the main game loop

As you can see, the last part of the preceding code is the game loop (`while (window.isOpen()) {}`). We will turn our attention to this now. Specifically, we will be coding the input handling section of the game loop.

The code that we will be adding is quite long. There is nothing complicated about it, though, and we will examine it all in a moment.

Add the following highlighted code to the game loop:

```
// The main game loop
while (window.isOpen())
{
    /*
    *****
    Handle input
    *****
    */

    // Handle events by polling
    Event event;
    while (window.pollEvent(event))
    {
        if (event.type == Event::KeyPressed)
        {
            // Pause a game while playing
            if (event.key.code == Keyboard::Return &&
                state == State::PLAYING)
            {
                state = State::PAUSED;
            }

            // Restart while paused
            else if (event.key.code == Keyboard::Return &&
                state == State::PAUSED)
            {
                state = State::PLAYING;
                // Reset the clock so there isn't a frame jump
                clock.restart();
            }

            // Start a new game while in GAME_OVER state
            else if (event.key.code == Keyboard::Return &&
                state == State::GAME_OVER)
```

```

        {
            state = State::LEVELING_UP;
        }

        if (state == State::PLAYING)
        {
        }

    }
} // End event polling

} // End game loop

```


In the preceding code, we instantiate an object of the `Event` type. We will use `event`, like we did in the previous projects, to poll for system events. To do so, we wrap the rest of the code from the previous block in a `while` loop with the `window.pollEvent(event)` condition. This will keep looping each frame until there are no more events to process.

Inside this `while` loop, we handle the events we are interested in. First, we test for `Event::KeyPressed` events. If the *Return* key is pressed while the game is in the `PLAYING` state, then we switch `state` to `PAUSED`.

If the *Return* key is pressed while the game is in the `PAUSED` state, then we switch `state` to `PLAYING` and restart the `clock` object. The reason we restart `clock` after switching from `PAUSED` to `PLAYING` is because, while the game is paused, the elapsed time still accumulates. If we didn't restart the clock, all our objects would update their locations as if the frame had just taken a very long time. This will become more apparent as we flesh out the rest of the code in this file.

We then have an `else if` block to test whether the *Return* key was pressed while the game was in the `GAME_OVER` state. If it was, then `state` is changed to `LEVELING_UP`.

[



Note that the `GAME_OVER` state is the state where the home screen is displayed. So, the `GAME_OVER` state is the state after the player has just died and when the player first runs the game. The first thing that the player gets to do each game is pick an attribute to improve (that is, level up).

]

In the previous code, there is a final `if` condition to test whether the state is equal to `PLAYING`. This `if` block is empty and we will add code to it throughout the project.



We will add code to lots of different parts of this file throughout the project. Therefore, it is worthwhile taking the time to understand the different states our game can be in and where we handle them. It will also be very beneficial to collapse and expand the different `if`, `else`, and `while` blocks as and when appropriate.

Spend some time thoroughly familiarizing yourself with the `while`, `if`, and `else if` blocks we have just coded. We will be referring to them regularly.

Next, immediately after the previous code and still inside the game loop, which is still dealing with handling input, add the following highlighted code. Note the existing code (not highlighted) that shows exactly where the new (highlighted) code goes:

```
// End event polling

// Handle the player quitting
if (Keyboard::isKeyPressed(Keyboard::Escape))
{
    window.close();
}

// Handle WASD while playing
if (state == State::PLAYING)
{
    // Handle the pressing and releasing of the WASD keys
    if (Keyboard::isKeyPressed(Keyboard::W))
    {
        player.moveUp();
    }
    else
    {
        player.stopUp();
    }

    if (Keyboard::isKeyPressed(Keyboard::S))
    {
        player.moveDown();
    }
    else
    {
        player.stopDown();
    }
}
```

```

        if (Keyboard::isKeyPressed(Keyboard::A))
        {
            player.moveLeft();
        }
        else
        {
            player.stopLeft();
        }

        if (Keyboard::isKeyPressed(Keyboard::D))
        {
            player.moveRight();
        }
        else
        {
            player.stopRight();
        }

    }// End WASD while playing

} // End game loop

```

In the preceding code, we first test to see whether the player has pressed the *Escape* key. If it is pressed, the game window will be closed.

Next, within one big `if(state == State::PLAYING)` block, we check each of the WASD keys in turn. If a key is pressed, we call the appropriate `player.move...` function. If it is not, we call the related `player.stop...` function.

This code ensures that, in each frame, the player object will be updated with the WASD keys that are pressed and those that are not. The `player.move...` and `player.stop...` functions store the information in the member Boolean variables (`m_LeftPressed`, `m_RightPressed`, `m_UpPressed`, and `m_DownPressed`). The `Player` class then responds to the value of these Booleans, in each frame, in the `player.update` function, which we will call in the update section of the game loop.

Now, we can handle the keyboard input to allow the player to level up at the start of each game and in-between each wave. Add and study the following highlighted code and then we will discuss it:

```

    } // End WASD while playing

    // Handle the LEVELING up state
    if (state == State::LEVELING_UP)
    {

```

```
// Handle the player LEVELING up
if (event.key.code == Keyboard::Num1)
{
    state = State::PLAYING;
}

if (event.key.code == Keyboard::Num2)
{
    state = State::PLAYING;
}

if (event.key.code == Keyboard::Num3)
{
    state = State::PLAYING;
}

if (event.key.code == Keyboard::Num4)
{
    state = State::PLAYING;
}

if (event.key.code == Keyboard::Num5)
{
    state = State::PLAYING;
}

if (event.key.code == Keyboard::Num6)
{
    state = State::PLAYING;
}

if (state == State::PLAYING)
{
    // Prepare the level
    // We will modify the next two lines later
    arena.width = 500;
    arena.height = 500;
    arena.left = 0;
    arena.top = 0;

    // We will modify this line of code later
    int tileSize = 50;
}
```

```

        // Spawn the player in the middle of the arena
        player.spawn(arena, resolution, tileSize);

        // Reset the clock so there isn't a frame jump
        clock.restart();
    }
} // End LEVELING up

} // End game loop

```

In the preceding code, which is all wrapped in a test to see whether the current value of state is equal to `LEVELING_UP`, we handle the keyboard keys 1, 2, 3, 4, 5, and 6. In the `if` block for each, we simply set state to `State::PLAYING`. We will add some code to deal with each level up option later in *Chapter 13, Sound Effects, File I/O, and Finishing the Game*.

This code does the following things:

1. If the state is equal to `LEVELING_UP`, wait for either the 1, 2, 3, 4, 5, or 6 keys to be pressed.
2. When pressed, change state to `PLAYING`.
3. When the state changes, still within the `if (state == State::LEVELING_UP)` block, the nested `if (state == State::PLAYING)` block will run.
4. Within this block, we set the location and size of arena, set the `tileSize` to 50, pass all the information to `player.spawn`, and call `clock.restart`.

Now, we have an actual spawned player object that is aware of its environment and can respond to key presses. We can now update the scene on each pass through the loop.

Be sure to neatly collapse the code from the input handling part of the game loop since we are done with that for now. The following code is in the updating part of the game loop. Add and study the following highlighted code and then we can discuss it:

```

} // End LEVELING up

/*
*****
UPDATE THE FRAME
*****
*/
if (state == State::PLAYING)
{

```

```
// Update the delta time
Time dt = clock.restart();

// Update the total game time
gameTimeTotal += dt;

// Make a decimal fraction of 1 from the delta time
float dtAsSeconds = dt.asSeconds();

// Where is the mouse pointer
mouseScreenPosition = Mouse::getPosition();

// Convert mouse position to world coordinates of mainView
mouseWorldPosition = window.mapPixelToCoords(
    Mouse::getPosition(), mainView);

// Update the player
player.update(dtAsSeconds, Mouse::getPosition());

// Make a note of the players new position
Vector2f playerPosition(player.getCenter());

// Make the view centre around the player
mainView.setCenter(player.getCenter());
} // End updating the scene

} // End game loop
```

First, note that the previous code is wrapped in a test to make sure the game is in the `PLAYING` state. We don't want this code to run if the game has been paused, it has ended, or if the player is choosing what to level up.

First, we restart the clock and store the time that the previous frame took in the `dt` variable:

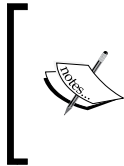
```
// Update the delta time
Time dt = clock.restart();
```

Next, we add the time that the previous frame took to the accumulated time the game has been running for, as held by `gameTimeTotal`:

```
// Update the total game time
gameTimeTotal += dt;
```

Now, we initialize a `float` variable called `dtAsSeconds` with the value returned by the `dt.AsSeconds` function. For most frames, this will be a fraction of one. This is perfect for passing into the `player.update` function to be used to calculate how much to move the player sprite.

Now, we can initialize `mouseScreenPosition` using the `MOUSE::getPosition` function.



You might be wondering about the slightly unusual syntax for getting the position of the mouse. This is called a **static function**. If we define a function in a class with the `static` keyword, we can call that function using the class name and without an instance of the class. C++ OOP has lots of quirks and rules like this. We will see more as we progress.

We then initialize `mouseWorldPosition` using the SFML `mapPixelToCoords` function on `window`. We discussed this function when talking about the `View` class earlier in this chapter.

At this point, we are now able to call `player.update` and pass in `dtAsSeconds` and the position of the mouse, as is required.

We store the player's new center in a `Vector2f` instance called `playerPosition`. At the moment, this is unused, but we will have a use for this later in the project.

We can then center the view around the center of the player's up-to-date position with `mainView.setCenter(player.getCenter())`.

We are now able to draw the player to the screen. Add the following highlighted code, which splits the draw section of the main game loop into different states:

```
// End updating the scene

/*
*****
Draw the scene
*****
*/

if (state == State::PLAYING)
{
    window.clear();
```

```
        // set the mainView to be displayed in the window
        // And draw everything related to it
        window.setView(mainView);

        // Draw the player
        window.draw(player.getSprite());
    }

    if (state == State::LEVELING_UP)
    {
    }

    if (state == State::PAUSED)
    {
    }

    if (state == State::GAME_OVER)
    {
    }

    window.display();

} // End game loop

return 0;
}
```

Within the `if (state == State::PLAYING)` section of the previous code, we clear the screen, set the view of the window to `mainView`, and then draw the player sprite with `window.draw(player.getSprite())`.

After all the different states have been handled, the code shows the scene in the usual manner with `window.display();`.

You can run the game and see our player character spin around in response to moving the mouse.



When you run the game, you need to press *Enter* to start the game, and then select a number from 1 to 6 to simulate choosing an upgrade option. Then, the game will start.

You can also move the player around within the (empty) 500 x 500 pixel arena. You can see our lonely player in the center of the screen, as shown here:



You can't, however, get any sense of movement because we haven't implemented the background. We will do so in the next chapter.

Summary

Phew! That was a long one. We have done a lot in this chapter: we built our first class for the Zombie Arena project, `Player`, and put it to use in the game loop. We also learned about and used an instance of the `View` class, although we haven't explored the benefits this gives us just yet.

In the next chapter, we will build our arena background by exploring what sprite sheets are. We will also learn about C++ **references**, which allow us to manipulate variables, even when they are out of scope (that is, in another function).

FAQ

Q) I noticed we have coded quite a few functions of the `Player` class that we don't use. Why is this?

A) Rather than keep coming back to the `Player` class, we have added all the code that we will need throughout the project. By the end of *Chapter 13, Sound Effects, File I/O, and Finishing the Game*, we will have made full use of all of these functions.

9

C++ References, Sprite Sheets, and Vertex Arrays

In *Chapter 4, Loops, Arrays, Switches, Enumerations, and Functions – Implementing Game Mechanics*, we talked about scope. This is the concept that variables declared in a function or inner block of code only have scope (that is, can be *seen* or used) in that function or block. Using only the C++ knowledge we have currently, this can cause a problem. What do we do if we need to work on a few complex objects that are needed in the `main` function? This could imply all the code must be in the `main` function.

In this chapter, we will explore C++ **references**, which allow us to work on variables and objects that are otherwise out of scope. In addition to this, these references will help us avoid having to pass large objects between functions, which is a slow process. It is slow because each time we do this, a copy of the variable or object must be made.

Armed with this new knowledge of references, we will look at the SFML `VertexArray` class, which allows us to build up a large image that can be quickly and efficiently drawn to the screen using multiple parts in a single image file. By the end of this chapter, we will have a scalable, random, scrolling background that's been made using references and a `VertexArray` object.

In this chapter, we will discuss the following topics:

- C++ references
- SFML vertex arrays
- Coding a random, scrolling background

C++ references

When we pass values to a function or return values from a function, that is exactly what we are doing – passing/returning by **value**. What happens is that a copy of the value held by the variable is made and then sent to the function, where it is used.

The significance of this is twofold:

1. If we want the function to make a permanent change to a variable, this system is no good to us.
2. When a copy is made to pass in as an argument or returned from the function, processing power and memory are consumed. For a simple `int`, or even perhaps a `Sprite`, this is insignificant. However, for a complex object, perhaps an entire game world (or background), the copying process will seriously affect our game's performance.

References are the solution to these two problems. A **reference** is a special type of variable. A reference *refers* to another variable. Here is an example to help you understand this better:

```
int numZombies = 100;
int& rNumZombies = numZombies;
```

In the preceding code, we declare and initialize a regular `int` called `numZombies`. We then declare and initialize an `int` reference called `rNumZombies`. The reference operator, `&`, which follows the type, determines that a reference is being declared.



The `r` prefix at the front of the reference name is optional but is useful for remembering that we are dealing with a reference.

Now, we have an `int` variable called `numZombies`, which stores the value 100, and an `int` reference called `rNumZombies`, which refers to `numZombies`.

Anything we do to `numZombies` can be seen through `rNumZombies`, and anything we do to `rNumZombies` we are actually doing to `numZombies`. Take a look at the following code:

```
int score = 10;
int& rScore = score;
score ++;
rScore ++;
```

In the previous code, we declare an `int` called `score`. Next, we declare an `int` reference called `rScore` that refers to `score`. Remember that anything we do to `score` can be seen by `rScore` and anything we do to `rScore` is actually being done to `score`.

Therefore, consider what happens when we increment `score` like this:

```
score ++;
```

The `score` variable now stores the value 11. In addition to this, if we were to output `rScore`, it would also output 11. The next line of code is as follows:

```
rScore ++;
```

Now, `score` actually holds the value 12 because anything we do to `rScore` is actually done to `score`.



If you want to know *how* this works, then more will be revealed in the next chapter when we discuss **pointers**. But simply put, you can consider a reference as storing a place/address in the computer's memory. That place in memory is the same place where the variable it refers to stores its value. Therefore, an operation on either the reference or the variable has exactly the same effect.

For now, it is much more important to talk about the *why* of references. There are two reasons to use references, and we have already mentioned them. Here they are, summarized again:

1. Changing/reading the value of a variable/object in another function, which is otherwise out of scope.
2. Passing/returning to/from a function without making a copy (and, therefore, more efficiently).

Study the following code and then we will discuss it:

```
void add(int n1, int n2, int a);
void referenceAdd(int n1, int n2, int& a);

int main()
{
    int number1 = 2;
    int number2 = 2;
    int answer = 0;

    add(number1, number2, answer);
    // answer still equals zero because it is passed as a copy
    // Nothing happens to answer in the scope of main

    referenceAdd(number1, number2, answer);
    // Now answer equals 4 because it was passed by reference
    // When the referenceAdd function did this:
    // answer = num1 + num 2;
    // It is actually changing the value stored by answer
    return 0;
}

// Here are the two function definitions
// They are exactly the same except that
// the second passes a reference to a
void add(int n1, int n2, int a)
{
    a = n1 + n2;
    // a now equals 4
    // But when the function returns a is lost forever
}

void referenceAdd(int n1, int n2, int& a)
{
```

```
a = n1 + n2;  
// a now equals 4  
// But a is a reference!  
// So, it is actually answer, back in main, that equals 4  
}
```

The previous code begins with the prototypes of two functions: `add` and `referenceAdd`. The `add` function takes three `int` variables while the `referenceAdd` function takes two `int` variables and an `int` reference.

When the `add` function is called and the `number1`, `number2`, and `answer` variables are passed in, a copy of the values is made and new variables local to `add` (that is, `n1`, `n2`, and `a`) are manipulated. As a result of this, `answer`, back in `main`, remains at zero.

When the `referenceAdd` function is called, `number1` and `number2` are again passed by value. However, `answer` is passed by reference. When the value of `n1` that's added to `n2` is assigned to the reference, `a`, what is really happening is that the value is assigned to `answer` back in the `main` function.

It is probably obvious that we would never need to use a reference for something this simple. It does, however, demonstrate the mechanics of passing by reference.

Let's summarize what we know about references.

References summary

The previous code demonstrated how a reference can be used to alter the value of a variable in one scope using code in another. As well as being extremely convenient, passing by reference is also very efficient because no copy is made. Our example, that is, using a reference to an `int`, is a bit ambiguous because, as an `int` is so small, there is no real efficiency gain. Later on in this chapter, we will use a reference to pass an entire level layout and the efficiency gain will be significant.



There is one gotcha with references! You must assign the reference to a variable at the time you create it. This means it is not completely flexible. Don't worry about this for now. We will explore references further alongside their more flexible (and slightly more complicated) relations, such as **pointers**, in the next chapter.

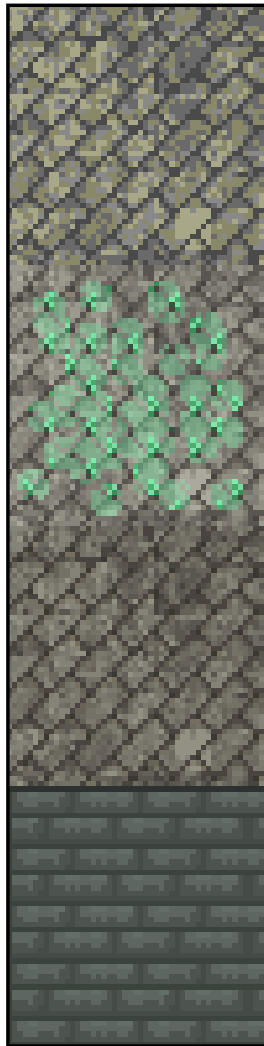
This is largely irrelevant for an `int`, but potentially significant for a large object of a class. We will use this exact technique when we implement the scrolling background of the *Zombie Arena* game later on in this chapter.

SFML vertex arrays and sprite sheets


We are nearly ready to implement the scrolling background. We just need to learn about SFML vertex arrays and sprite sheets.

What is a sprite sheet?

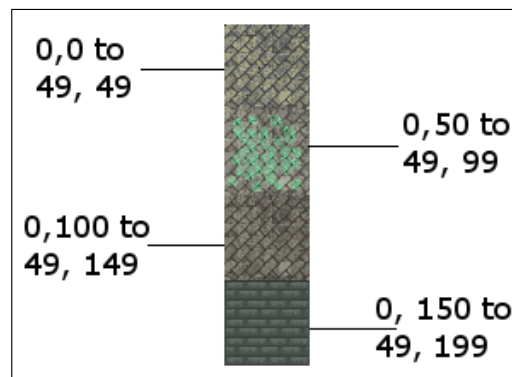
A **sprite sheet** is a set of images, either frames of animation, or totally individual graphics contained in one image file. Take a closer look at this sprite sheet, which contains four separate images that will be used to draw the background in our Zombie Arena game:



SFML allows us to load a sprite sheet as a regular texture, in the same way we have done for every texture in this book so far. When we load multiple images as a single texture, the GPU can handle it much more efficiently.


 Actually, a modern PC could handle these four textures without using a sprite sheet. It is worth learning these techniques, however, as our games are going to start getting progressively more demanding on our hardware.

What we need to do when we draw an image from the sprite sheet is make sure we refer to the precise pixel coordinates of the part of the sprite sheet we require, like so:



The previous diagram labels each part/tile with the coordinates and their position within the sprite sheet. These coordinates are called **texture coordinates**. We will use these texture coordinates in our code to draw just the right parts that we require.

What is a vertex array?

First, we need to ask, what is a vertex? A **vertex** is a single graphical point, that is, a coordinate. This point is defined by a horizontal and vertical position. The plural of vertex is vertices. A vertex array, then, is a whole collection of vertices.

In SFML, each vertex in a vertex array also has a color and a related additional vertex (that is, a pair of coordinates) called **texture coordinates**. Texture coordinates are the position of the image we want to use in terms of a sprite sheet. Later, we will see how we can position graphics and choose a part of the sprite sheet to display at each position, all with a single vertex array.

The SFML `VertexArray` class can hold different types of vertex sets. But each `VertexArray` should only hold one type of set. We use the type of set that suits the occasion.

Common scenarios in video games include, but are not limited to, the following **primitive** types:

- **Point:** A single vertex per point.
- **Line:** Two vertices per set that define the start and endpoint of the line.
- **Triangle:** Three vertices per point. This is the most commonly used (in the thousands) for complex 3D models, or in pairs to create a simple rectangle such as a sprite.
- **Quad:** Four vertices per set. This is a convenient way to map rectangular areas from a sprite sheet.

We will use quads in this project.

Building a background from tiles

The Zombie Arena background will be made up of a random arrangement of square images. You can think of this arrangement like tiles on a floor.

In this project, we will be using vertex arrays with **quad** sets. Each vertex will be part of a set of four (that is, a quad). Each vertex will define one corner of a tile from our background, while each texture coordinate will hold an appropriate value based on a specific image from the sprite sheet.

Let's look at some code to get us started. This isn't the exact code we will use in the project, but it is close and allows us to study vertex arrays before we move on to the actual implementation we will use.

Building a vertex array

As we do when we create an instance of a class, we declare our new object. The following code declares a new object of the `VertexArray` type, called `background`:

```
// Create a vertex array
VertexArray background;
```

We want to let our instance of `VertexArray` know which type of primitive we will be using. Remember that points, lines, triangles, and quads all have a different number of vertices. By setting the `VertexArray` instance to hold a specific type, it will be possible to know the start of each primitive. In our case, we want quads. Here is the code that will do this:

```
// What primitive type are we using
background.setPrimitiveType(Quads);
```

As with regular C++ arrays, a `VertexArray` instance needs to be set to a particular size. The `VertexArray` class is more flexible than a regular array, however. It allows us to change its size while the game is running. The size could be configured at the same time as the declaration, but our background needs to expand with each wave. The `VertexArray` class provides this functionality with the `resize` function. Here is the code that would set the size of our arena to a 10 by 10 tile size:

```
// Set the size of the vertex array
background.resize(10 * 10 * 4);
```

In the previous line of code, the first 10 is the width, the second 10 is the height, and 4 is the number of vertices in a quad. We could have just passed in 400, but showing the calculation like this makes it clear what we are doing. When we code the project for real, we will go a step further to aid clarity and declare variables for each part of the calculation.

We now have a `VertexArray` instance ready to have its hundreds of vertices configured. Here is how we set the position coordinates on the first four vertices (that is, the first quad):

```
// Position each vertex in the current quad
background[0].position = Vector2f(0, 0);
background[1].position = Vector2f(49, 0);
background[2].position = Vector2f(49,49);
background[3].position = Vector2f(0, 49);
```

Here is how we set the texture coordinates of these same vertices to the first image in the sprite sheet. These coordinates in the image file are from 0,0 (in the top-left corner) to 49,49 (in the bottom-right corner):

```
// Set the texture coordinates of each vertex
background[0].texCoords = Vector2f(0, 0);
background[1].texCoords = Vector2f(49, 0);
background[2].texCoords = Vector2f(49, 49);
background[3].texCoords = Vector2f(0, 49);
```

If we wanted to set the texture coordinates to the second image in the sprite sheet, we would have written the code like this:

```
// Set the texture coordinates of each vertex
background[0].texCoords = Vector2f(0, 50);
background[1].texCoords = Vector2f(49, 50);
background[2].texCoords = Vector2f(49, 99);
background[3].texCoords = Vector2f(0, 99);
```

Of course, if we define each and every vertex like this individually, then we are going to be spending a long time configuring even a simple 10 by 10 arena.

When we implement our background for real, we will devise a set of nested `for` loops that loop through each quad, pick a random background image, and assign the appropriate texture coordinates.

The code will need to be quite smart. It will need to know when it is an edge tile so that it can use the wall image from the sprite sheet. It will also need to use appropriate variables that know the position of each background tile in the sprite sheet, as well as the overall size of the required arena.

We will make this complexity manageable by putting all the code in both a separate function and a separate file. We will make the `VertexArray` instance usable in `main` by using a C++ reference.

We will examine these details later. You may have noticed that at no point have we associated a texture (the sprite sheet with the vertex array). Let's see how to do that now.

Using the vertex array to draw

We can load the sprite sheet as a texture in the same way that we load any other texture, as shown in the following code:

```
// Load the texture for our background vertex array
Texture textureBackground;
textureBackground.loadFromFile("graphics/background_sheet.png");
```

We can then draw the entire `VertexArray` with one call to `draw`:

```
// Draw the background
window.draw(background, &textureBackground);
```

The previous code is much more efficient than drawing every tile as an individual sprite.



Before we move on, notice the slightly odd-looking & notation before the `textureBackground` code. Your immediate thought might be that this has something to do with references. What is going on here is we are passing the address of the `Texture` instance instead of the actual `Texture` instance. We will learn more about this in the next chapter.

We are now able to use our knowledge of references and vertex arrays to implement the next stage of the Zombie Arena project.

Creating a randomly generated scrolling background

In this section, we will create a function that makes a background in a separate file. We will ensure the background will be available (in scope) to the main function by using a vertex array reference.

As we will be writing other functions that share data with the main function, we will write them all in their own `.cpp` files. We will provide prototypes for these functions in a new header file that we will include (with an `#include` directive) in `ZombieArena.cpp`.

To achieve this, let's make a new header file called `ZombieArena.h`. We are now ready to code the header file for our new function.

In this new `ZombieArena.h` header file, add the following highlighted code, including the function prototype:

```
#pragma once
using namespace sf;
int createBackground(VertexArray& rVA, IntRect arena);
```

The previous code allows us to write the definition of a function called `createBackground`. To match the prototype, the function definition must return an `int` value, and receive a `VertexArray` reference and an `IntRect` object as parameters.

Now, we can create a new `.cpp` file in which we will code the function definition. Create a new file called `CreateBackground.cpp`. We are now ready to code the function definition that will create our background.

Add the following code to the `CreateBackground.cpp` file, and then we will review it:

```
#include "ZombieArena.h"

int createBackground(VertexArray& rVA, IntRect arena)
{
    // Anything we do to rVA we are really doing
    // to background (in the main function)

    // How big is each tile/texture
    const int TILE_SIZE = 50;
    const int TILE_TYPES = 3;
    const int VERTS_IN_QUAD = 4;

    int worldWidth = arena.width / TILE_SIZE;
    int worldHeight = arena.height / TILE_SIZE;

    // What type of primitive are we using?
    rVA.setPrimitiveType(Quads);

    // Set the size of the vertex array
    rVA.resize(worldWidth * worldHeight * VERTS_IN_QUAD);

    // Start at the beginning of the vertex array
    int currentVertex = 0;

    return TILE_SIZE;
}
```

In the previous code, we write the function signature as well as the opening and closing curly brackets that mark the function body.

Within the function body, we declare and initialize three new `int` constants to hold values that we will need to refer to throughout the rest of the function. They are `TILE_SIZE`, `TILE_TYPES`, and `VERTS_IN_QUAD`.

The `TILE_SIZE` constant refers to the size in pixels of each tile within the sprite sheet. The `TILE_TYPES` constant refers to the number of different tiles within the sprite sheet. We could add more tiles into our sprite sheet and change `TILE_TYPES` to match the change, and the code we are about to write would still work. `VERTS_IN_QUAD` refers to the fact that there are four vertices in every quad. It is less error-prone to use this constant compared to always typing the number 4, which is less clear.

We then declare and initialize two `int` variables: `worldWidth` and `worldHeight`. These variables might appear obvious as to their use. They are betrayed by their names, but it is worth pointing out that they refer to the width and height of the world in the number of tiles, not pixels. The `worldWidth` and `worldHeight` variables are initialized by dividing the height and width of the passed-in arena by the `TILE_SIZE` constant.

Next, we get to use our reference for the first time. Remember that anything we do to `rVA`, we are really doing to the variable that was passed in, which is in scope in the `main` function (or will be when we code it).

Then, we prepare the vertex array to use quads using `rVA.setType` and then we make it just the right size by calling `rVA.resize`. To the `resize` function, we pass in the result of `worldWidth * worldHeight * VERTS_IN_QUAD`, which equates to exactly the number of vertices that our vertex array will have when we are done preparing it.

The last line of code declares and initializes `currentVertex` to zero. We will use `currentVertex` as we loop through the vertex array, initializing all the vertices.

We can now write the first part of a nested `for` loop that will prepare the vertex array. Add the following highlighted code and, based on what we have learned about vertex arrays, try and work out what it does:

```
// Start at the beginning of the vertex array
int currentVertex = 0;

for (int w = 0; w < worldWidth; w++)
{
    for (int h = 0; h < worldHeight; h++)
    {
        // Position each vertex in the current quad
        rVA[currentVertex + 0].position =
            Vector2f(w * TILE_SIZE, h * TILE_SIZE);

        rVA[currentVertex + 1].position =
            Vector2f((w * TILE_SIZE) + TILE_SIZE, h * TILE_SIZE);

        rVA[currentVertex + 2].position =
            Vector2f((w * TILE_SIZE) + TILE_SIZE, (h * TILE_SIZE)
                + TILE_SIZE);
```

```
        rVA[currentVertex + 3].position =
            Vector2f((w * TILE_SIZE), (h * TILE_SIZE)
                + TILE_SIZE);

        // Position ready for the next four vertices
        currentVertex = currentVertex + VERTS_IN_QUAD;
    }
}

return TILE_SIZE;
}
```

The code that we just added steps through the vertex array by using a nested for loop, which first steps through the first four vertices: `currentVertex + 1`, `currentVertex + 2`, and so on.

We access each vertex in the array using the array notation, `rVA[currentVertex + 0]` . . . , and so on. Using the array notation, we call the `position` function, `rVA[currentVertex + 0].position`

To the `position` function, we pass the horizontal and vertical coordinates of each vertex. We can work these coordinates out programmatically by using a combination of `w`, `h`, and `TILE_SIZE`.

At the end of the previous code, we position `currentVertex`, ready for the next pass through the nested for loop by advancing it four places (that is, adding four) with the code, that is, `currentVertex = currentVertex + VERTS_IN_QUAD`.

Of course, all this does is set the coordinates of our vertices; it doesn't assign a texture coordinate from the sprite sheet. This is what we will do next.

To make it absolutely clear where the new code goes, I have shown it in context, along with all the code that we wrote a moment ago. Add and study the following highlighted code:

```
for (int w = 0; w < worldWidth; w++)
{
    for (int h = 0; h < worldHeight; h++)
    {
        // Position each vertex in the current quad
        rVA[currentVertex + 0].position =
            Vector2f(w * TILE_SIZE, h * TILE_SIZE);
```

```

    rVA[currentVertex + 1].position =
        Vector2f((w * TILE_SIZE) + TILE_SIZE, h * TILE_SIZE);

    rVA[currentVertex + 2].position =
        Vector2f((w * TILE_SIZE) + TILE_SIZE, (h * TILE_SIZE)
            + TILE_SIZE);

    rVA[currentVertex + 3].position =
        Vector2f((w * TILE_SIZE), (h * TILE_SIZE)
            + TILE_SIZE);

    // Define the position in the Texture for current quad
    // Either grass, stone, bush or wall
    if (h == 0 || h == worldHeight-1 ||
        w == 0 || w == worldWidth-1)
    {
        // Use the wall texture
        rVA[currentVertex + 0].texCoords =
            Vector2f(0, 0 + TILE_TYPES * TILE_SIZE);

        rVA[currentVertex + 1].texCoords =
            Vector2f(TILE_SIZE, 0 +
                TILE_TYPES * TILE_SIZE);

        rVA[currentVertex + 2].texCoords =
            Vector2f(TILE_SIZE, TILE_SIZE +
                TILE_TYPES * TILE_SIZE);

        rVA[currentVertex + 3].texCoords =
            Vector2f(0, TILE_SIZE +
                TILE_TYPES * TILE_SIZE);
    }

    // Position ready for the next for vertices
    currentVertex = currentVertex + VERTS_IN_QUAD;
}

return TILE_SIZE;
}

```


The preceding highlighted code sets up the coordinates within the sprite sheet that each vertex is related to. Notice the somewhat long `if` condition. The condition checks whether the current quad is either one of the very first or the very last quads in the arena. If it is (one of the first or last), then this means it is part of the boundary. We can then use a simple formula using `TILE_SIZE` and `TILE_TYPES` to target the wall texture from the sprite sheet.

The array notation and the `texCoords` member are initialized for each vertex, in turn, to assign the appropriate corner of the wall texture within the sprite sheet.

The following code is wrapped in an `else` block. This means that it will run through the nested `for` loop each time the quad does not represent a border/wall tile. Add the following highlighted code among the existing code, and then we will examine it:

```
// Define position in Texture for current quad
// Either grass, stone, bush or wall
if (h == 0 || h == worldHeight-1 ||
    w == 0 || w == worldWidth-1)
{
    // Use the wall texture
    rVA[currentVertex + 0].texCoords =
        Vector2f(0, 0 + TILE_TYPES * TILE_SIZE);

    rVA[currentVertex + 1].texCoords =
        Vector2f(TILE_SIZE, 0 +
            TILE_TYPES * TILE_SIZE);

    rVA[currentVertex + 2].texCoords =
        Vector2f(TILE_SIZE, TILE_SIZE +
            TILE_TYPES * TILE_SIZE);

    rVA[currentVertex + 3].texCoords =
        Vector2f(0, TILE_SIZE +
            TILE_TYPES * TILE_SIZE);
}
else
{
    // Use a random floor texture
    srand((int)time(0) + h * w - h);
    int mOrG = (rand() % TILE_TYPES);
    int verticalOffset = mOrG * TILE_SIZE;

    rVA[currentVertex + 0].texCoords =
        Vector2f(0, 0 + verticalOffset);
```

```

        rVA[currentVertex + 1].texCoords =
            Vector2f(TILE_SIZE, 0 + verticalOffset);

        rVA[currentVertex + 2].texCoords =
            Vector2f(TILE_SIZE, TILE_SIZE + verticalOffset);

        rVA[currentVertex + 3].texCoords =
            Vector2f(0, TILE_SIZE + verticalOffset);
    }

    // Position ready for the next for vertices
    currentVertex = currentVertex + VERTS_IN_QUAD;
}

return TILE_SIZE;
}

```

The preceding highlighted code starts by seeding the random number generator with a formula that will be different in each pass through the loop. Then, the `mOrG` variable is initialized with a number between 0 and `TILE_TYPES`. This is just what we need to pick one of the tile types randomly.



`mOrG` stands for "mud or grass". The name is arbitrary.

Now, we declare and initialize a variable called `verticalOffset` by multiplying `mOrG` by `TileSize`. We now have a vertical reference point within the sprite sheet to the starting height of the randomly chosen texture for the current quad.

Now, we use a simple formula involving `TILE_SIZE` and `verticalOffset` to assign the precise coordinates of each corner of the texture to the appropriate vertex.

We can now put our new function to work in the game engine.

Using the background

We have done the tricky stuff, so this will be simple. There are three steps, as follows:

1. Create a `VertexArray`.
2. Initialize it after leveling up each wave.
3. Draw it in each frame.

Add the following highlighted code to declare a `VertexArray` instance called `background` and load the `background_sheet.png` file as a texture:

```
// Create an instance of the Player class
Player player;

// The boundaries of the arena
IntRect arena;

// Create the background
VertexArray background;
// Load the texture for our background vertex array
Texture textureBackground;
textureBackground.loadFromFile("graphics/background_sheet.png");

// The main game loop
while (window.isOpen())
```

Add the following code to call the `createBackground` function, passing in `background` as a reference and `arena` by value. Notice that, in the highlighted code, we have also modified the way that we initialize the `tileSize` variable. Add the highlighted code exactly as shown:

```
if (state == State::PLAYING)
{
    // Prepare the level
    // We will modify the next two lines later
    arena.width = 500;
    arena.height = 500;
    arena.left = 0;
    arena.top = 0;


    // Pass the vertex array by reference
    // to the createBackground function
    int tileSize = createBackground(background, arena);

    // We will modify this line of code later
    // int tileSize = 50;

    // Spawn the player in the middle of the arena
    player.spawn(arena, resolution, tileSize);

    // Reset the clock so there isn't a frame jump
    clock.restart();
}
```

Note that we have replaced the `int tileSize = 50` line of code because we get the value directly from the return value of the `createBackground` function.

 For the sake of future code clarity, you should delete the `int tileSize = 50` line of code and its related comment. I just commented it out to give the new code a clearer context.

Finally, it is time to do the drawing. This is really simple. All we do is call `window.draw` and pass the `VertexArray` instance, along with the `textureBackground` texture:


```
/*
*****
Draw the scene
*****
*/

if (state == State::PLAYING)
{
    window.clear();

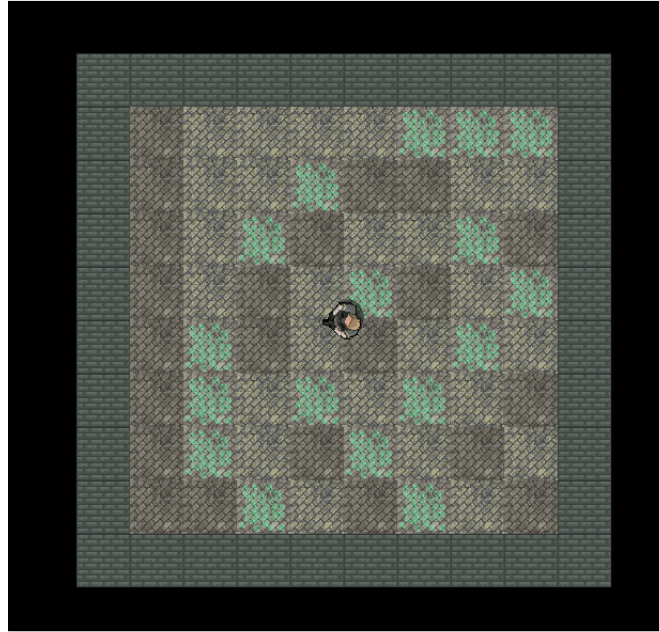
    // Set the mainView to be displayed in the window
    // And draw everything related to it
    window.setView(mainView);

    // Draw the background
    window.draw(background, &textureBackground);

    // Draw the player
    window.draw(player.getSprite());
}
```

 If you are wondering what is going on with the odd-looking `&` sign in front of `textureBackground`, then all will be made clear in the next chapter.

You can now run the game. You will see the following output:



Here, note how the player sprite glides and rotates smoothly within the arena's confines. Although the current code in the main function draws a small arena, the `CreateBackground` function can create an arena of any size. We will see arenas bigger than the screen in *Chapter 13, Sound Effects, File I/O, and Finishing the Game*.

Summary

In this chapter, we discovered C++ references, which are special variables that act as an alias to another variable. When we pass a variable by reference instead of by value, then anything we do on the reference happens to the variable back in the calling function.

We also learned about vertex arrays and created a vertex array full of quads to draw the tiles from a sprite sheet as a background.

The elephant in the room, of course, is that our zombie game doesn't have any zombies. We'll fix that in the next chapter by learning about C++ pointers and the **Standard Template Library (STL)**.

FAQ

Here are some questions that might be on your mind:

Q) Can you summarize these references again?

A) You must initialize a reference immediately, and it cannot be changed to reference another variable. Use references with functions so you are not working on a copy. This is good for efficiency because it avoids making copies and helps us abstract our code into functions more easily.

Q) Is there an easy way to remember the main benefit of using references?

A) To help you remember what a reference is used for, consider this short rhyme:

Moving large objects can make our games choppy,

passing by reference is faster than copy.

10

Pointers, the Standard Template Library, and Texture Management

We will learn a lot as well as get plenty done in terms of the game in this chapter. We will first learn about the fundamental C++ topic of **pointers**. Pointers are variables that hold a memory address. Typically, a pointer will hold the memory address of another variable. This sounds a bit like a reference, but we will see how they are much more powerful and use a pointer to handle an ever-expanding horde of zombies.

We will also learn about the **Standard Template Library (STL)**, which is a collection of classes that allow us to quickly and easily implement common data management techniques.

Once we understand the basics of the STL, we will be able to use that new knowledge to manage all the textures from the game because, if we have 1,000 zombies, we don't really want to load a copy of a zombie graphic into the GPU for each and every one.


We will also dig a little deeper into OOP and use a **static** function, which is a function of a class that can be called without an instance of the class. At the same time, we will see how we can design a class to ensure that only one instance can ever exist. This is ideal when we need to guarantee that different parts of our code will use the same data.

In this chapter, we will cover the following topics:


- Learning about pointers
- Learning about the STL
- Implementing the `TextureHolder` class using static functions and a **singleton** class
- Implementing a pointer to a horde of zombies
- Editing some existing code to use the `TextureHolder` class for the player and background

Learning about Pointers


Pointers can be the cause of frustration while learning to code C++. However, the concept is simple.

 A **pointer** is a variable that holds a memory address.

That's it! There's nothing to be concerned about. What probably causes the frustration to beginners is the syntax—the code we use to handle pointers. We will step through each part of the code for using pointers. You can then begin the ongoing process of mastering them.

 In this section, we will actually learn more about pointers than we need to for this project. In the next project, we will make greater use of pointers. Despite this, we will only scratch the surface of this topic. Further study is definitely recommended, and we will talk more about that in the final chapter.

Rarely do I suggest that memorizing facts, figures, or syntax is the best way to learn. However, memorizing the brief but crucial syntax related to pointers might be worthwhile. This will ensure that the information sinks so deep into our brains that we can never forget it. We can then talk about why we would need pointers at all and examine their relationship to references. A pointer analogy might help:

 If a variable is a house and its contents are the value it holds, then a pointer is the address of the house.

In the previous chapter, while discussing references, we learned that when we pass values to, or return values from, a function, we are actually making a completely new house, but it's exactly the same as the previous one. We are making a copy of the value that's passed to or from a function.

At this point, pointers are probably starting to sound a bit like references. That's because they are a bit like references. Pointers, however, are much more flexible, powerful, and have their own special and unique uses. These special and unique uses require a special and unique syntax.

Pointer syntax

There are two main operators associated with pointers. The first is the **address of** operator:


&

The second is the **dereference** operator:

*


We will now look at the different ways in which we can use these operators with pointers.

The first thing you will notice is that the address of the operator is the same as the reference operator. To add to the woes of an aspiring C++ game programmer, the operators do different things in different contexts. Knowing this from the outset is valuable. If you are staring at some code involving pointers and it seems like you are going mad, know this:

[ You are perfectly sane! You just need to look at the detail of the context.]

Now, you know that if something isn't clear and immediately obvious, it is not your fault. Pointers are not clear and immediately obvious but looking carefully at the context will reveal what is going on.

Armed with the knowledge that you need to pay more attention to pointers than to previous syntax, as well as what the two operators are (**address of** and **dereference**), we can now start to look at some real pointer code.

[ Make sure you have memorized the two operators before proceeding.]

Declaring a pointer

To declare a new pointer, we use the dereference operator, along with the type of variable the pointer will be holding the address of. Take a look at the following code before we talk about pointers some more:

```
// Declare a pointer to hold
// the address of a variable of type int

int* pHealth;
```

The preceding code declares a new pointer called `pHealth` that can hold the address of a variable of the `int` type. Notice I said *can* hold a variable of the `int` type. Like other variables, a pointer also needs to be initialized with a value to make proper use of it.

The name `pHealth`, just like other variables, is arbitrary.



It is common practice to prefix the names of variables that are pointers with a `p`. It is then much easier to remember when we are dealing with a pointer and can then distinguish them from regular variables.

The white space that's used around the dereference operator is optional because C++ rarely cares about spaces in syntax. However, it's recommended because it aids readability. Look at the following three lines of code that all do the same thing.

We have just seen the following format in the previous example, with the dereference operator next to the type:

```
int* pHealth;
```

The following code shows white space either side of the dereference operator:

```
int * pHealth;
```

The following code shows the dereference operator next to the name of the pointer:

```
int *pHealth;
```

It is worth being aware of these possibilities so that when you read code, perhaps on the web, you will understand they are all the same. In this book, we will always use the first option with the dereference operator next to the type.

Just like a regular variable can only successfully contain data of the appropriate type, a pointer should only hold the address of a variable of the appropriate type.

A pointer to the `int` type should not hold the address of a `String`, `Zombie`, `Player`, `Sprite`, `float`, or any other type, except `int`.

Let's see how we can initialize our pointers.

Initializing a pointer

Next, we will see how we can get the address of a variable into a pointer. Take a look at the following code:

```
// A regular int variable called health
int health = 5;

// Declare a pointer to hold the address of a variable of type int
int* pHealth;

// Initialize pHealth to hold the address of health,
// using the "address of" operator
pHealth = &health;
```

In the previous code, we declare an `int` variable called `health` and initialize it to 5. It makes sense, although we have never discussed it before, that this variable must be somewhere in our computer's memory. It must have a memory address.

We can access this address using the **address of** operator. Look closely at the last line of the previous code. We initialize `pHealth` with the address of `health`, like this:

```
pHealth = &health;
```

Our `pHealth` pointer now holds the address of the regular `int`, `health`.



In C++ terminology, we say that `pHealth` points to `health`.

We can use `pHealth` by passing it to a function so that the function can work on `health`, just like we did with references.

There would be no reason for pointers if that was all we were going to do with them, so let's take a look at reinitializing them.

Reinitializing pointers

A pointer, unlike a reference, can be reinitialized to point to a different address. Look at this following code:

```
// A regular int variable called health
int health = 5;
int score = 0;

// Declare a pointer to hold the address of a variable of type int
int* pHealth;

// Initialize pHealth to hold the address of health
pHealth = &health;

// Re-initialize pHealth to hold the address of score
pHealth = &score;
```

Now, `pHealth` points to the `int` variable, `score`.

Of course, the name of our pointer, `pHealth`, is now ambiguous and should perhaps have been called `pIntPtr`. The key thing to understand here is that we *can* do this reassignment.

At this stage, we haven't actually used a pointer for anything other than simply pointing (holding a memory address). Let's see how we can access the value stored at the address that's pointed to by a pointer. This will make them genuinely useful.

Dereferencing a pointer

We know that a pointer holds an address in memory. If we were to output this address in our game, perhaps in our HUD, after it has been declared and initialized, it might look something like this: **9876**.

It is just a value – a value that represents an address in memory. On different operating systems and hardware types, the range of these values will vary. In the context of this book, we never need to manipulate an address directly. We only care about what the value stored at the address that is pointed to is.

The actual addresses used by variables are determined when the game is executed (at runtime) and so there is no way of knowing the address of a variable and hence the value stored in a pointer while we are coding the game.

We can access the value stored at the address that's pointed to by a pointer by using the **dereference** operator:

*

The following code manipulates some variables directly and by using a pointer. Try and follow along and then we will go through it:



Warning! The code that follows is pointless (pun intended). It just demonstrates using pointers.

```
// Some regular int variables
int score = 0;
int hiScore = 10;

// Declare 2 pointers to hold the addresses of int
int* pIntPtr1;
int* pIntPtr2;

// Initialize pIntPtr1 to hold the address of score
pIntPtr1 = &score;

// Initialize pIntPtr2 to hold the address of hiScore
pIntPtr2 = &hiScore;

// Add 10 to score directly
score += 10;
// Score now equals 10

// Add 10 to score using pIntPtr1
*pIntPtr1 += 10;
// score now equals 20. A new high score

// Assign the new hi score to hiScore using only pointers
*pIntPtr2 = *pIntPtr1;
// hiScore and score both equal 20
```

In the previous code, we declare two int variables, `score` and `hiScore`. We then initialize them with the values 0 and 10, respectively. Next, we declare two pointers to int. These are `pIntPtr1` and `pIntPtr2`. We initialize them in the same step as declaring them to hold the addresses of (point to) the `score` and `hiScore` variables, respectively.

Following on, we add 10 to `score` in the usual way, `score += 10`. Then, we can see that by using the dereference operator on a pointer, we can access the value stored at the address they point to. The following code changed the value stored by the variable that's pointed to by `pIntPtr1`:

```
// Add 10 to score using pIntPtr1
*pIntPtr1 += 10;
// score now equals 20, A new high score
```

The last part of the preceding code dereferences both pointers to assign the value that's pointed to by `pIntPtr1` as the value that's pointed to by `pIntPtr2`:

```
// Assign the new hi-score to hiScore with only pointers
*pIntPtr2 = *pIntPtr1;
// hiScore and score both equal 20
```

Both `score` and `hiScore` are now equal to 20.

Pointers are versatile and powerful

We can do so much more with pointers. Here are just a few useful things we can do.

Dynamically allocated memory

All the pointers we have seen so far point to memory addresses that have a scope limited only to the function they are created in. So, if we declare and initialize a pointer to a local variable, when the function returns, the pointer, the local variable, and the memory address will be gone. They are out of scope.

Up until now, we have been using a fixed amount of memory that is decided in advance of the game being executed. Furthermore, the memory we have been using is controlled by the operating system, and variables are lost and created as we call and return from functions. What we need is a way to use memory that is always in scope until we are finished with it. We want to have access to memory we can call our own and take responsibility for.

When we declare variables (including pointers), they are in an area of memory known as **the stack**. There is another area of memory which, although allocated and controlled by the operating system, can be allocated at runtime. This other area of memory is called the **free store**, or sometimes, the **heap**.



Memory on the heap does not have scope to a specific function.
Returning from a function does not delete the memory on the heap.

This gives us great power. With access to memory that is only limited by the resources of the computer our game is running on, we can plan games with huge amounts of objects. In our case, we want a vast horde of zombies. As Spiderman's uncle wouldn't hesitate to remind us, however, "with great power comes great responsibility."

Let's look at how we can use pointers to take advantage of the memory on the free store and how we can release that memory back to the operating system when we are finished with it.

To create a pointer that points to a value on the heap, we need a pointer:

```
int* pToInt = nullptr;
```

In the previous line of code, we declare a pointer in the same way we have seen before, but since we are not initializing it to point to a variable, we initialize it to `nullptr`. We do this because it is good practice. Consider dereferencing a pointer (changing a value at the address it points to) when you don't even know what it is pointing to. It would be the programming equivalent of going to the shooting range, blindfolding someone, spinning them around, and telling them to shoot. By pointing a pointer to nothing (`nullptr`), we can't do any harm with it.

When we are ready to request memory on the free store, we use the `new` keyword, as shown in the following line of code:

```
pToInt = new int;
```

`pToInt` now holds the memory address of space on the free store that is just the right size to hold an `int` value.



Any allocated memory is returned when the program ends. It is, however, important to realize that this memory will never be freed (within the execution of our game) unless we free it. If we continue to take memory from the free store without giving it back, eventually it will run out and the game will crash.

It is unlikely that we would ever run out of memory by occasionally taking `int` sized chunks of the free store. But if our program has a function or loop that requests memory and this function or loop is executed regularly throughout the game, eventually the game will slow and then crash. Furthermore, if we allocate lots of objects on the free store and don't manage them correctly, then this situation can happen quite quickly.

The following line of code hands back (deletes) the memory on the free store that was previously pointed to by `pToInt`:

```
delete pToInt;
```

Now, the memory that was previously pointed to by `pToInt` is no longer ours to do what we like with; we must take precautions. Although the memory has been handed back to the operating system, `pToInt` still holds the address of this memory, which no longer belongs to us.

The following line of code ensures that `pToInt` can't be used to attempt to manipulate or access this memory:

```
pToInt = nullptr;
```



If a pointer points to an address that is invalid, it is called a **wild** or **dangling** pointer. If you attempt to dereference a dangling pointer and if you are lucky, the game will crash, and you will get a memory access violation error. If you are unlucky, you will create a bug that will be incredibly difficult to find. Furthermore, if we use memory on the free store that will persist beyond the life of a function, we must make sure to keep a pointer to it or we will have leaked memory.

Now, we can declare pointers and point them to newly allocated memory on the free store. We can manipulate and access the memory they point to by dereferencing them. We can also return memory to the free store when we are done with it, and we know how to avoid having a dangling pointer.

Let's look at some more advantages of pointers.

Passing a pointer to a function

In order to pass a pointer to a function, we need to write a function that has a pointer in the prototype, like in the following code:

```
void myFunction(int *pInt)
{
    // Dereference and increment the value stored
    // at the address pointed to by the pointer
    *pInt ++
    return;
}
```

The preceding function simply dereferences the pointer and adds 1 to the value stored at the pointed to address.

Now, we can use that function and pass the address of a variable or another pointer to a variable explicitly:

```
int someInt = 10;
int* pToInt = &someInt;

myFunction(&someInt);
// someInt now equals 11

myFunction(pToInt);
// someInt now equals 12
```

As shown in the previous code, within the function, we are manipulating the variable from the calling code and can do so using the address of a variable or a pointer to that variable, since both actions amount to the same thing.

Pointers can also point to instances of a class.

Declaring and using a pointer to an object

Pointers are not just for regular variables. We can also declare pointers to user-defined types such as our classes. This is how we would declare a pointer to an object of the `Player` type:

```
Player player;
Player* pPlayer = &Player;
```

We can even access the member functions of a `Player` object directly from the pointer, as shown in the following code:

```
// Call a member function of the player class
pPlayer->moveLeft()
```

Notice the subtle but vital difference: accessing a function with a pointer to an object rather than an object directly uses the `->` operator. We won't need to use pointers to objects in this project, but we will explore them more carefully before we do, which will be in the next project.

Let's go over one more new pointer topic before we talk about something completely new.

Pointers and arrays

Arrays and pointers have something in common. An array's name is a memory address. More specifically, the name of an array is the memory address of the first element in that array. To put this yet another way, an array name points to the first element of an array. The best way to understand this is to read on and look at the following example.

We can create a pointer to the type that an array holds and then use the pointer in the same way using exactly the same syntax that we would use for the array:

```
// Declare an array of ints
int arrayOfInts[100];

// Declare a pointer to int and initialize it
// with the address of the first
// element of the array, arrayOfInts
int* pToIntArray = arrayOfInts;

// Use pToIntArray just as you would arrayOfInts
arrayOfInts[0] = 999;
// First element of arrayOfInts now equals 999

pToIntArray[0] = 0;
// First element of arrayOfInts now equals 0
```

This also means that a function that has a prototype that accepts a pointer also accepts arrays of the type the pointer is pointing to. We will use this fact when we build our ever-increasing horde of zombies.



Regarding the relationship between pointers and references, the compiler actually uses pointers when implementing our references. This means that references are just a handy tool (that uses pointers "under the hood"). You could think of a reference as an automatic gearbox that is fine and convenient for driving around town, whereas pointers are a manual gearbox – more complicated, but with the correct use, they can provide better results/performance/flexibility.

Summary of pointers

Pointers are a bit fiddly at times. In fact, our discussion of pointers was only an introduction to the subject. The only way to get comfortable with them is to use them as much as possible. All you need to understand about pointers in order to complete this project is the following:

- Pointers are variables that store a memory address.
- We can pass pointers to functions to directly manipulate values from the calling function's scope, within the called function.
- Array names hold the memory address of the first element. We can pass this address as a pointer because that is exactly what it is.
- We can use pointers to point to memory on the free store. This means we can dynamically allocate large amounts of memory while the game is running.



There are yet more ways to use pointers. We will learn about **smart pointers** in the final project, once we have got used to using regular pointers.

There is just one more topic to cover before we can start coding the Zombie Arena project again.

The Standard Template Library

The **Standard Template Library (STL)** is a collection of data containers and ways to manipulate the data we put in those containers. If we want to be more specific, it is a way to store and manipulate different types of C++ variables and classes.

We can think of the different containers as customized and more advanced arrays. The STL is part of C++. It is not an optional thing that needs to be set up like SFML.

The STL is part of C++ because its containers and the code that manipulates them are fundamental to many types of code that many apps will need to use.


In short, the STL implements code that we and just about every C++ programmer is almost bound to need, at least at some point, and probably quite regularly.

If we were to write our own code to contain and manage our data, then it is unlikely we would write it as efficiently as the people who wrote the STL.


So, by using the STL, we guarantee that we are using the best written code possible to manage our data. Even SFML uses the STL. For example, under the hood, the `VertexArray` class uses the STL.

All we need to do is choose the right type of container from those that are available. The types of container that are available through the STL include the following:

- **Vector:** This is like an array with boosters. It handles dynamic resizing, sorting, and searching. This is probably the most useful container.
- **List:** A container that allows for the ordering of the data.
- **Map:** An associative container that allows the user to store data as key/value pairs. This is where one piece of data is the "key" to finding the other piece. A map can also grow and shrink, as well as be searched.
- **Set:** A container that guarantees that every element is unique.

 For a full list of STL container types, their different uses, and explanations, take a look at the following link: http://www.tutorialspoint.com/cplusplus/cpp_stl_tutorial.htm.

In the Zombie Arena game, we will use a map.

 If you want a glimpse into the kind of complexity that the STL is sparing us, then take a look at this tutorial, which implements the kind of thing that a list would do. Note that the tutorial implements only the very simplest bare-bones implementation of a list: <http://www.sanfoundry.com/cpp-program-implement-single-linked-list/>.

We can easily see that we will save a lot of time and end up with a better game if we explore the STL. Let's take a closer look at how to use a Map instance, and then we will see how it will be useful to us in the Zombie Arena game.

What is a map?

A **map** is a container that is dynamically resizable. We can add and remove elements with ease. What makes the map class special compared to the other containers in the STL is the way that we access the data within it.

The data in a map instance is stored in pairs. Consider a situation where you log in to an account, perhaps with a username and password. A map would be perfect for looking up the username and then checking the value of the associated password.

A map would also be just right for things such as account names and numbers, or perhaps company names and share prices.

Note that when we use `map` from the STL, we decide the type of values that form the key-value pairs. The values could be `string` instances and `int` instances, such as account numbers; `string` instances and other `string` instances such as usernames and passwords; or user-defined types such as objects.

What follows is some real code to make us familiar with `map`.

Declaring a map

This is how we could declare a `map`:

```
map<string, int> accounts;
```

The previous line of code declares a new `map` called `accounts` that has a key of `string` objects, each of which will refer to a value that is an `int`.

We can now store key-value pairs of the `string` type that refer to values of the `int` type. We will see how we can do this next.

Adding data to a Map

Let's go ahead and add a key-value pair to `accounts`:

```
accounts["John"] = 1234567;
```

Now, there is an entry in the `map` that can be accessed using the key of `John`. The following code adds two more entries to the `accounts` `map`:

```
accounts["Smit"] = 7654321;  
accounts["Larissa"] = 8866772;
```

Our `map` has three entries in it. Let's see how we can access the account numbers.

Finding data in a map

We would access the data in the same way that we added it: by using the key. As an example, we could assign the value stored by the `Smit` key to a new `int`, `accountNumber`, like this:

```
int accountNumber = accounts["Smit"];
```

The `int` variable, `accountNumber`, now stores the value `7654321`. We can do anything to a value stored in a `map` instance that we can do to that type.

Removing data from a map

Taking values out of our map is also straightforward. The following line of code removes the key, John, and its associated value:

```
accounts.erase("John");
```

Let's look at a few more things we can do with a map.

Checking the size of a map

We might like to know how many key-value pairs we have in our map. The following line of code does just that:

```
int size = accounts.size();
```

The `int` variable, `size`, now holds the value of 2. This is because `accounts` holds values for Smit and Larissa, because we deleted John.

Checking for keys in a map

The most relevant feature of `map` is its ability to find a value using a key. We can test for the presence or otherwise of a specific key like this:

```
if(accounts.find("John") != accounts.end())
{
    // This code won't run because John was erased
}

if(accounts.find("Smit") != accounts.end())
{
    // This code will run because Smit is in the map
}
```

In the previous code, the `!= accounts.end` value is used to determine when a key does or doesn't exist. If the searched for key is not present in the map, then `accounts.end` will be the result of the `if` statement.

Let's see how we can test or use all the values in a map by looping through a map.

Looping/iterating through the key-value pairs of a map

We have seen how we can use a `for` loop to loop/iterate through all the values of an array. But, what if we want to do something like this to a map?

The following code shows how we could loop through each key-value pair of the account's map and add one to each of the account numbers:

```
for (map<string,int>::iterator it = accounts.begin();
    it != accounts.end();
    ++ it)
{
    it->second += 1;
}
```

The condition of the `for` loop is probably the most interesting part of the previous code. The first part of the condition is the longest part. `map<string,int>::iterator it = accounts.begin()` is more understandable if we break it down.

`map<string,int>::iterator` is a type. We are declaring an iterator that's suitable for a map with key-value pairs of `string` and `int`. The iterator's name is `it`. We assign the value that's returned by `accounts.begin()` to `it`. The iterator, `it`, now holds the first key-value pair from the `accounts` map.

The rest of the condition of the `for` loop works as follows. `it != accounts.end()` means the loop will continue until the end of the map is reached, and `++it` simply steps to the next key-value pair in the map, each pass through the loop.

Inside the `for` loop, `it->second` accesses the value of the key-value pair and `+= 1` adds one to the value. Note that we can access the key (which is the first part of the key-value pair) with `it->first`.

You might have noticed that the syntax for setting up a loop through a map is quite verbose. C++ has a way to cut down on this verbosity.

The auto keyword

The code in the condition of the `for` loop was quite verbose – especially in terms of `map<string, int>::iterator`. C++ supplies a neat way to reduce verbosity with the `auto` keyword. Using the `auto` keyword, we can improve the previous code:

```
for (auto it = accounts.begin(); it != accounts.end(); ++ it)
{
    it->second += 1;
}
```

The `auto` keyword instructs the compiler to automatically deduce the type for us. This will be especially useful with the next class that we write.

STL summary

As with almost every C++ concept that we have covered in this book, the STL is a massive topic. Whole books have been written covering just the STL. At this point, however, we know enough to build a class that uses the STL `map` to store SFML Texture objects. We can then have textures that can be retrieved/loaded by using the filename as the key of the key-value pair.

The reason why we would go to this extra level of complexity and not just carry on using the `Texture` class the same way as we have been so far will become apparent as we proceed.

The TextureHolder class

Thousands of zombies represent a new challenge. Not only would loading, storing, and manipulating thousands of copies of three different zombie textures take up a lot of memory, but also a lot of processing power. We will create a new type of class that overcomes this problem and allows us to store just one of each texture.

We will also code the class in such a way that there can only ever be one instance of it. This type of class is called a **singleton**.



A singleton is a design pattern. A design pattern is a way to structure our code that is proven to work.

Furthermore, we will also code the class so that it can be used anywhere in our game code directly through the class name, without access to an instance.

Coding the TextureHolder header file

Let's make a new header file. Right-click **Header Files** in the **Solution Explorer** and select **Add | New Item...** In the **Add New Item** window, highlight (by left-clicking) **Header File (.h)**, and then in the **Name** field, type `TextureHolder.h`.

Add the code that follows into the `TextureHolder.h` file, and then we can discuss it:

```
#pragma once
#ifndef TEXTURE_HOLDER_H
#define TEXTURE_HOLDER_H

#include <SFML/Graphics.hpp>
#include <map>

using namespace sf;
using namespace std;

class TextureHolder
{
private:
    // A map container from the STL,
    // that holds related pairs of String and Texture
    map<    string, Texture> m_Textures;

    // A pointer of the same type as the class itself
    // the one and only instance
    static TextureHolder* m_s_Instance;

public:
    TextureHolder();
    static Texture& GetTexture(string const& filename);

};

#endif
```

In the previous code, notice that we have an `include` directive for `map` from the STL. We declare a `map` instance that holds the `string` type and the SFML `Texture` type, as well as the key-value pairs. The `map` is called `m_Textures`.

In the preceding code, this line follows on:

```
static TextureHolder* m_s_Instance;
```

The previous line of code is quite interesting. We are declaring a static pointer to an object of the `TextureHolder` type called `m_s_Instance`. This means that the `TextureHolder` class has an object that is the same type as itself. Not only that, but because it is static, it can be used through the class itself, without an instance of the class. When we code the related `.cpp` file, we will see how we can use this.

In the `public` part of the class, we have the prototype for the constructor function, `TextureHolder`. The constructor takes no arguments and, as usual, has no return type. This is the same as the default constructor. We are going to override the default constructor with a definition that makes our singleton work how we want it to.

We have another function called `GetTexture`. Let's look at the signature again and analyze exactly what is happening:

```
static Texture& GetTexture(string const& filename);
```

First, notice that the function returns a reference to a `Texture`. This means that `GetTexture` will return a reference, which is efficient because it avoids making a copy of what could be a large graphic. Also, notice that the function is declared as `static`. This means that the function can be used without an instance of the class. The function takes a `string` as a constant reference, as a parameter. The effect of this is two-fold. Firstly, the operation is efficient and secondly, because the reference is constant, it can't be changed.

Coding the TextureHolder function definitions

Now, we can create a new `.cpp` file that will contain the function definition. This will allow us to see the reasons behind our new types of functions and variables. Right-click **Source Files** in the **Solution Explorer** and select **Add | New Item....** In the **Add New Item** window, highlight (by left-clicking) **C++ File (.cpp)**, and then in the **Name** field, type `TextureHolder.cpp`. Finally, click the **Add** button. We are now ready to code the class.

Add the following code, and then we can discuss it:

```
#include "TextureHolder.h"

// Include the "assert feature"
#include <assert.h>

TextureHolder* TextureHolder::m_s_Instance = nullptr;

TextureHolder::TextureHolder()
{
```

```

        assert(m_s_Instance == nullptr);
        m_s_Instance = this;
    }

```

In the previous code, we initialize our pointer of the `TextureHolder` type to `nullptr`. In the constructor, `assert(m_s_Instance == nullptr)` ensures that `m_s_Instance` equals `nullptr`. If it doesn't the game will exit execution. Then, `m_s_Instance = this` assigns the pointer to this instance. Now, consider where this code is taking place. The code is in the constructor. The constructor is the way that we create instances of objects from classes. So, effectively, we now have a pointer to a `TextureHolder` that points to the one and only instance of itself.

Add the final part of the code to the `TextureHolder.cpp` file. There are more comments than code here. Examine the following code and read the comments as you add the code, and then we can go through it:

```

Texture& TextureHolder::GetTexture(string const& filename)
{
    // Get a reference to m_Textures using m_s_Instance
    auto& m = m_s_Instance->m_Textures;
    // auto is the equivalent of map<string, Texture>


    // Create an iterator to hold a key-value-pair (kvp)
    // and search for the required kvp
    // using the passed in file name
    auto keyValuePair = m.find(filename);
    // auto is equivalent of map<string, Texture>::iterator

    // Did we find a match?
    if (keyValuePair != m.end())
    {
        // Yes
        // Return the texture,
        // the second part of the kvp, the texture
        return keyValuePair->second;
    }
    else
    {
        // File name not found
        // Create a new key value pair using the filename
        auto& texture = m[filename];
        // Load the texture from file in the usual way
        texture.loadFromFile(filename);
    }
}

```

```
        // Return the texture to the calling code
        return texture;
    }
}
```

The first thing you will probably notice about the previous code is the `auto` keyword. The `auto` keyword was explained in the previous section.

 If you want to know what the actual types that have been replaced by `auto` are, then look at the comments immediately after each use of `auto` in the previous code.

At the start of the code, we get a reference to `m_textures`. Then, we attempt to get an iterator to the key-value pair represented by the passed-in filename (`filename`). If we find a matching key, we return the texture with `return keyValuePair->second`. Otherwise, we add the texture to the map and then return it to the calling code.

Admittedly, the `TextureHolder` class introduced lots of new concepts (singletons, static functions, constant references, `this`, and the `auto` keyword,) and syntax. Add to this the fact that we have only just learned about pointers and the STL, and this section's code might have been a little daunting.

So, was it all worth it?

What have we achieved with `TextureHolder`?

The point is that now that we have this class, we can go wild using textures from wherever we like in our code and not worry about running out of memory or having access to any texture in a particular function or class. We will see how to use `TextureHolder` soon.

Building a horde of zombies

Now, we are armed with the `TextureHolder` class to make sure that our zombie textures are easily available as well as only loaded into the GPU once. Then, we can investigate creating a whole horde of them.

We will store zombies in an array. Since the process of building and spawning a horde of zombies involves quite a few lines of code, it is a good candidate for abstracting to a separate function. Soon, we will code the `CreateHorde` function but first, of course, we need a `Zombie` class.

Coding the Zombie.h file

The first step to building a class to represent a zombie is to code the member variables and function prototypes in a header file.

Right-click **Header Files** in the **Solution Explorer** and select **Add | New Item....** In the **Add New Item** window, highlight (by left-clicking) **Header File (.h)**, and then in the **Name** field, type `Zombie.h`.

Add the following code to the `Zombie.h` file:

```
#pragma once
#include <SFML/Graphics.hpp>

using namespace sf;

class Zombie
{
private:
    // How fast is each zombie type?
    const float BLOATER_SPEED = 40;
    const float CHASER_SPEED = 80;
    const float CRAWLER_SPEED = 20;

    // How tough is each zombie type
    const float BLOATER_HEALTH = 5;
    const float CHASER_HEALTH = 1;
    const float CRAWLER_HEALTH = 3;

    // Make each zombie vary its speed slightly
    const int MAX_VARRIANCE = 30;
    const int OFFSET = 101 - MAX_VARRIANCE;

    // Where is this zombie?
    Vector2f m_Position;

    // A sprite for the zombie
    Sprite m_Sprite;

    // How fast can this one run/crawl?
    float m_Speed;

    // How much health has it got?
    float m_Health;
```

```
// Is it still alive?
bool m_Alive;

// Public prototypes go here
};
```

The previous code declares all the private member variables of the `Zombie` class. At the top of the previous code, we have three constant variables to hold the speed of each type of zombie: a very slow `Crawler`, a slightly faster `Bloater`, and a somewhat speedy `Chaser`. We can experiment with the value of these three constants to help balance the difficulty level of the game. It's also worth mentioning here that these three values are only used as a starting value for the speed of each zombie type. As we will see later in this chapter, we will vary the speed of every zombie by a small percentage from these values. This stops zombies of the same type from bunching up together as they pursue the player.

The next three constants determine the health level for each zombie type. Note that `Bloaters` are the toughest, followed by `Crawlers`. As a matter of balance, the `Chaser` zombies will be the easiest to kill.

Next, we have two more constants, `MAX_VARRIANCE` and `OFFSET`. These will help us determine the individual speed of each zombie. We will see exactly how when we code the `Zombie.cpp` file.

After these constants, we declare a bunch of variables that should look familiar because we had very similar variables in our `Player` class. The `m_Position`, `m_Sprite`, `m_Speed`, and `m_Health` variables are for what their names imply: the position, sprite, speed, and health of the zombie object.

Finally, in the preceding code, we declare a Boolean called `m_Alive`, which will be true when the zombie is alive and hunting, but false when its health gets to zero and it is just a splurge of blood on our otherwise pretty background.

Now, we can complete the `Zombie.h` file. Add the function prototypes highlighted in the following code, and then we will talk about them:

```
// Is it still alive?
bool m_Alive;

// Public prototypes go here
public:

// Handle when a bullet hits a zombie
bool hit();

// Find out if the zombie is alive
```

```
bool isAlive();

// Spawn a new zombie
void spawn(float startX, float startY, int type, int seed);

// Return a rectangle that is the position in the world
FloatRect getPosition();

// Get a copy of the sprite to draw
Sprite getSprite();

// Update the zombie each frame
void update(float elapsedTime, Vector2f playerLocation);

};
```

In the previous code, there is a `hit` function, which we can call every time the zombie is hit by a bullet. The function can then take the necessary steps, such as taking health from the zombie (reducing the value of `m_Health`) or killing it dead (setting `m_Alive` to false).

The `isAlive` function returns a Boolean that lets the calling code know whether the zombie is alive or dead. We don't want to perform collision detection or remove health from the player for walking over a blood splat.

The `spawn` function takes a starting position, a type (Crawler, Bloater, or Chaser, represented by an `int`), as well as a seed to use in some random number generation that we will see in the next section.

Just like we have in the `Player` class, the `Zombie` class has `getPosition` and `getSprite` functions to get a rectangle that represents the space occupied by the zombie and the sprite that can be drawn each frame.

The last prototype in the previous code is the `update` function. We could have probably guessed that it would receive the elapsed time since the last frame, but also notice that it receives a `Vector2f` vector called `playerLocation`. This vector will indeed be the exact coordinates of the center of the player. We will soon see how we can use this vector to chase after the player.

Now, we can code the function definitions in the `.cpp` file.

Coding the Zombie.cpp file

Next, we will code the functionality of the `Zombie` class — the function definitions.

Create a new `.cpp` file that will contain the function definitions. Right-click **Source Files** in the **Solution Explorer** and select **Add | New Item....** In the **Add New Item** window, highlight (by left-clicking) **C++ File (.cpp)**, and then in the **Name** field, type `Zombie.cpp`. Finally, click the **Add** button. We are now ready to code the class.

Add the following code to the `Zombie.cpp` file:

```
#include "zombie.h"
#include "TextureHolder.h"
#include <cstdlib>
#include <ctime>

using namespace std;
```

First, we add the necessary include directives and then `using namespace std`. You might remember a few instances when we prefixed our object declarations with `std::`. This `using` directive means we don't need to do that for the code in this file.

Now, add the following code, which is the definition of the `spawn` function. Study the code once you have added it, and then we will discuss it:

```
void Zombie::spawn(float startX, float startY, int type, int seed)
{

    switch (type)
    {
    case 0:
        // Bloater
        m_Sprite = Sprite(TextureHolder::GetTexture(
            "graphics/bloater.png"));

        m_Speed = BLOATER_SPEED;
        m_Health = BLOATER_HEALTH;
        break;

    case 1:
        // Chaser
        m_Sprite = Sprite(TextureHolder::GetTexture(
            "graphics/chaser.png"));

        m_Speed = CHASER_SPEED;
```

```

        m_Health = CHASER_HEALTH;
        break;

    case 2:
        // Crawler
        m_Sprite = Sprite(TextureHolder::GetTexture(
            "graphics/crawler.png"));

        m_Speed = CRAWLER_SPEED;
        m_Health = CRAWLER_HEALTH;
        break;
    }

    // Modify the speed to make the zombie unique
    // Every zombie is unique. Create a speed modifier
    srand((int)time(0) * seed);

    // Somewhere between 80 and 100
    float modifier = (rand() % MAX_VARRIANCE) + OFFSET;

    // Express this as a fraction of 1
    modifier /= 100; // Now equals between .7 and 1
    m_Speed *= modifier;

    // Initialize its location
    m_Position.x = startX;
    m_Position.y = startY;

    // Set its origin to its center
    m_Sprite.setOrigin(25, 25);

    // Set its position
    m_Sprite.setPosition(m_Position);
}

```

The first thing the function does is switch paths of execution based on the `int` value, which is passed in as a parameter. Within the `switch` block, there is a `case` for each type of zombie. Depending on the type of zombie, the appropriate texture, speed, and health is initialized to the relevant member variables.



We could have used an enumeration for the different types of zombie. Feel free to upgrade your code when the project is finished.

Of interest here is that we use the static `TextureHolder::GetTexture` function to assign the texture. This means that no matter how many zombies we spawn, there will be a maximum of three textures in the memory of the GPU.

The next three lines of code (excluding comments) do the following:

- Seed the random number generator with the `seed` variable that was passed in as a parameter.
- Declare and initialize the `modifier` variable using the `rand` function and the `MAX_VARRIANCE` and `OFFSET` constants. The result is a fraction between zero and one, which can be used to make each zombie's speed unique. The reason we want to do this is so that the zombies don't bunch up together on top of each other too much.
- We can now multiply `m_Speed` by `modifier` and we will have a zombie whose speed is within the `MAX_VARRIANCE` percent of the constant defined for this type of zombie's speed.

After we have resolved the speed, we assign the passed-in position held in `startX` and `startY` to `m_Position.x` and `m_Position.y`, respectively.

The last two lines of code in the previous listing set the origin of the sprite to the center and use the `m_Position` vector to set the position of the sprite.

Now, add the following code for the `hit` function to the `Zombie.cpp` file:

```
bool Zombie::hit()
{
    m_Health--;

    if (m_Health < 0)
    {
        // dead
        m_Alive = false;
        m_Sprite.setTexture(TextureHolder::GetTexture(
            "graphics/blood.png"));

        return true;
    }

    // injured but not dead yet
    return false;
}
```

The `hit` function is nice and simple: reduce `m_Health` by one and then check whether `m_Health` is below zero.

If it is below zero, then it sets `m_Alive` to false, swaps the zombie's texture for a blood splat, and returns `true` to the calling code so that it knows the zombie is now dead. If the zombie has survived, the hit returns `false`.

Add the following three getter functions, which just return a value to the calling code:

```
bool Zombie::isAlive()
{
    return m_Alive;
}

FloatRect Zombie::getPosition()
{
    return m_Sprite.getGlobalBounds();
}

Sprite Zombie::getSprite()
{
    return m_Sprite;
}
```

The previous three functions are quite self-explanatory, perhaps with the exception of the `getPosition` function, which uses the `m_Sprite.getLocalBounds` function to get the `FloatRect` instance, which is then returned to the calling code.

Finally, for the `Zombie` class, we need to add the code for the `update` function. Look closely at the following code, and then we will go through it:

```
void Zombie::update(float elapsedTime,
    Vector2f playerLocation)
{
    float playerX = playerLocation.x;
    float playerY = playerLocation.y;

    // Update the zombie position variables
    if (playerX > m_Position.x)
    {
        m_Position.x = m_Position.x +
            m_Speed * elapsedTime;
    }

    if (playerY > m_Position.y)
    {
        m_Position.y = m_Position.y +
            m_Speed * elapsedTime;
    }
}
```

```
    }

    if (playerX < m_Position.x)
    {
        m_Position.x = m_Position.x -
            m_Speed * elapsedTime;
    }

    if (playerY < m_Position.y)
    {
        m_Position.y = m_Position.y -
            m_Speed * elapsedTime;
    }

    // Move the sprite
    m_Sprite.setPosition(m_Position);

    // Face the sprite in the correct direction
    float angle = (atan2(playerY - m_Position.y,
        playerX - m_Position.x)
        * 180) / 3.141;

    m_Sprite.setRotation(angle);

}
```

In the preceding code, we copy `playerLocation.x` and `playerLocation.y` into the local variables called `playerX` and `playerY`.

Next, there are four `if` statements. They test to see whether the zombie is to the left, right, above, or below the current player's position. These four `if` statements, when they evaluate to true, adjust the zombie's `m_Position.x` and `m_Position.y` values appropriately using the usual formula, that is, speed multiplied by time since last frame. More specifically, the code is `m_Speed * elapsedTime`.

After the four `if` statements, `m_Sprite` is moved to its new location.

We then use the same calculation we previously used with the player and the mouse pointer, but this time, we do so for the zombie and the player. This calculation finds the angle that's needed to face the zombie toward the player.

Finally, for this function and the class, we call `m_Sprite.setRotation` to actually rotate the zombie sprite. Remember that this function will be called for every zombie (that is alive) on every frame of the game.

But, we want a whole horde of zombies.

Using the Zombie class to create a horde

Now that we have a class to create a living, attacking, and killable zombie, we want to spawn a whole horde of them.

To achieve this, we will write a separate function and we will use a pointer so that we can refer to our horde that will be declared in `main` but configured in a different scope.

Open the `ZombieArena.h` file in Visual Studio and add the following highlighted lines of code:

```
#pragma once
#include "Zombie.h"

using namespace sf;

int createBackground(VertexArray& rVA, IntRect arena);
Zombie* createHorde(int numZombies, IntRect arena);
```

Now that we have a prototype, we can code the function definition.

Create a new `.cpp` file that will contain the function definition. Right-click **Source Files** in the **Solution Explorer** and select **Add | New Item....** In the **Add New Item** window, highlight (by left-clicking) **C++ File (.cpp)**, and then in the **Name** field, type `CreateHorde.cpp`. Finally, click the **Add** button.

Add in the following code to the `CreateHorde.cpp` file and study it. Afterward, we will break it down into chunks and discuss it:

```
#include "ZombieArena.h"
#include "Zombie.h"

Zombie* createHorde(int numZombies, IntRect arena)
{
    Zombie* zombies = new Zombie[numZombies];

    int maxY = arena.height - 20;
    int minY = arena.top + 20;
```

```
int maxX = arena.width - 20;
int minX = arena.left + 20;

for (int i = 0; i < numZombies; i++)
{
    // Which side should the zombie spawn
    srand((int)time(0) * i);
    int side = (rand() % 4);
    float x, y;

    switch (side)
    {
    case 0:
        // left
        x = minX;
        y = (rand() % maxY) + minY;
        break;

    case 1:
        // right
        x = maxX;
        y = (rand() % maxY) + minY;
        break;

    case 2:
        // top
        x = (rand() % maxX) + minX;
        y = minY;
        break;

    case 3:
        // bottom
        x = (rand() % maxX) + minX;
        y = maxY;
        break;
    }

    // Bloater, crawler or runner
    srand((int)time(0) * i * 2);
    int type = (rand() % 3);

    // Spawn the new zombie into the array
```

```

        zombies[i].spawn(x, y, type, i);
    }
    return zombies;
}

```

Let's look at all the previous code again, in bite-size pieces. First, we added the now familiar include directives:

```

#include "ZombieArena.h"
#include "Zombie.h"

```

Next comes the function signature. Notice that the function must return a pointer to a `Zombie` object. We will be creating an array of `Zombie` objects. Once we are done creating the horde, we will return the array. When we return the array, we are actually returning the address of the first element of the array. This, as we learned in the section on pointers earlier in this chapter, is the same thing as a pointer. The signature also shows that we have two parameters. The first, `numZombies`, will be the number of zombies this current horde requires and the second, `arena`, is an `IntRect` that holds the size of the current arena in which to create this horde.

After the function signature, we declare a pointer to the `Zombie` type called `zombies` and initialize it with the memory address of the first element of an array, which we dynamically allocate on the heap:

```

Zombie* createHorde(int numZombies, IntRect arena)
{
    Zombie* zombies = new Zombie[numZombies];
}

```

The next part of the code simply copies the extremities of the arena into `maxY`, `minY`, `maxX`, and `minX`. We subtract twenty pixels from the right and bottom while adding twenty pixels to the top and left. We use these four local variables to help position each of the zombies. We made the twenty-pixel adjustments to stop the zombies appearing on top of the walls:

```

int maxY = arena.height - 20;
int minY = arena.top + 20;
int maxX = arena.width - 20;
int minX = arena.left + 20;

```

Now, we enter a `for` loop that will loop through each of the `Zombie` objects in the `zombies` array from zero through to `numZombies`:

```

for (int i = 0; i < numZombies; i++)

```


Inside the `for` loop, the first thing the code does is seed the random number generator and then generate a random number between zero and three. This number is stored in the `side` variable. We will use the `side` variable to decide whether the zombie spawns at the left, top, right, or bottom of the arena. We also declare two `int` variables, `x` and `y`. These two variables will temporarily hold the actual horizontal and vertical coordinates of the current zombie:

```
// Which side should the zombie spawn
srand((int)time(0) * i);
int side = (rand() % 4);
float x, y;
```

Still inside the `for` loop, we have a `switch` block with four `case` statements. Note that the `case` statements are for 0, 1, 2, and 3, and that the argument in the `switch` statement is `side`. Inside each of the `case` blocks, we initialize `x` and `y` with one predetermined value, either `minX`, `maxX`, `minY`, or `maxY`, and one randomly generated value. Look closely at the combinations of each predetermined and random value. You will see that they are appropriate for positioning the current zombie randomly across either the left side, top side, right side, or bottom side. The effect of this will be that each zombie can spawn randomly, anywhere on the outside edge of the arena:

```
switch (side)
{
    case 0:
        // left
        x = minX;
        y = (rand() % maxY) + minY;
        break;

    case 1:
        // right
        x = maxX;
        y = (rand() % maxY) + minY;
        break;

    case 2:
        // top
        x = (rand() % maxX) + minX;
        y = minY;
        break;
```

```

        case 3:
            // bottom
            x = (rand() % maxX) + minX;
            y = maxY;
            break;
    }

```

Still inside the `for` loop, we seed the random number generator again and generate a random number between 0 and 2. We store this number in the `type` variable. The `type` variable will determine whether the current zombie will be a Chaser, Bloater, or Crawler.

After the type is determined, we call the `spawn` function on the current `Zombie` object in the `zombies` array. As a reminder, the arguments that are sent into the `spawn` function determine the starting location of the zombie and the type of zombie it will be. The apparently arbitrary `i` is passed in as it is used as a unique seed that randomly varies the speed of a zombie within an appropriate range. This stops our zombies "bunching up" and becoming a blob rather than a horde:

```

// Bloater, crawler or runner
srand((int)time(0) * i * 2);
int type = (rand() % 3);

// Spawn the new zombie into the array
zombies[i].spawn(x, y, type, i);

```

The `for` loop repeats itself once for each zombie, controlled by the value contained in `numZombies`, and then we return the array. The array, as another reminder, is simply an address of the first element of itself. The array is dynamically allocated on the heap, so it persists after the function returns:

```

return zombies;

```

Now, we can bring our zombies to life.

Bringing the horde to life (back to life)

We have a `Zombie` class and a function to make a randomly spawning horde of them. We have the `TextureHolder` singleton as a neat way to hold just three textures that can be used for dozens or even thousands of zombies. Now, we can add the horde to our game engine in `main`.

Add the following highlighted code to include the `TextureHolder` class. Then, just inside `main`, we will initialize the one and only instance of `TextureHolder`, which can be used from anywhere within our game:

```
#include <SFML/Graphics.hpp>
#include "ZombieArena.h"
#include "Player.h"
#include "TextureHolder.h"

using namespace sf;

int main()
{
    // Here is the instance of TextureHolder
    TextureHolder holder;

    // The game will always be in one of four states
    enum class State { PAUSED, LEVELING_UP, GAME_OVER, PLAYING };
    // Start with the GAME_OVER state
    State state = State::GAME_OVER;
```

The following few lines of highlighted code declare some control variables for the number of zombies at the start of the wave, the number of zombies still to be killed, and, of course, a pointer to `Zombie` called `zombies` that we initialize to `nullptr`:

```
// Create the background
VertexArray background;
// Load the texture for our background vertex array
Texture textureBackground;
textureBackground.loadFromFile("graphics/background_sheet.png");

// Prepare for a horde of zombies
int numZombies;
int numZombiesAlive;
Zombie* zombies = nullptr;

// The main game loop
while (window.isOpen())
```

Next, in the `PLAYING` section, nested inside the `LEVELING_UP` section, we add code that does the following:

- Initializes `numZombies` to 10. As the project progresses, this will eventually be dynamic and based on the current wave number.
- Delete any preexisting allocated memory. Otherwise, each new call to `createHorde` would take up progressively more memory but without freeing up the previous horde's memory.
- Then, we call `createHorde` and assign the returned memory address to `zombies`.
- We also initialize `zombiesAlive` with `numZombies` because we haven't killed any at this point.

Add the following highlighted code, which we have just discussed:

```
if (state == State::PLAYING)
{
    // Prepare the level
    // We will modify the next two lines later
    arena.width = 500;
    arena.height = 500;
    arena.left = 0;
    arena.top = 0;

    // Pass the vertex array by reference
    // to the createBackground function
    int tileSize = createBackground(background, arena);

    // Spawn the player in the middle of the arena
    player.spawn(arena, resolution, tileSize);

    // Create a horde of zombies
    numZombies = 10;

    // Delete the previously allocated memory (if it exists)
    delete[] zombies;
    zombies = createHorde(numZombies, arena);
    numZombiesAlive = numZombies;

    // Reset the clock so there isn't a frame jump
    clock.restart();
}
```

Now, add the following highlighted code to the `ZombieArena.cpp` file:

```
/*
*****
UPDATE THE FRAME
*****
*/
if (state == State::PLAYING)
{
    // Update the delta time
    Time dt = clock.restart();
    // Update the total game time
    gameTimeTotal += dt;
    // Make a decimal fraction of 1 from the delta time
    float dtAsSeconds = dt.asSeconds();

    // Where is the mouse pointer
    mouseScreenPosition = Mouse::getPosition();

    // Convert mouse position to world coordinates of mainView
    mouseWorldPosition = window.mapPixelToCoords(
        Mouse::getPosition(), mainView);

    // Update the player
    player.update(dtAsSeconds, Mouse::getPosition());

    // Make a note of the players new position
    Vector2f playerPosition(player.getCenter());

    // Make the view centre around the player
    mainView.setCenter(player.getCenter());

    // Loop through each Zombie and update them
    for (int i = 0; i < numZombies; i++)
    {
        if (zombies[i].isAlive())
        {
            zombies[i].update(dt.asSeconds(), playerPosition);
        }
    }
}

} // End updating the scene
```

All the new preceding code does is loop through the array of zombies, check whether the current zombie is alive and, if it is, calls its update function with the necessary arguments.

Add the following code to draw all the zombies:

```
/*
*****
Draw the scene
*****
*/

if (state == State::PLAYING)
{
    window.clear();

    // set the mainView to be displayed in the window
    // And draw everything related to it
    window.setView(mainView);

    // Draw the background
    window.draw(background, &textureBackground);

    // Draw the zombies
    for (int i = 0; i < numZombies; i++)
    {
        window.draw(zombies[i].getSprite());
    }

    // Draw the player
    window.draw(player.getSprite());
}
```

The preceding code loops through all the zombies and calls the `getSprite` function to allow the `draw` function to do its work. We don't check whether the zombie is alive because even if the zombie is dead, we want to draw the blood splatter.

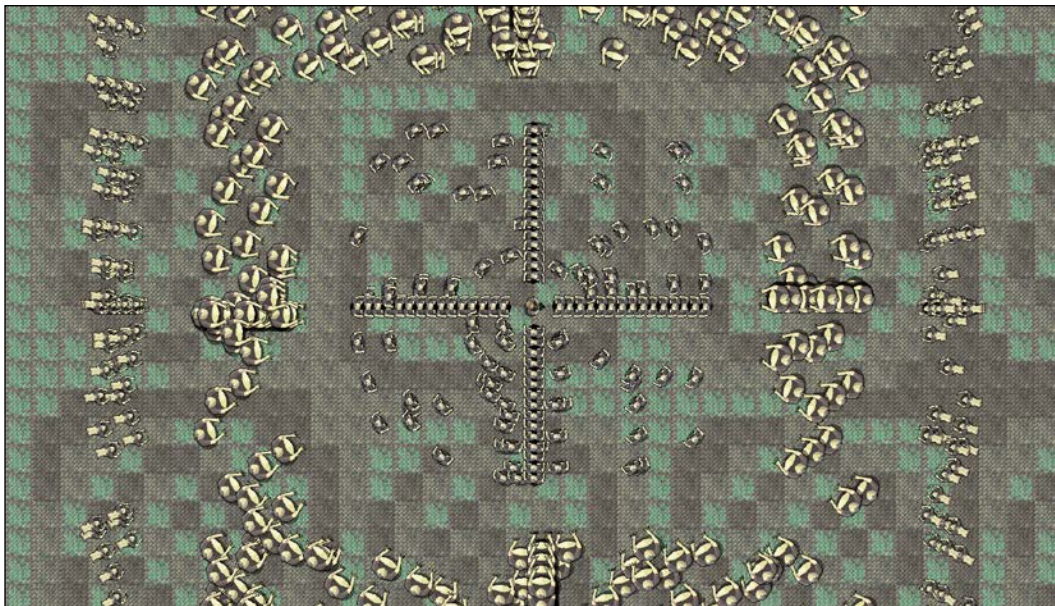
At the end of the main function, we need to make sure to delete our pointer because it is a good practice as well as often being essential. However, technically, this isn't essential because the game is about to exit, and the operating system will reclaim all the memory that's used after the `return 0` statement:

```
    } // End of main game loop

    // Delete the previously allocated memory (if it exists)
    delete[] zombies;

    return 0;
}
```

You can run the game and see the zombies spawn around the edge of the arena. They will immediately head straight toward the player at their various speeds. Just for fun, I increased the size of the arena and increased the number of zombies to 1,000 as you can see in the following screenshot:



This is going to end badly!

Note that you can also pause and resume the onslaught of the horde using the *Enter* key because of the code we wrote in *Chapter 8, SFML Views – Starting the Zombie Shooter Game*.

Let's fix the fact that some classes still use a `Texture` instance directly and modify it to use the new `TextureHolder` class.

Using the `TextureHolder` class for all textures

Since we have our `TextureHolder` class, we might as well be consistent and use it to load all our textures. Let's make some very small alterations to the existing code that loads textures for the background sprite sheet and the player.

Changing the way the background gets its textures

In the `ZombieArena.cpp` file, find the following code:

```
// Load the texture for our background vertex array
Texture textureBackground;
textureBackground.loadFromFile("graphics/background_sheet.png");
```

Delete the code highlighted previously and replace it with the following highlighted code, which uses our new `TextureHolder` class:

```
// Load the texture for our background vertex array
Texture textureBackground = TextureHolder::GetTexture(
    "graphics/background_sheet.png");
```

Let's update the way the `Player` class gets a texture.

Changing the way the Player gets its texture

In the `Player.cpp` file, inside the constructor, find this code:

```
#include "player.h"

Player::Player()
{
    m_Speed = START_SPEED;
    m_Health = START_HEALTH;
    m_MaxHealth = START_HEALTH;

    // Associate a texture with the sprite
    // !!Watch this space!!
    m_Texture.loadFromFile("graphics/player.png");
    m_Sprite.setTexture(m_Texture);

    // Set the origin of the sprite to the centre,
    // for smooth rotation
    m_Sprite.setOrigin(25, 25);
}
```

Delete the code highlighted previously and replace it with the following highlighted code, which uses our new `TextureHolder` class. In addition, add the `include` directive to add the `TextureHolder` header to the file. The new code is shown highlighted, in context, as follows:

```
#include "player.h"
#include "TextureHolder.h"

Player::Player()
{
    m_Speed = START_SPEED;
    m_Health = START_HEALTH;
    m_MaxHealth = START_HEALTH;

    // Associate a texture with the sprite
    // !!Watch this space!!
    m_Sprite = Sprite(TextureHolder::GetTexture(
        "graphics/player.png"));
}
```

```
// Set the origin of the sprite to the centre,  
// for smooth rotation  
m_Sprite.setOrigin(25, 25);  
}
```



From now on, we will use the `TextureHolder` class for loading all textures.

Summary

In this chapter, we have covered pointers and discussed that they are variables that hold a memory address to a specific type of object. The full significance of this will begin to reveal itself as this book progresses and the power of pointers is revealed. We also used pointers in order to create a huge horde of zombies that can be accessed using a pointer, which it turns out is also the same thing as the first element of an array.

We learned about the STL, and in particular the `map` class. We implemented a class that will store all our textures, as well as provide access to them.

You might have noticed that the zombies don't appear to be very dangerous. They just drift through the player without leaving a scratch. Currently, this is a good thing because the player has no way to defend themselves.

In the next chapter, we will make two more classes: one for ammo and health pickups and one for bullets that the player can shoot. After we have done that, we will learn how to detect collisions so that the bullets and zombies do some damage and the pickups can be collected by the player.

FAQ

Here are some questions that might be on your mind:

Q) What's the difference between pointers and references?

A) Pointers are like references with boosters. Pointers can be changed to point to different variables (memory addresses), as well as point to dynamically allocated memory on the free store.

Q) What's the deal with arrays and pointers?

A) Arrays are really constant pointers to their first element.

Q) Can you remind me about the `new` keyword and memory leaks?

A) When we use memory on the free store using the `new` keyword, it persists even when the function it was created in has returned and all the local variables are gone. When we are done with using memory on the free store, we must release it. So, if we use memory on the free store that we want to persist beyond the life of a function, we must make sure to keep a pointer to it or we will have leaked memory. It would be like putting all our belongings in our house and then forgetting where we live! When we return the `zombies` array from `createHorde`, it is like passing the relay baton (memory address) from `createHorde` to `main`. It's like saying, *OK, here is your horde of zombies – they are your responsibility now*. And, we wouldn't want any leaked zombies running around in our RAM! So, we must remember to call `delete` on pointers to dynamically allocated memory.

11

Collision Detection, Pickups, and Bullets

So far, we have implemented the main visual aspects of our game. We have a controllable character running around in an arena full of zombies that chase them. The problem is that they don't interact with each other. A zombie can wander right through the player without leaving a scratch. We need to detect collisions between the zombies and the player.

If the zombies are going to be able to injure and eventually kill the player, it is only fair that we give the player some bullets for their gun. We will then need to make sure that the bullets can hit and kill the zombies.

At the same time, if we are writing collision detection code for bullets, zombies, and the player, it would be a good time to add a class for health and ammo pickups as well.

Here is what we will do and the order in which we will cover things in this chapter:

- Shooting Bullets
- Adding a crosshair and hiding the mouse pointer
- Spawning pickups
- Detecting collisions

Let's start with the `Bullet` class.

Coding the Bullet class

We will use the SFML `RectangleShape` class to visually represent a bullet. We will code a `Bullet` class that has a `RectangleShape` member, as well as other member data and functions. Then, we will add bullets to our game in a few steps, as follows:

1. First, we will code the `Bullet.h` file. This will reveal all the details of the member data and the prototypes for the functions.
2. Next, we will code the `Bullet.cpp` file, which, of course, will contain the definitions for all the functions of the `Bullet` class. As we step through this, I will explain exactly how an object of the `Bullet` type will work and be controlled.
3. Finally, we will declare a whole array full of bullets in the `main` function. We will also implement a control scheme for shooting, managing the player's remaining ammo, and reloading.

Let's get started with step 1.

Coding the Bullet header file

To make the new header file, right-click **Header Files** in the **Solution Explorer** and select **Add | New Item....** In the **Add New Item** window, highlight (by left-clicking) **Header File (.h)**, and then, in the **Name** field, type `Bullet.h`.

Add the following private member variables, along with the `Bullet` class declaration, to the `Bullet.h` file. We can then run through them and explain what they are for:

```
#pragma once
#include <SFML/Graphics.hpp>

using namespace sf;

class Bullet
{
private:
    // Where is the bullet?
    Vector2f m_Position;

    // What each bullet looks like
    RectangleShape m_BulletShape;
```

```
// Is this bullet currently whizzing through the air
bool m_InFlight = false;

// How fast does a bullet travel?
float m_BulletSpeed = 1000;

// What fraction of 1 pixel does the bullet travel,
// Horizontally and vertically each frame?
// These values will be derived from m_BulletSpeed
float m_BulletDistanceX;
float m_BulletDistanceY;

// Some boundaries so the bullet doesn't fly forever
float m_MaxX;
float m_MinX;
float m_MaxY;
float m_MinY;

// Public function prototypes go here

};
```

In the previous code, the first member is a `Vector2f` called `m_Position`, which will hold the bullet's location in the game world.

Next, we declare a `RectangleShape` called `m_BulletShape` as we are using a simple non-texture graphic for each bullet, a bit like we did for the time-bar in *Timber!!!*.

The code then declares a `Boolean`, `m_InFlight`, which will keep track of whether the bullet is currently whizzing through the air or not. This will allow us to decide whether we need to call its update function each frame and whether we need to run collision detection checks.

The `float` variable, `m_BulletSpeed`, will (you can probably guess) hold the speed in pixels per second that the bullet will travel at. It is initialized to the value of 1000, which is a little arbitrary, but it works well.

Next, we have two more `float` variables, `m_BulletDistanceX` and `m_BulletDistanceY`. As the calculations to move a bullet are a little more complex than those used to move a zombie or the player, we will benefit from having these two variables, which we will perform calculations on. They will be used to decide the horizontal and vertical changes in the bullet's position in each frame.

Finally, we have four more `float` variables (`m_MaxX`, `m_MinX`, `m_MaxY`, and `m_MinY`), which will later be initialized to hold the maximum and minimum horizontal and vertical positions for the bullet.

It is likely that the need for some of these variables is not immediately apparent, but it will become clearer when we see each of them in action in the `Bullet.cpp` file.

Now, add all the public function prototypes to the `Bullet.h` file:

```
// Public function prototypes go here
public:
    // The constructor
    Bullet();

    // Stop the bullet
    void stop();

    // Returns the value of m_InFlight
    bool isInFlight();

    // Launch a new bullet
    void shoot(float startX, float startY,
              float xTarget, float yTarget);

    // Tell the calling code where the bullet is in the world
    FloatRect getPosition();

    // Return the actual shape (for drawing)
    RectangleShape getShape();

    // Update the bullet each frame
    void update(float elapsedTime);

};
```

Let's run through each of the functions in turn, and then we can move on to coding their definitions.

First, we have the `Bullet` function, which is, of course, the constructor. In this function, we will set up each `Bullet` instance, ready for action.

The `stop` function will be called when the bullet has been in action but needs to stop.

The `isInFlight` function returns a Boolean and will be used to test whether a bullet is currently in flight or not.

The `shoot` function's use is given away by its name, but how it will work deserves some discussion. For now, just note that it has four `float` parameters that will be passed in. The four values represent the starting (where the player is) horizontal and vertical position of the bullet, as well as the vertical and horizontal target position (where the crosshair is).

The `getPosition` function returns a `FloatRect` that represents the location of the bullet. This function will be used to detect collisions with zombies. You might remember from *Chapter 10, Pointers, the Standard Template Library, and Texture Management*, that zombies also had a `getPosition` function.

Following on, we have the `getShape` function, which returns an object of the `RectangleShape` type. As we have discussed, each bullet is represented visually by a `RectangleShape` object. The `getShape` function, therefore, will be used to grab a copy of the current state of `RectangleShape` in order to draw it.

Finally, and hopefully as expected, there is the `update` function, which has a `float` parameter that represents the fraction of a second that has passed since the last time update was called. The `update` method will change the position of the bullet each frame.

Let's look at and code the function definitions.

Coding the Bullet source file

Now, we can create a new `.cpp` file that will contain the function definitions. Right-click **Source Files** in the **Solution Explorer** and select **Add | New Item....** In the **Add New Item** window, highlight (by left-clicking) **C++ File (.cpp)**, and then, in the **Name** field, type `Bullet.cpp`. Finally, click the **Add** button. We are now ready to code the class.

Add the following code, which is for the include directives and the constructor. We know it is a constructor because the function has the same name as the class:

```
#include "bullet.h"

// The constructor
Bullet::Bullet()
{
    m_BulletShape.setSize(sf::Vector2f(2, 2));
}
```


The only thing that the `Bullet` constructor needs to do is set the size of `m_BulletShape`, which is the `RectangleShape` object. The code sets the size to two pixels by two pixels.

Next, we will code the more substantial `shoot` function. Add the following code to the `Bullet.cpp` file and study it, and then we can talk about it:

```
void Bullet::shoot(float startX, float startY,
                  float targetX, float targetY)
{
    // Keep track of the bullet
    m_InFlight = true;
    m_Position.x = startX;
    m_Position.y = startY;

    // Calculate the gradient of the flight path
    float gradient = (startX - targetX) / (startY - targetY);

    // Any gradient less than 1 needs to be negative
    if (gradient < 0)
    {
        gradient *= -1;
    }

    // Calculate the ratio between x and y
    float ratioXY = m_BulletSpeed / (1 + gradient);

    // Set the "speed" horizontally and vertically
    m_BulletDistanceY = ratioXY;
    m_BulletDistanceX = ratioXY * gradient;

    // Point the bullet in the right direction
    if (targetX < startX)
    {
        m_BulletDistanceX *= -1;
    }

    if (targetY < startY)
    {
        m_BulletDistanceY *= -1;
    }

    // Set a max range of 1000 pixels
    float range = 1000;
    m_MinX = startX - range;
```

```

        m_MaxX = startX + range;
        m_MinY = startY - range;
        m_MaxY = startY + range;

        // Position the bullet ready to be drawn
        m_BulletShape.setPosition(m_Position);
    }

```

In order to demystify the shoot function, we will split it up and talk about the code we have just added in chunks.

First, let's remind ourselves about the signature. The shoot function receives the starting and target horizontal and vertical positions of a bullet. The calling code will supply these based on the position of the player sprite and the position of the crosshair. Here it is again:

```

void Bullet::shoot(float startX, float startY,
                  float targetX, float targetY)

```

Inside the shoot function, we set `m_InFlight` to true and position the bullet using the `startX` and `startY` parameters. Here is that piece of code again:

```

// Keep track of the bullet
m_InFlight = true;
m_Position.x = startX;
m_Position.y = startY;

```

Now, we use a bit of trigonometry to determine the gradient of travel for a bullet. The progression of the bullet, both horizontally and vertically, must vary based on the slope of the line that's created by drawing between the start and target of a bullet. The rate of change cannot be the same or very steep shots will arrive at the horizontal location before the vertical location, and vice versa for shallow shots.

The code that follows derives the gradient based on the equation of a line. Then, it checks whether the gradient is less than zero and if it is, multiplies it by -1. This is because the start and target coordinates that are passed in can be negative or positive, and we always want the amount of progression each frame to be positive. Multiplying by -1 simply makes the negative number into its positive equivalent because a minus multiplied by a minus gives a positive. The actual direction of travel will be handled in the `update` function by adding or subtracting the positive values we arrive at in this function.

Next, we calculate a ratio of horizontal to vertical distance by dividing our bullet's speed (`m_BulletSpeed`) by one, plus the gradient. This will allow us to change the bullet's horizontal and vertical position by the correct amount each frame, based on the target the bullet is heading toward.

Finally, in this part of the code, we assign the values to `m_BulletDistanceY` and `m_BulletDistanceX`:

```
// Calculate the gradient of the flight path
float gradient = (startX - targetX) / (startY - targetY);

// Any gradient less than zero needs to be negative
if (gradient < 0)
{
    gradient *= -1;
}

// Calculate the ratio between x and y
float ratioXY = m_BulletSpeed / (1 + gradient);

// Set the "speed" horizontally and vertically
m_BulletDistanceY = ratioXY;
m_BulletDistanceX = ratioXY * gradient;
```

The following code is much more straightforward. We simply set a maximum horizontal and vertical location that the bullet can reach. We don't want a bullet carrying on forever. In the update function, we will see whether a bullet has passed its maximum or minimum locations:

```
// Set a max range of 1000 pixels in any direction
float range = 1000;
m_MinX = startX - range;
m_MaxX = startX + range;
m_MinY = startY - range;
m_MaxY = startY + range;
```

The following code moves the sprite that represents the bullet to its starting location. We use the `setPosition` function of `Sprite`, as we have often done before:

```
// Position the bullet ready to be drawn
m_BulletShape.setPosition(m_Position);
```

Next, we have four straightforward functions. Let's add the `stop`, `isInFlight`, `getPosition`, and `getShape` functions:

```
void Bullet::stop()
{
    m_InFlight = false;
}

bool Bullet::isInFlight()
```

```

{
    return m_InFlight;
}

FloatRect Bullet::getPosition()
{
    return m_BulletShape.getGlobalBounds();
}

RectangleShape Bullet::getShape()
{
    return m_BulletShape;
}

```

The `stop` function simply sets the `m_InFlight` variable to `false`. The `isInFlight` function returns whatever the value of this same variable currently is. So, we can see that `shoot` sets the bullet going, `stop` makes it stop, and `isInFlight` informs us what the current state is.

The `getPosition` function returns a `FloatRect`. We will see how we can use the `FloatRect` from each game object to detect collisions soon.

Finally, for the previous code, `getShape` returns a `RectangleShape` so that we can draw the bullet once each frame.

The last function we need to implement before we can start using `Bullet` objects is `update`. Add the following code, study it, and then we can talk about it:

```

void Bullet::update(float elapsedTime)
{
    // Update the bullet position variables
    m_Position.x += m_BulletDistanceX * elapsedTime;
    m_Position.y += m_BulletDistanceY * elapsedTime;

    // Move the bullet
    m_BulletShape.setPosition(m_Position);

    // Has the bullet gone out of range?
    if (m_Position.x < m_MinX || m_Position.x > m_MaxX ||
        m_Position.y < m_MinY || m_Position.y > m_MaxY)
    {
        m_InFlight = false;
    }
}

```

In the `update` function, we use `m_BulletDistanceX` and `m_BulletDistanceY`, multiplied by the time since the last frame to move the bullet. Remember that the values of the two variables were calculated in the `shoot` function and represent the gradient (ratio to each other) that's required to move the bullet at just the right angle. Then, we use the `setPosition` function to actually move `RectangleShape`.

The last thing we do in `update` is a test to see whether the bullet has moved beyond its maximum range. The slightly convoluted `if` statement checks `m_Position.x` and `m_Position.y` against the maximum and minimum values that were calculated in the `shoot` function. These maximum and minimum values are stored in `m_MinX`, `m_MaxX`, `m_MinY`, and `m_MaxY`. If the test is true, then `m_InFlight` is set to false.

The `Bullet` class is done. Now, we will look at how we can shoot some in the `main` function.

Making the bullets fly

We will make the bullets usable by following these six steps:

1. Add the necessary include directive for the `Bullet` class.
2. Add some control variables and an array to hold some `Bullet` instances.
3. Handle the player pressing `R` to reload.
4. Handle the player pressing the left mouse button to fire a bullet.
5. Update all bullets that are in flight in each frame.
6. Draw the bullets that are in flight in each frame.

Including the Bullet class

Add the include directive to make the `Bullet` class available:

```
#include <SFML/Graphics.hpp>
#include "ZombieArena.h"
#include "Player.h"
#include "TextureHolder.h"
#include "Bullet.h"

using namespace sf;
```

Let's move on to the next step.

Control variables and the bullet array

Here are some variables to keep track of clip sizes, spare bullets, bullets, the remaining bullets in the clip, the current rate of fire (starting at one per second), and the time when the last bullet was fired.

Add the following highlighted code. Then, we can move on and see all these variables in action throughout the rest of this section:

```
// Prepare for a horde of zombies
int numZombies;
int numZombiesAlive;
Zombie* zombies = NULL;

// 100 bullets should do
Bullet bullets[100];
int currentBullet = 0;
int bulletsSpare = 24;
int bulletsInClip = 6;
int clipSize = 6;
float fireRate = 1;
// When was the fire button last pressed?
Time lastPressed;

// The main game loop
while (window.isOpen())
```

Next, let's handle what happens when the player presses the *R* keyboard key, which is used for reloading a clip.

Reloading the gun

Now, we will handle the player input related to shooting bullets. First, we will handle pressing the *R* key to reload the gun. We will do so with an SFML event.

Add the following highlighted code. It is shown with lots of context to make sure the code goes in the right place. Study the code and then we can talk about it:

```
// Handle events
Event event;
while (window.pollEvent(event))
{
    if (event.type == Event::KeyPressed)
    {
```

```
// Pause a game while playing
if (event.key.code == Keyboard::Return &&
    state == State::PLAYING)
{
    state = State::PAUSED;
}

// Restart while paused
else if (event.key.code == Keyboard::Return &&
    state == State::PAUSED)
{
    state = State::PLAYING;
    // Reset the clock so there isn't a frame jump
    clock.restart();
}

// Start a new game while in GAME_OVER state
else if (event.key.code == Keyboard::Return &&
    state == State::GAME_OVER)
{
    state = State::LEVELING_UP;
}

if (state == State::PLAYING)
{
    // Reloading
    if (event.key.code == Keyboard::R)
    {
        if (bulletsSpare >= clipSize)
        {
            // Plenty of bullets. Reload.
            bulletsInClip = clipSize;
            bulletsSpare -= clipSize;
        }
        else if (bulletsSpare > 0)
        {
            // Only few bullets left
            bulletsInClip = bulletsSpare;
            bulletsSpare = 0;
        }
        else
        {
            // More here soon?!
        }
    }
}
```

```
        }  
    }  
  
}  
} // End event polling
```

The previous code is nested within the event handling part of the game loop (`while(window.pollEvent())`), within the block that only executes when the game is actually being played (`if(state == State::Playing)`). It is obvious that we don't want the player reloading when the game has finished or is paused, and wrapping the new code as we've described achieves this.

In the new code itself, the first thing we do is test for the *R* key being pressed with `if(event.key.code == Keyboard::R)`. Once we have detected that the *R* key was pressed, the remaining code is executed. Here is the structure of the `if`, `else if`, and `else` blocks:

```
if(bulletsSpare >= clipSize)  
    ...  
else if(bulletsSpare > 0)  
    ...  
else  
    ...
```

The previous structure allows us to handle three possible scenarios, as shown here:

- The player has pressed *R* and they have more bullets spare than the clip can take. In this scenario, the clip is refilled, and the number of spare bullets is reduced.
- The player has some spare bullets but not enough to fill the clip completely. In this scenario, the clip is filled with as many spare bullets as the player has and the number of spare bullets is set to zero.
- The player has pressed *R* but they have no spare bullets at all. For this scenario, we don't actually need to alter the variables. However, we will play a sound effect here when we implement the sound in *Chapter 13, Sound Effects, File I/O, and Finishing the Game*, so we will leave the empty `else` block ready.

Now, let's shoot a bullet.

Shooting a bullet

Here, we will handle the left mouse button being clicked to fire a bullet. Add the following highlighted code and study it carefully:

```
        if (Keyboard::isKeyPressed(Keyboard::D))
        {
            player.moveRight();
        }
    else
    {
        player.stopRight();
    }

    // Fire a bullet
    if (Mouse::isButtonPressed(sf::Mouse::Left))
    {
        if (gameTimeTotal.asMilliseconds()
            - lastPressed.asMilliseconds()
            > 1000 / fireRate && bulletsInClip > 0)
        {

            // Pass the centre of the player
            // and the centre of the cross-hair
            // to the shoot function
            bullets[currentBullet].shoot(
                player.getCenter().x, player.getCenter().y,
                mouseWorldPosition.x, mouseWorldPosition.y);

            currentBullet++;
            if (currentBullet > 99)
            {
                currentBullet = 0;
            }
            lastPressed = gameTimeTotal;

            bulletsInClip--;
        }

    }

} // End fire a bullet

} // End WASD while playing
```

All the previous code is wrapped in an `if` statement that executes whenever the left mouse button is pressed, that is, `if (Mouse::isButtonPressed(sf::Mouse::Left))`. Note that the code will execute repeatedly, even if the player just holds down the button. The code we will go through now controls the rate of fire.

In the preceding code, we then check whether the total time elapsed in the game (`gameTimeTotal`) minus the time the player last shot a bullet (`lastPressed`) is greater than 1,000, divided by the current rate of fire and that the player has at least one bullet in the clip. We use 1,000 because this is the number of milliseconds in a second.

If this test is successful, the code that actually fires a bullet is executed. Shooting a bullet is easy because we did all the hard work in the `Bullet` class. We simply call `shoot` on the current bullet from the `bullets` array. We pass in the player's and the cross-hair's current horizontal and vertical locations. The bullet will be configured and set in flight by the code in the `shoot` function of the `Bullet` class.

All we must do is keep track of the array of bullets. We incremented the `currentBullet` variable. Then, we need to check to see whether we fired the last bullet (99) with the `if (currentBullet > 99)` statement. If it was the last bullet, we set `currentBullet` to zero. If it wasn't the last bullet, then the next bullet is ready to go whenever the rate of fire permits it and the player presses the left mouse button.

Finally, in the preceding code, we store the time that the bullet was fired into `lastPressed` and decrement `bulletsInClip`.

Now, we can update every bullet, each frame.

Updating the bullets each frame

Add the following highlighted code to loop through the `bullets` array, check whether the bullet is in flight, and if it is, call its update function:

```
// Loop through each Zombie and update them
for (int i = 0; i < numZombies; i++)
{
    if (zombies[i].isAlive())
    {
        zombies[i].update(dt.asSeconds(), playerPosition);
    }
}

// Update any bullets that are in-flight
for (int i = 0; i < 100; i++)
{
```

```
        if (bullets[i].isInFlight())
        {
            bullets[i].update(dtAsSeconds);
        }
    }

    }// End updating the scene
```

Finally, we will draw all the bullets.

Drawing the bullets each frame

Add the following highlighted code to loop through the `bullets` array, check whether the bullet is in flight, and if it is, draw it:

```
/*
*****
Draw the scene
*****
*/

if (state == State::PLAYING)
{
    window.clear();

    // set the mainView to be displayed in the window
    // And draw everything related to it
    window.setView(mainView);

    // Draw the background
    window.draw(background, &textureBackground);

    // Draw the zombies
    for (int i = 0; i < numZombies; i++)
    {
        window.draw(zombies[i].getSprite());
    }

    for (int i = 0; i < 100; i++)
    {
        if (bullets[i].isInFlight())
        {
            window.draw(bullets[i].getShape());
        }
    }
}
```

```
    }

    // Draw the player
    window.draw(player.getSprite());
}
```

Run the game to try out the bullets. Notice that you can fire six shots before you need to press *R* to reload. The obvious things that are missing is some visual indicator of the number of bullets in the clip and the number of spare bullets. Another problem is that the player can very quickly run out of bullets, especially since the bullets have no stopping power whatsoever. They fly straight through the zombies. Add to this that the player is expected to aim at a mouse pointer instead of a precision crosshair and it is clear that we have work to do.

In the next chapter, we will give visual feedback through a HUD. We will replace the mouse cursor with a crosshair next and then spawn some pickups to replenish bullets and health after that. Finally, in this chapter, we will handle collision detection to make the bullets and the zombies do damage and make the player able to actually get the pickups.

Giving the player a crosshair

Adding a crosshair is easy and only requires one new concept. Add the following highlighted code, and then we can run through it:

```
// 100 bullets should do
Bullet bullets[100];
int currentBullet = 0;
int bulletsSpare = 24;
int bulletsInClip = 6;
int clipSize = 6;
float fireRate = 1;
// When was the fire button last pressed?
Time lastPressed;

// Hide the mouse pointer and replace it with crosshair
window.setMouseCursorVisible(true);
Sprite spriteCrosshair;
Texture textureCrosshair = TextureHolder::GetTexture(
    "graphics/crosshair.png");

spriteCrosshair.setTexture(textureCrosshair);
```

```
spriteCrosshair.setOrigin(25, 25);
```

```
// The main game loop  
while (window.isOpen())
```

First, we call the `setMouseCursorVisible` function on our window object. We then load a Texture and declare a Sprite instance and initialize it in the usual way. Furthermore, we set the sprite's origin to its center to make it convenient and simpler to make the bullets fly to the middle, as you would expect to happen.

Now, we need to update the crosshair each frame with the world coordinates of the mouse. Add the following highlighted line of code, which uses the `mouseWorldPosition` vector to set the crosshair's position each frame:

```
/*  
*****  
UPDATE THE FRAME  
*****  
*/  
if (state == State::PLAYING)  
{  
    // Update the delta time  
    Time dt = clock.restart();  
    // Update the total game time  
    gameTimeTotal += dt;  
    // Make a decimal fraction of 1 from the delta time  
    float dtAsSeconds = dt.asSeconds();  
  
    // Where is the mouse pointer  
    mouseScreenPosition = Mouse::getPosition();  
  
    // Convert mouse position to world coordinates of mainView  
    mouseWorldPosition = window.mapPixelToCoords(  
        Mouse::getPosition(), mainView);  
  
    // Set the crosshair to the mouse world location  
    spriteCrosshair.setPosition(mouseWorldPosition);  
  
    // Update the player  
    player.update(dtAsSeconds, Mouse::getPosition());
```

Next, as you have probably come to expect, we can draw the crosshair each frame. Add the following highlighted line of code in the position shown. This line of code needs no explanation, but its position after all the other game objects is important, so it is drawn on top:

```

/*
*****
Draw the scene
*****
*/

if (state == State::PLAYING)
{
    window.clear();

    // set the mainView to be displayed in the window
    // And draw everything related to it
    window.setView(mainView);

    // Draw the background
    window.draw(background, &textureBackground);

    // Draw the zombies
    for (int i = 0; i < numZombies; i++)
    {
        window.draw(zombies[i].getSprite());
    }

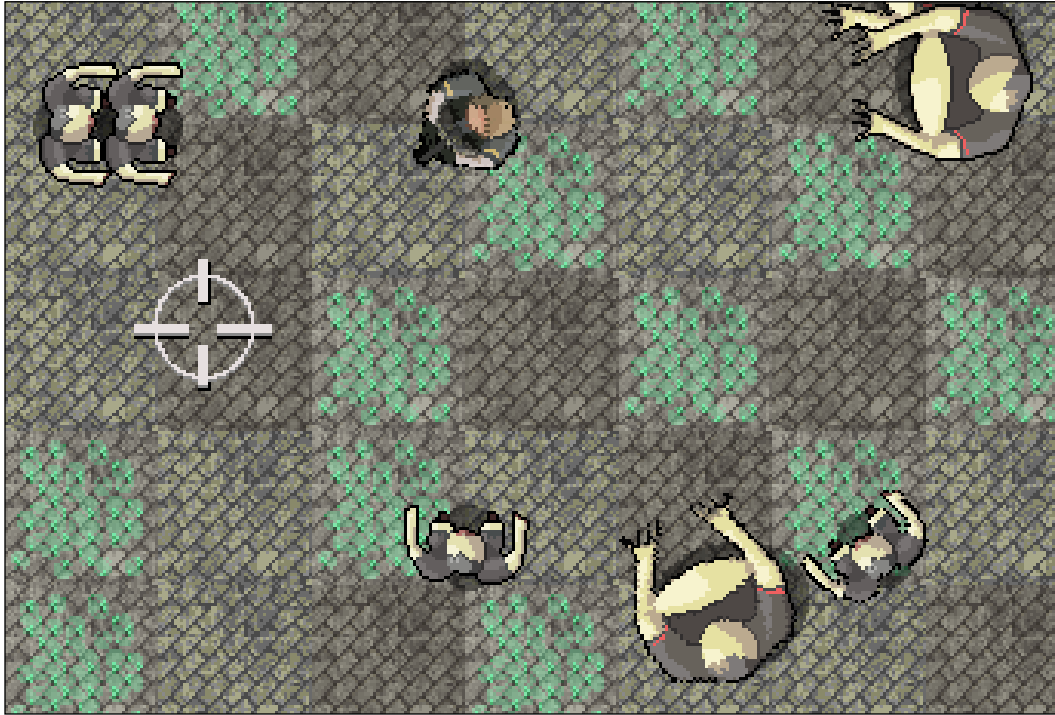
    for (int i = 0; i < 100; i++)
    {
        if (bullets[i].isInFlight())
        {
            window.draw(bullets[i].getShape());
        }
    }

    // Draw the player
    window.draw(player.getSprite());

    //Draw the crosshair
    window.draw(spriteCrosshair);
}

```

Now, you can run the game and will see a cool crosshair instead of a mouse cursor:



Notice how the bullet fires neatly through the center of the crosshair. The way the shooting mechanism works is analogous to allowing the player to choose to shoot from the hip or aim down the sights. If the player keeps the crosshair close to the center, they can fire and turn rapidly, yet must carefully judge the position of distant zombies.

Alternatively, the player can hover their crosshair directly over the head of a distant zombie and score a precise hit; however, they then have much further to move the crosshair back if a zombie attacks from another direction.

An interesting improvement to the game would be to add a small random amount of inaccuracy to each shot. This inaccuracy could perhaps be mitigated with an upgrade between waves.

Coding a class for pickups

In this section, we will code a `Pickup` class that has a `Sprite` member, as well as other member data and functions. We will add pickups to our game in just a few steps:

1. First, we will code the `Pickup.h` file. This will reveal all the details of the member data and the prototypes for the functions.
2. Then, we will code the `Pickup.cpp` file which, of course, will contain the definitions for all the functions of the `Pickup` class. As we step through this, I will explain exactly how an object of the `Pickup` type will work and be controlled.
3. Finally, we will use the `Pickup` class in the main function to spawn them, update them, and draw them.

Let's get started with step 1.

Coding the Pickup header file

To make the new header file, right-click **Header Files** in the **Solution Explorer** and select **Add | New Item....** In the **Add New Item** window, highlight (by left-clicking) **Header File (.h)**, and then, in the **Name** field, type `Pickup.h`.

Add and study the following code to the `Pickup.h` file and then we can go through it:

```
#pragma once
#include <SFML/Graphics.hpp>

using namespace sf;

class Pickup
{
private:
    //Start value for health pickups
    const int HEALTH_START_VALUE = 50;
    const int AMMO_START_VALUE = 12;
    const int START_WAIT_TIME = 10;
    const int START_SECONDS_TO_LIVE = 5;

    // The sprite that represents this pickup
    Sprite m_Sprite;
```



```
// The arena it exists in
IntRect m_Arena;

// How much is this pickup worth?
int m_Value;

// What type of pickup is this?
// 1 = health, 2 = ammo
int m_Type;

// Handle spawning and disappearing
bool m_Spawned;
float m_SecondsSinceSpawn;
float m_SecondsSinceDeSpawn;
float m_SecondsToLive;
float m_SecondsToWait;

// Public prototypes go here
};
```

The previous code declares all the private variables of the `Pickup` class. Although the names should be quite intuitive, it might not be obvious why many of them are needed at all. Let's go through them, starting from the top:

- `const int HEALTH_START_VALUE = 50`: This constant variable is used to set the starting value of all health pickups. The value will be used to initialize the `m_Value` variable, which will need to be manipulated throughout the course of a game.
- `const int AMMO_START_VALUE = 12`: This constant variable is used to set the starting value of all ammo pickups. The value will be used to initialize the `m_Value` variable, which will need to be manipulated throughout the course of a game.
- `const int START_WAIT_TIME = 10`: This variable determines how long a pickup will wait before it respawns after disappearing. It will be used to initialize the `m_SecondsToWait` variable, which can be manipulated throughout the game.
- `const int START_SECONDS_TO_LIVE = 5`: This variable determines how long a pickup will last between spawning and being de-spawned. Like the previous three constants, it has a non-constant associated with it that can be manipulated throughout the course of the game. The non-constant it's used to initialize is `m_SecondsToLive`.
- `Sprite m_Sprite`: This is the sprite to visually represent the object.

- `IntRect m_Arena`: This will hold the size of the current arena to help the pickup to spawn in a sensible position.
- `int m_Value`: How much health or ammo is this pickup worth? This value is used when the player levels up the value of the health or ammo pickup.
- `int m_Type`: This will be either 1 or 2 for health or ammo, respectively. We could have used an enumeration class, but that seemed like overkill for just two options.
- `bool m_Spawned`: Is the pickup currently spawned?
- `float m_SecondsSinceSpawn`: How long is it since the pickup was spawned?
- `float m_SecondsSinceDeSpawn`: How long is it since the pickup was de-spawned (disappeared)?
- `float m_SecondsToLive`: How long should this pickup stay spawned before de-spawning?
- `float m_SecondsToWait`: How long should this pickup stay de-spawned before respawning?



Note that most of the complexity of this class is due to the variable spawn time and its upgradeable nature. If the pickups just respawned when collected and had a fixed value, this would be a very simple class. We need our pickups to be upgradeable so that the player is forced to develop a strategy to progress through the waves.

Next, add the following public function prototypes to the `Pickup.h` file. Be sure to familiarize yourself with the new code so that we can go through it:

```
// Public prototypes go here
public:

    Pickup::Pickup(int type);

    // Prepare a new pickup
    void setArena(IntRect arena);

    void spawn();

    // Check the position of a pickup
    FloatRect getPosition();
```

```
// Get the sprite for drawing
Sprite getSprite();

// Let the pickup update itself each frame
void update(float elapsedTime);

// Is this pickup currently spawned?
bool isSpawned();

// Get the goodness from the pickup
int gotIt();

// Upgrade the value of each pickup
void upgrade();

};
```

Let's talk briefly about each of the function definitions.

- The first function is the constructor and is named after the class. Note that it takes a single `int` parameter. This will be used to initialize the type of pickup it will be (health or ammo).
- The `setArena` function receives an `IntRect`. This function will be called for each `Pickup` instance at the start of each wave. The `Pickup` objects will then "know" the areas into which they can spawn.
- The `spawn` function will, of course, handle spawning the pickup.
- The `getPosition` function, just like in the `Player`, `Zombie`, and `Bullet` classes, will return a `FloatRect` instance that represents the current location of the object in the game world.
- The `getSprite` function returns a `Sprite` object that allows the pickup to be drawn once each frame.
- The `update` function receives the time the previous frame took. It uses this value to update its private variables and make decisions about when to spawn and de-spawn.
- The `isSpawned` function returns a `Boolean` that will let the calling code know whether or not the pickup is currently spawned.
- The `gotIt` function will be called when a collision is detected with the player. The code of the `Pickup` class can then prepare itself for respawning at the appropriate time. Note that it returns an `int` value so that the calling code knows how much the pickup is "worth" in either health or ammo.
- The `upgrade` function will be called when the player chooses to level up the properties of a pickup during the leveling up phase of the game.

Now that we have gone through the member variables and function prototypes, it should be quite easy to follow along as we code the function definitions.

Coding the Pickup class function definitions

Now, we can create a new .cpp file that will contain the function definitions.

Right-click **Source Files** in the **Solution Explorer** and select **Add | New Item....** In the **Add New Item** window, highlight (by left-clicking) **C++ File (.cpp)**, and then, in the **Name** field, type `Pickup.cpp`. Finally, click the **Add** button. We are now ready to code the class.

Add the following code to the `Pickup.cpp` file. Be sure to review the code so that we can discuss it:

```
#include "Pickup.h"
#include "TextureHolder.h"

Pickup::Pickup(int type)
{
    // Store the type of this pickup
    m_Type = type;

    // Associate the texture with the sprite
    if (m_Type == 1)
    {
        m_Sprite = Sprite(TextureHolder::GetTexture(
            "graphics/health_pickup.png"));

        // How much is pickup worth
        m_Value = HEALTH_START_VALUE;
    }
    else
    {
        m_Sprite = Sprite(TextureHolder::GetTexture(
            "graphics/ammo_pickup.png"));

        // How much is pickup worth
        m_Value = AMMO_START_VALUE;
    }

    m_Sprite.setOrigin(25, 25);

    m_SecondsToLive = START_SECONDS_TO_LIVE;
    m_SecondsToWait = START_WAIT_TIME;
}
```

In the previous code, we added the familiar include directives. Then, we added the `Pickup` constructor. We know it is the constructor because it has the same name as the class.

The constructor receives an `int` called `type` and the first thing the code does is assign the value that's received from `type` to `m_Type`. After this, there is an `if else` block that checks whether `m_Type` is equal to 1. If it is, `m_Sprite` is associated with the health pickup texture and `m_Value` is set to `HEALTH_START_VALUE`.

If `m_Type` is not equal to 1, the `else` block associates the ammo pickup texture with `m_Sprite` and assigns the value of `AMMO_START_VALUE` to `m_Value`.

After the `if else` block, the code sets the origin of `m_Sprite` to the center using the `setOrigin` function and assigns `START_SECONDS_TO_LIVE` and `START_WAIT_TIME` to `m_SecondsToLive` and `m_SecondsToWait`, respectively.

The constructor has successfully prepared a `Pickup` object that is ready for use.

Now, we will add the `setArena` function. Examine the code as you add it:

```
void Pickup::setArena(IntRect arena)
{
    // Copy the details of the arena to the pickup's m_Arena
    m_Arena.left = arena.left + 50;
    m_Arena.width = arena.width - 50;
    m_Arena.top = arena.top + 50;
    m_Arena.height = arena.height - 50;

    spawn();
}
```

The `setArena` function that we just coded simply copies the values from the passed in `arena` object but varies the values by + 50 on the left and top and - 50 on the right and bottom. The `Pickup` object is now aware of the area in which it can spawn. The `setArena` function then calls its own `spawn` function to make the final preparations for being drawn and updated each frame.

The `spawn` function is next. Add the following code after the `setArena` function:

```
void Pickup::spawn()
{
    // Spawn at a random location
    srand((int)time(0) / m_Type);
    int x = (rand() % m_Arena.width);
    srand((int)time(0) * m_Type);
```

```
int y = (rand() % m_Arena.height);

m_SecondsSinceSpawn = 0;
m_Spawned = true;

m_Sprite.setPosition(x, y);
}
```

The `spawn` function does everything necessary to prepare the pickup. First, it seeds the random number generator and gets a random number for both the horizontal and vertical position of the object. Notice that it uses the `m_Arena.width` and `m_Arena.height` variables as the ranges for the possible horizontal and vertical positions.

The `m_SecondsSinceSpawn` variable is set to zero so that the length of time that's allowed before it is de-spawned is reset. The `m_Spawned` variable is set to `true` so that, when we call `isSpawned`, from `main`, we will get a positive response. Finally, `m_Sprite` is moved into position with `setPosition`, ready for being drawn to the screen.

In the following block of code, we have three simple getter functions. The `getPosition` function returns a `FloatRect` of the current position of `m_Sprite`, `getSprite` returns a copy of `m_Sprite` itself, and `isSpawned` returns `true` or `false`, depending on whether the object is currently spawned.

Add and examine the code we have just discussed:

```
FloatRect Pickup::getPosition()
{
    return m_Sprite.getGlobalBounds();
}

Sprite Pickup::getSprite()
{
    return m_Sprite;
}

bool Pickup::isSpawned()
{
    return m_Spawned;
}
```

Next, we will code the `gotIt` function. This function will be called from `main` when the player touches/collides (gets) with the pickup. Add the `gotIt` function after the `isSpawned` function:

```
int Pickup::gotIt()
{
    m_Spawned = false;
    m_SecondsSinceDespawn = 0;
    return m_Value;
}
```

The `gotIt` function sets `m_Spawned` to `false` so that we know not to draw and check for collisions anymore. `m_SecondsSinceDespawn` is set to zero so that the countdown to spawning begins again from the start. `m_Value` is then returned to the calling code so that the calling code can handle adding extra ammunition or health, as appropriate.

Following this, we need to code the `update` function, which ties together many of the variables and functions we have seen so far. Add and familiarize yourself with the `update` function, and then we can talk about it:

```
void Pickup::update(float elapsedTime)
{
    if (m_Spawned)
    {
        m_SecondsSinceSpawn += elapsedTime;
    }
    else
    {
        m_SecondsSinceDespawn += elapsedTime;
    }

    // Do we need to hide a pickup?
    if (m_SecondsSinceSpawn > m_SecondsToLive && m_Spawned)
    {
        // Remove the pickup and put it somewhere else
        m_Spawned = false;
        m_SecondsSinceDespawn = 0;
    }

    // Do we need to spawn a pickup
    if (m_SecondsSinceDespawn > m_SecondsToWait && !m_Spawned)
    {

```

```

        // spawn the pickup and reset the timer
        spawn();
    }
}

```

The update function is divided into four blocks that are considered for execution each frame:

1. An `if` block that executes if `m_Spawned` is true: `if (m_Spawned)`. This block of code adds the time this frame to `m_SecondsSinceSpawned`, which keeps track of how long the pickup has been spawned.
2. A corresponding `else` block that executes if `m_Spawned` is false. This block adds the time this frame took to `m_SecondsSinceDeSpawn`, which keeps track of how long the pickup has waited since it was last de-spawned (hidden).
3. Another `if` block that executes when the pickup has been spawned for longer than it should have been: `if (m_SecondsSinceSpawn > m_SecondsToLive && m_Spawned)`. This block sets `m_Spawned` to false and resets `m_SecondsSinceDeSpawn` to zero. Now, block 2 will execute until it is time to spawn it again.
4. A final `if` block that executes when the time to wait since de-spawning has exceeded the necessary wait time, and the pickup is not currently spawned: `if (m_SecondsSinceDeSpawn > m_SecondsToWait && !m_Spawned)`. When this block is executed, it is time to spawn the pick up again, and the spawn function is called.

These four tests are what control the hiding and showing of a pickup.

Finally, add the definition for the upgrade function:

```

void Pickup::upgrade()
{
    if (m_Type == 1)
    {
        m_Value += (HEALTH_START_VALUE * .5);
    }
    else
    {
        m_Value += (AMMO_START_VALUE * .5);
    }

    // Make them more frequent and last longer
    m_SecondsToLive += (START_SECONDS_TO_LIVE / 10);
    m_SecondsToWait -= (START_WAIT_TIME / 10);
}

```


The upgrade function tests for the type of pickup, either health or ammo, and then adds 50% of the (appropriate) starting value on to `m_Value`. The next two lines after the `if else` blocks increase the amount of time the pickup will remain spawned and decreases the amount of time the player must wait between spawns.

This function is called when the player chooses to level up the pickups during the `LEVELING_UP` state.

Our `Pickup` class is ready for use.

Using the Pickup class

After all that hard work implementing the `Pickup` class, we can now go ahead and write code in the game engine to put some pickups into the game.

The first thing we will do is add an include directive to the `ZombieArena.cpp` file:

```
#include <SFML/Graphics.hpp>
#include "ZombieArena.h"
#include "Player.h"
#include "TextureHolder.h"
#include "Bullet.h"
#include "Pickup.h"

using namespace sf;
```

In this following code, we are adding two `Pickup` instances: one called `healthPickup` and another called `ammoPickup`. We pass the values 1 and 2, respectively, into the constructor so that they are initialized to the correct type of pickup. Add the following highlighted code, which we have just discussed:

```
// Hide the mouse pointer and replace it with crosshair
window.setMouseCursorVisible(true);
Sprite spriteCrosshair;
Texture textureCrosshair = TextureHolder::GetTexture(
    "graphics/crosshair.png");
spriteCrosshair.setTexture(textureCrosshair);
spriteCrosshair.setOrigin(25, 25);

// Create a couple of pickups
Pickup healthPickup(1);
Pickup ammoPickup(2);

// The main game loop
while (window.isOpen())
```

In the `LEVELING_UP` state of the keyboard handling, add the following highlighted lines within the nested `PLAYING` code block:

```
if (state == State::PLAYING)
{
    // Prepare the level
    // We will modify the next two lines later
    arena.width = 500;
    arena.height = 500;
    arena.left = 0;
    arena.top = 0;

    // Pass the vertex array by reference
    // to the createBackground function
    int tileSize = createBackground(background, arena);

    // Spawn the player in the middle of the arena
    player.spawn(arena, resolution, tileSize);

    // Configure the pick-ups
    healthPickup.setArena(arena);
    ammoPickup.setArena(arena);

    // Create a horde of zombies
    numZombies = 10;

    // Delete the previously allocated memory (if it exists)
    delete[] zombies;
    zombies = createHorde(numZombies, arena);
    numZombiesAlive = numZombies;

    // Reset the clock so there isn't a frame jump
    clock.restart();
}
```

The preceding code simply passes `arena` into the `setArena` function of each pickup. The pickups now know where they can spawn. This code executes for each new wave, so, as the arena's size grows, the `Pickup` objects will get updated.

The following code simply calls the `update` function for each `Pickup` object on each frame:

```
// Loop through each Zombie and update them
for (int i = 0; i < numZombies; i++)
{
```

```
        if (zombies[i].isAlive())
        {
            zombies[i].update(dt.asSeconds(), playerPosition);
        }
    }

    // Update any bullets that are in-flight
    for (int i = 0; i < 100; i++)
    {
        if (bullets[i].isInFlight())
        {
            bullets[i].update(dtAsSeconds);
        }
    }

    // Update the pickups
    healthPickup.update(dtAsSeconds);
    ammoPickup.update(dtAsSeconds);

} // End updating the scene
```

The following code in the draw part of the game loop checks whether the pickup is currently spawned and if it is, draws it. Let's add it:

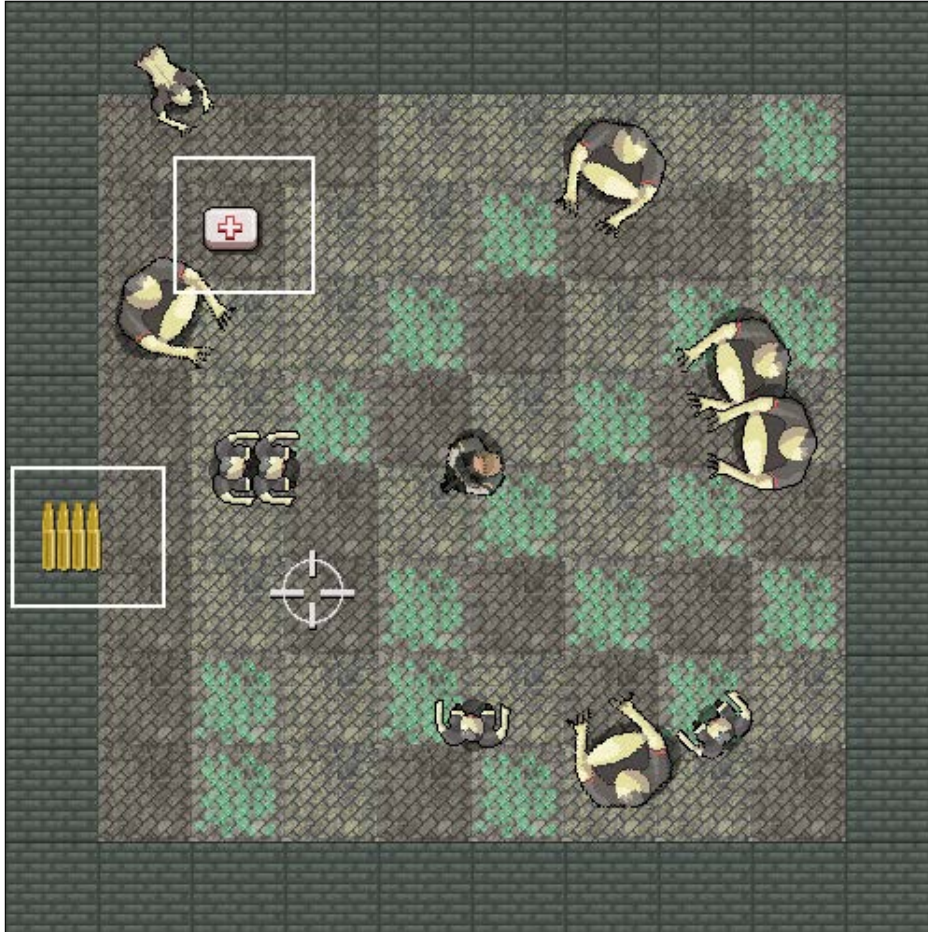
```
    // Draw the player
    window.draw(player.getSprite());

    // Draw the pick-ups, if currently spawned
    if (ammoPickup.isSpawned())
    {
        window.draw(ammoPickup.getSprite());
    }

    if (healthPickup.isSpawned())
    {
        window.draw(healthPickup.getSprite());
    }

    //Draw the crosshair
    window.draw(spriteCrosshair);
}
```

Now, you can run the game and see the pickups spawn and de-spawn. You can't, however, actually pick them up yet:



Now that we have all the objects in our game, it is a good time to make them interact (collide) with each other.

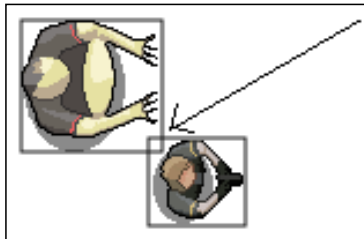
Detecting collisions

We just need to know when certain objects from our game touch certain other objects. We can then respond to that event in an appropriate manner. In our classes, we have already added functions that will be called when our objects collide. They are as follows:

- The `Player` class has a `hit` function. We will call it when a zombie collides with the player.
- The `Zombie` class has a `hit` function. We will call it when a bullet collides with a zombie.
- The `Pickup` class has a `gotIt` function. We will call it when the player collides with a pickup.

If necessary, look back to refresh your memory regarding how each of those functions works. All we need to do now is detect the collisions and call the appropriate functions.

We will use **rectangle intersection** to detect collisions. This type of collision detection is straightforward (especially with SFML). We will use the same technique that we used in the Pong game. The following image shows how a rectangle can reasonably accurately represent the zombies and the player:



We will deal with this in three sections of code that will all follow on from one another. They will all go at the end of the update part of our game engine.

We need to know the answers to the following three questions for each frame:

1. Has a Zombie been shot?
2. Has the player been touched by a Zombie?
3. Has the player touched a pickup?

First, let's add a couple more variables for `score` and `hiScore`. We can then change them when a zombie is killed. Add the following code:

```
// Create a couple of pickups
Pickup healthPickup(1);
Pickup ammoPickup(2);

// About the game
int score = 0;
int hiScore = 0;

// The main game loop
while (window.isOpen())
```

Now, let's start by detecting whether a zombie is colliding with a bullet.

Has a zombie been shot?

The following code might look complicated but, when we step through it, we will see it is nothing we haven't seen before. Add the following code just after the call to update the pickups each frame. Then, we can go through it:

```
// Update the pickups
healthPickup.update(dtAsSeconds);
ammoPickup.update(dtAsSeconds);

// Collision detection
// Have any zombies been shot?
for (int i = 0; i < 100; i++)
{
    for (int j = 0; j < numZombies; j++)
    {
        if (bullets[i].isInFlight() &&
            zombies[j].isAlive())
        {
            if (bullets[i].getPosition().intersects(
                zombies[j].getPosition()))
            {
                // Stop the bullet
                bullets[i].stop();

                // Register the hit and see if it was a kill
                if (zombies[j].hit())
                {
```


Within the nested `for` loops, we do the following.

We check whether the current bullet is in flight and the current zombie is still alive with the following code:

```
if (bullets[i].isInFlight() && zombies[j].isAlive())
```

Provided the zombie is alive and the bullet is in flight, we test for a rectangle intersection with the following code:

```
if (bullets[i].getPosition().intersects(zombies[j].getPosition()))
```

If the current bullet and zombie have collided, then we take a number of steps, as detailed next.

Stop the bullet with the following code:

```
// Stop the bullet
bullets[i].stop();
```

Register a hit with the current zombie by calling its `hit` function. Note that the `hit` function returns a `Boolean` that lets the calling code know whether the zombie is dead yet. This is shown in the following line of code:

```
// Register the hit and see if it was a kill
if (zombies[j].hit()) {
```

Inside this `if` block, which detects when the zombie is dead and hasn't just wounded us, do the following:

- Add ten to `score`.
- Change `hiScore` if the score the player has achieved has exceeded (beaten) `score`.
- Reduce `numZombiesAlive` by one.
- Check whether all the zombies are dead with `(numZombiesAlive == 0)` and if so, change state to `LEVELING_UP`.

Here is the block of code inside `if(zombies[j].hit())` that we have just discussed:

```
// Not just a hit but a kill too
score += 10;
if (score >= hiScore)
{
    hiScore = score;
}
```



```
numZombiesAlive--;

// When all the zombies are dead (again)
if (numZombiesAlive == 0)
{
    state = State::LEVELING_UP;
}
```

That's the zombies and the bullets taken care of. You can now run the game and see the blood. Of course, you won't see the score until we implement the HUD in the next chapter.

Has the player been touched by a zombie?

This code is much shorter and simpler than the zombie and bullet collision detection code. Add the following highlighted code just after the previous code we wrote:

```
}// End zombie being shot

// Have any zombies touched the player
for (int i = 0; i < numZombies; i++)
{
    if (player.getPosition().intersects
        (zombies[i].getPosition()) && zombies[i].isAlive())
    {

        if (player.hit(gameTimeTotal))
        {
            // More here later
        }

        if (player.getHealth() <= 0)
        {
            state = State::GAME_OVER;
        }
    }
}
}// End player touched
```

Here, we detect whether a zombie has collided with the player by using a `for` loop to go through all the zombies. For each zombie that is alive, the code uses the `intersects` function to test for a collision with the player. When a collision has occurred, we call `player.hit`. Then, we check whether the player is dead by calling `player.getHealth`. If the player's health is equal to or less than zero, then we change state to `GAME_OVER`.

You can run the game and collisions will be detected. However, as there is no HUD or sound effects yet, it is not clear that this is happening. In addition, we need to do some more work resetting the game when the player had died, and a new game is starting. So, although the game runs, the results are not especially satisfying right now. We will improve this over the next two chapters.

Has the player touched a pickup?

The collision detection code between the player and each of the two pickups is shown here. Add the following highlighted code just after the previous code that we added:

```

    }// End player touched

    // Has the player touched health pickup
    if (player.getPosition().intersects
        (healthPickup.getPosition()) && healthPickup.isSpawned())
    {
        player.increaseHealthLevel(healthPickup.gotIt());
    }

    // Has the player touched ammo pickup
    if (player.getPosition().intersects
        (ammoPickup.getPosition()) && ammoPickup.isSpawned())
    {
        bulletsSpare += ammoPickup.gotIt();
    }

    }// End updating the scene

```

The preceding code uses two simple `if` statements to see whether either `healthPickup` or `ammoPickup` have been touched by the player.

If a health pickup has been collected, then the `player.increaseHealthLevel` function uses the value returned from the `healthPickup.gotIt` function to increase the player's health.

If an ammo pickup has been collected, then `bulletsSpare` is increased by the value that's returned from `ammoPickup.gotIt`.



You can now run the game, kill zombies, and collect pickups! Note that, when your health equals zero, the game will enter the `GAME_OVER` state and pause. To restart it, you will need to press `Enter`, followed by a number between 1 and 6. When we implement the HUD, the home screen, and the leveling up screen, these steps will be intuitive and straightforward for the player. We will do so in the next chapter.

Summary

This was a busy chapter, but we achieved a lot. Not only did we add bullets and pickups to the game through two new classes, but we also made all the objects interact as they should by detecting when they collide with each other.

Despite these achievements, we need to do more work to set up each new game and to give the player feedback through a HUD. In the next chapter, we will build the HUD.

FAQ

Here are some questions that might be on your mind:

Q) Are there any better ways of doing collision detection?

A) Yes. There are lots more ways to do collision detection, including but not limited to the following.

- You can divide objects up into multiple rectangles that fit the shape of the sprite better. It is perfectly manageable for C++ to check on thousands of rectangles each frame. This is especially the case when you use techniques such as neighbor checking to reduce the number of tests that are necessary each frame.
- For circular objects, you can use the radius overlap method.
- For irregular polygons, you can use the passing number algorithm.

You can review all of these techniques, if you wish, by taking a look at the following links:

- Neighbor checking: <http://gamecodeschool.com/essentials/collision-detection-neighbor-checking/>
- Radius overlap method: <http://gamecodeschool.com/essentials/collision-detection-radius-overlap/>
- Crossing number algorithm: <http://gamecodeschool.com/essentials/collision-detection-crossing-number/>

12

Layering Views and Implementing the HUD

In this chapter, we will get to see the real value of SFML Views. We will add a large array of SFML Text objects and manipulate them, like we did before in the Timber!!! project and the Pong project. What's new is that we will draw the HUD using a second View instance. This way, the HUD will stay neatly positioned over the top of the main game action, regardless of what the background, player, zombies, and other game objects are doing.

Here is what we will do in this chapter:

- Add text and a background to the home/game over screen
- Add text to the level-up screen
- Create the second View
- Add a HUD

Adding all the Text and HUD objects

We will be manipulating a few strings in this chapter. We are doing this so we can format the HUD and the level-up screen with the necessary text.

Add the extra `include` directive highlighted in the following code so that we can make some `sstream` objects to achieve this:

```
#include <sstream>
#include <SFML/Graphics.hpp>
#include "ZombieArena.h"
#include "Player.h"
#include "TextureHolder.h"
```

```
#include "Bullet.h"
#include "Pickup.h"
```

```
using namespace sf;
```

Next, add this rather lengthy, but easily explainable, piece of code. To help identify where you should add the code, the new code is highlighted, and the existing code is not:

```
int score = 0;
int hiScore = 0;

// For the home/game over screen
Sprite spriteGameOver;
Texture textureGameOver = TextureHolder::GetTexture(
    "graphics/background.png");
spriteGameOver.setTexture(textureGameOver);
spriteGameOver.setPosition(0, 0);

// Create a view for the HUD
View hudView(sf::FloatRect(0, 0, resolution.x, resolution.y));

// Create a sprite for the ammo icon
Sprite spriteAmmoIcon;
Texture textureAmmoIcon = TextureHolder::GetTexture(
    "graphics/ammo_icon.png");
spriteAmmoIcon.setTexture(textureAmmoIcon);
spriteAmmoIcon.setPosition(20, 980);

// Load the font
Font font;
font.loadFromFile("fonts/zombiecontrol.ttf");

// Paused
Text pausedText;
pausedText.setFont(font);
pausedText.setCharacterSize(155);
pausedText.setFillColor(Color::White);
pausedText.setPosition(400, 400);
pausedText.setString("Press Enter \nto continue");

// Game Over
Text gameOverText;
gameOverText.setFont(font);
gameOverText.setCharacterSize(125);
```

```
gameOverText.setFillColor(Color::White);
gameOverText.setPosition(250, 850);
gameOverText.setString("Press Enter to play");

// LEVELING up
Text levelUpText;
levelUpText.setFont(font);
levelUpText.setCharacterSize(80);
levelUpText.setFillColor(Color::White);
levelUpText.setPosition(150, 250);
std::stringstream levelUpStream;
levelUpStream <<
    "1- Increased rate of fire" <<
    "\n2- Increased clip size(next reload)" <<
    "\n3- Increased max health" <<
    "\n4- Increased run speed" <<
    "\n5- More and better health pickups" <<
    "\n6- More and better ammo pickups";
levelUpText.setString(levelUpStream.str());

// Ammo
Text ammoText;
ammoText.setFont(font);
ammoText.setCharacterSize(55);
ammoText.setFillColor(Color::White);
ammoText.setPosition(200, 980);

// Score
Text scoreText;
scoreText.setFont(font);
scoreText.setCharacterSize(55);
scoreText.setFillColor(Color::White);
scoreText.setPosition(20, 0);

// Hi Score
Text hiScoreText;
hiScoreText.setFont(font);
hiScoreText.setCharacterSize(55);
hiScoreText.setFillColor(Color::White);
hiScoreText.setPosition(1400, 0);
std::stringstream s;
s << "Hi Score:" << hiScore;
hiScoreText.setString(s.str());
```

```
// Zombies remaining
Text zombiesRemainingText;
zombiesRemainingText.setFont(font);
zombiesRemainingText.setCharacterSize(55);
zombiesRemainingText.setFillColor(Color::White);
zombiesRemainingText.setPosition(1500, 980);
zombiesRemainingText.setString("Zombies: 100");

// Wave number
int wave = 0;
Text waveNumberText;
waveNumberText.setFont(font);
waveNumberText.setCharacterSize(55);
waveNumberText.setFillColor(Color::White);
waveNumberText.setPosition(1250, 980);
waveNumberText.setString("Wave: 0");

// Health bar
RectangleShape healthBar;
healthBar.setFillColor(Color::Red);
healthBar.setPosition(450, 980);

// The main game loop
while (window.isOpen())
```

The previous code is very simple and nothing new. It basically creates a whole bunch of SFML Text objects. It assigns their colors and sizes and then formats their positions using functions we have seen before.

The most important thing to note is that we create another View object called `hudView` and initialize it to fit the resolution of the screen.

As we have seen, the main View object scrolls around as it follows the player. In contrast, we will never move `hudView`. The result of this is that if we switch to this view before we draw the elements of the HUD, we will create the effect of allowing the game world to scroll by underneath while the player's HUD remains stationary.



As an analogy, you can think of laying a transparent sheet of plastic with some writing on it over a TV screen. The TV will carry on as normal with moving pictures, and the text on the plastic sheet will stay in the same place, regardless of what goes on underneath it. We will take this concept a step further in the next project when we split the screen and separate moving views of the game world.

The next thing to notice, however, is that the hi-score is not set in any meaningful way. We will need to wait until the next chapter, when we investigate file I/O, to save and retrieve the high score.

Another point worth noting is that we declare and initialize a `RectangleShape` called `healthBar`, which will be a visual representation of the player's remaining health. This will work in almost the same way that the time-bar worked in the `Timber!!!` project, except it will represent health instead of time.

In the previous code, there is a new `Sprite` instance called `ammoIcon` that gives context to the bullet and clip statistics that we will draw next to it, at the bottom-left of the screen.

Although there is nothing new or technical about the large amount of code that we just added, be sure to familiarize yourself with the details – especially the variable names – to make the rest of this chapter easier to follow.

Updating the HUD

As you might expect, we will update the HUD variables in the update section of our code. We will not, however, do so every frame. The reason for this is that it is unnecessary, and it also slows our game loop down.

As an example, consider the scenario when the player kills a zombie and gets some more points. It doesn't matter whether the `Text` object that holds the score is updated in one-thousandth, one-hundredth, or even one-tenth of a second. The player will discern no difference. This means there is no point rebuilding strings that we set for the `Text` objects every frame.

Therefore, we can time when and how often we update the HUD. Add the following highlighted variables:

```
// Debug HUD
Text debugText;
debugText.setFont(font);
debugText.setCharacterSize(25);
debugText.setFillColor(Color::White);
debugText.setPosition(20, 220);
std::ostringstream ss;

// When did we last update the HUD?
int framesSinceLastHUDUpdate = 0;
```



```
// How often (in frames) should we update the HUD
int fpsMeasurementFrameInterval = 1000;

// The main game loop
while (window.isOpen())
```

In the previous code, we have variables to track how many frames it has been since the last time the HUD was updated, and the interval, measured in frames, we would like to wait between HUD updates.

Now, we can use these new variables and update the HUD each frame. We won't see all the HUD elements change, however, until we begin to manipulate the final variables, such as `wave`, in the next chapter.

Add the following highlighted code in the update section of the game loop, as follows:

```
// Has the player touched ammo pickup
if (player.getPosition().intersects
    (ammoPickup.getPosition()) && ammoPickup.isSpawned())
{
    bulletsSpare += ammoPickup.gotIt();
}

// size up the health bar
healthBar.setSize(Vector2f(player.getHealth() * 3, 50));

// Increment the number of frames since the previous update
framesSinceLastHUDUpdate++;

// re-calculate every fpsMeasurementFrameInterval frames
if (framesSinceLastHUDUpdate > fpsMeasurementFrameInterval)
{
    // Update game HUD text
    std::stringstream ssAmmo;
    std::stringstream ssScore;
    std::stringstream ssHiScore;
    std::stringstream ssWave;
    std::stringstream ssZombiesAlive;

    // Update the ammo text
    ssAmmo << bulletsInClip << "/" << bulletsSpare;
    ammoText.setString(ssAmmo.str());
```

```
// Update the score text
ssScore << "Score:" << score;
scoreText.setString(ssScore.str());

// Update the high score text
ssHiScore << "Hi Score:" << hiScore;
hiScoreText.setString(ssHiScore.str());

// Update the wave
ssWave << "Wave:" << wave;
waveNumberText.setString(ssWave.str());

// Update the high score text
ssZombiesAlive << "Zombies:" << numZombiesAlive;
zombiesRemainingText.setString(ssZombiesAlive.str());

framesSinceLastHUDUpdate = 0;

} // End HUD update

} // End updating the scene
```

In the new code, we update the size of the healthBar sprite then increment the framesSinceLastHUDUpdate variable.

Next, we start an `if` block that tests whether framesSinceLastHUDUpdate is greater than our preferred interval, which is stored in `fpsMeasurementFrameInterval`.

Inside this `if` block is where all the action takes place. First, we declare a `stringstream` object for each string that we need to set to a `Text` object.

Then, we use each of those `stringstream` objects in turn and use the `setString` function to set the result to the appropriate `Text` object.

Finally, before the `if` block is exited, framesSinceLastHUDUpdate is set back to zero so that the count can begin again.

Now, when we redraw the scene, the new values will appear in the player's HUD.

Drawing the HUD, home, and level-up screens

All the code in the following three code blocks goes in the drawing phase of our game loop. All we need to do is draw the appropriate `Text` objects during the appropriate states, in the draw section of the main game loop.

In the `PLAYING` state, add the following highlighted code:

```
//Draw the crosshair
window.draw(spriteCrosshair);

// Switch to the HUD view
window.setView(hudView);

// Draw all the HUD elements
window.draw(spriteAmmoIcon);
window.draw(ammoText);
window.draw(scoreText);
window.draw(hiScoreText);
window.draw(healthBar);
window.draw(waveNumberText);
window.draw(zombiesRemainingText);
}

if (state == State::LEVELING_UP)
{
}
```

The vital thing to notice in the preceding block of code is that we switch views to the HUD view. This causes everything to be drawn at the precise screen positions we gave each of the elements of the HUD. They will never move.

In the `LEVELING_UP` state, add the following highlighted code:

```
if (state == State::LEVELING_UP)
{
    window.draw(spriteGameOver);
    window.draw(levelUpText);
}
```

In the PAUSED state, add the following highlighted code:

```
if (state == State::PAUSED)
{
    window.draw(pausedText);
}
```

In the GAME_OVER state, add the following highlighted code:

```
if (state == State::GAME_OVER)
{
    window.draw(spriteGameOver);
    window.draw(gameOverText);
    window.draw(scoreText);
    window.draw(hiScoreText);
}
```

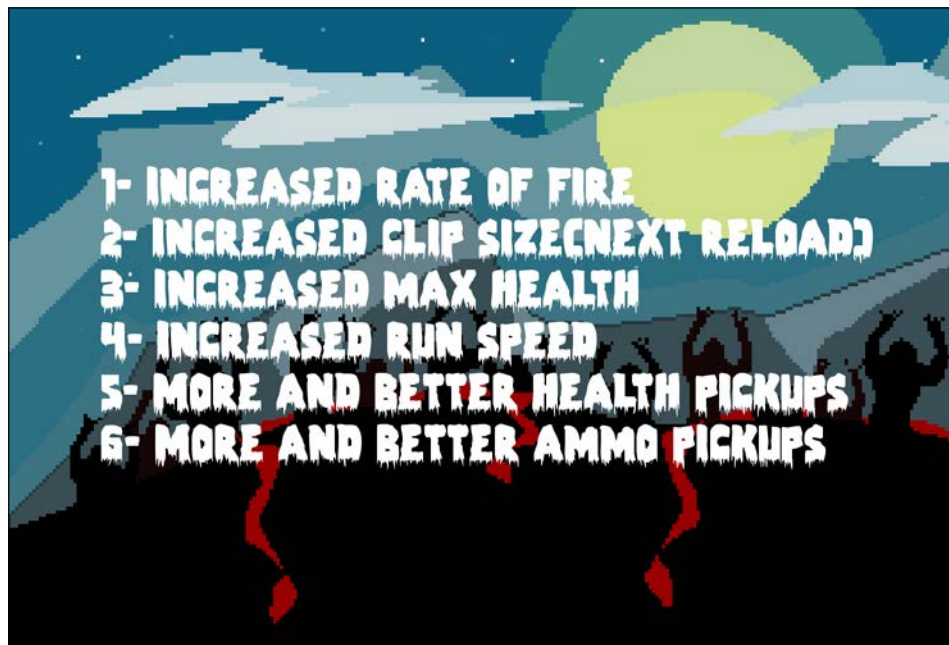
Now, we can run the game and see our HUD update during gameplay:



The following screenshot shows the high score and score on the home/game over screen:



Next, we see text that tells the player what their level-up options are, although these options don't do anything yet:



Here, we can see a helpful message on the pause screen:



SFML Views are more powerful than this simple HUD can demonstrate. For an insight into the potential of the SFML View class and how easy they are to use, look at the SFML website's tutorial on View at <https://www.sfml-dev.org/tutorials/2.5/graphics-view.php>.

Summary

This was a quick and simple chapter. We looked at how to display the values that are held by variables of different types using `sstream` and then learned how to draw them over the top of the main game action using a second SFML View object.

We are nearly done with Zombie Arena now. All the screenshots in this chapter show a small arena that doesn't take advantage of the full monitor.

In the next chapter, the final one for this project, we will put in some finishing touches, such as leveling up, sound effects, and saving the high score. The arena can then grow to the same size as the monitor and far beyond.

FAQ

Here is a question that might be on your mind:

Q) Where can I see more of the power of the `View` class in action?

A) Take a look at the enhanced edition of the Zombie Arena game, in the download bundle. You can use the cursor keyboard keys to spin and zoom the game. Warning! Spinning the scene makes the controls awkward, but you get to see some of the things that can be done with the `View` class:



The zoom and rotate functionality were achieved with just a few lines of code in the input handling section of the main game loop. You can see the code in the `Zombie Arena Enhanced Version` folder of the download bundle or run the enhanced version from the `Runnable Games/Zombie Arena` folder.

13

Sound Effects, File I/O, and Finishing the Game

We are nearly there. This short chapter will demonstrate how we can easily manipulate files stored on the hard drive using the C++ standard library, and we will also add sound effects. Of course, we know how to add sound effects, but we will discuss exactly where in the code the calls to the `play` function will go. We will also tie up a few loose ends to make the game complete.

In this chapter, we will cover the following topics:

- Saving and loading the hi-score using file input and file output
- Adding sound effects
- Allowing the player to level up
- Creating multiple never-ending waves

Saving and loading the high score

File **i/o** or **input/output** is a fairly technical subject. Fortunately for us, as it is such a common requirement in programming, there is a library that handles all this complexity for us. Like concatenating strings for our HUD, it is the C++ Standard Library that provides the necessary functionality through `fstream`.

First, we include `fstream` in the same way we included `sstream`:

```
#include <sstream>
#include <fstream>
#include <SFML/Graphics.hpp>
#include "ZombieArena.h"
#include "Player.h"
```



```
#include "TextureHolder.h"
#include "Bullet.h"
#include "Pickup.h"
```

```
using namespace sf;
```

Now, add a new folder in the `ZombieArena` folder called `gamedata`. Next, right-click in this folder and create a new file called `scores.txt`. It is in this file that we will save the player's high score. You can easily open the file and add a score to it. If you do, make sure it is quite a low score so that we can easily test whether beating that score results in the new score being added. Be sure to close the file once you are done with it or the game will not be able to access it.

In the following code, we will create an `ifstream` object called `inputFile` and send the folder and file we just created as a parameter to its constructor.

`if(inputFile.is_open())` checks that the file exists and is ready to read from. We then put the contents of the file into `hiScore` and close the file. Add the following highlighted code:

```
// Score
Text scoreText;
scoreText.setFont(font);
scoreText.setCharacterSize(55);
scoreText.setColor(Color::White);
scoreText.setPosition(20, 0);

// Load the high score from a text file
std::ifstream inputFile("gamedata/scores.txt");
if (inputFile.is_open())
{
    // >> Reads the data
    inputFile >> hiScore;
    inputFile.close();
}

// Hi Score
Text hiScoreText;
hiScoreText.setFont(font);
hiScoreText.setCharacterSize(55);
hiScoreText.setColor(Color::White);
hiScoreText.setPosition(1400, 0);
std::stringstream s;
s << "Hi Score:" << hiScore;
hiScoreText.setString(s.str());
```

Now, we can handle saving a potentially new high score. Within the block that handles the player's health being less than or equal to zero, we need to create an `ofstream` object called `outputFile`, write the value of `hiScore` to the text file, and then close the file, like so:

```
// Have any zombies touched the player
for (int i = 0; i < numZombies; i++)
{
    if (player.getPosition().intersects
        (zombies[i].getPosition()) && zombies[i].isAlive())
    {

        if (player.hit(gameTimeTotal))
        {
            // More here later
        }

        if (player.getHealth() <= 0)
        {
            state = State::GAME_OVER;

            std::ofstream outputFile("gamedata/scores.txt");
            // << writes the data
            outputFile << hiScore;
            outputFile.close();

        }
    }
}
} // End player touched
```

You can play the game and your hi-score will be saved. Quit the game and notice that your hi-score is still there if you play it again.

Let's make some noise.

Preparing sound effects

In this section, we will create all the `SoundBuffer` and `Sound` objects that we need to add a range of sound effects to the game.

Start by adding the required SFML `#include` statements:

```
#include <sstream>
#include <fstream>
```

```
#include <SFML/Graphics.hpp>
#include <SFML/Audio.hpp>
#include "ZombieArena.h"
#include "Player.h"
#include "TextureHolder.h"
#include "Bullet.h"
#include "Pickup.h"
```

Now, go ahead and add the seven `SoundBuffer` and `Sound` objects that load and prepare the seven sound files that we prepared in *Chapter 8, SFML Views – Starting the Zombie Shooter Game*:

```
// When did we last update the HUD?
int framesSinceLastHUDUpdate = 0;
// What time was the last update
Time timeSinceLastUpdate;
// How often (in frames) should we update the HUD
int fpsMeasurementFrameInterval = 1000;

// Prepare the hit sound
SoundBuffer hitBuffer;
hitBuffer.loadFromFile("sound/hit.wav");
Sound hit;
hit.setBuffer(hitBuffer);

// Prepare the splat sound
SoundBuffer splatBuffer;
splatBuffer.loadFromFile("sound/splat.wav");
Sound splat;
splat.setBuffer(splatBuffer);

// Prepare the shoot sound
SoundBuffer shootBuffer;
shootBuffer.loadFromFile("sound/shoot.wav");
Sound shoot;
shoot.setBuffer(shootBuffer);

// Prepare the reload sound
SoundBuffer reloadBuffer;
reloadBuffer.loadFromFile("sound/reload.wav");
Sound reload;
reload.setBuffer(reloadBuffer);

// Prepare the failed sound
SoundBuffer reloadFailedBuffer;
```

```

reloadFailedBuffer.loadFromFile("sound/reload_failed.wav");
Sound reloadFailed;
reloadFailed.setBuffer(reloadFailedBuffer);

// Prepare the powerup sound
SoundBuffer powerupBuffer;
powerupBuffer.loadFromFile("sound/powerup.wav");
Sound powerup;
powerup.setBuffer(powerupBuffer);

// Prepare the pickup sound
SoundBuffer pickupBuffer;
pickupBuffer.loadFromFile("sound/pickup.wav");
Sound pickup;
pickup.setBuffer(pickupBuffer);

// The main game loop
while (window.isOpen())

```

Now, the seven sound effects are ready to play. We just need to work out where in our code each of the calls to the `play` function will go.

Leveling up

The following code we will add allows the player to level up between waves. It is because of the work we have already done that this is straightforward to achieve.

Add the following highlighted code to the `LEVELING_UP` state where we handle player input:

```

// Handle the LEVELING up state
if (state == State::LEVELING_UP)
{
    // Handle the player LEVELING up
    if (event.key.code == Keyboard::Num1)
    {
        // Increase fire rate
        fireRate++;
        state = State::PLAYING;
    }

    if (event.key.code == Keyboard::Num2)
    {
        // Increase clip size
    }
}

```

```
        clipSize += clipSize;
        state = State::PLAYING;
    }

    if (event.key.code == Keyboard::Num3)
    {
        // Increase health
        player.upgradeHealth();
        state = State::PLAYING;
    }

    if (event.key.code == Keyboard::Num4)
    {
        // Increase speed
        player.upgradeSpeed();
        state = State::PLAYING;
    }

    if (event.key.code == Keyboard::Num5)
    {
        // Upgrade pickup
        healthPickup.upgrade();
        state = State::PLAYING;
    }

    if (event.key.code == Keyboard::Num6)
    {
        // Upgrade pickup
        ammoPickup.upgrade();
        state = State::PLAYING;
    }

    if (state == State::PLAYING)
    {
```

The player can now level up each time a wave of zombies is cleared. We can't, however, increase the number of zombies or the size of the level just yet.

In the next part of the `LEVELING_UP` state, right after the code we have just added, amend the code that runs when the state changes from `LEVELING_UP` to `PLAYING`.

Here is the code in full. I have highlighted the lines that are either new or have been slightly amended.

Add or amend the following highlighted code:

```

    if (event.key.code == Keyboard::Num6)
    {
        ammoPickup.upgrade();
        state = State::PLAYING;
    }

    if (state == State::PLAYING)
    {
        // Increase the wave number
        wave++;

        // Prepare the level
        // We will modify the next two lines later
        arena.width = 500 * wave;
        arena.height = 500 * wave;
        arena.left = 0;
        arena.top = 0;

        // Pass the vertex array by reference
        // to the createBackground function
        int tileSize = createBackground(background, arena);

        // Spawn the player in the middle of the arena
        player.spawn(arena, resolution, tileSize);

        // Configure the pick-ups
        healthPickup.setArena(arena);
        ammoPickup.setArena(arena);

        // Create a horde of zombies
        numZombies = 5 * wave;

        // Delete the previously allocated memory (if it exists)
        delete[] zombies;
        zombies = createHorde(numZombies, arena);
        numZombiesAlive = numZombies;

        // Play the powerup sound
        powerup.play();

        // Reset the clock so there isn't a frame jump
        clock.restart();
    }
} // End LEVELING up

```

The previous code starts by incrementing the `wave` variable. Then, the code is amended to make the number of zombies and size of the arena relative to the new value of `wave`. Finally, we add the call to `powerup.play()` to play the leveling up sound effect.

Restarting the game

We already determine the size of the arena and the number of zombies by the value of the `wave` variable. We must also reset the ammo and gun-related variables, as well as setting `wave` and `score` to zero at the start of each new game. Find the following code in the event-handling section of the game loop and add the following highlighted code:

```
// Start a new game while in GAME_OVER state
else if (event.key.code == Keyboard::Return &&
        state == State::GAME_OVER)
{
    state = State::LEVELING_UP;
    wave = 0;
    score = 0;

    // Prepare the gun and ammo for next game
    currentBullet = 0;
    bulletsSpare = 24;
    bulletsInClip = 6;
    clipSize = 6;
    fireRate = 1;

    // Reset the player's stats
    player.resetPlayerStats();
}
```

Now, we can play the game, the player can get even more powerful, and the zombies will get ever more numerous within an arena of increasing size – until they die. Then, the game starts all over again.

Playing the rest of the sounds

Now, we will add the rest of the calls to the `play` function. We will deal with each of them individually, as locating exactly where they go is key to playing them at the right moment.

Adding sound effects while the player is reloading

Add the following highlighted code in three places to play the appropriate `reload` or `reloadFailed` sound when the player presses the `R` key to attempt to reload their gun:

```
if (state == State::PLAYING)
{
    // Reloading
    if (event.key.code == Keyboard::R)
    {
        if (bulletsSpare >= clipSize)
        {
            // Plenty of bullets. Reload.
            bulletsInClip = clipSize;
            bulletsSpare -= clipSize;
            reload.play();
        }
        else if (bulletsSpare > 0)
        {
            // Only few bullets left
            bulletsInClip = bulletsSpare;
            bulletsSpare = 0;
            reload.play();
        }
        else
        {
            // More here soon?!
            reloadFailed.play();
        }
    }
}
```

The player will now get an audible response when they reload or attempt to reload. Let's move on to playing a shooting sound.

Making a shooting sound

Add the following highlighted call to `shoot.play()` near the end of the code that handles the player clicking the left mouse button:

```
// Fire a bullet
if (sf::Mouse::isButtonPressed(sf::Mouse::Left))
```



```
{

    if (gameTimeTotal.asMilliseconds()
        - lastPressed.asMilliseconds()
        > 1000 / fireRate && bulletsInClip > 0)
    {

        // Pass the centre of the player and crosshair
        // to the shoot function
        bullets[currentBullet].shoot(
            player.getCenter().x, player.getCenter().y,
            mouseWorldPosition.x, mouseWorldPosition.y);

        currentBullet++;
        if (currentBullet > 99)
        {
            currentBullet = 0;
        }
        lastPressed = gameTimeTotal;

        shoot.play();

        bulletsInClip--;
    }

}

} // End fire a bullet
```

The game will now play a satisfying shooting sound. Next, we will play a sound when the player is hit by a zombie.

Playing a sound when the player is hit

In this following code, we wrap the call to `hit.play` in a test to see if the `player.hit` function returns true. Remember that the `player.hit` function tests to see if a hit has been recorded in the previous 100 milliseconds. This will have the effect of playing a fast-repeating thud sound, but not so fast that the sound blurs into one noise.

Add the call to `hit.play`, as highlighted in the following code:

```
// Have any zombies touched the player
for (int i = 0; i < numZombies; i++)
{
```

```

    if (player.getPosition().intersects
        (zombies[i].getPosition()) && zombies[i].isAlive())
    {

        if (player.hit(gameTimeTotal))
        {
            // More here later
            hit.play();
        }

        if (player.getHealth() <= 0)
        {
            state = State::GAME_OVER;

            std::ofstream OutputFile("gamedata/scores.txt");
            OutputFile << hiScore;
            OutputFile.close();
        }
    }
}
} // End player touched

```

The player will hear an ominous thudding sound when a zombie touches them, and this sound will repeat around five times per second if the zombie continues touching them. The logic for this is contained in the `hit` function of the `Player` class.

Playing a sound when getting a pickup

When the player picks up a health pickup, we will play the regular pickup sound. However, when the player gets an ammo pickup, we will play the reload sound effect.

Add the two calls to play sounds within the appropriate collision detection code:

```

// Has the player touched health pickup
if (player.getPosition().intersects
    (healthPickup.getPosition()) && healthPickup.isSpawned())
{
    player.increaseHealthLevel(healthPickup.gotIt());
    // Play a sound
    pickup.play();
}

```

```
// Has the player touched ammo pickup
if (player.getPosition().intersects
    (ammoPickup.getPosition()) && ammoPickup.isSpawned())
{
    bulletsSpare += ammoPickup.gotIt();
    // Play a sound
    reload.play();
}
```

Making a splat sound when a zombie is shot

Add a call to `splat.play` at the end of the section of code that detects a bullet colliding with a zombie:

```
// Have any zombies been shot?
for (int i = 0; i < 100; i++)
{
    for (int j = 0; j < numZombies; j++)
    {
        if (bullets[i].isInFlight() &&
            zombies[j].isAlive())
        {
            if (bullets[i].getPosition().intersects
                (zombies[j].getPosition()))
            {
                // Stop the bullet
                bullets[i].stop();

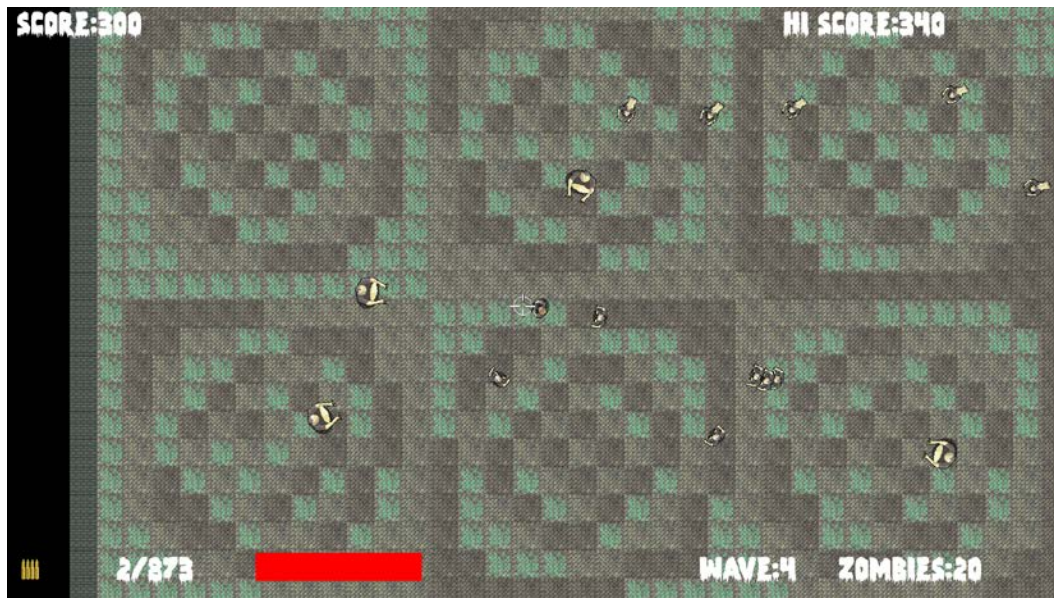
                // Register the hit and see if it was a kill
                if (zombies[j].hit()) {
                    // Not just a hit but a kill too
                    score += 10;
                    if (score >= hiScore)
                    {
                        hiScore = score;
                    }

                    numZombiesAlive--;

                    // When all the zombies are dead (again)
                    if (numZombiesAlive == 0) {
                        state = State::LEVELING_UP;
                    }
                }
            }
        }
    }
}
```

```
    }  
  
    // Make a splat sound  
    splat.play();  
}  
  
}  
  
}  
} // End zombie being shot
```

You can now play the completed game and watch the number of zombies and the arena increase each wave. Choose your level-ups carefully:



Congratulations!

Summary

We've finished the Zombie Arena game. It has been quite a journey. We have learned a whole bunch of C++ fundamentals, such as references, pointers, OOP, and classes. In addition, we have used SFML to manage cameras (views), vertex arrays, and collision detection. We learned how to use sprite sheets to reduce the number of calls to `window.draw` and speed up the frame rate. Using C++ pointers, the STL, and a little bit of OOP, we built a singleton class to manage our textures. In the next project, we will extend this idea to manage all of our game's assets.

Coming up in the penultimate project of this book, we will discover particle effects, directional sound, and split-screen co-op gaming. In C++, we will encounter inheritance, polymorphism, and a few more new concepts as well.

FAQ

Here are some questions that might be on your mind:

Q) Despite using classes, I am finding that the code is getting very long and unmanageable again.

A) One of the biggest issues is the structure of our code. As we learn more C++, we will also learn ways to make the code more manageable and generally less lengthy. We will do so in the next project and the final project too. By the end of this book, you will know about a number of strategies that you can use to manage your code.

Q) The sound effects seem a bit flat and unrealistic. How can they be improved?

A) One way to significantly improve the feeling the player gets from sound is to make the sound directional, as well as changing the volume based on the distance of the sound source to the player character. We will use SFML's advanced sound features in the next project.

14

Abstraction and Code Management – Making Better Use of OOP

In this chapter, we will take a first look at the penultimate project of this book. The project we will be building will use advanced features such as directional sound, which has the effect of appearing to play relative to the position of the player. It will also have split-screen cooperative gameplay. In addition, this project will introduce the concept of **Shaders**, which are programs written in another language that run directly on the graphics card. By the end of *Chapter 18, Particle Systems and Shaders*, you will have a fully functioning, multiplayer platform game built in the style of the hit classic *Thomas Was Alone*.

This chapter's focus will be getting the project started and exploring how the code will be structured to make better use of OOP. Here are the details of the topics that will be covered in this chapter:

- Introducing the final project, **Thomas Was Late**, including the gameplay features and project assets
- A detailed discussion of how we will improve the structure of the code compared to previous projects
- Coding the Thomas Was Late game engine
- Implementing split-screen functionality

The Thomas Was Late game



At this point, if you haven't already, I suggest that you go and watch a video of Thomas Was Alone at <http://store.steampowered.com/app/220780/>.

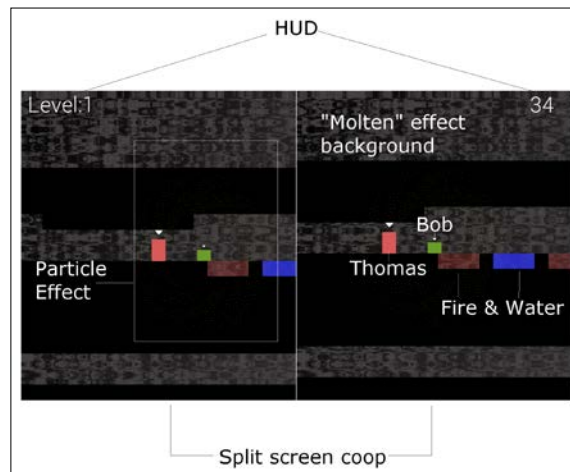
Notice the simple but aesthetically excellent graphics. The video also shows a variety of gameplay challenges such as using the character's different attributes (height, jump, power, and so on). To keep our game simple without losing the challenge, we will have fewer puzzle features than Thomas Was Alone but will have the additional challenge of creating the need for two players to play cooperatively. Just to make sure the game is not too easy, we will also make the players have to rush to beat the clock, which is why the name of our game is Thomas Was Late.

Features of Thomas Was Late

Our game will not be nearly as advanced as the masterpiece that we are attempting to emulate, but it will have a good selection of exciting game-play features, such as the following:

- A clock that counts down from a time appropriate to the challenge of the level.
- Fire pits that emit a roar relative to the position of the player and respawn the player at the start if they fall in. Water pits have the same effect but without the directional sound effects.
- Cooperative gameplay. Both players will have to get their characters to the goal within the allotted time. They will need to work together frequently so that the shorter, lower-jumping Bob will need to stand on his friend's (Thomas') head.
- The player will have the option of switching between full and split-screen so they can attempt to control both characters themselves.
- Each level will be designed in, and loaded from, a text file. This will make it easy to design varied and numerous levels.

Take a look at the following annotated screenshot of the game to see some of the features in action and the components/assets that make up the game:

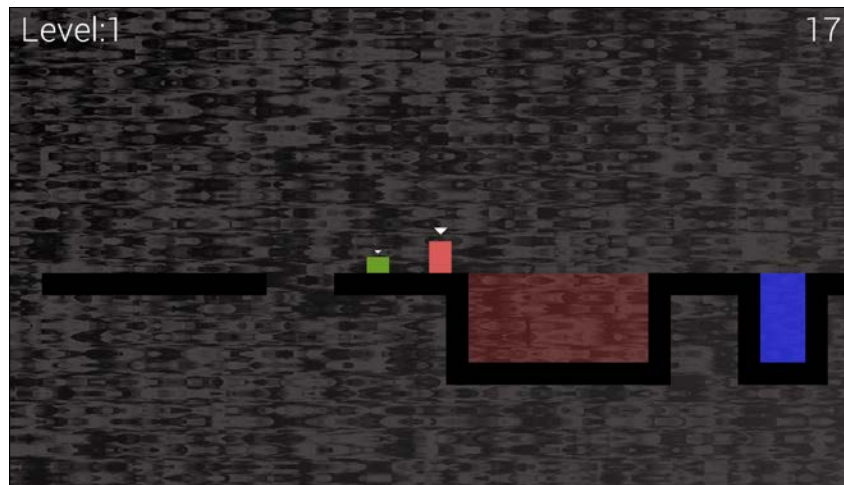


Let's look at each of these features and describe a few more:

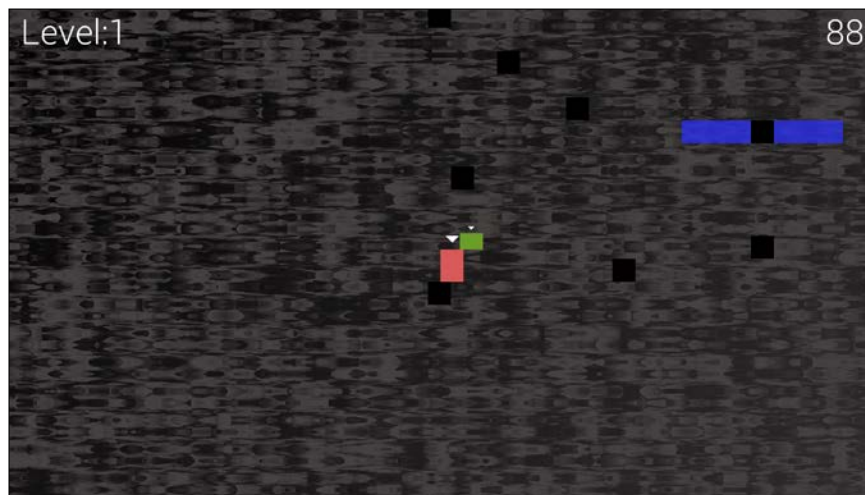
- The preceding screenshot shows a simple HUD that details the level number and the number of seconds remaining until the player(s) fail and must restart the level.
- You can also clearly see the split-screen coop in action. Remember that this is optional. A single player can take on the game, fullscreen, while switching the camera focus between Thomas and Bob.
- It is not very clear in the preceding screenshot (especially in print), but when a character dies, they will explode in a starburst/firework-like particle effect.
- The water and fire tiles can be strategically placed to make the level fun, as well as forcing cooperation between the characters. More on this will be covered in *Chapter 16, Building Playable Levels and Collision Detection*.
- Next, notice Thomas and Bob. They are not only different in height but also have significantly varied jumping abilities. This means that Bob is dependent upon Thomas for big jumps, and levels can be designed to force Thomas to take a specific route to avoid him "banging his head".
- In addition, the fire tiles will emit a roaring sound. These will be relative to the position of Thomas. Not only will they be directional and come from either the left or right speaker, they will also get louder and quieter as Thomas moves closer to or further away from the source.
- Finally, in the preceding annotated screenshot, you can see the background. Why not compare how that looks to the `background.png` file (shown later in this chapter)? You will see it is quite different. We will use OpenGL shader effects in *Chapter 18, Particle Systems and Shaders*, to achieve the moving, almost bubbling, effect in the background.

All of these features warrant a few more screenshots so that we can keep the finished product in mind as we write the C++ code.

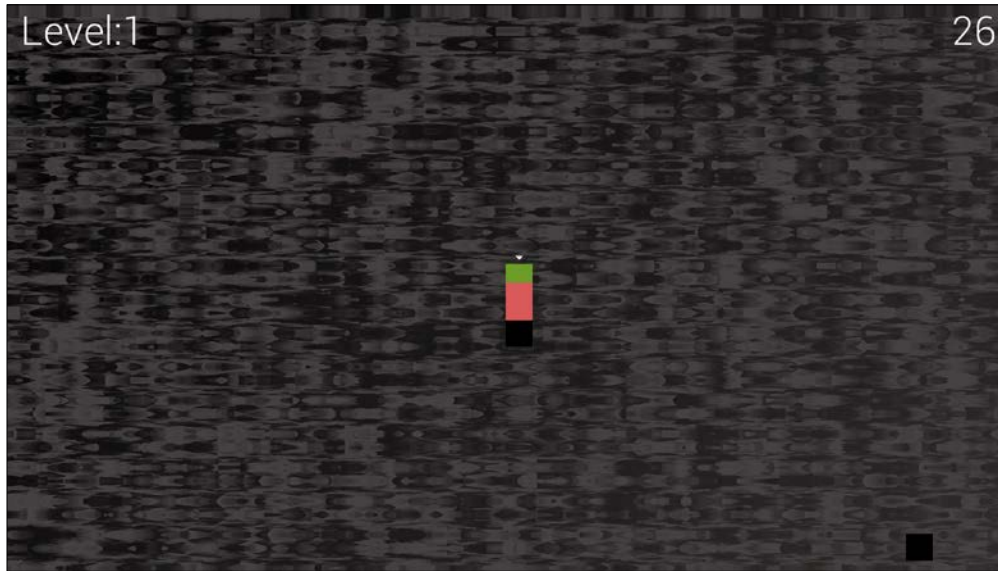
The following screenshot shows Thomas and Bob arriving at a fire pit that Bob has no chance of jumping over without help:



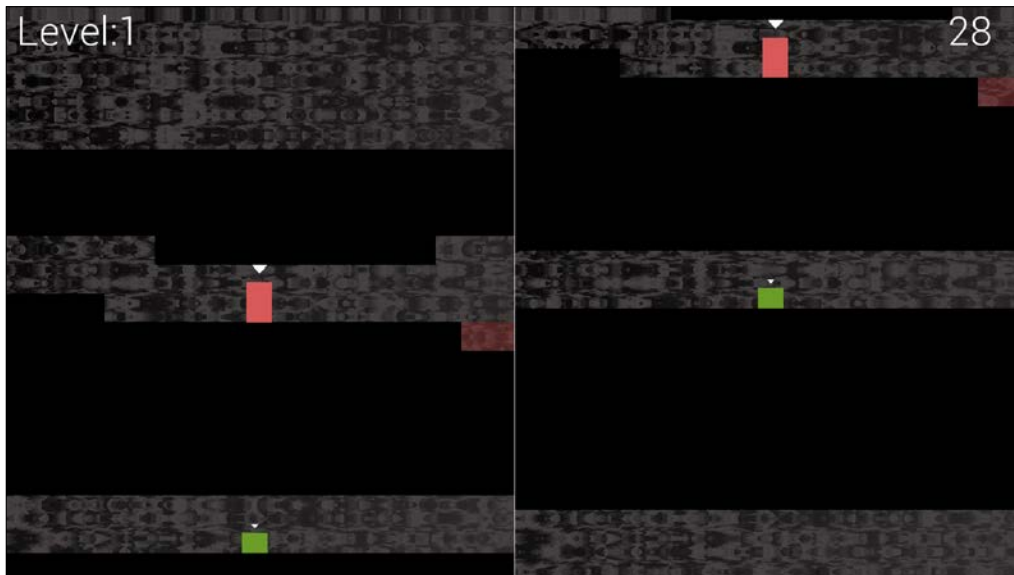
The following screenshot shows Bob and Thomas collaborating to clear a precarious jump:



The following screenshot shows how we can design puzzles where a "leap of faith" is required in order to reach the goal:



The following screenshot demonstrates how we can design oppressive cave systems of almost any size. We can also devise levels where Bob and Thomas are forced to split up and go different routes:



Creating the project

Creating the Thomas Was Late project will follow the same procedure that we used in the previous three projects. Since creating a project is a slightly fiddly process, I will detail all the steps again here. For even more detail and images, refer to setting up the Timber!!! project in *Chapter 1, C++, SFML, Visual Studio, and Starting the First Game*:

1. Start Visual Studio and click on the **Create New Project** button. If you have another project open, you can select **File | New project**.
2. In the window shown next, choose **Console app** and click the **Next** button. You will then see the **Configure your new project** window.
3. In the **Configure your new project** window, type TWL in the **Project name** field.
4. In the **Location** field, browse to the `VS Projects` folder.
5. Check the option to **Place solution and project in the same directory**.
6. When you have completed these steps, click **Create**.
7. We will now configure the project to use the SFML files that we put in the `SFML` folder. From the main menu, select **Project | TWL properties....** At this stage, you should have the **TWL Property Pages** window open.
8. In the **TWL Property Pages** window, take the following steps. Select **All Configurations** from the **Configuration:** dropdown.
9. Now, select **C/C++** and then **General** from the left-hand menu.
10. Now, locate the **Additional Include Directories** edit box and type the drive letter where your SFML folder is located, followed by `\SFML\include`. The full path to type, if you located your SFML folder on your D drive, is `D:\SFML\include`. Vary your path if you installed SFML on a different drive.
11. Click **Apply** to save your configurations so far.
12. Now, still in the same window, perform the following steps. From the left-hand menu, select **Linker** and then **General**.
13. Now, find the **Additional Library Directories** edit box and type the drive letter where your SFML folder is, followed by `\SFML\lib`. So, the full path to type if you located your SFML folder on your D drive is `D:\SFML\lib`. Vary your path if you installed SFML to a different drive.
14. Click **Apply** to save your configurations so far.
15. Next, still in the same window, perform these steps. Switch the **Configuration:** dropdown to **Debug** as we will be running and testing Pong in debug mode.
16. Select **Linker** and then **Input**.

17. Find the **Additional Dependencies** edit box and click into it at the far-left-hand side. Now, copy and paste/type the following: `sfml-graphics-d.lib;sfml-window-d.lib;sfml-system-d.lib;sfml-network-d.lib;sfml-audio-d.lib;`. Be extra careful to place the cursor exactly at the start of the edit box's current content so that you don't overwrite any of the text that is already there.
18. Click **OK**.
19. Click **Apply** and then **OK**.

That's the project properties configured and ready to go. Now, we need to copy the SFML .dll files into the main project directory by following these steps:

1. My main project directory is `D:\VS Projects\TWL`. This folder was created by Visual Studio in the previous steps. If you put your `Projects` folder somewhere else, then perform this step there instead. The files we need to copy into the project folder are located in our `SFML\bin` folder. Open a window for each of the two locations and highlight all the .dll files.
2. Now, copy and paste the highlighted files into the project.

The project is now set up and ready to go.

The project's assets

The assets in this project are even more numerous and diverse than the *Zombie Arena* game. As usual, the assets include a font for the writing on the screen, sound effects for different actions such as jumping, reaching the goal, or the distant roar of fire, and, of course, graphics for Thomas and Bob as well as a sprite sheet for all the background tiles.

All of the assets that are required for this game are included in the download bundle. They can be found in the `Chapter 14/graphics` and `Chapter 14/sound` folders.

In addition to the graphics, sounds, and fonts that we have come to expect, this game has two new asset types. They are level design files and GLSL shader programs. Let's find out about each of them.

Game level designs

Levels are all created in a text file. By using the numbers 0 through 3, we can build level designs to challenge players. All the level designs are in the `levels` folder in the same directory as the other assets. Feel free to take a peek at one now, but we will look at them in detail in *Chapter 18, Particle Systems and Shaders*.

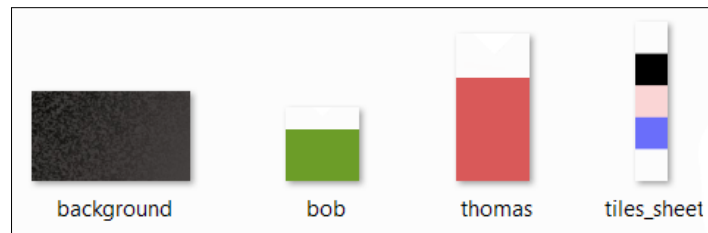
In addition to these level design assets, we have a special type of graphical asset called **shaders**.

GLSL shaders

Shaders are programs written in **GLSL (Graphics Library Shading Language)**. Don't worry about having to learn another language as we don't need to get too in-depth to take advantage of shaders. Shaders are special as they are entire programs, separate from our C++ code, that are executed by the GPU each and every frame. In fact, some of these shader programs are run every frame, for every pixel! We will find out more about these details in *Chapter 18, Particle Systems and Shaders*. If you can't wait that long, take a look at the files in the `Chapter 14/shaders` folder of the download bundle.

The graphical assets close up

The graphical assets make up the parts of the scene of our game. If you take a look at the graphical assets, it should be clear where in our game they will be used:



If the tiles on the `tiles_sheet` graphic look a little different to the screenshots of the game, this is because they are partly transparent and the background showing through changes them a little. If the background graphic looks totally different to the actual background in the game screenshots, that is because the shader programs we will write will manipulate each and every pixel, each and every frame, to create a kind of "molten" effect.

The sound assets close up

The sound files are all in `.wav` format. These files contain the sound effects that we will play at certain events throughout the game. They are as follows:

- `fallinfire.wav`: A sound that will be played when the player's head goes into fire and the player has no chance of escape.
- `fallinwater.wav`: Water has the same end effect as fire: death. This sound effect notifies the player that they need to start from the beginning of the level.

- `fire1.wav`: This sound effect is recorded in mono. It will be played at different volumes, based on the player's distance from fire tiles and from different speakers based on whether the player is to the left or the right of the fire tile. Clearly, we will need to learn a few more tricks to implement this functionality.
- `jump.wav`: A pleasing (slightly predictable) whooping sound for when the player jumps.
- `reachgoal.wav`: A pleasing victory sound for when the player(s) get both characters (Thomas and Bob) to the goal tile.

The sound effects are very straightforward, and you can easily create your own. If you intend to replace the `fire1.wav` file, be sure to save your sounds in mono (not stereo) format. The reasons for this will be explained in *Chapter 17, Sound Spatialization and HUD*.

Adding the assets to the project

Once you have decided which assets you will use, it is time to add them to the project. The following instructions will assume you are using all the assets that were supplied in this book's download bundle.

Where you are using your own, simply replace the appropriate sound or graphic file with your own, using exactly the same filename. Let's get started:

1. Browse to the `D:\VS Projects\TWL` folder.
2. Create five new folders within this folder and name them `graphics`, `sound`, `fonts`, `shaders`, and `levels`.
3. From the download bundle, copy the entire contents of Chapter 14/`graphics` into the `D:\VS Projects\TWL\graphics` folder.
4. From the download bundle, copy the entire contents of Chapter 14/`sound` into the `D:\VS Projects\TWL\sound` folder.
5. Now, visit <http://www.dafont.com/roboto.font> in your web browser and download the **Roboto Light** font.
6. Extract the contents of the zipped download and add the `Roboto-Light.ttf` file to the `D:\VS Projects\TWL\fonts` folder.
7. From the download bundle, copy the entire contents of Chapter 12/`levels` into the `D:\VS Projects\TWL\levels` folder.
8. From the download bundle, copy the entire contents of Chapter 12/`shaders` into the `D:\VS Projects\TWL\shaders` folder.

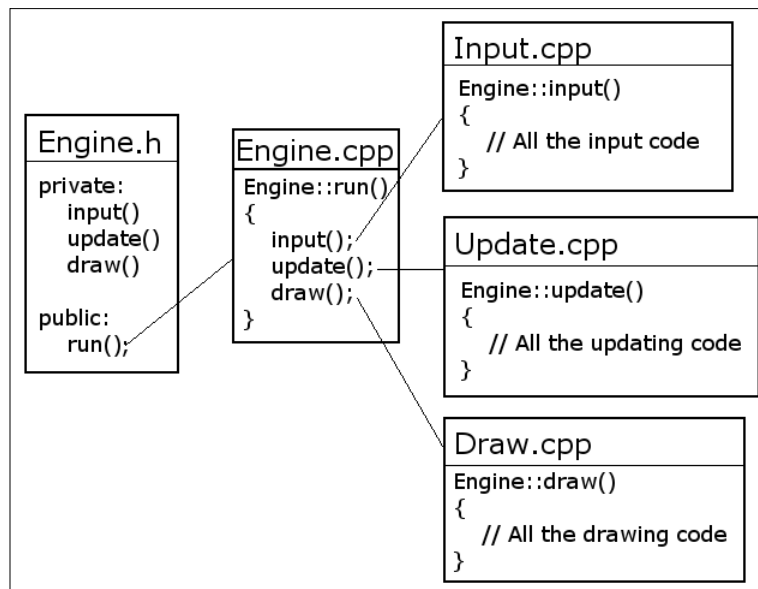
Now that we have a new project, along with all the assets we will need for the entire project, we can talk about how we will structure the game engine code.

Structuring the Thomas Was Late code

One of the problems that has been getting worse from project to project, despite taking measures to reduce the problem, is how long and unwieldy the code gets. **Object-oriented programming (OOP)** allows us to break our projects up into logical and manageable chunks, known as classes.

We will make a big improvement to the manageability of the code in this project with the introduction of an `Engine` class. Among other functions, the `Engine` class will have three private functions. These are `input`, `update`, and `draw`. These should sound very familiar. Each of these functions will hold a chunk of the code that was previously in the `main` function. Each of these functions will be in a code file of its own, that is, `Input.cpp`, `Update.cpp`, and `Draw.cpp`, respectively.

There will also be one public function in the `Engine` class, which can be called with an instance of `Engine`. This function is `run` and will be responsible for calling `input`, `update`, and `draw` once for each frame of the game:



Furthermore, because we have abstracted the major parts of the game engine to the `Engine` class, we can also move many of the variables from `main` and make them members of `Engine`. All we need to do to get our game engine fired up is create an instance of `Engine` and call its `run` function. Here is a sneak preview of the super-simple `main` function:

```
int main()
{
    // Declare an instance of Engine
    Engine engine;

    // Start the engine
    engine.run();

    // Quit in the usual way when the engine is stopped
    return 0;
}
```



Don't add the previous code just yet.

To make our code even more manageable and readable, we will also abstract responsibility for big tasks such as loading a level and collision detection to separate functions (in separate code files). These two functions are `loadLevel` and `detectCollisions`. We will also code other functions to handle some of the new features of the Thomas Was Late project. We will cover them in detail, as and when they occur.

To further take advantage of OOP, we will delegate responsibility for areas of the game entirely to new classes. You probably remember that the sound and HUD code was quite lengthy in previous projects. We will build a `SoundManager` and `HUD` class to handle these aspects in a cleaner manner. Exactly how they work will be explored in depth when we implement them.

The game levels themselves are also much more in-depth than previous games, so we will also code a `LevelManager` class.

As you would expect, the playable characters will be made with classes as well. For this project, however, we will learn some more C++ and implement a `PlayableCharacter` class with all the common functionality of Thomas and Bob. Then, the `Thomas` and `Bob` classes will *inherit* this common functionality as well as implementing their own unique functions and abilities. This technique, perhaps unsurprisingly, is called **inheritance**. I will go into more detail about inheritance in the next chapter: *Chapter 15, Advanced OOP – Inheritance and Polymorphism*.

We will also implement several other classes to perform specific responsibilities. For example, we will make some neat explosions using particle systems. You might be able to guess that, to do this, we will code a `Particle` class and a `ParticleSystem` class. All these classes will have instances that are members of the `Engine` class. Doing things this way will make all the features of the game accessible from the game engine but encapsulate the details into the appropriate classes.



Note that despite these new techniques to separate out the different aspects of our code, by the end of this project, we will still have some slightly unwieldy classes. The final project of this book, while a much simpler shooter game, will explore one more way of organizing our code to make it manageable.

The last thing to mention before we move on to seeing the actual code that will make the `Engine` class is that we will reuse, without any changes whatsoever, the `TextureHolder` class that we discussed and coded for the *Zombie Arena* game.

Building the game engine

As we suggested in the previous section, we will code a class called `Engine` that will control and bind the different parts of the *Thomas Was Late* game.

The first thing we will do is make the `TextureHolder` class from the previous project available in this one.

Reusing the `TextureHolder` class

The `TextureHolder` class that we discussed and coded for the *Zombie Arena* game will also be useful in this project. While it is possible to add the files (`TextureHolder.h` and `TextureHolder.cpp`) directly from the previous project, without recoding them or recreating the files, I don't want to assume that you haven't jumped straight to this project. What follows is very brief instructions, along with the complete code listing we need, to create the `TextureHolder` class. If you want the class or the code explained, please see *Chapter 10, Pointers, the Standard Template Library, and Texture Management*.



If you did complete the previous project and you *do* want to add the class from the Zombie Arena project, simply do the following. In the **Solution Explorer** window, right-click **Header Files** and select **Add | Existing Item....** Browse to `TextureHolder.h` from the previous project and select it. In the **Solution Explorer** window, right-click **Source Files** and select **Add | Existing Item....** Browse to `TextureHolder.cpp` from the previous project and select it. You can now use the `TextureHolder` class in this project. Note that the files are shared between projects and any changes will take effect in both projects.

To create the `TextureHolder` class from scratch, right-click **Header Files** in the **Solution Explorer** and select **Add | New Item....** In the **Add New Item** window, highlight (by left-clicking) **Header File (.h)**, and then, in the **Name** field, type `TextureHolder.h`. Finally, click the **Add** button.

Add the following code to `TextureHolder.h`:

```
#pragma once
#ifndef TEXTURE_HOLDER_H
#define TEXTURE_HOLDER_H

#include <SFML/Graphics.hpp>
#include <map>

class TextureHolder
{
private:
    // A map container from the STL,
    // that holds related pairs of String and Texture
    std::map<std::string, sf::Texture> m_Textures;

    // A pointer of the same type as the class itself
    // the one and only instance
    static TextureHolder* m_s_Instance;

public:
    TextureHolder();
    static sf::Texture& GetTexture(std::string const& filename);

};

#endif
```

Right-click **Source Files** in the **Solution Explorer** and select **Add | New Item....** In the **Add New Item** window, highlight (by left-clicking) **C++ File (.cpp)**, and then, in the **Name** field, type `TextureHolder.cpp`. Finally, click the **Add** button.

Add the following code to `TextureHolder.cpp`:

```
#include "TextureHolder.h"
#include <assert.h>

using namespace sf;
using namespace std;

TextureHolder* TextureHolder::m_s_Instance = nullptr;

TextureHolder::TextureHolder()
{
    assert(m_s_Instance == nullptr);
    m_s_Instance = this;
}

sf::Texture& TextureHolder::GetTexture(std::string const& filename)
{
    // Get a reference to m_Textures using m_S_Instance
    auto& m = m_s_Instance->m_Textures;
    // auto is the equivalent of map<string, Texture>

    // Create an iterator to hold a key-value-pair (kvp)
    // and search for the required kvp
    // using the passed in file name
    auto keyValuePair = m.find(filename);
    // auto is equivalent of map<string, Texture>::iterator

    // Did we find a match?
    if (keyValuePair != m.end())
    {
        // Yes
        // Return the texture,
        // the second part of the kvp, the texture
        return keyValuePair->second;
    }
    else
    {
        // File name not found
        // Create a new key value pair using the filename
    }
}
```

```

        auto& texture = m[filename];
        // Load the texture from file in the usual way
        texture.loadFromFile(filename);

        // Return the texture to the calling code
        return texture;
    }
}

```

We can now get on with our new Engine class.

Coding Engine.h

As usual, we will start with the header file, which holds the function declarations and member variables. Note that we will revisit this file throughout the project to add more functions and member variables. At this stage, we will add just the code that is necessary.

Right-click **Header Files** in the **Solution Explorer** and select **Add | New Item....** In the **Add New Item** window, highlight (by left-clicking) **Header File (.h)** and then, in the **Name** field, type `Engine.h`. Finally, click the **Add** button. We are now ready to code the header file for the Engine class.

Add the following member variables, as well as the function declarations. Many of them we have seen before in the other projects and some of them were discussed in the *Structuring the Thomas Was Late Code* section. Take note of the function and variable names, as well as whether they are private or public. Add the following code to the `Engine.h` file and then we will talk about it:

```

#pragma once
#include <SFML/Graphics.hpp>
#include "TextureHolder.h"

using namespace sf;

class Engine
{
private:
    // The texture holder
    TextureHolder th;

    const int TILE_SIZE = 50;
    const int VERTS_IN_QUAD = 4;

```

```
// The force pushing the characters down
const int GRAVITY = 300;

// A regular RenderWindow
RenderWindow m_Window;

// The main Views
View m_MainView;
View m_LeftView;
View m_RightView;

// Three views for the background
View m_BGMainView;
View m_BGLeftView;
View m_BGRightView;

View m_HudView;

// Declare a sprite and a Texture
// for the background
Sprite m_BackgroundSprite;
Texture m_BackgroundTexture;

// Is the game currently playing?
bool m_Playing = false;

// Is character 1 or 2 the current focus?
bool m_Character1 = true;

// Start in full screen (not split) mode
bool m_SplitScreen = false;

// Time left in the current level (seconds)
float m_TimeRemaining = 10;
Time m_GameTimeTotal;

// Is it time for a new/first level?
bool m_NewLevelRequired = true;

// Private functions for internal use only
void input();
void update(float dtAsSeconds);
void draw();
```

```

public:
    // The Engine constructor
    Engine();

    // Run will call all the private functions
    void run();

};

```

Here is a complete run down of all the private variables and functions. Where it is appropriate, I will spend a little longer on the explanation:

- `TextureHolder th`: The one and only instance of the `TextureHolder` class.
- `TILE_SIZE`: A useful constant to remind us that each tile in the sprite-sheet is 50 pixels wide and 50 pixels high.
- `VERTS_IN_QUAD`: A useful constant to make our manipulation of a `VertexArray` less error-prone. There are, in fact, four vertices in a quad. Now, we can't forget it.
- `GRAVITY`: A constant `int` value representing the number of pixels by which the game characters will be pushed downward each second. This is quite a fun value to play with once the game is done. We initialize it to 300 here as this works well for our initial level designs.
- `m_Window`: The usual `RenderWindow` object that we have had in all our projects.
- The SFML `View` objects, `m_MainView`, `m_LeftView`, `m_RightView`, `m_BGMainView`, `m_BGLeftView`, `m_BGRightView`, and `m_HudView`: The first three `View` objects are for the full screen view and the left and right and split-screen views of the game. We also have a separate SFML `View` object for each of those three, which will draw the background behind. The last `View` object, `m_HudView`, will be drawn on top of the appropriate combination of the other six views to display the score, the remaining time, and any messages to the players. Having seven different `View` objects might imply complexity, but when you see how we deal with them as the chapter progresses, you will see they are quite straightforward. We will have the whole split-screen/full screen conundrum sorted out by the end of this chapter.
- `Sprite m_BackgroundSprite` and `Texture m_BackgroundTexture`: Somewhat predictably, this combination of SFML `Sprite` and `Texture` will be for showing and holding the background graphic from the graphics assets folder.

- `m_Playing`: This Boolean will keep the game engine informed about whether the level has started yet (by pressing the *Enter* key). The player does not have the option to pause the game once they have started it.
- `m_Character1`: When the screen is full screen, should it center on Thomas (`m_Character1 = true`) or Bob (`m_Character1 = false`)? Initially, it is initialized to `true`, to center on Thomas.
- `m_SplitScreen`: This variable is used to determine whether the game currently being played is in split-screen mode or not. We will use this variable to decide how exactly to use all the View objects we declared a few steps ago.
- `m_TimeRemaining` variable: This `float` variable holds how much time (in seconds) is remaining to get to the goal of the current level. In the previous code, it is set to 10 for the purposes of testing, until we get to set a specific time for each level.
- `m_GameTimeTotal` variable: This variable is an SFML Time object. It keeps track of how long the game has been played for.
- `m_NewLevelRequired` Boolean variable: This variable keeps an eye on whether the player has just completed or failed a level. We can then use it to trigger loading the next level or restarting the current level.
- The `input` function: This function will handle all the player's input, which in this game is entirely from the keyboard. At first glance, it would appear that it handles all the keyboard input directly. In this game, however, we will be handling keyboard input that directly affects Thomas or Bob within the `Thomas` and `Bob` classes. This function will also handle keyboard input such as quitting, switching to split-screen, and any other keyboard input.
- The `update` function: This function will do all the work that we previously did in the update section of the `main` function. We will also call some other functions from the `update` function in order to keep the code organized. If you look back at the code, you will see that it receives a `float` parameter that will hold the fraction of a second that has passed since the previous frame. This, of course, is just what we need to update all our game objects.
- The `draw` function: This function will hold all the code that used to go in the drawing section of the `main` function in previous projects. We will, however, have some drawing code that is not kept in this function when we look at other ways to draw with SFML. We will see this new code when we learn about particle systems in *Chapter 18, Particle Systems and Shaders*.

Now, let's run through all the public functions:

- The `Engine` constructor function: As we have come to expect, this function will be called when we first declare an instance of `Engine`. It will do all the setup and initialization of the class. We will see exactly what when we code the `Engine.cpp` file shortly.
- The `run` function: This is the only public function that we need to call. It will trigger the execution of `input`, `update`, and `draw`, and will do all the work for us.

Next, we will see the definitions of all these functions and some of the variables in action.

Coding `Engine.cpp`

In all our previous classes, we have put all the function definitions into the `.cpp` file prefixed with the class name. As our aim for this project is to make the code more manageable, we are doing things a little differently.

In the `Engine.cpp` file, we will place the constructor (`Engine`) and the public `run` function. The rest of the functions will be going in their own `.cpp` file with a name that makes it clear which function goes where. This will not be a problem for the compiler if we add the appropriate include directive (`#include "Engine.h"`) at the top of all the files that contain function definitions from the `Engine` class.

Let's get started by coding `Engine` and running it in `Engine.cpp`. Right-click **Source Files** in the **Solution Explorer** and select **Add | New Item....** In the **Add New Item** window, highlight (by left-clicking) **C++ File (.cpp)** and then, in the **Name** field, type `Engine.cpp`. Finally, click the **Add** button. We are now ready to code the `.cpp` file for the `Engine` class.

Coding the `Engine` class constructor definition

The code for this function will go in the `Engine.cpp` file we have recently created.

Add the following code and then we can discuss it:

```
#include "Engine.h"

Engine::Engine()
{
    // Get the screen resolution
    // and create an SFML window and View
```



```
Vector2f resolution;
resolution.x = VideoMode::getDesktopMode().width;
resolution.y = VideoMode::getDesktopMode().height;

m_Window.create(VideoMode(resolution.x, resolution.y),
    "Thomas was late",
    Style::Fullscreen);

// Initialize the full screen view
m_MainView.setSize(resolution);
m_HudView.reset(
    FloatRect(0, 0, resolution.x, resolution.y));

// Initialize the split-screen Views
m_LeftView.setViewport(
    FloatRect(0.001f, 0.001f, 0.498f, 0.998f));

m_RightView.setViewport(
    FloatRect(0.5f, 0.001f, 0.499f, 0.998f));

m_BGLeftView.setViewport(
    FloatRect(0.001f, 0.001f, 0.498f, 0.998f));

m_BGRightView.setViewport(
    FloatRect(0.5f, 0.001f, 0.499f, 0.998f));

m_BackgroundTexture = TextureHolder::GetTexture(
    "graphics/background.png");

// Associate the sprite with the texture
m_BackgroundSprite.setTexture(m_BackgroundTexture);
}
```

We have seen much of this code before. For example, there are the usual lines of code to get the screen resolution, as well as to create a `RenderWindow`. At the end of the previous code, we use the now-familiar code to load a texture and assign it to a `Sprite`. In this case, we are loading the `background.png` texture and assigning it to `m_BackgroundSprite`.

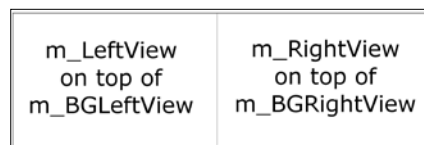
It is the code in between the four calls to the `setViewport` function that needs some explanation. The `setViewport` function assigns a portion of the screen to an SFML `View` object. It doesn't work with pixel coordinates, however. It works using a ratio. Here, "1" is the entire screen (width or height). The first two values in each call to `setViewport` are the starting position (horizontally then vertically), while the last two are the ending position.

Notice that `m_LeftView` and `m_BGLeftView` are placed in exactly the same place, that is, starting on virtually the far-left (0.001) of the screen and ending two-thousandths from the center (0.498).

`m_RightView` and `m_BGRightView` are also in exactly the same position as each other, starting just right of the previous two `View` objects (0.5) and extending to almost the far-right-hand side (0.998).

Furthermore, all the views leave a tiny slither of a gap at the top and bottom of the screen. When we draw these `View` objects on the screen, on top of a white background, it will have the effect of splitting the screen with a thin white line between the two sides of the screen, as well as a thin white border around the edges.

I have tried to represent this effect in the following diagram:



The best way to understand it is to finish this chapter, run the code, and see it in action.

Coding the run function definition

The code for this function will go in the `Engine.cpp` file we have recently created.

Add the following code immediately after the previous constructor code:

```
void Engine::run()
{
    // Timing
    Clock clock;

    while (m_Window.isOpen())
    {
        Time dt = clock.restart();
        // Update the total game time
        m_GameTimeTotal += dt;
        // Make a decimal fraction from the delta time
        float dtAsSeconds = dt.asSeconds();

        // Call each part of the game loop in turn
        input();
    }
}
```

```
        update(dtAsSeconds);
        draw();
    }
}
```

The `run` function is the center of our engine; it initiates all the other parts. First, we declare a `Clock` object. Next, we have the familiar `while(window.isOpen())` loop, which creates the game loop. Inside this while loop, we do the following:

1. Restart `clock` and save the time that the previous loop took in `dt`.
2. Keep track of the total time elapsed in `m_GameTimeTotal`.
3. Declare and initialize a `float` to represent the fraction of a second that elapsed during the previous frame.
4. Call `input`.
5. Call `update`, passing in the elapsed time (`dtAsSeconds`).
6. Call `draw`.

All of this should look very familiar. What's new is that it is wrapped in the `run` function.

Coding the input function definition

As we explained previously, the code for the `input` function will go in its own file because it is more extensive than the constructor or the `run` function. We will use `#include "Engine.h"` and prefix the function signature with `Engine::` to make sure the compiler is aware of our intentions.

Right-click **Source Files** in the **Solution Explorer** and select **Add | New Item....** In the **Add New Item** window, highlight (by left-clicking) **C++ File (.cpp)** and then, in the **Name** field, type `Input.cpp`. Finally, click the **Add** button. We are now ready to code the `input` function.

Add the following code:

```
void Engine::input()
{
    Event event;
    while (m_Window.pollEvent(event))
    {
        if (event.type == Event::KeyPressed)
        {
            // Handle the player quitting
            if (Keyboard::isKeyPressed(Keyboard::Escape))
            {

```

```

        m_Window.close();
    }

    // Handle the player starting the game
    if (Keyboard::isKeyPressed(Keyboard::Return))
    {
        m_Playing = true;
    }

    // Switch between Thomas and Bob
    if (Keyboard::isKeyPressed(Keyboard::Q))
    {
        m_Character1 = !m_Character1;
    }

    // Switch between full and split-screen
    if (Keyboard::isKeyPressed(Keyboard::E))
    {
        m_SplitScreen = !m_SplitScreen;
    }
}
}
}

```

Like the previous projects, we check the `RenderWindow` event queue each frame. Also like we've already done before, we detect specific keyboard keys using `if (Keyboard::isKeyPressed(...`. The most relevant information in the code we just added is what the keys do:

- As per usual, the *Esc* key closes the window and the game will quit.
- The *Enter* key sets `m_Playing` to true and eventually this will have the effect of starting the level.
- The *Q* key alternates the value of `m_Character1` between true and false. This key only has an effect in full screen mode. It will switch between Thomas and Bob being the center of the main `View`.
- The *E* keyboard key switches `m_SplitScreen` between true and false. This will have the effect of switching between full screen and split-screen views.

Most of this keyboard functionality will be fully working by the end of this chapter. We are getting close to being able to run our game engine. Next, let's code the update function.

Coding the update function definition

As we explained previously, the code for this function will go in its own file because it is more extensive than the constructor or the `run` function. We will use `#include "Engine.h"` and prefix the function signature with `Engine::` to make sure the compiler is aware of our intentions.

Right-click **Source Files** in the **Solution Explorer** and select **Add | New Item....** In the **Add New Item** window, highlight (by left-clicking) **C++ File (.cpp)** and then, in the **Name** field, type `Update.cpp`. Finally, click the **Add** button. We are now ready to write some code for the `update` function.

Add the following code to the `Update.cpp` file to implement the `update` function:

```
#include "Engine.h"
#include <SFML/Graphics.hpp>
#include <sstream>

using namespace sf;

void Engine::update(float dtAsSeconds)
{
    if (m_Playing)
    {
        // Count down the time the player has left
        m_TimeRemaining -= dtAsSeconds;

        // Have Thomas and Bob run out of time?
        if (m_TimeRemaining <= 0)
        {
            m_NewLevelRequired = true;
        }

        }// End if playing
    }
}
```

First, notice that the `update` function receives the time the previous frame took as a parameter. This, of course, will be essential for the `update` function to fulfill its role.

The previous code doesn't achieve anything visible at this stage. It does put in the structure that we will require for future chapters. It subtracts the time the previous frame took from `m_TimeRemaining` and checks whether time has run out. If it has, it sets `m_NewLevelRequired` to `true`. All this code is wrapped in an `if` statement that only executes when `m_Playing` is `true`. The reason for this is that, like the previous projects, we don't want time advancing and objects updating when the game has not started.

We will build on this code as the project continues.

Coding the draw function definition

As we explained previously, the code for this function will go in its own file because it is more extensive than the constructor or the `run` function. We will use `#include "Engine.h"` and prefix the function signature with `Engine::` to make sure the compiler is aware of our intentions.

Right-click **Source Files** in the **Solution Explorer** and select **Add | New Item....** In the **Add New Item** window, highlight (by left-clicking) **C++ File (.cpp)** and then, in the **Name** field, type `Draw.cpp`. Finally click the **Add** button. We are now ready to add some code to the `draw` function.

Add the following code to the `Draw.cpp` file to implement the `draw` function:

```
#include "Engine.h"

void Engine::draw()
{
    // Rub out the last frame
    m_Window.clear(Color::White);

    if (!m_SplitScreen)
    {
        // Switch to background view
        m_Window.setView(m_BGMainView);
        // Draw the background
        m_Window.draw(m_BackgroundSprite);
        // Switch to m_MainView
        m_Window.setView(m_MainView);
    }
    else
    {
```

```
        // Split-screen view is active

        // First draw Thomas' side of the screen

        // Switch to background view
        m_Window.setView(m_BGLeftView);
        // Draw the background
        m_Window.draw(m_BackgroundSprite);
        // Switch to m_LeftView
        m_Window.setView(m_LeftView);

        // Now draw Bob's side of the screen

        // Switch to background view
        m_Window.setView(m_BGRightView);
        // Draw the background
        m_Window.draw(m_BackgroundSprite);
        // Switch to m_RightView
        m_Window.setView(m_RightView);

    }

    // Draw the HUD
    // Switch to m_HudView
    m_Window.setView(m_HudView);

    // Show everything we have just drawn
    m_Window.display();
}
```

In the previous code, there is nothing we haven't seen before. The code starts, as usual, by clearing the screen. In this project, we clear the screen with White. What's new is the way the different drawing options are separated by a condition that checks whether the screen is currently split or not:

```
if (!m_SplitScreen)
{
}
else
{
}
```

If the screen is not split, we draw the background sprite in the background View (`m_BGView`) and then switch to the main full screen View (`m_MainView`). Note that, currently, we don't do any drawing in `m_MainView`.

If, on the other hand, the screen is split, the code in the `else` block is executed and we draw `m_BGLeftView` with the background sprite on the left of the screen, followed by switching to `m_LeftView`.

Then, still in the `else` block, we draw `m_BGRightView` with the background sprite on the right of the screen, followed by switching to `m_RightView`.

Outside of the `if else` structure we just described, we switch to the `m_HUDView`. At this stage, we are not actually drawing anything in `m_HUDView`.

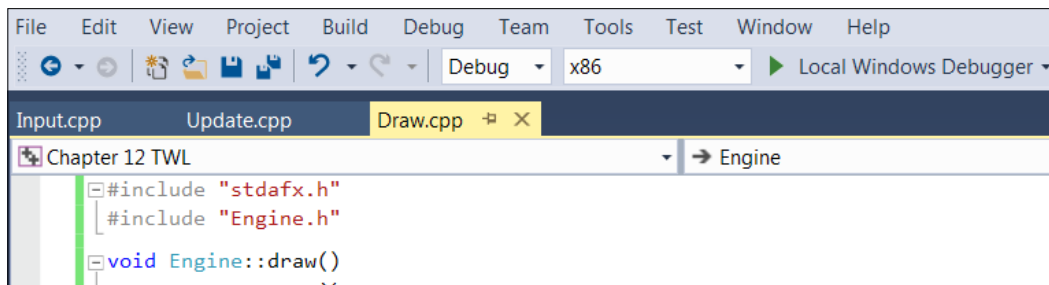
Like the other two (`input`, `update`) of the three most significant functions, we will go back to the `draw` function often. We will add new elements for our game that need to be drawn. You will notice that, each time we do, we will add code into each of the main, left-hand, and right-hand sections.

Let's quickly recap on the `Engine` class and then we can fire it up.

The Engine class so far

What we have done is abstract all the code that used to be in the `main` function into the `input`, `update`, and `draw` functions. The continuous looping of these functions, as well as the timing, is handled by the `run` function.

Consider leaving the **Input.cpp**, **Update.cpp**, and **Draw.cpp** tabs open in Visual Studio, perhaps organized in order, as shown in the following screenshot:



We will revisit each of these functions throughout the course of the project and add more code. Now that we have the basic structure and functionality of the `Engine` class, we can create an instance of it in the `main` function and see it in action.

Coding the main function

Let's rename the `TFL.cpp` file that was autogenerated when the project was created to `Main.cpp`. Right-click the `TFL` file in the **Solution Explorer** and select **Rename**. Change the name to `Main.cpp`. This will be the file that contains our main function and the code that instantiates the `Engine` class.

Add the following code to `Main.cpp`:

```
#include "Engine.h"

int main()
{
    // Declare an instance of Engine
    Engine engine;

    // Start the engine VRRrrrrmmm
    engine.run();

    // Quit in the usual way when the engine is stopped
    return 0;
}
```

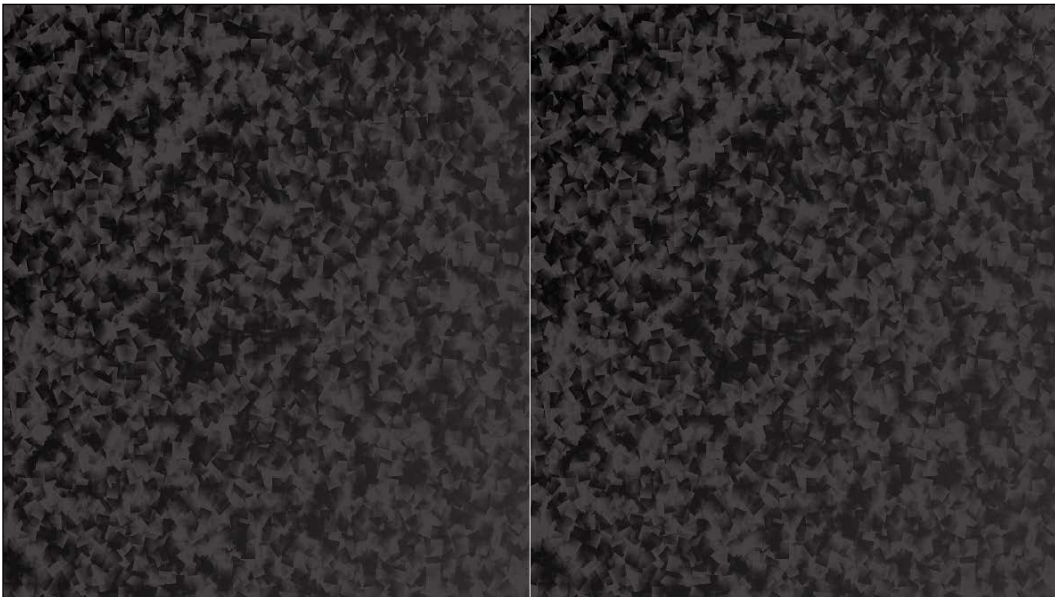
All we do is add an `include` directive for the `Engine` class, declare an instance of `Engine`, and then call its `run` function. Everything will be handled by the `Engine` class until the player quits and the execution returns to `main` and the `return 0` statement.

That was easy. Now, we can run the game and see the empty background, either in full screen or split-screen, which will eventually contain all the action.

Here is the game so far in full screen mode, showing just the background:



Now, tap the *E* key. You will be able to see the screen neatly partitioned into two halves, ready for split-screen coop gameplay:



Summary

In this chapter, we introduced the Thomas Was Late game and laid the foundations of understanding as well as the code structure for the rest of the project. It is certainly true that there are a lot of files in the Solution Explorer but, provided we understand the purpose of each, we will find the implementation of the rest of the project quite easy.

In the next chapter, we will learn about two more fundamental C++ topics, inheritance and polymorphism. We will also begin to put them to use by building three classes to represent two playable characters.

FAQ

Here is a question that might be on your mind:

Q) I don't fully understand the structure of the code files. What should I do?

A) It is true that abstraction can make the structure of our code less clear, but the actual code itself becomes so much easier. Instead of cramming everything into the `main` function like we did in the previous projects, we will split the code up into `Input.cpp`, `Update.cpp`, and `Draw.cpp`. Furthermore, we will use more classes to group together related code as we proceed. Study the *Structuring the Thomas Was Late code* section again, especially the diagrams.

15

Advanced OOP – Inheritance and Polymorphism

In this chapter, we will further extend our knowledge of OOP by looking at the slightly more advanced concepts of **inheritance** and **polymorphism**. We will then be able to use this new knowledge to implement the star characters of our game, Thomas and Bob. Here is what we will cover in this chapter:

- Learn how to extend and modify a class using inheritance
- Treat an object of a class as if it is more than one type of class by using polymorphism
- Learn about abstract classes and how designing classes that are never instantiated can actually be useful
- Build an abstract `PlayableCharacter` class
- Put inheritance to work with the `Thomas` and `Bob` classes
- Add Thomas and Bob to the game project

Inheritance

We have already seen how we can use other people's hard work by instantiating objects from the classes of the SFML library. But this whole OOP thing goes even further than that.

What if there is a class that has loads of useful functionality in it, but is not quite what we want? In this situation, we can **inherit** from the other class. Just like it sounds, **inheritance** means we can harness all the features and benefits of other people's classes, including the encapsulation, while further refining or extending the code specifically to our situation. In this project, we will inherit from and extend some SFML classes; we will also do so with our own classes.

Let's look at some code that uses inheritance.

Extending a class

With all this in mind, let's look at an example class and see how we can extend it, just to see the syntax and as a first step.

First, we define a class to inherit from. This is no different from how we created any of our other classes. Take a look at this hypothetical `Soldier` class declaration:

```
class Soldier
{
    private:
        // How much damage can the soldier take
        int m_Health;
        int m_Armour;
        int m_Range;
        int m_ShotPower;

    Public:
        void setHealth(int h);
        void setArmour(int a);
        void setRange(int r);
        void setShotPower(int p);
};
```

In the previous code, we define a `Soldier` class. It has four private variables: `m_Health`, `m_Armour`, `m_Range`, and `m_ShotPower`. It has also four public functions: `setHealth`, `setArmour`, `setRange`, and `setShotPower`. We don't need to see the definitions of these functions; they will simply initialize the appropriate variable that their name makes obvious.

We can also imagine that a fully implemented `Soldier` class would be much more in-depth than this. It would probably have functions such as `shoot`, `goProne`, and so on. If we implemented a `Soldier` class in an SFML project, it would likely have a `Sprite` object, as well as an `update` and a `getPostion` function.

The simple scenario that we've presented here is suitable if we wish to learn about inheritance. Now, let's look at something new: inheriting from the `Soldier` class. Look at the following code, especially the highlighted part:

```
class Sniper : public Soldier
{
    public:
```

```
// A constructor specific to Sniper
Sniper::Sniper();

};
```

By adding `: public Soldier` to the `Sniper` class declaration, `Sniper` inherits from `Soldier`. But what does this mean, exactly? `Sniper` **is a** `Soldier`. It has all the variables and functions of `Soldier`. Inheritance is even more than this, however.

Also note that, in the previous code, we declare a `Sniper` constructor. This constructor is unique to `Sniper`. We have not only inherited from `Soldier`; we have **extended** `Soldier`. All the functionality (definitions) of the `Soldier` class would be handled by the `Soldier` class, but the definition of the `Sniper` constructor must be handled by the `Sniper` class.

Here is what the hypothetical `Sniper` constructor definition might look like:

```
// In Sniper.cpp
Sniper::Sniper()
{
    setHealth(10);
    setArmour(10);
    setRange(1000);
    setShotPower(100);
}
```

We could go ahead and write a bunch of other classes that are extensions of the `Soldier` class, perhaps `Commando` and `Infantryman`. Each would have the exact same variables and functions, but each could also have a unique constructor that initializes those variables appropriate to the specific type of `Soldier`. `Commando` might have very high `m_Health` and `m_ShotPower` but really puny `m_Range`. `Infantryman` might be in between `Commando` and `Sniper` with mediocre values for each variable.

As if OOP wasn't useful enough already, we can now model real-world objects, including their hierarchies. We can achieve this by sub-classing/extending/inheriting from other classes.

The terminology we might like to learn here is that the class that is extended from is the **super-class**, and the class that inherits from the super-class is the **sub-class**. We can also say **parent** and **child** class.



You might find yourself asking this question about inheritance: why? The reason is something like this: we can write common code once; in the parent class, we can update that common code and all the classes that inherit from it are also updated. Furthermore, a sub-class only gets to use public and **protected** instance variables and functions. So, designed properly, this also enhances the goals of encapsulation.

Did you say protected? Yes. There is an access specifier for class variables and functions called **protected**. You can think of protected variables as being somewhere between public and private. Here is a quick summary of access specifiers, along with more details about the **protected** specifier:

- Public variables and functions can be accessed and used by anyone with an instance of the class.
- Private variables and functions can only accessed/used by the internal code of the class, and not directly from an instance. This is good for encapsulation and when we need to access/change private variables, since we can provide public getter and setter functions (such as `getSprite`). If we extend a class that has private variables and functions, that child class **cannot** directly access the private data of its parent.
- Protected variables and functions are almost the same as private. They cannot be accessed/used directly by an instance of the class. However, they **can** be used directly by any class that extends the class they are declared in. So, it is like they are private, except to child classes.

To fully understand what protected variables and functions are and how they can be useful, let's look at another topic first. Then, we will see them in action.

Polymorphism

Polymorphism allows us to write code that is less dependent on the types we are trying to manipulate. This can make our code clearer and more efficient. Polymorphism means many forms. If the objects that we code can be more than one type of thing, then we can take advantage of this.



But what does polymorphism mean to us? Boiled down to its simplest definition, polymorphism means the following: any sub-class can be used as part of the code that uses the super-class. This means we can write code that is simpler and easier to understand and also easier to modify or change. Also, we can write code for the super-class and rely on the fact that no matter how many times it is sub-classed, within certain parameters, the code will still work.

Let's discuss an example.

Suppose we want to use polymorphism to help write a zoo management game where we must feed and tend to the needs of animals. We will probably want to have a function such as `feed`. We will also probably want to pass an instance of the animal to be fed into the `feed` function.

A zoo, of course, has lots of animals, such as `Lion`, `Elephant`, and `Three-toed Sloth`. With our new knowledge of C++ inheritance, it makes sense to code an `Animal` class and have all the different types of animal inherit from it.

If we want to write a function (`feed`) that we can pass `Lion`, `Elephant`, and `ThreeToedSloth` into as a parameter, it might seem like we need to write a `feed` function for each type of `Animal`. However, we can write polymorphic functions with polymorphic return types and arguments. Take a look at the following definition of the hypothetical `feed` function:

```
void feed(Animal& a)
{
    a.decreaseHunger();
}
```

The preceding function has an `Animal` reference as a parameter, meaning that any object that is built from a class that extends `Animal` can be passed into it.

This means you can write code today and make another subclass in a week, month, or year, and the very same functions and data structures will still work. Also, we can enforce a set of rules upon our subclasses regarding what they can and cannot do, as well as how they do it. So, good design in one stage can influence it at other stages.

But will we ever really want to instantiate an actual `Animal`?

Abstract classes – virtual and pure virtual functions

An **abstract class** is a class that cannot be instantiated and therefore cannot be made into an object.



Some terminology we might like to learn about here is *concrete* class. A **concrete class** is any class that isn't abstract. In other words, all the classes we have written so far have been concrete classes and can be instantiated into usable objects.

So, it's code that will never be used, then? But that's like paying an architect to design your home and then never building it!

If we, or the designer of a class, wants to force its users to inherit it before using their class, they can make a class **abstract**. If this happens, we cannot make an object from it; therefore, we must inherit from it first and make an object from the sub-class.

To do so, we can make a function **pure virtual** and not provide any definition. Then, that function must be **overridden (rewritten)** in any class that inherits from it.

Let's look at an example; it will help. We can make a class abstract by adding a pure virtual function such as the abstract `Animal` class, which can only perform the generic action of `makeNoise`:

```
Class Animal
private:
    // Private stuff here

public:

    void virtual makeNoise() = 0;

    // More public stuff here
};
```

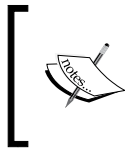
As you can see, we add the C++ keyword `virtual`, before, and `= 0` after the function declaration. Now, any class that extends/inherits from `Animal` must override the `makeNoise` function. This might make sense since different types of animal make very different types of noise. We could have assumed that anybody who extends the `Animal` class is smart enough to notice that the `Animal` class cannot make a noise and that they will need to handle it, but what if they don't notice? The point is that by making a pure virtual function, we guarantee that they will, because they must.

Abstract classes are also useful because, sometimes, we want a class that can be used as a polymorphic type, but we need to guarantee it can never be used as an object. For example, `Animal` doesn't really make sense on its own. We don't talk about animals; we talk about types of animals. We don't say, "Ooh, look at that lovely, fluffy, white animal!", or, "Yesterday we went to the pet shop and got an animal and an animal bed." It's just too, well, abstract.

So, an abstract class is kind of like a **template** to be used by any class that extends it (inherits from it). If we were building an *Industrial Empire* type game where the player manages businesses and their employees, we might want a `Worker` class, for example, and extend it to make `Miner`, `Steelworker`, `OfficeWorker`, and, of course, `Programmer`. But what exactly does a plain `Worker` do? Why would we ever want to instantiate one?

The answer is we wouldn't want to instantiate one, but we might want to use it as a polymorphic type so that we can pass multiple `Worker` sub-classes between functions and have data structures that can hold all types of workers.

All pure virtual functions must be overridden by any class that extends the parent class that contains the pure virtual function. This means that the abstract class can provide some of the common functionality that would be available in all its subclasses. For example, the `Worker` class might have the `m_AnnualSalary`, `m_Productivity`, and `m_Age` member variables. It might also have the `getPayCheck` function, which is not pure virtual and is the same in all the sub-classes, but a `doWork` function, which is pure virtual and must be overridden, because all the different types of `Worker` will `doWork` very differently.



By the way, **virtual** as opposed to pure virtual is a function that can be **optionally overridden**. You declare a virtual function the same way as a pure virtual function but leave the `= 0` off to the end. In the current game project, we will use a pure virtual function.

If any of this virtual, pure virtual, or abstract stuff is unclear, using it is probably the best way to understand it.

Building the PlayableCharacter class

Now that we know the basics of inheritance, polymorphism, and pure virtual functions, we will put them to use. We will build a `PlayableCharacter` class that has most of the functionality that any character from our game is going to need. It will have one pure virtual function, known as `handleInput`. The `handleInput` function will need to be quite different in the sub-classes, so this makes sense.

As `PlayableCharacter` will have a pure virtual function, it will be an abstract class and no objects of it will be possible. We will then build the `Thomas` and `Bob` classes, which will inherit from `PlayableCharacter`, implement the definition of the pure virtual function, and allow us to instantiate `Bob` and `Thomas` objects in our game. It will not be possible to instantiate a `PlayableCharacter` instance directly, but we wouldn't want to because it is too abstract anyway.

Coding PlayableCharacter.h

As usual when creating a class, we will start off with the header file that will contain the member variable and function declarations. What is new is that, in this class, we will declare some **protected** member variables. Remember that protected variables can be used as if they were public in classes that inherit from the class with the protected variables.

Right-click **Header Files** in the **Solution Explorer** and select **Add | New Item...** In the **Add New Item** window, highlight (by left-clicking) **Header File (.h)** and then, in the **Name** field, type `PlayableCharacter.h`. Finally, click the **Add** button. We are now ready to code the header file for the `PlayableCharacter` class.

We will add and discuss the contents of the `PlayableCharacter.h` file in three sections. First will be the protected section, followed by private, and then public.

Add the following code to the `PlayableCharacter.h` file:

```
#pragma once
#include <SFML/Graphics.hpp>

using namespace sf;

class PlayableCharacter
{
protected:
    // Of course we will need a sprite
    Sprite m_Sprite;

    // How long does a jump last
    float m_JumpDuration;

    // Is character currently jumping or falling
    bool m_IsJumping;
    bool m_IsFalling;


    // Which directions is the character currently moving in
    bool m_LeftPressed;
    bool m_RightPressed;

    // How long has this jump lasted so far
    float m_TimeThisJump;

    // Has the player just initiated a jump
    bool m_JustJumped = false;

    // Private variables and functions come next
```

The first thing to notice in the code we just wrote is that all the variables are protected. This means that when we inherit from the class, all the variables we just wrote will be accessible to those classes that extend it. We will extend this class with the Thomas and Bob classes.

 The terms *inherit from* and *extend* are virtually synonymous in most contexts in this book. Sometimes, one seems more appropriate than the other, however.

Apart from the protected access specification, there is nothing new or complicated about the previous code. It is worth paying attention to some of the details, however. If we do, it will be easy to understand how the class works as we progress. So, let's run through those protected variables, one at a time.

We have our somewhat predictable `Sprite`, `m_Sprite`. We have a float variable called `m_JumpDuration`, which will hold a value representing the time that the character is able to jump for. The greater the value, the further/higher the character will be able to jump.

Next, we have a Boolean `m_IsJumping`, which is `true` when the character is jumping and `false` otherwise. This will be useful for making sure that the character can't jump while in mid-air.

The `m_IsFalling` variable has a similar use to `m_IsJumping`. It will be used to know when a character is falling.

Next, we have two Booleans that will be `true` if the character's left or right keyboard buttons are currently being pressed. These are relatively dependent upon the character (*A* and *D* for Thomas, and the *Left* and *Right* arrow keys for Bob, respectively). How we respond to these Booleans will be seen in the Thomas and Bob classes.

The `m_TimeThisJump` float variable is updated each frame that `m_IsJumping` is `true`. We can then find out when `m_JumpDuration` has been reached.

The final protected variable is the `m_JustJumped` Boolean. This will be `true` if a jump was initiated in the current frame. It will be used so that we know when to play a jump sound effect.

Next, add the following private variables to the `PlayableCharacter.h` file:

```
private:
    // What is the gravity
    float m_Gravity;

    // How fast is the character
```

```
float m_Speed = 400;

// Where is the player
Vector2f m_Position;

// Where are the characters various body parts?
FloatRect m_Feet;
FloatRect m_Head;
FloatRect m_Right;
FloatRect m_Left;

// And a texture
Texture m_Texture;

// All our public functions will come next
```

In the previous code, we have some interesting private variables. Remember that these variables will only be directly accessible to the code in the `PlayableCharacter` class. The `Thomas` and `Bob` classes will not be able to access them directly.

The `m_Gravity` variable will hold the number of pixels per second that the character will fall. The `m_Speed` variable will hold the number of pixels per second that the character can move left or right.

The `Vector2f, m_Position` variable is the position in the world (not the screen) where the center of the character is.

The next four `FloatRect` objects are important to discuss. When we did collision detection in the *Zombie Arena* game, we simply checked to see if two `FloatRect` objects intersected. Each `FloatRect` object represented an entire character, a pickup, or a bullet. For the non-rectangular shaped objects (zombies and the player), this was a little bit inaccurate.

In this game, we will need to be more precise. The `m_Feet, m_Head, m_Right, m_Left`, and `FloatRect` objects will hold the coordinates of the different parts of a character's body. These coordinates will be updated each frame.

Through these coordinates, we will be able to tell exactly when a character lands on a platform, bumps their head during a jump, or rubs shoulders with a tile to their side.

Lastly, we have a Texture. The Texture is private as it is not used directly by the Thomas or Bob classes. However, as we saw, the Sprite is protected because it is used directly.

Now, add all the public functions to the PlayableCharacter.h file. Then, we will discuss them:

```
public:
    void spawn(Vector2f startPosition, float gravity);

    // This is a pure virtual function
    bool virtual handleInput() = 0;
    // This class is now abstract and cannot be instantiated

    // Where is the player
    FloatRect getPosition();

    // A rectangle representing the position
    // of different parts of the sprite
    FloatRect getFeet();
    FloatRect getHead();
    FloatRect getRight();
    FloatRect getLeft();

    // Send a copy of the sprite to main
    Sprite getSprite();

    // Make the character stand firm
    void stopFalling(float position);
    void stopRight(float position);
    void stopLeft(float position);
    void stopJump();

    // Where is the center of the character
    Vector2f getCenter();

    // We will call this function once every frame
    void update(float elapsedTime);

}; // End of the class
```

Let's talk about each of the function declarations that we just added. This will make coding their definitions easier to follow:

- The `spawn` function receives a `Vector2f` called `startPosition` and a float value called `gravity`. As the names suggest, `startPosition` will be the coordinates in the level that the character will start, and `gravity` will be the number of pixels per second at which the character will fall.
- `bool virtual handleInput() = 0` is, of course our pure virtual function. Since `PlayableCharacter` has this function, any class that extends it, if we want to instantiate it, must provide a definition for this function. Therefore, when we write all the function definitions for `PlayableCharacter` in a minute, we will not provide a definition for `handleInput`. There will need to be definitions in both the `Thomas` and `Bob` classes.
- The `getPosition` function returns a `FloatRect` object that represents the position of the whole character.
- The `getFeet()` function, as well as `getHead`, `getRight`, and `getLeft`, return a `FloatRect` object that represents the location of a specific part of the character's body. This is just what we need for detailed collision detection.
- The `getSprite` function, as usual, returns a copy of `m_Sprite` to the calling code.
- The `stopFalling`, `stopRight`, `stopLeft`, and `stopJump` functions receive a single float value that the function will use to reposition the character and stop it walking or jumping through a solid tile.
- The `getCenter` function returns a `Vector2f` object to the calling code to let it know exactly where the center of the character is. This value is held in `m_Position`. As we will see later, it is used by the `Engine` class to center the appropriate `View` around the appropriate character.
- We have seen the `update` function many times before and, as usual, it takes a float parameter, which is the fraction of a second the current frame has taken. This update function will need to do more work than previous update functions (from our other projects), however. It will need to handle jumping as well as updating the `FloatRect` objects that represent the head, feet, and left- and right-hand sides of the character.

Now, we can write the definitions for all the functions, except, of course, `handleInput`.

Coding PlayableCharacter.cpp

Right-click **Source Files** in the **Solution Explorer** and select **Add | New Item....** In the **Add New Item** window, highlight (by left-clicking) **C++ File (.cpp)** and then, in the **Name** field, type `PlayableCharacter.cpp`. Finally, click the **Add** button. We are now ready to code the `.cpp` file for the `PlayableCharacter` class.

We will break up the code and discuss it in several chunks. First, add the include directives and the definition of the `spawn` function:

```
#include "PlayableCharacter.h"

void PlayableCharacter::spawn(
    Vector2f startPosition, float gravity)
{
    // Place the player at the starting point
    m_Position.x = startPosition.x;
    m_Position.y = startPosition.y;

    // Initialize the gravity
    m_Gravity = gravity;

    // Move the sprite in to position
    m_Sprite.setPosition(m_Position);
}
```

The `spawn` function initializes `m_Position` with the passed-in position, and also initializes `m_Gravity`. The final line of code moves `m_Sprite` to its starting position.

Next, add the definition for the `update` function immediately after the previous code:

```
void PlayableCharacter::update(float elapsedTime)
{
    if (m_RightPressed)
    {
        m_Position.x += m_Speed * elapsedTime;
    }
}
```



```
    if (m_LeftPressed)
    {
        m_Position.x -= m_Speed * elapsedTime;
    }

    // Handle Jumping
    if (m_IsJumping)
    {
        // Update how long the jump has been going
        m_TimeThisJump += elapsedTime;

        // Is the jump going upwards
        if (m_TimeThisJump < m_JumpDuration)
        {
            // Move up at twice gravity
            m_Position.y -= m_Gravity * 2 * elapsedTime;
        }
        else
        {
            m_IsJumping = false;
            m_IsFalling = true;
        }
    }

    // Apply gravity
    if (m_IsFalling)
    {
        m_Position.y += m_Gravity * elapsedTime;
    }

    // Update the rect for all body parts
    FloatRect r = getPosition();

    // Feet
    m_Feet.left = r.left + 3;
    m_Feet.top = r.top + r.height - 1;
    m_Feet.width = r.width - 6;
    m_Feet.height = 1;
```

```

    // Head
    m_Head.left = r.left;
    m_Head.top = r.top + (r.height * .3);
    m_Head.width = r.width;
    m_Head.height = 1;

    // Right
    m_Right.left = r.left + r.width - 2;
    m_Right.top = r.top + r.height * .35;
    m_Right.width = 1;
    m_Right.height = r.height * .3;

    // Left
    m_Left.left = r.left;
    m_Left.top = r.top + r.height * .5;
    m_Left.width = 1;
    m_Left.height = r.height * .3;

    // Move the sprite into position
    m_Sprite.setPosition(m_Position);
}

```

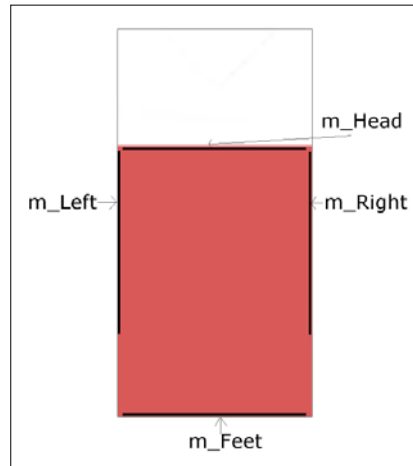
The first two parts of the code check whether `m_RightPressed` or `m_LeftPressed` is true. If either of them is, then `m_Position` is changed using the same formula as the previous project (elapsed time multiplied by speed).

Next, we see whether the character is currently executing a jump. We know this from `if (m_IsJumping)`. If this `if` statement is true, these are the steps the code takes:

1. Updates `m_TimeThisJump` with `elapsedTime`.
2. Checks if `m_TimeThisJump` is still less than `m_JumpDuration`. If it is, it changes the y coordinate of `m_Position` by 2x gravity, multiplied by the elapsed time.
3. In the else clause that executes when `m_TimeThisJump` is not lower than `m_JumpDuration`, `m_Falling` is set to true. The effect of doing this will be seen next. Also, `m_Jumping` is set to false. This prevents the code we have just been discussing from executing, because `if (m_IsJumping)` is now false.

The `if (m_IsFalling)` block moves `m_Position` down each frame. It is moved using the current value of `m_Gravity` and the elapsed time.

The code that follows (most of the remaining code) updates the "body parts" of the character, relative to the current position of the sprite as a whole. Take a look at the following diagram to see how the code calculates the position of the virtual head, feet, and left- and right-hand sides of the character:



The final line of code uses the `setPosition` function to move the sprite to its correct location after all the possibilities of the update function.

Now, add the definitions for the `getPosition`, `getCenter`, `getFeet`, `getHead`, `getLeft`, `getRight`, and `getSprite` functions, immediately after the previous code:

```
FloatRect PlayableCharacter::getPosition()
{
    return m_Sprite.getGlobalBounds();
}

Vector2f PlayableCharacter::getCenter()
{
    return Vector2f(
        m_Position.x + m_Sprite.getGlobalBounds().width / 2,
        m_Position.y + m_Sprite.getGlobalBounds().height / 2
    );
}

FloatRect PlayableCharacter::getFeet()
{
    return m_Feet;
}
```

```
FloatRect PlayableCharacter::getHead()
{
    return m_Head;
}

FloatRect PlayableCharacter::getLeft()
{
    return m_Left;
}

FloatRect PlayableCharacter::getRight()
{
    return m_Right;
}

Sprite PlayableCharacter::getSprite()
{
    return m_Sprite;
}
```

The `getPosition` function returns a `FloatRect` that wraps the entire sprite, while `getCenter` returns a `Vector2f` that contains the center of the sprite. Notice that we divide the height and width of the sprite by 2 in order to dynamically arrive at this result. This is because Thomas and Bob will be of different heights.

The `getFeet`, `getHead`, `getLeft`, and `getRight` functions return the `FloatRect` objects that represent the body parts of the character that we update each frame in the update function. We will write the collision detection code that uses these functions in the next chapter.

The `getSprite` function, as usual, returns a copy of `m_Sprite`.

Finally, for the `PlayableCharacter` class, add the definitions for the `stopFalling`, `stopRight`, `stopLeft`, and `stopJump` functions. Do so immediately after the previous code:

```
void PlayableCharacter::stopFalling(float position)
{
    m_Position.y = position - getPosition().height;
    m_Sprite.setPosition(m_Position);
    m_IsFalling = false;
}
```

```
void PlayableCharacter::stopRight(float position)
{
    m_Position.x = position - m_Sprite.getGlobalBounds().width;
    m_Sprite.setPosition(m_Position);
}

void PlayableCharacter::stopLeft(float position)
{
    m_Position.x = position + m_Sprite.getGlobalBounds().width;
    m_Sprite.setPosition(m_Position);
}

void PlayableCharacter::stopJump()
{
    // Stop a jump early
    m_IsJumping = false;
    m_IsFalling = true;
}
```

Each of the previous functions receives a value as a parameter that is used to reposition either the top, bottom, left, or right of the sprite. Exactly what these values are and how they are obtained will be something for the next chapter. Each of the previous functions also repositions the sprite.

The final function is the `stopJump` function, which will also be used in collision detection. It sets the necessary values for `m_IsJumping` and `m_IsFalling` to end a jump.

Building the Thomas and Bob classes

Now, we get to use inheritance for real. We will build a class for Thomas and a class for Bob. They will both inherit from the `PlayableCharacter` class we just coded. They will then have all the functionality of the `PlayableCharacter` class, including direct access to its protected variables. We will also add the definition for the pure virtual function, `handleInput`. You will notice that the `handleInput` functions for Thomas and Bob will be different.

Coding Thomas.h

Right-click **Header Files** in the **Solution Explorer** and select **Add | New Item....** In the **Add New Item** window, highlight (by left-clicking) **Header File (.h)** and then, in the **Name** field, type `Thomas.h`. Finally, click the **Add** button. We are now ready to code the header file for the Thomas class.

Add the following code to the `Thomas.h` class:

```
#pragma once
#include "PlayableCharacter.h"

class Thomas : public PlayableCharacter
{
public:
    // A constructor specific to Thomas
    Thomas::Thomas();

    // The overridden input handler for Thomas
    bool virtual handleInput();

};
```

The previous code is very short and sweet. We can see that we have a constructor and that we are going to implement the pure virtual `handleInput` function. So, let's do that now.

Coding Thomas.cpp

Right-click **Source Files** in the **Solution Explorer** and select **Add | New Item....** In the **Add New Item** window, highlight (by left-clicking) **C++ File (.cpp)** and then, in the **Name** field, type `Thomas.cpp`. Finally, click the **Add** button. We are now ready to code the `.cpp` file for the Thomas class.

Add the Thomas constructor to the `Thomas.cpp` file, as follows:

```
#include "Thomas.h"
#include "TextureHolder.h"

Thomas::Thomas()
{
    // Associate a texture with the sprite
    m_Sprite = Sprite(TextureHolder::GetTexture(
```

```
        "graphics/thomas.png")) ;

    m_JumpDuration = .45;
}
```

All we need to do is load the `thomas.png` graphic and set the duration of a jump (`m_JumpDuration`) to `.45` (nearly half a second).

Add the definition of the `handleInput` function as follows:

```
// A virtual function
bool Thomas::handleInput()
{
    m_JustJumped = false;

    if (Keyboard::isKeyPressed(Keyboard::W))
    {
        // Start a jump if not already jumping
        // but only if standing on a block (not falling)
        if (!m_IsJumping && !m_IsFalling)
        {
            m_IsJumping = true;
            m_TimeThisJump = 0;
            m_JustJumped = true;
        }
    }
    else
    {
        m_IsJumping = false;
        m_IsFalling = true;
    }

    if (Keyboard::isKeyPressed(Keyboard::A))
    {
        m_LeftPressed = true;
    }
    else
    {
        m_LeftPressed = false;
    }

    if (Keyboard::isKeyPressed(Keyboard::D))
    {

```

```

        m_RightPressed = true;
    }
    else
    {
        m_RightPressed = false;
    }

    return m_JustJumped;
}

```

This code should look quite familiar to you. We are using the SFML `isKeyPressed` function to see whether any of the *W*, *A*, or *D* keys are being pressed.

When *W* is pressed, the player is attempting to jump. The code then uses the `if(!m_IsJumping && !m_IsFalling)` code to check that the character is not already jumping and that it is not falling either. When these tests are both true, `m_IsJumping` is set to `true`, `m_TimeThisJump` is set to 0, and `m_JustJumped` is set to `true`.

When the previous two tests don't evaluate to `true`, the `else` clause is executed and `m_Jumping` is set to `false`, and `m_IsFalling` is set to `true`.

Handling how the *A* and *D* keys are being pressed is as simple as setting `m_LeftPressed` and/or `m_RightPressed` to `true` or `false`. The update function will now be able to handle moving the character.

The last line of code in the function returns the value of `m_JustJumped`. This will let the calling code know if it needs to play a jumping sound effect.

We will now code the `Bob` class. It is nearly identical to the `Thomas` class, except it has different jumping abilities and a different `Texture`, and uses different keys on the keyboard.

Coding Bob.h

The `Bob` class is identical in structure to the `Thomas` class. It inherits from `PlayableCharacter`, it has a constructor, and it provides the definition of the `handleInput` function. The difference compared to `Thomas` is that we initialize some of `Bob`'s member variables differently and we handle input (in the `handleInput` function) differently as well. Let's code the class and look at the details.

Right-click **Header Files** in the **Solution Explorer** and select **Add | New Item....** In the **Add New Item** window, highlight (by left-clicking) **Header File (.h)** and then, in the **Name** field, type `Bob.h`. Finally, click the **Add** button. We are now ready to code the header file for the `Bob` class.

Add the following code to the `Bob.h` file:

```
#pragma once
#include "PlayableCharacter.h"

class Bob : public PlayableCharacter
{
public:
    // A constructor specific to Bob
    Bob::Bob();

    // The overridden input handler for Bob
    bool virtual handleInput();

};
```

The previous code is identical to the `Thomas.h` file apart from the class name and therefore the constructor name.

Coding Bob.cpp

Right-click **Source Files** in the **Solution Explorer** and select **Add | New Item....** In the **Add New Item** window, highlight (by left-clicking) **C++ File (.cpp)** and then, in the **Name** field, type `Bob.cpp`. Finally, click the **Add** button. We are now ready to code the `.cpp` file for the `Bob` class.

Add the following code for the `Bob` constructor to the `Bob.cpp` file. Notice that the texture is different (`bob.png`) and that `m_JumpDuration` is initialized to a significantly smaller value. `Bob` is now his own unique self:

```
#include "Bob.h"
#include "TextureHolder.h"

Bob::Bob()
{
    // Associate a texture with the sprite
    m_Sprite = Sprite(TextureHolder::GetTexture(
        "graphics/bob.png"));

    m_JumpDuration = .25;
}
```

Add the `handleInput` code immediately after the `Bob` constructor:

```
bool Bob::handleInput()
{
    m_JustJumped = false;

    if (Keyboard::isKeyPressed(Keyboard::Up))
    {
        // Start a jump if not already jumping
        // but only if standing on a block (not falling)
        if (!m_IsJumping && !m_IsFalling)
        {
            m_IsJumping = true;
            m_TimeThisJump = 0;
            m_JustJumped = true;
        }
    }
    else
    {
        m_IsJumping = false;
        m_IsFalling = true;
    }

    if (Keyboard::isKeyPressed(Keyboard::Left))
    {
        m_LeftPressed = true;
    }
    else
    {
        m_LeftPressed = false;
    }

    if (Keyboard::isKeyPressed(Keyboard::Right))
    {
        m_RightPressed = true;;
    }
    else
    {
```

```
        m_RightPressed = false;
    }

    return m_JustJumped;
}
```

Notice that the code is nearly identical to the code in the `handleInput` function of the `Thomas` class. The only difference is that we respond to different keys (the *Left* arrow key and *Right* arrow key for left and right movement, respectively, and the *Up* arrow key for jumping).

Now that we have a `PlayableCharacter` class that has been extended by the `Bob` and `Thomas` classes, we can add a `Bob` and a `Thomas` instance to the game.

Updating the game engine to use Thomas and Bob

In order to be able to run the game and see our new characters, we have to declare instances of them, call their `spawn` functions, update them each frame, and draw them each frame. Let's do that now.

Updating Engine.h to add an instance of Bob and Thomas

Open up the `Engine.h` file and add the following highlighted lines of code:

```
#pragma once
#include <SFML/Graphics.hpp>
#include "TextureHolder.h"
#include "Thomas.h"
#include "Bob.h"

using namespace sf;

class Engine
{
private:
    // The texture holder
    TextureHolder th;

    // Thomas and his friend, Bob
    Thomas m_Thomas;
```

```

    Bob m_Bob;

    const int TILE_SIZE = 50;
    const int VERTS_IN_QUAD = 4;
    ...
    ...

```

Now, we have an instance of both Thomas and Bob that are derived from `PlayableCharacter`.

Updating the input function to control Thomas and Bob

Now, we will add the ability to control the two characters. This code will go in the input part of the code. Of course, for this project, we have a dedicated input function. Open `Input.cpp` and add the following highlighted code:

```

void Engine::input()
{
    Event event;
    while (m_Window.pollEvent(event))
    {
        if (event.type == Event::KeyPressed)
        {
            // Handle the player quitting
            if (Keyboard::isKeyPressed(Keyboard::Escape))
            {
                m_Window.close();
            }

            // Handle the player starting the game
            if (Keyboard::isKeyPressed(Keyboard::Return))
            {
                m_Playing = true;
            }

            // Switch between Thomas and Bob
            if (Keyboard::isKeyPressed(Keyboard::Q))
            {
                m_Character1 = !m_Character1;
            }
        }
    }
}

```

```
        // Switch between full and split-screen
        if (Keyboard::isKeyPressed(Keyboard::E))
        {
            m_SplitScreen = !m_SplitScreen;
        }
    }

    // Handle input specific to Thomas
    if(m_Thomas.handleInput())
    {
        // Play a jump sound
    }

    // Handle input specific to Bob
    if(m_Bob.handleInput())
    {
        // Play a jump sound
    }
}
```

Note how simple the previous code is: all the functionality is contained within the Thomas and Bob classes. All the code must do is add an include directive for each of the Thomas and Bob classes. Then, within the input function, the code just calls the pure virtual `handleInput` functions on `m_Thomas` and `m_Bob`. The reason we wrap each of the calls in an `if` statement is that they return `true` or `false` based on whether a new jump has just been successfully initiated. We will handle playing the jump sound effects in *Chapter 17, Sound Spatialization and the HUD*.

Updating the update function to spawn and update the PlayableCharacter instances

This is broken into two parts. First, we need to spawn Bob and Thomas at the start of a new level, and second, we need to update them (by calling their update functions) each frame.

Spawning Thomas and Bob

We need to call the `spawn` functions of our `Thomas` and `Bob` objects in a few different places as the project progresses. Most obviously, we need to spawn the two characters when a new level begins. In the next chapter, as the number of tasks we need to perform at the beginning of a level increases, we will write a `loadLevel` function. For now, let's just call `spawn` on `m_Thomas` and `m_Bob` in the `update` function, as highlighted in the following code. Add the following code, but keep in mind that it will eventually be deleted and replaced:

```
void Engine::update(float dtAsSeconds)
{
    if (m_NewLevelRequired)
    {
        // These calls to spawn will be moved to a new
        // loadLevel() function soon
        // Spawn Thomas and Bob
        m_Thomas.spawn(Vector2f(0,0), GRAVITY);
        m_Bob.spawn(Vector2f(100, 0), GRAVITY);

        // Make sure spawn is called only once
        m_TimeRemaining = 10;
        m_NewLevelRequired = false;
    }

    if (m_Playing)
    {
        // Count down the time the player has left
        m_TimeRemaining -= dtAsSeconds;

        // Have Thomas and Bob run out of time?
        if (m_TimeRemaining <= 0)
        {
            m_NewLevelRequired = true;
        }
    }

    }// End if playing
}
```

The previous code simply calls `spawn` and passes in a location in the game world, along with the gravity. The code is wrapped in an `if` statement that checks whether a new level is required. The spawning code will be moved to a dedicated `loadLevel` function, but the `if` condition will be part of the finished project. Also, `m_TimeRemaining` is set to an arbitrary 10 seconds for now.

Now, we can update the instances each frame of the game loop.

Updating Thomas and Bob each frame

Next, we will update Thomas and Bob. All we need to do is call their `update` functions and pass in the time this frame has taken.

Add the following highlighted code:

```
void Engine::update(float dtAsSeconds)
{
    if (m_NewLevelRequired)
    {
        // These calls to spawn will be moved to a new
        // LoadLevel function soon
        // Spawn Thomas and Bob
        m_Thomas.spawn(Vector2f(0,0), GRAVITY);
        m_Bob.spawn(Vector2f(100, 0), GRAVITY);

        // Make sure spawn is called only once
        m_NewLevelRequired = false;
    }

    if (m_Playing)
    {
        // Update Thomas
        m_Thomas.update(dtAsSeconds);

        // Update Bob
        m_Bob.update(dtAsSeconds);

        // Count down the time the player has left
        m_TimeRemaining -= dtAsSeconds;
    }
}
```

```

        // Have Thomas and Bob run out of time?
        if (m_TimeRemaining <= 0)
        {
            m_NewLevelRequired = true;
        }

    } // End if playing
}

```

Now that the characters can move, we need to update the appropriate `View` objects to center around the characters and make them the center of attention. Of course, until we have some objects in our game world, the sensation of actual movement will not be achieved.

Add the following highlighted code:

```

void Engine::update(float dtAsSeconds)
{
    if (m_NewLevelRequired)
    {
        // These calls to spawn will be moved to a new
        // LoadLevel function soon
        // Spawn Thomas and Bob
        m_Thomas.spawn(Vector2f(0,0), GRAVITY);
        m_Bob.spawn(Vector2f(100, 0), GRAVITY);

        // Make sure spawn is called only once
        m_NewLevelRequired = false;
    }

    if (m_Playing)
    {
        // Update Thomas
        m_Thomas.update(dtAsSeconds);

        // Update Bob
        m_Bob.update(dtAsSeconds);

        // Count down the time the player has left
        m_TimeRemaining -= dtAsSeconds;
    }
}

```



```
        // Have Thomas and Bob run out of time?
        if (m_TimeRemaining <= 0)
        {
            m_NewLevelRequired = true;
        }

    } // End if playing

    // Set the appropriate view around the appropriate character
    if (m_SplitScreen)
    {
        m_LeftView.setCenter(m_Thomas.getCenter());
        m_RightView.setCenter(m_Bob.getCenter());
    }
    else
    {
        // Centre full screen around appropriate character
        if (m_Character1)
        {
            m_MainView.setCenter(m_Thomas.getCenter());
        }
        else
        {
            m_MainView.setCenter(m_Bob.getCenter());
        }
    }
}
```

The previous code handles the two possible situations. First, the `if (mSplitScreen)` condition positions the left-hand view around `m_Thomas` and the right-hand view around `m_Bob`. The `else` clause that executes when the game is in full screen mode tests to see if `m_Character1` is true. If it is, then the full screen view (`m_MainView`) is centered around Thomas, otherwise it is centered around Bob. You probably remember that the player can use the *E* key to toggle split-screen mode and the *Q* key to toggle between Bob and Thomas in full screen mode. We coded this in the input function of the Engine class, back in *Chapter 12, Layering Views and Implementing the HUD*.

Now, we can draw the graphics for Thomas and Bob to the screen.

Drawing Bob and Thomas

Make sure the `Draw.cpp` file is open and add the following highlighted code:

```
void Engine::draw()
{
    // Rub out the last frame
    m_Window.clear(Color::White);

    if (!m_SplitScreen)
    {
        // Switch to background view
        m_Window.setView(m_BGMainView);
        // Draw the background
        m_Window.draw(m_BackgroundSprite);
        // Switch to m_MainView
        m_Window.setView(m_MainView);

        // Draw thomas
        m_Window.draw(m_Thomas.getSprite());

        // Draw bob
        m_Window.draw(m_Bob.getSprite());
    }
    else
    {
        // Split-screen view is active

        // First draw Thomas' side of the screen

        // Switch to background view
        m_Window.setView(m_BGLeftView);
        // Draw the background
        m_Window.draw(m_BackgroundSprite);
        // Switch to m_LeftView
        m_Window.setView(m_LeftView);

        // Draw bob
        m_Window.draw(m_Bob.getSprite());

        // Draw thomas
        m_Window.draw(m_Thomas.getSprite());
    }
}
```

```
        // Now draw Bob's side of the screen

        // Switch to background view
        m_Window.setView(m_BGRightView);
        // Draw the background
        m_Window.draw(m_BackgroundSprite);
        // Switch to m_RightView
        m_Window.setView(m_RightView);

        // Draw thomas
        m_Window.draw(m_Thomas.getSprite());

        // Draw bob
        m_Window.draw(m_Bob.getSprite());

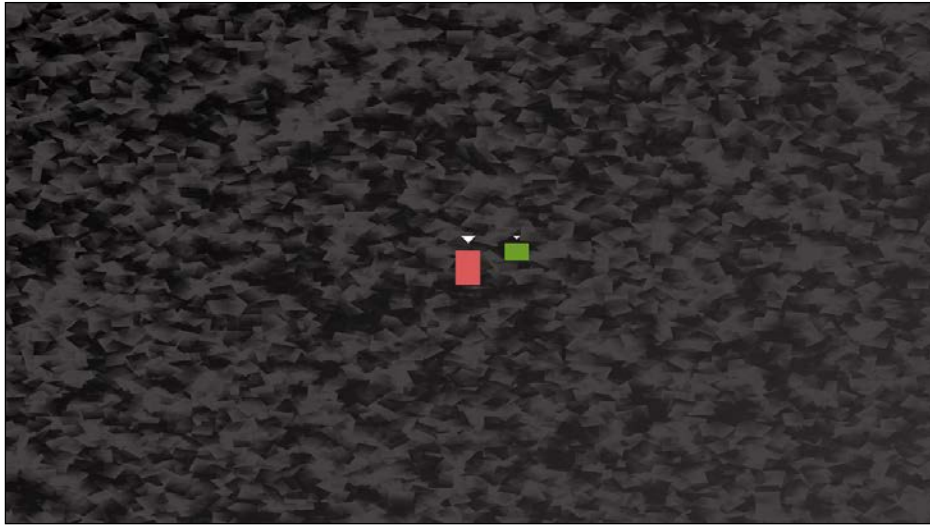
    }

    // Draw the HUD
    // Switch to m_HudView
    m_Window.setView(m_HudView);

    // Show everything we have just drawn
    m_Window.display();
}
```

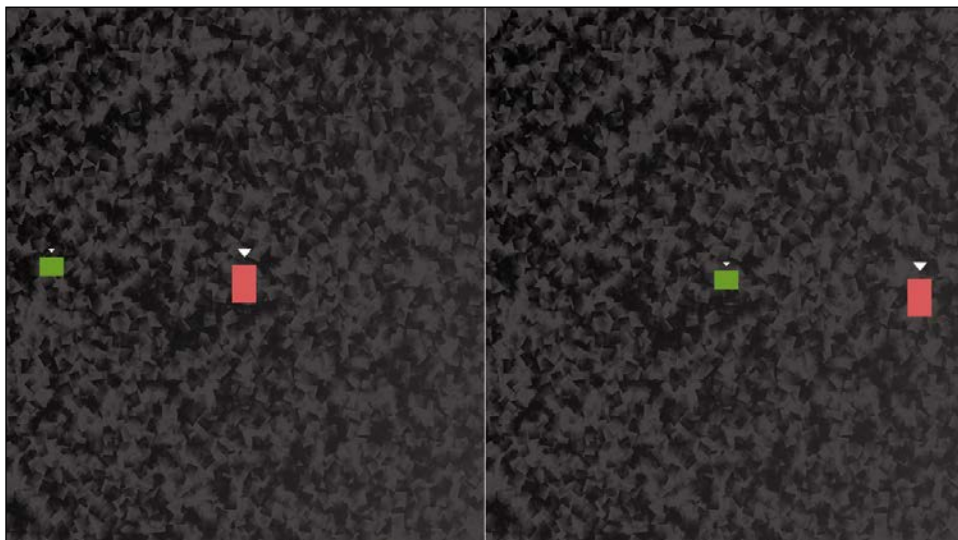
Notice that we draw both Thomas and Bob for full screen, the left, and the right. Also, notice the very subtle difference in the way that we draw the characters in split-screen mode. When drawing the left-hand side of the screen, we switch the order the characters are drawn and draw Thomas after Bob. So, Thomas will always be "on top" on the left and Bob will always be on top on the right. This is because the player controlling Thomas is catered for on the left and Bob the right, respectively.

You can now run the game and see Thomas and Bob in the center of the screen, as follows:



If you press the *Q* key to switch focus from Thomas to Bob, you will see the *View* make the slight adjustment. If you move either of the characters left or right (Thomas with *A* and *D*, and Bob with the arrow keys) you will see them move relative to each other.

Try pressing the *E* key to toggle between full screen and split-screen. Then, try moving both characters again to see the effect. In the following screenshot, you can see that Thomas is always centered in the left-hand window and Bob is always centered in the right-hand window:



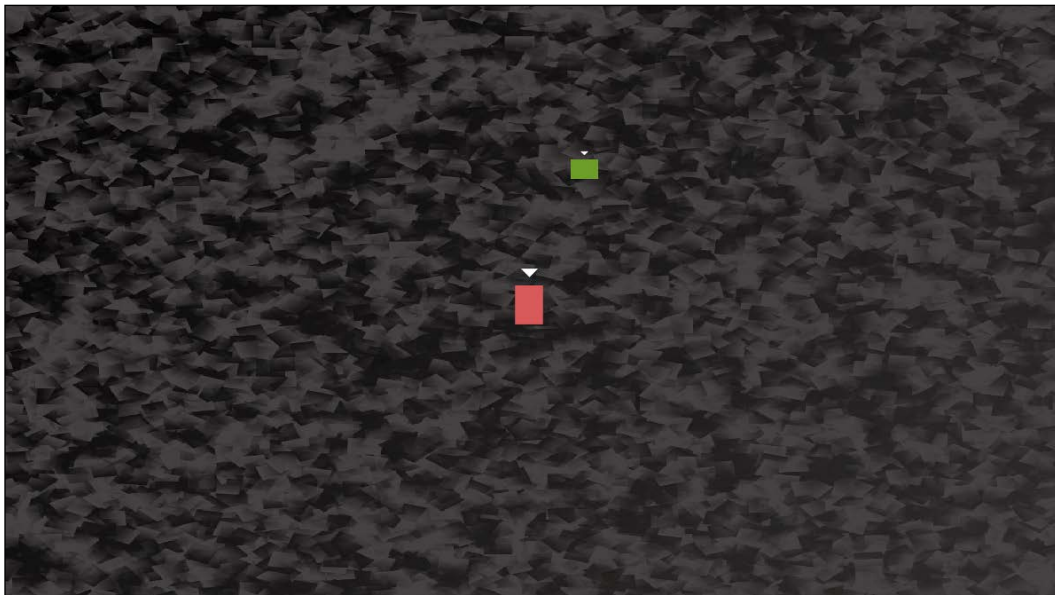
If you leave the game running long enough, the characters will respawn in their original positions every 10 seconds. This is the beginning of the functionality we will need for the finished game. This behavior is caused by `m_TimeRemaining` going below 0 and then setting the `m_NewLevelRequired` variable to `true`.

Also note that we can't see the full effect of movement until we draw the details of the level. In fact, although it can't be seen, both characters are continuously falling at 300 pixels per second. Since the camera is centering around them every frame and there are no other objects in the game world, we cannot see this downward movement.

If you want to see this for yourself, just change the call to `m_Bob.spawn`, as follows:

```
m_Bob.spawn(Vector2f(0,0), 0);
```

Now that Bob has no gravitational effect, Thomas will visibly fall away from him. This is shown in the following screenshot:



We will add some playable levels to interact with in the next chapter.

Summary

In this chapter, we learned about some new C++ concepts, such as inheritance, which allows us to extend a class and gain all its functionality. We also learned that we can declare variables as protected and that this will give the child class access to them, but they will still be encapsulated (hidden) from all other code. We also used pure virtual functions, which make a class abstract, meaning that the class cannot be instantiated and must therefore be inherited from/extended. We were also introduced to the concept of polymorphism, but will need to wait until the next chapter to use it in our game.

In the next chapter, we will add some major functionality to the game. By the end of the next chapter, Thomas and Bob will be walking, jumping, and falling. They will even be able to jump on each other's heads, as well as exploring some level designs that have been loaded from a text file.

FAQ

Q) We learned about Polymorphism, but why didn't I notice anything polymorphic in the game code so far?

A) We will see polymorphism in action in the next chapter when we write a function that takes `PlayerCharacter` as a parameter. We will see how we can pass both Bob and Thomas to this new function. It will work the same with both of them.

16

Building Playable Levels and Collision Detection

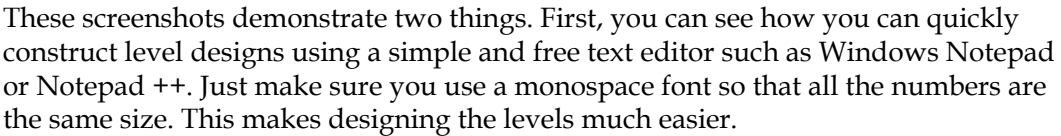
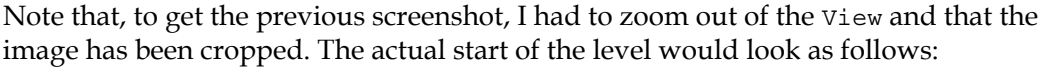
This chapter will probably be one of the most satisfying chapters of this project. The reason for this is that, by the end of it, we will have a playable game. Although there will still be features to implement (sound, particle effects, the HUD, and shader effects), Bob and Thomas will be able to run, jump, and explore the world. Furthermore, you will be able to create your very own level designs of any size or complexity by simply making platforms and obstacles in a text file.

We will achieve all this by covering the following topics:

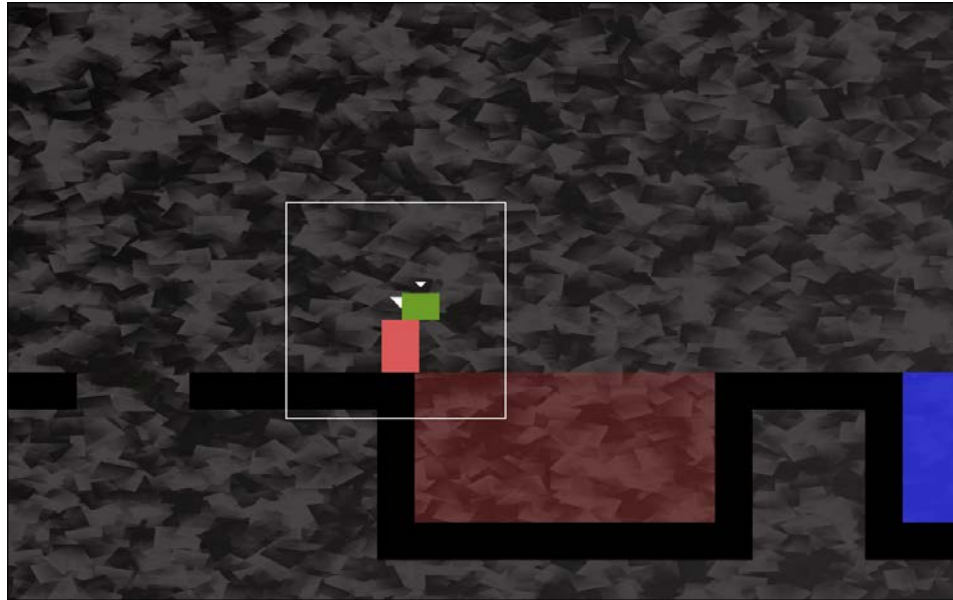
- Exploring how to design levels in a text file
- Building a `LevelManager` class that will load levels from a text file, convert them into data that our game can use, and keep track of the level details such as spawn position, current level, and allowed time limit
- Updating the game engine to use `LevelManager`
- Coding a polymorphic function to handle collision detection for both Bob and Thomas

[illegible]


The previous code translates into the following level layout:



Secondly, these screenshots demonstrate the gameplay aspects of the design. From left to right in the level, Thomas and Bob need to jump over a small hole or they will fall to their death (respawn). Then, they have a large expanse of fire to traverse. It is impossible for Bob to jump that many tiles. The players will need to work together to find the solution. The only way that Bob will clear the fire tiles is by standing on Thomas's head and jumping from there, as shown in the following screenshot:



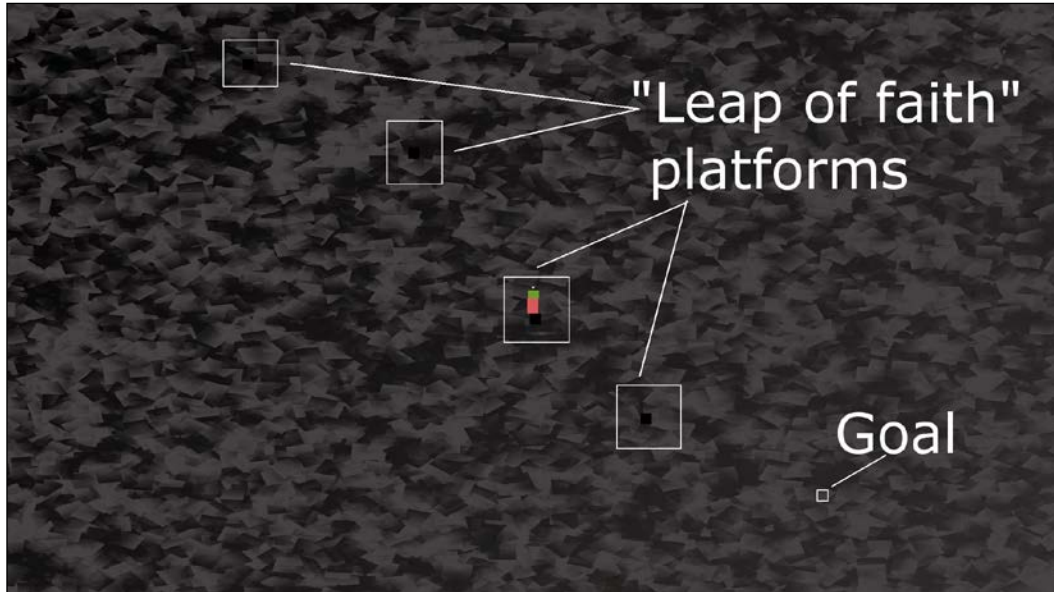
It is then quite simple to get to the goal and move on to the next level.

[ I strongly encourage you to complete this chapter and then spend some time designing your own levels.]

I have included a few level designs to get us started. They are in the `levels` folder that we added to the project back in *Chapter 14, Abstraction and Code Management – Making Better Use of OOP*.

There are some zoomed-out views of the game there, along with a screenshot of the code of the level design. The screenshot of the code is probably more useful than reproducing the textual content. If the code has to be checked, just open the files in the `levels` folder.

The code for the in-game platforms has been highlighted, as they are not very clear in the zoomed-out screenshot that follows:



The provided designs are simple. The game engine will be able to handle very large designs, but we have the freedom to use our imagination and build some long and challenging levels.

Of course, these designs won't do anything until we learn how to load them and convert the text into a playable level. Additionally, it won't be possible to stand on any platforms until we have implemented the collision detection.

First, let's handle loading the level designs.

Building the LevelManager class

It will take several phases of coding to make our level designs work.

The first thing we will do is code the `LevelManager` header file. This will allow us to look at and discuss the member variables and functions that will be in the `LevelManager` class.

Next, we will code the `LevelManager.cpp` file, which will have all the function definitions in it. Since this is a long file, we will break it up into several sections to code and discuss them.

Once the `LevelManager` class is complete, we will add an instance of it to the game engine (`Engine` class). We will also add a new function to the `Engine` class, `loadLevel`, which we can call from the `update` function whenever a new level is required. The `loadLevel` function will not only use the `LevelManager` instance to load the appropriate level – it will also take care of aspects such as spawning the player characters and preparing the clock.

Now, let's get an overview of `LevelManager` by coding the `LevelManager.h` file.

Coding LevelManager.h

Right-click **Header Files** in the **Solution Explorer** and select **Add | New Item....** In the **Add New Item** window, highlight (by left-clicking) **Header File (.h)** and then, in the **Name** field, type `LevelManager.h`. Finally, click the **Add** button. We are now ready to code the header file for the `LevelManager` class.

Add the following include directives and private variables and then we will discuss them:

```
#pragma once

#include <SFML/Graphics.hpp>
using namespace sf;
using namespace std;

class LevelManager
{
private:
    Vector2i m_LevelSize;
    Vector2f m_StartPosition;
    float m_TimeModifier = 1;
    float m_BaseTimeLimit = 0;
    int m_CurrentLevel = 0;
    const int NUM_LEVELS = 4;

    // public declarations go here
```

The preceding code declares a `Vector2i, m_LevelSize` to hold two integer values that will hold the horizontal and vertical number of tiles that the current map contains. The `Vector2f, m_StartPosition` contains the coordinates in the world where Bob and Thomas should be spawned. Note that this is not a tile position relatable to `m_LevelSize` units but a horizontal and vertical pixel position in the level.

The `m_TimeModifier` member variable is a float type variable that will be used to multiply the time that's available in the current level. The reason we want to do this is so that we can change (decrease) this value so that we can shorten the time that's available each time the player attempts the same level. As an example, if the player gets 60 seconds for the first time they attempt level 1, then 60 multiplied by 1 is, of course, 60. When the player completes all the levels and comes back to level 1 for the second time, `m_TimeModifier` will have been reduced by 10 percent. Then, when the time available is multiplied by 0.9, the amount of time that's available to the player will be 54 seconds. This is 10 percent less. The game will get steadily harder.

The `m_BaseTimeLimit` float variable holds the original, unmodified time limit we have just discussed.

We can probably guess that `m_CurrentLevel` will hold the current level number that is being played.

The `int, NUM_LEVELS` constant will be used to flag when it is appropriate to go back to level 1 again and reduce the value of `m_TimeModifier`.

Now, add the following public variables and function declarations after the previous code we added:

```
public:

    const int TILE_SIZE = 50;
    const int VERTS_IN_QUAD = 4;

    float getTimeLimit();

    Vector2f getStartPosition();

    int** nextLevel(VertexArray& rVaLevel);

    Vector2i getLevelSize();

    int getCurrentLevel();

};
```

In the previous code, there are two constant `int` members. `TILE_SIZE` is a useful constant to remind us that each tile in the sprite-sheet is fifty pixels wide and fifty pixels high. `VERTS_IN_QUAD` is a useful constant to make our manipulation of a `VertexArray` less error-prone. There are, in fact, four vertices in a quad. Now, we can't forget this.

The `getTimeLimit`, `getStartPosition`, `getLevelSize`, and `getCurrentLevel` functions are simple getter functions which return the current value of the private member variables we declared in the previous block of code.

The function that deserves to be talked about more is `nextLevel`. This function receives a `VertexArray` reference, just like we used in the *Zombie Arena* game. The function can then work on the `VertexArray` reference and all the changes will be present in the `VertexArray` reference from the calling code.

The `nextLevel` function returns a pointer to a pointer, which means we can return an address of the first element of a two-dimensional array of `int` values. We will be building a two-dimensional array of `int` values that will represent the layout of each level. Of course, these `int` values will be read from the level design text files.

Coding the `LevelManager.cpp` file

Right-click **Source Files** in the **Solution Explorer** and select **Add | New Item....** In the **Add New Item** window, highlight (by left-clicking) **C++ File (.cpp)** and then, in the **Name** field, type `LevelManager.cpp`. Finally, click the **Add** button. We are now ready to code the `.cpp` file for the `LevelManager` class.

As this is quite a long class, we will break it up to discuss it in six chunks. The first five will cover the `nextLevel` function, while the sixth will cover the rest of the functions.

Add the following include directives and the first (of five) part of the `nextLevel` function:

```
#include <SFML/Graphics.hpp>
#include <SFML/Audio.hpp>
#include "TextureHolder.h"
#include <sstream>
#include <fstream>
#include "LevelManager.h"

using namespace sf;
using namespace std;

int** LevelManager::nextLevel(VertexArray& rVaLevel)
{
    m_LevelSize.x = 0;
    m_LevelSize.y = 0;

    // Get the next level
    m_CurrentLevel++;
```



```
if (m_CurrentLevel > NUM_LEVELS)
{
    m_CurrentLevel = 1;
    m_TimeModifier -= .1f;
}

// Load the appropriate level from a text file
string levelToLoad;
switch (m_CurrentLevel)
{

case 1:
    levelToLoad = "levels/level1.txt";
    m_StartPosition.x = 100;
    m_StartPosition.y = 100;
    m_BaseTimeLimit = 30.0f;
    break;

case 2:
    levelToLoad = "levels/level2.txt";
    m_StartPosition.x = 100;
    m_StartPosition.y = 3600;
    m_BaseTimeLimit = 100.0f;
    break;

case 3:
    levelToLoad = "levels/level3.txt";
    m_StartPosition.x = 1250;
    m_StartPosition.y = 0;
    m_BaseTimeLimit = 30.0f;
    break;


case 4:
    levelToLoad = "levels/level4.txt";
    m_StartPosition.x = 50;
    m_StartPosition.y = 200;
    m_BaseTimeLimit = 50.0f;
    break;

} // End switch
```

After the include directives, the code initializes the `m_LevelSize.x` and `m_LevelSize.y` variables to zero.

Next, `m_CurrentLevel` is incremented. The `if` statement that follows checks whether `m_CurrentLevel` is greater than `NUM_LEVELS`. If it is, then `m_CurrentLevel` is set back to 1 and `m_TimeModifier` is reduced by 0.1 in order to shorten the allowed time for all levels.

The code then switches based on the value held by `m_CurrentLevel`. Each case statement initializes the name of the text file that holds the level design, the starting position for Thomas and Bob, as well as `m_BaseTimeLimit`, which is the unmodified time limit for the level in question.

 If you design your own levels, add a case statement and the appropriate values for it here. Also, edit the `NUM_LEVELS` constant in the `LevelManager.h` file.

Now, add the second part of the `nextLevel` function, as follows. Add this code immediately after the previous code. Study the code as we add it so we can discuss it:

```
ifstream inputFile(levelToLoad);
string s;

// Count the number of rows in the file
while (getline(inputFile, s))
{
    ++m_LevelSize.y;
}

// Store the length of the rows
m_LevelSize.x = s.length();
```

In the preceding (second part) code, we declare an `ifstream` object called `inputFile` which opens a stream to the file name contained in `levelToLoad`.

The code loops through each line of the file using `getline`, but doesn't record any of its content. All it does is count the number of lines by incrementing `m_LevelSize.y`. After the `for` loop, the width of the level is saved in `m_LevelSize.x` using the `s.length` function. This implies that the length of all the lines must be the same or we would run into trouble.

At this point, we know and have saved the length and width of the current level in `m_LevelSize`.

Now, add the third part of the `nextLevel` function, as shown in the following code. Add the code immediately underneath the previous code. Study the code as we add it so we can discuss it:

```
// Go back to the start of the file
inputFile.clear();
inputFile.seekg(0, ios::beg);

// Prepare the 2D array to hold the int values from the file
int** arrayLevel = new int*[m_LevelSize.y];
for (int i = 0; i < m_LevelSize.y; ++i)
{
    // Add a new array into each array element
    arrayLevel[i] = new int[m_LevelSize.x];
}
```

First, we clear `inputFile` using its `clear` function. The `seekg` function, which is called with the `0, ios::beg` parameters, moves the file cursor's position (where characters will be read from next) to the beginning of the file.

Next, we declare a pointer to a pointer called `arrayLevel`. Note that this is done on the free store/heap using the `new` keyword. Once we have initialized this two-dimensional array, we will be able to return its address to the calling code and it will persist until we either delete it or the game is closed.

The `for` loop loops from 0 to `m_LevelSize.y - 1`. In each pass of the loop, it adds a new array of `int` values, on the heap, to match the value of `m_LevelSize.x`. We now have a perfectly configured (for the current level) two-dimensional array. The only problem is that there is nothing in it yet.

Now, add the fourth part of the `nextLevel` function, as shown in the following code. Add this code immediately after the previous code. Study the code as we add it so we can discuss it:

```
// Loop through the file and store all
// the values in the 2d array
string row;
int y = 0;
while (inputFile >> row)
{
    for (int x = 0; x < row.length(); x++) {
```

```

        const char val = row[x];
        arrayLevel[y][x] = atoi(&val);
    }

    y++;
}

// Close the file
inputFile.close();

```

First, the code initializes a string called `row` that will hold only one row of the level design at a time. We also declare and initialize an `int` called `y` that will help us count the rows.

The while loop executes repeatedly until `inputFile` gets past the last row. Inside the while loop, there is a for loop which goes through each character of the current row and stores it in the two-dimensional array, `arrayLevel`. Notice that we access the right element of the two-dimensional array with `arrayLevel[y][x]`. The `atoi` function converts the `char val` into an `int`. This is required because we have a two-dimensional array for `int`, and not for `char`.

Now, let's add the fifth part of the `nextLevel` function, as shown here. Add this code immediately after the previous code. Study the code as we add it, so we can discuss it:

```

// What type of primitive are we using?
rVaLevel.setPrimitiveType(Quads);

// Set the size of the vertex array
rVaLevel.resize(m_LevelSize.x *
    m_LevelSize.y * VERTS_IN_QUAD);

// Start at the beginning of the vertex array
int currentVertex = 0;

for (int x = 0; x < m_LevelSize.x; x++)
{
    for (int y = 0; y < m_LevelSize.y; y++)
    {
        // Position each vertex in the current quad
        rVaLevel[currentVertex + 0].position =
            Vector2f(x * TILE_SIZE,
                y * TILE_SIZE);
    }
}

```

```
        rVaLevel[currentVertex + 1].position =
            Vector2f((x * TILE_SIZE) + TILE_SIZE,
                    y * TILE_SIZE);

        rVaLevel[currentVertex + 2].position =
            Vector2f((x * TILE_SIZE) + TILE_SIZE,
                    (y * TILE_SIZE) + TILE_SIZE);

        rVaLevel[currentVertex + 3].position =
            Vector2f((x * TILE_SIZE),
                    (y * TILE_SIZE) + TILE_SIZE);

        // Which tile from the sprite sheet should we use
        int verticalOffset = arrayLevel[y][x] * TILE_SIZE;

        rVaLevel[currentVertex + 0].texCoords =
            Vector2f(0, 0 + verticalOffset);

        rVaLevel[currentVertex + 1].texCoords =
            Vector2f(TILE_SIZE, 0 + verticalOffset);

        rVaLevel[currentVertex + 2].texCoords =
            Vector2f(TILE_SIZE, TILE_SIZE + verticalOffset);

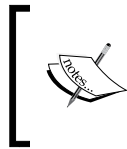
        rVaLevel[currentVertex + 3].texCoords =
            Vector2f(0, TILE_SIZE + verticalOffset);

        // Position ready for the next four vertices
        currentVertex = currentVertex + VERTS_IN_QUAD;
    }
}

return arrayLevel;
} // End of nextLevel function
```

Although this is the longest section of code from the five sections (we divided `nextLevel` in two), it is also the most straightforward. This is because we have seen very similar code in the *Zombie Arena* project.

The process for the preceding code is that the nested `for` loops loop from zero through to the width and height of the level. For each position in the array, four vertices are put into the `VertexArray` and four texture coordinates are assigned from the sprite-sheet. The positions of the vertices and texture coordinates are calculated using the `currentVertex` variable, `TILE_SIZE`, and the `VERTS_IN_QUAD` constants. At the end of each loop of the inner `for` loop, `currentVertex` is increased by `VERTS_IN_QUAD`, moving nicely on to the next tile.



The important thing to remember about `VertexArray` is that it was passed in to `nextLevel` by reference. Therefore, `VertexArray` will be available in the calling code. We will call `nextLevel` from the code in the `Engine` class.

Once this function has been called, the `Engine` class will have a `VertexArray` to represent the level graphically and a two-dimensional array of `int` values as a numerical representation of all the platforms and obstacles in the level.

The rest of the `LevelManager` functions are all simple getter functions but do take the time to familiarize yourself with which private value is returned by which function. Add the remaining functions from the `LevelManager` class, as follows:

```
Vector2i LevelManager::getLevelSize()
{
    return m_LevelSize;
}

int LevelManager::getCurrentLevel()
{
    return m_CurrentLevel;
}

float LevelManager::getTimeLimit()
{
    return m_BaseTimeLimit * m_TimeModifier;
}

Vector2f LevelManager::getStartPosition()
{
    return m_StartPosition;
}
```

Now that the `LevelManager` class is complete, we can move on to using it. We will code another function in the `Engine` class to do so.

Coding the loadLevel function

To be clear, this function is part of the `Engine` class, although it will delegate much of its work to other functions, including those of the `LevelManager` class that we just built.

First, let's add the declaration for the new function, along with some other new pieces of code, to the `Engine.h` file. Open the `Engine.h` file and add the highlighted lines of code shown in the abbreviated snapshot of the `Engine.h` file, as follows:

```
#pragma once
#include <SFML/Graphics.hpp>
#include "TextureHolder.h"
#include "Thomas.h"
#include "Bob.h"
#include "LevelManager.h"

using namespace sf;

class Engine
{
private:
    // The texture holder
    TextureHolder th;

    // Thomas and his friend, Bob
    Thomas m_Thomas;
    Bob m_Bob;

    // A class to manage all the levels
    LevelManager m_LM;

    const int TILE_SIZE = 50;
    const int VERTS_IN_QUAD = 4;

    // The force pushing the characters down
    const int GRAVITY = 300;

    // A regular RenderWindow
    RenderWindow m_Window;
```

```
// The main Views
View m_MainView;
View m_LeftView;
View m_RightView;

// Three views for the background
View m_BGMainView;
View m_BGLeftView;
View m_BGRightView;

View m_HudView;

// Declare a sprite and a Texture for the background
Sprite m_BackgroundSprite;
Texture m_BackgroundTexture;

// Is the game currently playing?
bool m_Playing = false;

// Is character 1 or 2 the current focus?
bool m_Character1 = true;

// Start in full screen mode
bool m_SplitScreen = false;

// How much time is left in the current level
float m_TimeRemaining = 10;
Time m_GameTimeTotal;

// Is it time for a new/first level?
bool m_NewLevelRequired = true;

// The vertex array for the level tiles
VertexArray m_VAlevel;

// The 2d array with the map for the level
// A pointer to a pointer
int** m_ArrayLevel = NULL;

// Texture for the level tiles
Texture m_TextureTiles;
```



```
// Private functions for internal use only
void input();
void update(float dtAsSeconds);
void draw();

// Load a new level
void loadLevel();

public:
    // The Engine constructor
    Engine();

    ...
    ...
    ...
```

This is what we can see in the previous code:

- We included the `LevelManager.h` file.
- We added an instance of `LevelManager` called `m_LM`.
- We added a `VertexArray` called `m_VAlevel`.
- We added a pointer to a pointer to `int` that will hold the two-dimensional array that is returned from `nextLevel`.
- We added a new `Texture` object for the sprite-sheet.
- We added the declaration for the `loadLevel` function that we will write now.

Right-click **Source Files** in the **Solution Explorer** and select **Add | New Item....** In the **Add New Item** window, highlight (by left-clicking) **C++ File (.cpp)** and then, in the **Name** field, type `LoadLevel.cpp`. Finally, click the **Add** button. We are now ready to code the `loadLevel` function.

Add the code for the `loadLevel` function to the `LoadLevel.cpp` file. Then, we can discuss it:

```
#include "Engine.h"

void Engine::loadLevel()
{
    m_Playing = false;

    // Delete the previously allocated memory
    for (int i = 0; i < m_LM.getLevelSize().y; ++i)
    {
```

```

        delete[] m_ArrayLevel[i];
    }
    delete[] m_ArrayLevel;

    // Load the next 2d array with the map for the level
    // And repopulate the vertex array as well
    m_ArrayLevel = m_LM.nextLevel(m_VALevel);

    // How long is this new time limit
    m_TimeRemaining = m_LM.getTimeLimit();

    // Spawn Thomas and Bob
    m_Thomas.spawn(m_LM.getStartPosition(), GRAVITY);
    m_Bob.spawn(m_LM.getStartPosition(), GRAVITY);

    // Make sure this code isn't run again
    m_NewLevelRequired = false;
}

```

First, we set `m_Playing` to false to stop parts of the update function from executing. Next, we loop through all the horizontal arrays within `m_ArrayLevel` and delete them. After the for loop, we delete `m_ArrayLevel` itself.

`m_ArrayLevel = m_LM.nextLevel(m_VALevel)` calls `nextLevel` and prepares the `VertexArray` `m_VALevel`, as well as the two-dimensional array known as `m_ArrayLevel`. The level is set up and ready to go.

`m_TimeRemaining` is initialized by calling `getTimeLimit` and Thomas and Bob are spawned using the `spawn` function, along with the value returned from `getStartPosition`.

Finally, `m_NewLevelRequired` is set to false. As we will see in a few pages time, `m_NewLevelRequired` being set to true causes `loadLevel` to be called. We only want to run this function once.

Updating the engine

Open the `Engine.cpp` file and add the following highlighted code to load the sprite-sheet texture at the end of the Engine constructor:

```

Engine::Engine()
{
    // Get the screen resolution and create an SFML window and View
    Vector2f resolution;

```

```
resolution.x = VideoMode::getDesktopMode().width;
resolution.y = VideoMode::getDesktopMode().height;

m_Window.create(VideoMode(resolution.x, resolution.y),
    "Thomas was late",
    Style::Fullscreen);

// Initialize the full screen view
m_MainView.setSize(resolution);
m_HudView.reset(
    FloatRect(0, 0, resolution.x, resolution.y));

// Initialize the split-screen Views
m_LeftView.setViewport(
    FloatRect(0.001f, 0.001f, 0.498f, 0.998f));

m_RightView.setViewport(
    FloatRect(0.5f, 0.001f, 0.499f, 0.998f));

m_BGLeftView.setViewport(
    FloatRect(0.001f, 0.001f, 0.498f, 0.998f));

m_BGRightView.setViewport(
    FloatRect(0.5f, 0.001f, 0.499f, 0.998f));

// Can this graphics card use shaders?
if (!sf::Shader::isAvailable())
{
    // Time to get a new PC
    m_Window.close();
}

m_BackgroundTexture = TextureHolder::GetTexture(
    "graphics/background.png");

// Associate the sprite with the texture
m_BackgroundSprite.setTexture(m_BackgroundTexture);

// Load the texture for the background vertex array
m_TextureTiles = TextureHolder::GetTexture(
    "graphics/tiles_sheet.png");
}
```

All we do in the previous code is load the sprite-sheet into `m_TextureTiles`.

Open the `Update.cpp` file and make the following highlighted changes and additions:

```
void Engine::update(float dtAsSeconds)
{
    if (m_NewLevelRequired)
    {
        // These calls to spawn will be moved to a new
        // loadLevel function soon
        // Spawn Thomas and Bob
        //m_Thomas.spawn(Vector2f(0,0), GRAVITY);
        //m_Bob.spawn(Vector2f(100, 0), GRAVITY);

        // Make sure spawn is called only once
        //m_TimeRemaining = 10;
        //m_NewLevelRequired = false;

        // Load a level
        loadLevel();
    }
}
```

Actually, we should delete, rather than comment out, the lines we are no longer using. I have just shown it in this way so that the changes are clear. All there should be in the previous `if` statement is the call to `loadLevel`.

Finally, before we can see the results of the work we've done so far in this chapter, open the `Draw.cpp` file and make the following highlighted additions to draw the vertex array that represents a level:

```
void Engine::draw()
{
    // Rub out the last frame
    m_Window.clear(Color::White);

    if (!m_SplitScreen)
    {
        // Switch to background view
        m_Window.setView(m_BGMainView);
        // Draw the background
        m_Window.draw(m_BackgroundSprite);
        // Switch to m_MainView
        m_Window.setView(m_MainView);
    }
}
```

```
        // Draw the Level
        m_Window.draw(m_VAlevel, &m_TextureTiles);

        // Draw thomas
        m_Window.draw(m_Thomas.getSprite());

        // Draw bob
        m_Window.draw(m_Bob.getSprite());
    }
    else
    {
        // Split-screen view is active

        // First draw Thomas' side of the screen

        // Switch to background view
        m_Window.setView(m_BGLeftView);
        // Draw the background
        m_Window.draw(m_BackgroundSprite);
        // Switch to m_LeftView
        m_Window.setView(m_LeftView);

        // Draw the Level
        m_Window.draw(m_VAlevel, &m_TextureTiles);

        // Draw bob
        m_Window.draw(m_Bob.getSprite());

        // Draw thomas
        m_Window.draw(m_Thomas.getSprite());

        // Now draw Bob's side of the screen

        // Switch to background view
        m_Window.setView(m_BGRightView);
        // Draw the background
        m_Window.draw(m_BackgroundSprite);
        // Switch to m_RightView
        m_Window.setView(m_RightView);

        // Draw the Level
        m_Window.draw(m_VAlevel, &m_TextureTiles);
    }
}
```

```

        // Draw thomas
        m_Window.draw(m_Thomas.getSprite());

        // Draw bob
        m_Window.draw(m_Bob.getSprite());

    }

    // Draw the HUD
    // Switch to m_HudView
    m_Window.setView(m_HudView);

    // Show everything we have just drawn
    m_Window.display();
}

```

Note that we need to draw the `VertexArray` for all the screen options (full, left, and right).

Now, you can run the game. Unfortunately, however, Thomas and Bob fall straight through all our lovingly designed platforms. Due to this, we can't try and progress through the levels and beat the clock.

Collision detection

We will handle collision detection using rectangle intersection and the SFML `intersects` function. What will be different in this project is that we will abstract the collision detection code into its own function. Thomas and Bob, as we have already seen, have multiple rectangles (`m_Head`, `m_Feet`, `m_Left`, and `m_Right`) that we need to check for collisions.

Coding the `detectCollisions` function

To be clear, this function is part of the `Engine` class. Open the `Engine.h` file and add a declaration for a function called `detectCollisions`. This is highlighted in the following code snippet:

```

// Private functions for internal use only
void input();
void update(float dtAsSeconds);
void draw();

```

```
// Load a new level
void loadLevel();

bool detectCollisions(PlayableCharacter& character);

public:
    // The Engine constructor
    Engine();
```

Notice from the signature that the `detectCollision` function takes a polymorphic argument as a `PlayerCharacter` object. As we know, `PlayerCharacter` is abstract and can never be instantiated. We do, however, inherit from it with the `Thomas` and `Bob` classes. We will be able to pass either `m_Thomas` or `m_Bob` to `detectCollisions`.

Right-click **Source Files** in the **Solution Explorer** and select **Add | New Item....** In the **Add New Item** window, highlight (by left-clicking) **C++ File (.cpp)** and then, in the **Name** field, type `DetectCollisions.cpp`. Finally, click the **Add** button. We are now ready to code the `detectCollisions` function.

Add the following code to `DetectCollisions.cpp`. Note that this is just the first part of this function:

```
#include "Engine.h"

bool Engine::detectCollisions(PlayableCharacter& character)
{
    bool reachedGoal = false;
    // Make a rect for all his parts
    FloatRect detectionZone = character.getPosition();

    // Make a FloatRect to test each block
    FloatRect block;

    block.width = TILE_SIZE;
    block.height = TILE_SIZE;

    // Build a zone around thomas to detect collisions
    int startX = (int)(detectionZone.left / TILE_SIZE) - 1;
    int startY = (int)(detectionZone.top / TILE_SIZE) - 1;
    int endX = (int)(detectionZone.left / TILE_SIZE) + 2;

    // Thomas is quite tall so check a few tiles vertically
    int endY = (int)(detectionZone.top / TILE_SIZE) + 3;
```

```
// Make sure we don't test positions lower than zero
// Or higher than the end of the array
if (startX < 0)startX = 0;
if (startY < 0)startY = 0;
if (endX >= m_LM.getLevelSize().x)
    endX = m_LM.getLevelSize().x;
if (endY >= m_LM.getLevelSize().y)
    endY = m_LM.getLevelSize().y;
```

The first thing that we do is declare a Boolean called `reachedGoal`. This is the value that the `detectCollisions` function returns to the calling code. It is initialized to `false`.

Next, we declare a `FloatRect` object called `detectionZone` and initialize it with the same rectangle that represents the entire rectangle of the character sprite. Note that we will not actually do intersection tests with this rectangle. After, we declare another `FloatRect` called `block`. We initialize `block` as a 50 by 50 game unit rectangle. We will see `block` in use shortly.

Next, we will look at how we will use `detectionZone`. We initialize four `int` variables, `startX`, `startY`, `endX`, and `endY`, by expanding the area around `detectionZone` by a few blocks. In the four `if` statements that follow, we check that it is not possible to try and do collision detection on a tile that does not exist. We will achieve this by making sure we never check positions less than zero or greater than the value returned by `getLevelSize().x` or `.y`.

What all this previous code has done is create an area that is used to do collision detection. There is no point doing collision detection on a block that is hundreds or thousands of pixels away from the character. In addition, if we try and do collision detection where an array position doesn't exist (less than zero or greater than `getLevelSize().x`), the game will crash.

Next, add the following code, which handles the player falling out of the level:

```
// Has the character fallen out of the map?
FloatRect level(0, 0,
    m_LM.getLevelSize().x * TILE_SIZE,
    m_LM.getLevelSize().y * TILE_SIZE);

if (!character.getPosition().intersects(level))
{
    // respawn the character
    character.spawn(m_LM.getStartPosition(), GRAVITY);
}
```


For a character to stop falling, it must collide with a platform. Therefore, if the player moves out of the map (where there are no platforms), they will continuously fall. The previous code checks whether the character *does not* intersect with the `FloatRect`, `level`. If it does not, then it has fallen out of the level and the `spawn` function sends it back to the start.

Add the following, quite long, code block and then we will go through what it does:

```
// Loop through all the local blocks
for (int x = startX; x < endX; x++)
{
    for (int y = startY; y < endY; y++)
    {
        // Initialize the starting position of the current block
        block.left = x * TILE_SIZE;
        block.top = y * TILE_SIZE;

        // Has character been burnt or drowned?
        // Use head as this allows him to sink a bit
        if (m_ArrayLevel[y][x] == 2 || m_ArrayLevel[y][x] == 3)
        {
            if (character.getHead().intersects(block))
            {
                character.spawn(m_LM.getStartPosition(), GRAVITY);
                // Which sound should be played?
                if (m_ArrayLevel[y][x] == 2) // Fire, ouch!
                {
                    // Play a sound

                }
                else // Water
                {
                    // Play a sound
                }
            }
        }

        // Is character colliding with a regular block
        if (m_ArrayLevel[y][x] == 1)
        {
            if (character.getRight().intersects(block))
            {
```

```

        character.stopRight(block.left);
    }
    else if (character.getLeft().intersects(block))
    {
        character.stopLeft(block.left);
    }

    if (character.getFeet().intersects(block))
    {
        character.stopFalling(block.top);
    }
    else if (character.getHead().intersects(block))
    {
        character.stopJump();
    }
}

// More collision detection here once we have
// learned about particle effects

// Has the character reached the goal?
if (m_ArrayLevel[y][x] == 4)
{
    // Character has reached the goal
    reachedGoal = true;
}

}
}

```

The previous code does three things using the same techniques. It loops through all the values contained between `startX`, `endX`, and `startY`, `endY`. For each pass, it checks and does the following:

- Has the character been burned or drowned? `if (m_ArrayLevel[y][x] == 2 || m_ArrayLevel[y][x] == 3)` determines whether the current position being checked is a fire or a water tile. If the character's head intersects with one of these tiles, the player is respawned. We also code an empty `if/else` block in preparation to add sound in the next chapter.

- Has the character touched a regular tile? code `if (m_ArrayLevel[y][x] == 1)` determines whether the current position being checked holds a regular tile. If it intersects with any of the rectangles that represent the various body parts of the character, then the related function is called (`stopRight`, `stopLeft`, `stopFalling`, or `stopJump`). The value that is passed to each of these functions and how the function uses that value to reposition the character is quite nuanced. While it is not necessary to closely examine these values to understand the code, we might like to look at the values that are passed in and then refer to the appropriate function of the `PlayableCharacter` class in the previous chapter. This will help you appreciate exactly what is going on.
- Has the character touched the goal tile? This is determined with `if (m_ArrayLevel[y][x] == 4)`. All we need to do is set `reachedGoal` to `true`. The update function of the `Engine` class will keep track of whether both characters (Thomas and Bob) have reached the goal simultaneously. We will write this code in the update function in just a minute.

Add the following line of code to the `detectCollisions` function:

```
        // All done, return, whether or
        // not a new level might be required
        return reachedGoal;
    }
```

The previous line of code returns the `reachedGoal` Boolean value so that the calling code can keep track and respond appropriately if both characters simultaneously reach the goal.

All we need to do now is call the `detectCollision` function once per character, per frame. Add the following highlighted code in the `Update.cpp` file within the `if (m_Playing)` block of code:

```
if (m_Playing)
{
    // Update Thomas
    m_Thomas.update(dtAsSeconds);

    // Update Bob
    m_Bob.update(dtAsSeconds);

    // Detect collisions and see if characters
    // have reached the goal tile
    // The second part of the if condition is only executed
    // when thomas is touching the home tile
```

```

    if (detectCollisions(m_Thomas) && detectCollisions(m_Bob))
    {
        // New level required
        m_NewLevelRequired = true;

        // Play the reach goal sound

    }
    else
    {
        // Run bobs collision detection
        detectCollisions(m_Bob);
    }

    // Count down the time the player has left
    m_TimeRemaining -= dtAsSeconds;

    // Have Thomas and Bob run out of time?
    if (m_TimeRemaining <= 0)
    {
        m_NewLevelRequired = true;
    }

} // End if playing

```

The previous code calls the `detectCollision` function and checks if both Bob and Thomas have simultaneously reached the goal. If they have, then the next level is prepared by setting `m_NewLevelRequired` to `true`.

You can run the game and walk on the platforms. You can reach the goal and start a new level. Also, for the first time, the jump button (*W* or Up arrow) will work.

If you reach the goal, then the next level will load. If you reach the goal of the last level, then the first level will load with a 10% reduced time limit. Of course, there is no visual feedback for the time or the current level because we haven't built a HUD yet. We will do so in the next chapter.

Many of the levels, however, require Thomas and Bob to work as a team. More specifically, Thomas and Bob need to be able to climb on each other's heads.

More collision detection

Add the following code just after you added the previous code in the `Update.cpp` file, within the `if (m_Playing)` section:

```
if (m_Playing)
{
    // Update Thomas
    m_Thomas.update(dtAsSeconds);

    // Update Bob
    m_Bob.update(dtAsSeconds);

    // Detect collisions and see if characters
    // have reached the goal tile
    // The second part of the if condition is only executed
    // when thomas is touching the home tile
    if (detectCollisions(m_Thomas) && detectCollisions(m_Bob))
    {
        // New level required
        m_NewLevelRequired = true;

        // Play the reach goal sound

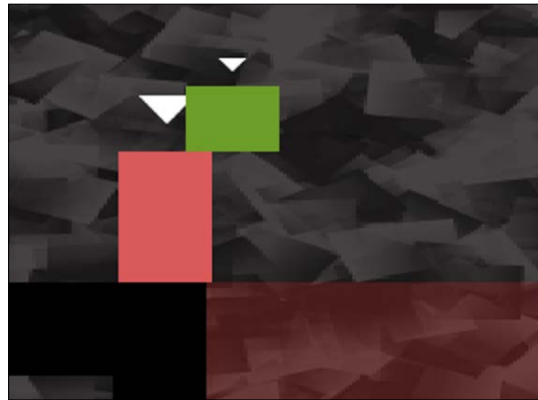
    }
    else
    {
        // Run bobs collision detection
        detectCollisions(m_Bob);
    }

    // Let bob and thomas jump on each others heads
    if (m_Bob.getFeet().intersects(m_Thomas.getHead()))
    {
        m_Bob.stopFalling(m_Thomas.getHead().top);
    }
    else if (m_Thomas.getFeet().intersects(m_Bob.getHead()))
    {
        m_Thomas.stopFalling(m_Bob.getHead().top);
    }

    // Count down the time the player has left
    m_TimeRemaining -= dtAsSeconds;
```

```
// Have Thomas and Bob run out of time?  
if (m_TimeRemaining <= 0)  
{  
    m_NewLevelRequired = true;  
}  
  
} // End if playing
```

You can run the game again and stand on the heads of Thomas and Bob to get to the hard-to-reach places that were previously not possible to get to:



Summary

There was quite a lot of code in this chapter. We learned how to read from a file and convert strings of text into `char` values and then into `int` values. Once we had a two-dimensional array of `int` values, we were able to populate a `VertexArray` instance to show the level on the screen. We then used the same two-dimensional array of `int` values to implement collision detection. We used rectangle intersection, just like we did in the *Zombie Arena* project, although this time, for more precision, we gave each character four collision zones – one each to represent their head, feet, left, and right-hand sides.

Now that the game is totally playable, we need to represent the state of the game (score and time) on the screen. In the next chapter, we will implement the HUD, along with some much more advanced sound effects than we have used so far.

17

Sound Spatialization and the HUD

In this chapter, we will be adding all the sound effects and the HUD. We have done this in two of the previous projects, but we will do things a bit differently this time. We will explore the concept of sound **spatialization** and how SFML makes this otherwise complicated concept nice and easy. In addition, we will build a HUD class to encapsulate our code that draws information to the screen.

We will complete these tasks in the following order.

- What is spatialization?
- How SFML handles spatialization
- Building a SoundManager class
- Deploying emitters
- Using the SoundManager class
- Building a HUD class
- Using the HUD class

What is spatialization?

Spatialization is the act of making something relative to the space it is a part of, or within. In our daily lives, everything in the natural world, by default, is spatialized. If a motorbike whizzes past from left to right, we will hear the sound grow from faint to loud from one side to the other. As it passes by, it will become more prominent in the other ear, before fading into the distance once more. If we were to wake up one morning and the world was no longer spatialized, it would be exceptionally weird.

If we can make our video games a little bit more like the real world, our players can become more immersed. Our zombie game would have been a lot more fun if the player could have heard them faintly in the distance and their inhuman wailing grew louder as they drew closer, from one direction or another.

It is probably obvious that the mathematics of spatialization will be complex. How do we calculate how loud a given sound will be in a specific speaker based on the distance and direction from the player (the hearer of the sound) to the object that is making the sound (the emitter)?

Fortunately, SFML does all the complicated processes for us. All we need to do is get familiar with a few technical terms and then we can start using SFML to spatialize our sound effects.

Emitters, attenuation, and listeners

We will need to be aware of a few pieces of information in order to give SFML what it needs to do its work. We will need to be aware of where the sound is coming from in our game world. This source of the sound is called an **emitter**. In a game, the emitter could be a zombie, a vehicle, or in the case of our current project, a fire tile. We have already been keeping track of the position of the objects in our game, so giving SFML the emitter's location will be quite straightforward.

The next factor we need to be aware of is **attenuation**. Attenuation is the rate at which a wave deteriorates. You could simplify that statement and make it specific to sound and say that attenuation is how quickly the sound reduces in volume. It isn't technically accurate, but it is a good enough description for the purposes of this chapter and our game.

The final factor that we need to consider is the **listener**. When SFML spatializes the sound, where is it spatializing it relative to; where are the "ears" of the game.? In most games, the logical thing to do is use the player character. In our game, we will use Thomas (our player character).

Handling spatialization using SFML

SFML has several functions that allow us to handle emitters, attenuation, and listeners. Let's take a look at them hypothetically and then we will write some code to add spatialized sound to our project for real.

We can set up a sound effect ready to be played, as we have done so often already, like this:

```
// Declare SoundBuffer in the usual way
SoundBuffer zombieBuffer;
// Declare a Sound object as-per-usual
Sound zombieSound;
// Load the sound from a file like we have done so often
zombieBuffer.loadFromFile("sound/zombie_growl.wav");
// Associate the Sound object with the Buffer
zombieSound.setBuffer(zombieBuffer);
```

We can set the position of the emitter using the `setPosition` function shown in the following code:

```
// Set the horizontal and vertical positions of the emitter
// In this case the emitter is a zombie
// In the Zombie Arena project we could have used
// getPosition().x and getPosition().y
// These values are arbitrary
float x = 500;
float y = 500;
zombieSound.setPosition(x, y, 0.0f);
```

As suggested in the comments of the previous code, how exactly we can obtain the coordinates of the emitter will probably be dependent on the type of game. As shown in the previous code, this would be quite simple in the Zombie Arena project. We will have a few challenges to overcome when we set the position in this project.

We can set the attenuation level as follows:

```
zombieSound.setAttenuation(15);
```

The actual attenuation level can be a little ambiguous. The effect that we want the player to get might be different from the accurate scientific formula that is used to reduce the volume over distance based on attenuation. Getting the right attenuation level is usually achieved by experimenting. The higher the level of attenuation, the quicker the sound level reduces to silence.

Also, we might want to set a zone around the emitter where the volume is not attenuated at all. We might do this if the feature isn't appropriate beyond a certain range or if we have many sound sources and don't want to "overdo" the feature. To do so, we can use the `setMinimumDistance` function as shown here:

```
zombieSound.setMinDistance(150);
```

With the previous line of code, attenuation would not be calculated until the listener is 150 pixels/units away from the emitter.

Some other useful functions from the SFML library include the `setLoop` function. This function will tell SFML to keep playing the sound over and over when `true` is passed in as a parameter, like in the following code:

```
zombieSound.setLoop(true);
```

The sound would continue to play until we end it with the following code:

```
zombieSound.stop();
```

From time to time, we would want to know the status of a sound (playing or stopped). We can achieve this with the `getStatus` function, as demonstrated in the following code:

```
if (zombieSound.getStatus() == Sound::Status::Stopped)
{
    // The sound is NOT playing
    // Take whatever action here
}

if (zombieSound.getStatus() == Sound::Status::Playing)
{
    // The sound IS playing
    // Take whatever action here
}
```

There is just one more aspect of using sound spatialization with SFML that we need to cover. The listener. Where is the listener? We can set the position of the listener with the following code:

```
// Where is the listener?
// How we get the values of x and y varies depending upon the game
// In the Zombie Arena game or the Thomas Was Late game
// We can use getPosition()
Listener::setPosition(m_Thomas.getPosition().x,
    m_Thomas.getPosition().y, 0.0f);
```

The preceding code will make all the sounds play relative to that location. This is just what we need for the distant roar of a fire tile or incoming zombie, but for regular sound effects like jumping, this is a problem. We could start handling an emitter for the location of the player, but SFML makes things simple for us. Whenever we want to play a "normal" sound, we simply call `setRelativeToListener`, as shown in the following code, and then play the sound in the exact same way we have done so far. Here is how we might play a "normal" unspatialized jump sound effect:

```
jumpSound.setRelativeToListener(true);  
jumpSound.play();
```

All we need to do is call `Listener::setPosition` again before we play any spatialized sounds.

We now have a wide repertoire of SFML sound functions, and we are ready to make some spatialized noise for real.

Building the SoundManager class

You might recall from the previous project that all the sound code took up quite a few lines of code. Now, consider that, with spatialization, it's going to get longer still. To keep our code manageable, we will code a class to manage all our sound effects being played. In addition, to help us with spatialization, we will add a function to the `Engine` class as well, but we will discuss that when we come to it, later in this chapter.

Coding SoundManager.h

Let's get started by coding and examining the header file.

Right-click **Header Files** in the **Solution Explorer** and select **Add | New Item...** In the **Add New Item** window, highlight (by left-clicking) **Header File (.h)** and then, in the **Name** field, type `SoundManager.h`. Finally, click the **Add** button. We are now ready to code the header file for the `SoundManager` class.

Add and examine the following code:

```
#pragma once  
#include <SFML/Audio.hpp>  
  
using namespace sf;  
  
class SoundManager  
{
```

```
private:
    // The buffers
    SoundBuffer m_FireBuffer;
    SoundBuffer m_FallInFireBuffer;
    SoundBuffer m_FallInWaterBuffer;
    SoundBuffer m_JumpBuffer;
    SoundBuffer m_ReachGoalBuffer;

    // The Sounds
    Sound m_Fire1Sound;
    Sound m_Fire2Sound;
    Sound m_Fire3Sound;
    Sound m_FallInFireSound;

    Sound m_FallInWaterSound;
    Sound m_JumpSound;
    Sound m_ReachGoalSound;

    // Which sound should we use next, fire 1, 2 or 3
    int m_NextSound = 1;

public:

    SoundManager();

    void playFire(Vector2f emitterLocation,
                  Vector2f listenerLocation);

    void playFallInFire();
    void playFallInWater();
    void playJump();
    void playReachGoal();
};
```

There is nothing tricky in the code we just added. There are five `SoundBuffer` objects and eight `Sound` objects. Three of the `Sound` objects will play the same `SoundBuffer`. This explains the reason for the different number of `Sound`/`SoundBuffer` objects. We do this so that we can have multiple roaring sound effects playing, with different spatialized parameters, simultaneously.

Note the `m_NextSound` variable, which will help us keep track of which of these simultaneous sounds we should use next.

There is a constructor, `SoundManager`, where we will set up all our sound effects, and there are five functions that will play the sound effects. Four of these functions simply play "normal" sound effects and their code will be simpler.

One of the functions, `playFire`, will handle the spatialized sound effects and will be a bit more in-depth. Notice the parameters of the `playFire` function. It receives a `Vector2f`, which is the location of the emitter and a second `Vector2f`, which is the location of the listener.

Coding the `SoundManager.cpp` file

Now, we can code the function definitions. The constructor and the `playFire` functions have a large amount of code, so we will look at them individually. The other functions are short and sweet, so we will handle them all at once.

Right-click **Source Files** in the **Solution Explorer** and select **Add | New Item....** In the **Add New Item** window, highlight (by left-clicking) **C++ File (.cpp)** and then, in the **Name** field, type `SoundManager.cpp`. Finally, click the **Add** button. We are now ready to code the `.cpp` file for the `SoundManager` class.

Coding the constructor

Add the following code for the include directives and the constructor to `SoundManager.cpp`:

```
#include "SoundManager.h"
#include <SFML/Audio.hpp>

using namespace sf;

SoundManager::SoundManager()
{
    // Load the sound in to the buffers
    m_FireBuffer.loadFromFile("sound/fire1.wav");
    m_FallInFireBuffer.loadFromFile("sound/fallinfire.wav");
    m_FallInWaterBuffer.loadFromFile("sound/fallinwater.wav");
    m_JumpBuffer.loadFromFile("sound/jump.wav");
    m_ReachGoalBuffer.loadFromFile("sound/reachgoal.wav");

    // Associate the sounds with the buffers
    m_Fire1Sound.setBuffer(m_FireBuffer);
    m_Fire2Sound.setBuffer(m_FireBuffer);
    m_Fire3Sound.setBuffer(m_FireBuffer);
```

```
m_FallInFireSound.setBuffer(m_FallInFireBuffer);
m_FallInWaterSound.setBuffer(m_FallInWaterBuffer);
m_JumpSound.setBuffer(m_JumpBuffer);
m_ReachGoalSound.setBuffer(m_ReachGoalBuffer);

// When the player is 50 pixels away sound is full volume
float minDistance = 150;
// The sound reduces steadily as the player moves further away
float attenuation = 15;


// Set all the attenuation levels
m_Fire1Sound.setAttenuation(attenuation);
m_Fire2Sound.setAttenuation(attenuation);
m_Fire3Sound.setAttenuation(attenuation);

// Set all the minimum distance levels
m_Fire1Sound.setMinDistance(minDistance);
m_Fire2Sound.setMinDistance(minDistance);
m_Fire3Sound.setMinDistance(minDistance);

// Loop all the fire sounds
// when they are played
m_Fire1Sound.setLoop(true);
m_Fire2Sound.setLoop(true);
m_Fire3Sound.setLoop(true);
}
```

In the previous code, we loaded five sound files into the five `SoundBuffer` objects. Next, we associated the eight `Sound` objects with one of the `SoundBuffer` objects. Notice that `m_Fire1Sound`, `m_Fire2Sound`, and `m_Fire3Sound` are all going to be playing from the same `SoundBuffer`, `m_FireBuffer`.

Next, we set the attenuation and minimum distance for the three fire sounds.

 The values of 150 and 15, respectively, were arrived at through experimentation. Once the game is running, it is advisable to experiment with these values by changing them around and seeing (or rather, hearing) the difference.

Finally, for the constructor, we use the `setLoop` function on each of the fire-related `Sound` objects. Now, when we call `play`, they will play continuously.

Coding the playFire function

Add the playFire function as follows. Then, we can discuss it:

```
void SoundManager::playFire(
    Vector2f emitterLocation, Vector2f listenerLocation)
{
    // Where is the listener? Thomas.
    Listener::setPosition(listenerLocation.x,
        listenerLocation.y, 0.0f);

    switch(m_NextSound)
    {

    case 1:
        // Locate/move the source of the sound
        m_Fire1Sound.setPosition(emitterLocation.x,
            emitterLocation.y, 0.0f);

        if (m_Fire1Sound.getStatus() == Sound::Status::Stopped)
        {
            // Play the sound, if its not already
            m_Fire1Sound.play();
        }
        break;

    case 2:
        // Do the same as previous for the second sound
        m_Fire2Sound.setPosition(emitterLocation.x,
            emitterLocation.y, 0.0f);

        if (m_Fire2Sound.getStatus() == Sound::Status::Stopped)
        {
            m_Fire2Sound.play();
        }
        break;

    case 3:
        // Do the same as previous for the third sound
        m_Fire3Sound.setPosition(emitterLocation.x,
            emitterLocation.y, 0.0f);

        if (m_Fire3Sound.getStatus() == Sound::Status::Stopped)
        {
```



```
        m_Fire3Sound.play();
    }
    break;
}

// Increment to the next fire sound

m_NextSound++;

// Go back to 1 when the third sound has been started
if (m_NextSound > 3)
{
    m_NextSound = 1;
}
}
```

The first thing we do is call `Listener::setPosition` and set the location of the listener based on the `Vector2f` that is passed in as a parameter.

Next, the code enters a switch block that tests the value of `m_NextSound`. Each of the case statements does the exact same thing but to either `m_Fire1Sound`, `m_Fire2Sound`, or `m_Fire3Sound`.

In each of the case blocks, we set the position of the emitter using the passed in parameter with the `setPosition` function. The next part of the code in each case block checks whether the sound is currently stopped, and, if it is, plays the sound. Soon, we will see how we arrive at the positions for the emitter and listener that are passed into this function.

The final part of the `playFire` function increments `m_NextSound` and ensures that it can only be equal to 1, 2, or 3, as required by the switch block.

Coding the rest of the SoundManager functions

Add these four simple functions:

```
void SoundManager::playFallInFire()
{
    m_FallInFireSound.setRelativeToListener(true);
    m_FallInFireSound.play();
}

void SoundManager::playFallInWater()
{
    m_FallInWaterSound.setRelativeToListener(true);
```

```
        m_FallInWaterSound.play();
    }

    void SoundManager::playJump()
    {
        m_JumpSound.setRelativeToListener(true);
        m_JumpSound.play();
    }

    void SoundManager::playReachGoal()
    {
        m_ReachGoalSound.setRelativeToListener(true);
        m_ReachGoalSound.play();
    }
```

The `playFallInFire`, `playFallInWater`, and `playReachGoal` functions do just two things. First, they each call `setRelativeToListener` so that the sound effect is not spatialized, making the sound effect "normal", not directional, and then they call `play` on the appropriate `Sound` object.

That concludes the `SoundManager` class. Now, we can use it in the `Engine` class.

Adding SoundManager to the game engine

Open the `Engine.h` file and add an instance of the new `SoundManager` class, as shown in the following highlighted code:

```
#pragma once
#include <SFML/Graphics.hpp>
#include "TextureHolder.h"
#include "Thomas.h"
#include "Bob.h"
#include "LevelManager.h"
#include "SoundManager.h"

using namespace sf;

class Engine
{
```

```
private:
    // The texture holder
    TextureHolder th;

    // Thomas and his friend, Bob
    Thomas m_Thomas;
    Bob m_Bob;

    // A class to manage all the levels
    LevelManager m_LM;

    // Create a SoundManager
    SoundManager m_SM;

    const int TILE_SIZE = 50;
    const int VERTS_IN_QUAD = 4;
```

At this point, we could use `m_SM` to call the various `play...` functions. Unfortunately, there is still a bit more work to be done in order to manage the locations of the emitters (fire tiles).

Populating the sound emitters

Open the `Engine.h` file and add a new prototype for a `populateEmitters` function and a new STL vector of `Vector2f` objects:

```
...
...
...
// Run will call all the private functions
bool detectCollisions(PlayableCharacter& character);

// Make a vector of the best places to emit sounds from
void populateEmitters(vector<Vector2f>& vSoundEmitters,
    int** arrayLevel);

// A vector of Vector2f for the fire emitter locations
vector<Vector2f> m_FireEmitters;

public:
    ...
    ...
    ...
```

The `populateEmitters` function takes a vector of `Vector2f` objects as a parameter, as well as a pointer to pointer to `int` (a two-dimensional array). The vector will hold the location of each emitter in a level. The array is the two-dimensional array that holds the layout of a level.

Coding the `populateEmitters` function

The job of the `populateEmitters` function is to scan through all the elements of `arrayLevel` and decide where to put the emitters. It will store its results in `m_FireEmitters`.

Right-click **Source Files** in the **Solution Explorer** and select **Add | New Item....** In the **Add New Item** window, highlight (by left-clicking) **C++ File (.cpp)** and then, in the **Name** field, type `PopulateEmitters.cpp`. Finally, click the **Add** button. Now, we can code the new function, `populateEmitters`.

Add the code in its entirety. Be sure to study the code as you do, so that we can discuss it:

```
#include "Engine.h"

using namespace sf;
using namespace std;

void Engine::populateEmitters(
    vector <Vector2f>& vSoundEmitters,
    int** arrayLevel)
{
    // Make sure the vector is empty
    vSoundEmitters.empty();

    // Keep track of the previous emitter
    // so we don't make too many
    FloatRect previousEmitter;

    // Search for fire in the level
    for (int x = 0; x < (int)m_LM.getLevelSize().x; x++)
    {
        for (int y = 0; y < (int)m_LM.getLevelSize().y; y++)
        {
            if (arrayLevel[y][x] == 2) // fire is present
            {
                // Skip over any fire tiles too
            }
        }
    }
}
```

```
        // near a previous emitter
        if (!FloatRect(x * TILE_SIZE,
            y * TILE_SIZE,
            TILE_SIZE,
            TILE_SIZE).intersects(previousEmitter))
        {
            // Add the coordinates of this water block
            vSoundEmitters.push_back(
                Vector2f(x * TILE_SIZE, y * TILE_SIZE));

            // Make a rectangle 6 blocks x 6 blocks,
            // so we don't make any more emitters
            // too close to this one
            previousEmitter.left = x * TILE_SIZE;
            previousEmitter.top = y * TILE_SIZE;
            previousEmitter.width = TILE_SIZE * 6;
            previousEmitter.height = TILE_SIZE * 6;
        }
    }

    }

    }
    return;
}
```

Some of the code might appear complex at first glance. Understanding the technique we are using to choose where an emitter will be makes this simpler. In our levels, there are large blocks of fire tiles. For example, in one of the levels, there are more than 30 fire tiles together in a group. The code makes sure that there is only one emitter within a given rectangle. This rectangle is stored in `previousEmitter` and is 300 pixels by 300 pixels (`TILE_SIZE * 6`).

The code sets up a nested `for` loop that loops through `arrayLevel`, looking for fire tiles. When it finds one, it makes sure that it does not intersect with `previousEmitter`. Only then does it use the `pushBack` function to add another emitter to `vSoundEmitters`. After doing so, it also updates `previousEmitter` to avoid getting large clusters of sound emitters.

Let's make some noise.

Playing sounds

Open the `LoadLevel.cpp` file and add the call to the new `populateEmitters` function, as highlighted in the following code:

```
void Engine::loadLevel()
{
    m_Playing = false;

    // Delete the previously allocated memory
    for (int i = 0; i < m_LM.getLevelSize().y; ++i)
    {
        delete[] m_ArrayLevel[i];
    }
    delete[] m_ArrayLevel;

    // Load the next 2d array with the map for the level
    // And repopulate the vertex array as well
    m_ArrayLevel = m_LM.nextLevel(m_VAlevel);

    // Prepare the sound emitters
populateEmitters(m_FireEmitters, m_ArrayLevel);

    // How long is this new time limit
    m_TimeRemaining = m_LM.getTimeLimit();

    // Spawn Thomas and Bob
    m_Thomas.spawn(m_LM.getStartPosition(), GRAVITY);
    m_Bob.spawn(m_LM.getStartPosition(), GRAVITY);

    // Make sure this code isn't run again
    m_NewLevelRequired = false;
}
```

The first sound to add is the jump sound. We remember that the keyboard handling code is in the pure virtual functions within both the `Bob` and `Thomas` classes and that the `handleInput` function returns `true` when a jump has been successfully initiated.

Open the `Input.cpp` file and add the following highlighted lines of code to play a jump sound when `Thomas` or `Bob` successfully begins a jump:

```
// Handle input specific to Thomas
if (m_Thomas.handleInput())
{
```

```
        // Play a jump sound
        m_SM.playJump();
    }

    // Handle input specific to Bob
    if (m_Bob.handleInput())
    {
        // Play a jump sound
        m_SM.playJump();
    }
```

Open the `Update.cpp` file and add the following highlighted line of code to play a success sound when Thomas and Bob have simultaneously reached the goal for the current level:

```
    // Detect collisions and see if characters have reached the goal tile
    // The second part of the if condition is only executed
    // when Thomas is touching the home tile
    if (detectCollisions(m_Thomas) && detectCollisions(m_Bob))
    {
        // New level required
        m_NewLevelRequired = true;

        // Play the reach goal sound
        m_SM.playReachGoal();
    }
    else
    {
        // Run Bobs collision detection
        detectCollisions(m_Bob);
    }
```

Also, within the `Update.cpp` file, we will add code to loop through the `m_FireEmitters` vector and decide when we need to call the `playFire` function of the `SoundManager` class.

Look closely at the small amount of context around the new highlighted code. It is essential to add this code in exactly the right place:

```
    } // End if playing

    // Check if a fire sound needs to be played
    vector<Vector2f>::iterator it;
```

```
// Iterate through the vector of Vector2f objects
for (it = m_FireEmitters.begin(); it != m_FireEmitters.end(); it++)
{
    // Where is this emitter?
    // Store the location in pos
    float posX = (*it).x;
    float posY = (*it).y;

    // is the emitter near the player?
    // Make a 500 pixel rectangle around the emitter
    FloatRect localRect(posX - 250, posY - 250, 500, 500);

    // Is the player inside localRect?
    if (m_Thomas.getPosition().intersects(localRect))
    {
        // Play the sound and pass in the location as well
        m_SM.playFire(Vector2f(posX, posY), m_Thomas.getCenter());
    }
}

// Set the appropriate view around the appropriate character
```

The preceding code is a bit like collision detection for sound. Whenever Thomas strays within a 500 by 500-pixel rectangle surrounding a fire emitter, the `playFire` function is called, passing in the coordinates of the emitter and of Thomas. The `playFire` function does the rest of the work and plays a spatialized, looping sound effect.

Open the `DetectCollisions.cpp` file, find the appropriate place, and add the following highlighted code. The two highlighted lines of code trigger the sound effect when either character falls into a water or fire tile:

```
// Has character been burnt or drowned?
// Use head as this allows him to sink a bit
if (m_ArrayLevel[y][x] == 2 || m_ArrayLevel[y][x] == 3)
{
    if (character.getHead().intersects(block))
    {
        character.spawn(m_LM.getStartPosition(), GRAVITY);
        // Which sound should be played?
        if (m_ArrayLevel[y][x] == 2) // Fire, ouch!
        {
            // Play a sound
            m_SM.playFallInFire();
        }
    }
}
```



```
    }
    else // Water
    {
        // Play a sound
        m_SM.playFallInWater();
    }
}
```

Playing the game will now allow you to hear all the sounds, including cool spatialization, when you're near a fire tile.

Implementing the HUD class

The HUD is super-simple and not really anything different compared to the Zombie Arena project. What we will do that is different is wrap all the code up in a new HUD class. If we declare all the `Font`, `Text`, and other variables as members of this new class, we can then initialize them in the constructor and provide getter functions to all their values. This will keep the `Engine` class clear from loads of declarations and initializations.

Coding HUD.h

First, we will code the `HUD.h` file with all the member variables and function declarations. Right-click **Header Files** in the **Solution Explorer** and select **Add | New Item...** In the **Add New Item** window, highlight (by left-clicking) **Header File (.h)** and then, in the **Name** field, type `HUD.h`. Finally, click the **Add** button. We are now ready to code the header file for the HUD class.

Add the following code to `HUD.h`:

```
#pragma once
#include <SFML/Graphics.hpp>

using namespace sf;

class Hud
{
private:
    Font m_Font;
    Text m_StartText;
    Text m_TimeText;
    Text m_LevelText;
```

```

public:
    Hud();
    Text getMessage();
    Text getLevel();
    Text getTime();

    void setLevel(String text);
    void setTime(String text);
};

```

In the preceding code, we added one `Font` instance and three `Text` instances. The `Text` objects will be used to show a message prompting the user to start, the time remaining, and the current level number.

The public functions are more interesting. First, there is the constructor where most of the code will go. The constructor will initialize the `Font` and `Text` objects, as well as position them on the screen relative to the current screen resolution.

The three getter functions, `getMessage`, `getLevel`, and `getTime`, will return a `Text` object to the calling code so that it can draw them to the screen.

The `setLevel` and `setTime` functions will be used to update the text shown in `m_LevelText` and `m_TimeText`, respectively.

Now, we can code all the definitions for the functions we have just declared.

Coding the HUD.cpp file

Right-click **Source Files** in the **Solution Explorer** and select **Add | New Item....** In the **Add New Item** window, highlight (by left-clicking) **C++ File (.cpp)** and then, in the **Name** field, type `HUD.cpp`. Finally, click the **Add** button. We are now ready to code the `.cpp` file for the `HUD` class.

Add the include directives and the following code. Then, we will discuss it:

```

#include "Hud.h"

Hud::Hud()
{
    Vector2u resolution;
    resolution.x = VideoMode::getDesktopMode().width;
    resolution.y = VideoMode::getDesktopMode().height;

    // Load the font
    m_Font.loadFromFile("fonts/Roboto-Light.ttf");
}

```

```
// when Paused
m_StartText.setFont(m_Font);
m_StartText.setCharacterSize(100);
m_StartText.setFillColor(Color::White);
m_StartText.setString("Press Enter when ready!");

// Position the text
FloatRect textRect = m_StartText.getLocalBounds();

m_StartText.setOrigin(textRect.left +
    textRect.width / 2.0f,
    textRect.top +
    textRect.height / 2.0f);

m_StartText.setPosition(
    resolution.x / 2.0f, resolution.y / 2.0f);

// Time
m_TimeText.setFont(m_Font);
m_TimeText.setCharacterSize(75);
m_TimeText.setFillColor(Color::White);
m_TimeText.setPosition(resolution.x - 150, 0);
m_TimeText.setString("-----");

// Level
m_LevelText.setFont(m_Font);
m_LevelText.setCharacterSize(75);
m_LevelText.setFillColor(Color::White);
m_LevelText.setPosition(25, 0);
m_LevelText.setString("1");
}
```

First, we store the horizontal and vertical resolution in a `Vector2u` called `resolution`. Next, we load the font from the `fonts` directory that we added back in *Chapter 14, Abstraction and Code Management – Making Better Use of OOP*.

The next four lines of code set the font, the color, the size, and the text of `m_StartText`. The block of code after this captures the size of the rectangle that wraps `m_StartText` and performs a calculation to work out how to position it centrally on the screen. If you want a more thorough explanation of this part of the code, then refer to *Chapter 3, C++ Strings and SFML Time – Player Input and HUD*.

In the final two blocks of code in the constructor, the font, text size, color, position, and actual text for `m_TimeText` and `m_LevelText` are set. In a moment, we will see that these two `Text` objects will be updatable through two setter functions, whenever it is required.

Add the following getter and setter functions immediately underneath the code we have just added:

```
Text Hud::getMessage()
{
    return m_StartText;
}

Text Hud::getLevel()
{
    return m_LevelText;
}

Text Hud::getTime()
{
    return m_TimeText;
}

void Hud::setLevel(String text)
{
    m_LevelText.setString(text);
}

void Hud::setTime(String text)
{
    m_TimeText.setString(text);
}
```

The first three functions in the previous code simply return the appropriate `Text` object, that is, `m_StartText`, `m_LevelText`, or `m_TimeText`. We will use these functions shortly when we draw the HUD to the screen. The final two functions, `setLevel` and `setTime`, use the `setString` functions to update the appropriate `Text` object with the value that will be passed in from the `update` function of the `Engine` class, every 500 frames.

With all that done, we can put the HUD class to work in our game engine.

Using the HUD class

Open `Engine.h`, add an include for our new class, declare an instance of the new HUD class, and declare and initialize two new member variables that will keep track of how often we update the HUD. As we learned in the previous projects, we don't need to update the HUD every frame.

Add the following highlighted code to `Engine.h`:

```
#pragma once
#include <SFML/Graphics.hpp>
#include "TextureHolder.h"
#include "Thomas.h"
#include "Bob.h"
#include "LevelManager.h"
#include "SoundManager.h"
#include "HUD.h"

using namespace sf;

class Engine
{
private:
    // The texture holder
    TextureHolder th;

    // Thomas and his friend, Bob
    Thomas m_Thomas;
    Bob m_Bob;

    // A class to manage all the levels
    LevelManager m_LM;

    // Create a SoundManager
    SoundManager m_SM;

    // The Hud
    Hud m_Hud;
    int m_FramesSinceLastHUDUpdate = 0;
    int m_TargetFramesPerHUDUpdate = 500;

    const int TILE_SIZE = 50;
```

Next, we need to add some code to the `update` function of the `Engine` class. Open `Update.cpp` and add the following highlighted code to update the HUD once every 500 frames:

```
// Set the appropriate view around the appropriate character
if (m_SplitScreen)
{
    m_LeftView.setCenter(m_Thomas.getCenter());
    m_RightView.setCenter(m_Bob.getCenter());
}
else
{
    // Centre full screen around appropriate character
    if (m_Character1)
    {
        m_MainView.setCenter(m_Thomas.getCenter());
    }
    else
    {
        m_MainView.setCenter(m_Bob.getCenter());
    }
}

// Time to update the HUD?
// Increment the number of frames since
// the last HUD calculation
m_FramesSinceLastHUDUpdate++;

// Update the HUD every m_TargetFramesPerHUDUpdate frames
if (m_FramesSinceLastHUDUpdate > m_TargetFramesPerHUDUpdate)
{
    // Update game HUD text
    stringstream ssTime;
    stringstream ssLevel;

    // Update the time text
    ssTime << (int)m_TimeRemaining;
    m_Hud.setTime(ssTime.str());

    // Update the level text
    ssLevel << "Level:" << m_LM.getCurrentLevel();
    m_Hud.setLevel(ssLevel.str());
}
```

```
        m_FramesSinceLastHUDUpdate = 0;
    }
} // End of update function
```

In the preceding code, `m_FramesSinceLastUpdate` is incremented each frame. When `m_FramesSinceLastUpdate` exceeds `m_TargetFramesPerHUDUpdate`, then execution enters the `if` block. Inside the `if` block, we use `stringstream` objects to update our `Text`, like we did in the previous projects. In this project, we are using the `HUD` class, so we call the `setTime` and `setLevel` functions by passing in the current values that the `Text` objects need to be set to.

The final step in the `if` block is to set `m_FramesSinceLastUpdate` back to zero so it can start counting toward the next update.

Finally, open the `Draw.cpp` file and add the following highlighted code to draw the HUD each frame:

```
else
{
    // Split-screen view is active

    // First draw Thomas' side of the screen

    // Switch to background view
    m_Window.setView(m_BGLeftView);
    // Draw the background
    m_Window.draw(m_BackgroundSprite);
    // Switch to m_LeftView
    m_Window.setView(m_LeftView);

    // Draw the Level
    m_Window.draw(m_VALevel, &m_TextureTiles);

    // Draw thomas
    m_Window.draw(m_Bob.getSprite());

    // Draw thomas
    m_Window.draw(m_Thomas.getSprite());

    // Now draw Bob's side of the screen

    // Switch to background view
    m_Window.setView(m_BGRightView);
    // Draw the background
```

```
m_Window.draw(m_BackgroundSprite);
// Switch to m_RightView
m_Window.setView(m_RightView);

// Draw the Level
m_Window.draw(m_VAlevel, &m_TextureTiles);

// Draw thomas
m_Window.draw(m_Thomas.getSprite());

// Draw bob
m_Window.draw(m_Bob.getSprite());

}

// Draw the HUD
// Switch to m_HudView
m_Window.setView(m_HudView);
m_Window.draw(m_Hud.getLevel());
m_Window.draw(m_Hud.getTime());
if (!m_Playing)
{
    m_Window.draw(m_Hud.getMessage());
}

// Show everything we have just drawn
m_Window.display();
} // End of draw
```

The preceding code draws the HUD by using the getter functions from the HUD class. Notice that the call to draw the message that prompts the player to start is only used when the game is not currently playing (`!m_Playing`).

Run the game and play a few levels to see the time tick down and the levels tick up. When you get back to level 1 again, notice that you have 10% less time than before.

Summary

In this chapter, we have explored sound spatialization. Our "Thomas Was Late" game is not only fully playable now, but we have added directional sound effects and a simple but informative HUD. We can also add new levels with ease. At this point, we could call it a day.

It would be nice to add a bit more sparkle. In the next chapter, we will look into two gaming concepts. First, we will look at particle systems, which are how we can handle things such as explosions or other special effects. To achieve this, we will need to learn a bit more C++. Due to this, the topic of multiple inheritance will be introduced.

After that, we will add the final flourish to the game when we learn about OpenGL and the programmable graphics pipeline. We will then be able to dip our toes into the **GLSL** language, which allows us to write code that executes directly on the GPU so that we can create some special effects.

18

Particle Systems and Shaders

In this chapter, we will look at what a particle system is and then go ahead and code one into our game. We will scratch the surface of the topic of OpenGL shaders and see how writing code in another language (**GLSL**), that can be run directly on the graphics card, can lead to smooth graphical effects that might otherwise be impossible. As usual, we will also use our new skills and knowledge to enhance the current project.

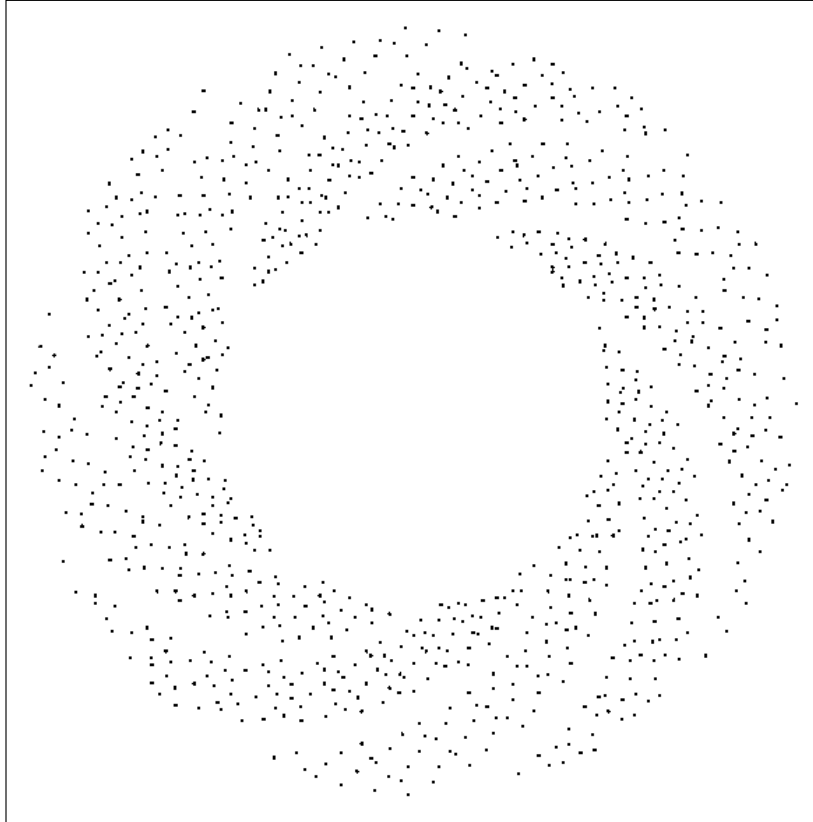
In this chapter, we will cover the following topics:

- Building a particle system
- OpenGL shaders and GLSL
- Using shaders in the Thomas Was Late game

Building a particle system

Before we start coding, it will be helpful to see exactly what it is that we are trying to achieve.

Take a look at the following diagram:



The previous illustration is a screenshot of the particle effect on a plain background. We will use this effect in our game. We will spawn one of these effects each time the player dies.

The way we achieve this effect is as follows:

1. First, we spawn 1,000 dots (particles), one on top of the other, at a chosen pixel position.
2. Each frame of the game moves each of the 1,000 particles outwards at a predetermined but random speed and angle.
3. Repeat step two for two seconds and then make the particles disappear.

We will use a `VertexArray` to draw all the dots and the primitive type of `Point` to represent each particle visually. Furthermore, we will inherit from the SFML `Drawable` class so that our particle system can take care of drawing itself.

Coding the Particle class

The `Particle` class will be a simple class that represents just one particle from a thousand particles. Let's get coding.

Coding Particle.h

Right-click **Header Files** in the **Solution Explorer** and select **Add | New Item....** In the **Add New Item** window, highlight (by left-clicking) **Header File (.h)** and then, in the **Name** field, type `Particle.h`. Finally, click the **Add** button. We are now ready to code the header file for the `Particle` class.

Add the following code to the `Particle.h` file:

```
#pragma once
#include <SFML/Graphics.hpp>

using namespace sf;

class Particle
{
private:
    Vector2f m_Position;
    Vector2f m_Velocity;

public:
    Particle(Vector2f direction);

    void update(float dt);

    void setPosition(Vector2f position);

    Vector2f getPosition();
};
```

In the preceding code, we have two `Vector2f` objects. One will represent the horizontal and vertical coordinate of the particle, while the other will represent the horizontal and vertical speed.



When you have a rate of change (speed) in more than one direction, the combined values also define a direction. This is called **velocity**. Hence, `Vector2f` is called `m_Velocity`.

We also have several public functions. First is the constructor. It takes a `Vector2f` and uses this to let it know which direction/velocity this particle will have. This implies that the system, not the particle itself, will be choosing the velocity.

Next is the `update` function, which takes the time the previous frame has taken. We will use this to move the particle by precisely the correct amount.

The final two functions, `setPosition` and `getPosition`, are used to move the particle in position and find out its position, respectively.

All of these functions will make complete sense when we code them.

Coding the Particle.cpp file

Right-click **Source Files** in the **Solution Explorer** and select **Add | New Item....** In the **Add New Item** window, highlight (by left-clicking) **C++ File (.cpp)** and then, in the **Name** field, type `Particle.cpp`. Finally, click the **Add** button. We are now ready to code the `.cpp` file for the `Particle` class.

Add the following code to `Particle.cpp`:

```
#include "Particle.h"

Particle::Particle(Vector2f direction)
{
    // Determine the direction

    m_Velocity.x = direction.x;
    m_Velocity.y = direction.y;
}

void Particle::update(float dtAsSeconds)
{
    // Move the particle
    m_Position += m_Velocity * dtAsSeconds;
}

void Particle::setPosition(Vector2f position)
{
    m_Position = position;
}

Vector2f Particle::getPosition()
```

```
{  
    return m_Position;  
}
```

All of these functions use concepts we have seen before. The constructor sets up the `m_Velocity.x` and `m_Velocity.y` values using the passed in `Vector2f` object.

The `update` function moves the horizontal and vertical positions of the particle by multiplying `m_Velocity` by the elapsed time (`dtAsSeconds`). Notice how, to achieve this, we simply add the two `Vector2f` objects together. There is no need to perform calculations for both the `x` and `y` members separately.

The `setPosition` function, as we explained previously, initializes the `m_Position` object with the passed in values. The `getPosition` function returns `m_Position` to the calling code.

We now have a fully functioning `Particle` class. Next, we will code a `ParticleSystem` class to spawn and control the particles.

Coding the ParticleSystem class

The `ParticleSystem` class does most of the work for our particle effects. It is this class that we will create an instance of in the `Engine` class. Before we do, however, let's talk a little bit more about OOP and the SFML `Drawable` class.

Exploring SFML's Drawable class and OOP

The `Drawable` class has just one function. It has no variables either. Furthermore, its one and only function is pure virtual. This means that, if we inherit from `Drawable`, we must implement its one and only function. The purpose, as a reminder from *Chapter 14, Abstraction and Code Management – Making Better Use of OOP*, is that we can then use our class that inherits from `Drawable` as a polymorphic type. Put more simply, anything that SFML allows us to do with a `Drawable` object, we will be able to do with our class that inherits from it. The only requirement is that we must provide a definition for the pure virtual function, `draw`.

Some classes that inherit from `Drawable` already include `Sprite` and `VertexArray` (among others). Whenever we have used `Sprite` or `VertexArray`, we passed them to the `draw` function of the `RenderWindow` class.

The reason that we have been able to draw every object we have ever drawn, in this entire book, is because they have all been inherited from `Drawable`. We can use this knowledge to our advantage.

We can inherit from `Drawable` with any object we like, as long as we implement the pure virtual draw function. This is also a straightforward process. Consider a hypothetical `SpaceShip` class. The header file (`SpaceShip.h`) of the `SpaceShip` class that inherits from `Drawable` would look like this:

```
class SpaceShip : public Drawable
{
private:
    Sprite m_Sprite;
    // More private members

public:

    virtual void draw(RenderTarget& target,
        RenderStates states) const;

    // More public members

};
```

In the previous code, we can see the pure virtual draw function and a `Sprite` instance. Notice there is no way to access the private `Sprite` outside of the class – not even a `getSprite` function!

The `SpaceShip.cpp` file would look something like this:

```
void SpaceShip::SpaceShip
{
    // Set up the spaceship
}

void SpaceShip::draw(RenderTarget& target, RenderStates states) const
{
    target.draw(m_Sprite, states);
}

// Any other functions
```

In the previous code, notice the simple implementation of the draw function. The parameters are beyond the scope of this book. Just note that the `target` parameter is used to call `draw` and passes in `m_Sprite` as well as `states`, the other parameter.



While it is not necessary to understand the parameters to take full advantage of `Drawable`, in the context of this book, you might be intrigued. You can read more about SFML `Drawable` on the SFML website here: <https://www.sfm1-dev.org/tutorials/2.5/graphics-vertex-array.php>.

In the main game loop, we could now treat a `SpaceShip` instance as if it were a `Sprite` or any other class that inherits from `Drawable`, like so:

```
SpaceShip m_SpaceShip;
// create other objects here
// ...

// In the draw function
// Rub out the last frame
m_Window.clear(Color::Black);

// Draw the spaceship
m_Window.draw(m_SpaceShip);
// More drawing here
// ...

// Show everything we have just drawn
m_Window.display();
```

It is because `SpaceShip` **is a** `Drawable` that we can treat it like it was a `Sprite` or `VertexArray` and, because we overrode the pure virtual `draw` function, everything just works as we want it to. You will use this approach in this chapter to draw the particle system.

While we are on the subject of OOP, let's look at an alternative way of encapsulating the drawing code into the game object that we will use in the next project.

An alternative to inheriting from `Drawable`

It is also possible to keep all the drawing functionality within the class that is the object to be drawn by implementing our own function, within our class, perhaps by using the following code:

```
void drawThisObject(RenderWindow window)
{
    window.draw(m_Sprite)
}
```


The previous code assumes that `m_Sprite` represents the visual appearance of the current class we are drawing, as it has throughout this and the previous project. Assuming that the instance of the class that contains the `drawThisObject` function is called `playerHero` and further assuming we have an instance of `RenderWindow` called `m_Window`, we could then draw the object from the main game loop with this code:

```
playerHero.draw(m_Window);
```

In this solution, we pass the `RenderWindow`, `m_Window`, into the `drawThisObject` function as a parameter. The `drawThisObject` function then uses `RenderWindow` to draw the `Sprite`, `m_Sprite`.

If we have a more complicated set of game objects, then passing a reference of `RenderWindow` to the object to be drawn, each frame, so it can draw itself, is a good tactic.

We will use this tactic in the final project of this book, which we will start in the next chapter. Let's finish the particle system by coding the `ParticleSystem` class, which will inherit from `Drawable`.

Coding ParticleSystem.h

Right-click **Header Files** in the **Solution Explorer** and select **Add | New Item....** In the **Add New Item** window, highlight (by left-clicking) **Header File (.h)** and then, in the **Name** field, type `ParticleSystem.h`. Finally, click the **Add** button. We are now ready to code the header file for the `ParticleSystem` class.

Add the code for the `ParticleSystem` class to `ParticleSystem.h`:

```
#pragma once
#include <SFML/Graphics.hpp>
#include "Particle.h"

using namespace sf;
using namespace std;

class ParticleSystem : public Drawable
{
private:
    vector<Particle> m_Particles;
    VertexArray m_Vertices;
    float m_Duration;
    bool m_IsRunning = false;

public:
```

```

    virtual void draw(RenderTarget& target,
        RenderStates states) const;

    void init(int count);

    void emitParticles(Vector2f position);

    void update(float elapsed);

    bool running();


};

```

Let's go through this a bit at a time. First, notice that we are inheriting from SFML's `Drawable` class. This is what will allow us to pass our `ParticleSystem` instance to `m_Window.draw`, because `ParticleSystem` **is a** `Drawable`. And, since we inherit from `Drawable`, we can override the `draw` function using the same function signature as the `Drawable` class uses internally. Shortly, when we use the `ParticleSystem` class, we will see the following code.

```
m_Window.draw(m_PS);
```

The `m_PS` object is an instance of our `ParticleSystem` class, and we will pass it directly to the `draw` function of the `RenderWindow` class, just like we have done for the `Sprite`, `VertexArray`, and `RectangleShape` instances. All this is made possible by the power of inheritance and polymorphism.

[ Don't add the `m_Window.draw...` code just yet; we have a bit more work to do first.]

There is a vector named `m_Particles` of the `Particle` type. This vector will hold each and every instance of `Particle`. Next, we have a `VertexArray` called `m_Vertices`. This will be used to draw all the particles in the form of a whole bunch of `Point` primitives.

The `m_Duration`, `float` variable is how long each effect will last. We will initialize it in the constructor function.

The `m_IsRunning` Boolean variable will be used to indicate whether the particle system is currently in use or not.

Next, in the public section, we have the pure virtual function, `draw`, that we will soon implement to handle what happens when we pass our instance of `ParticleSystem` to `m_Window.draw`.

The `init` function will prepare the `VertexArray` and the `vector`. It will also initialize all the `Particle` objects (held by the `vector`) with their velocities and initial positions.

The `update` function will loop through each `Particle` instance in the `vector` and call their individual `update` functions.

The `running` function provides access to the `m_IsRunning` variable so that the game engine can query whether the `ParticleSystem` is currently in use.

Let's code the function definitions to see what goes on inside `ParticleSystem`.

Coding the `ParticleSystem.cpp` file

Right-click **Source Files** in the **Solution Explorer** and select **Add | New Item....** In the **Add New Item** window, highlight (by left-clicking) **C++ File (.cpp)** and then, in the **Name** field, type `ParticleSystem.cpp`. Finally, click the **Add** button. We are now ready to code the `.cpp` file for the `ParticleSystem` class.

We will split this file into five sections so that we can code and discuss it in more detail. Add the first section of code, as follows:

```
#include <SFML/Graphics.hpp>
#include "ParticleSystem.h"

using namespace sf;
using namespace std;

void ParticleSystem::init(int numParticles)
{
    m_Vertices.setPrimitiveType(Points);
    m_Vertices.resize(numParticles);

    // Create the particles

    for (int i = 0; i < numParticles; i++)
    {
        srand(time(0) + i);
        float angle = (rand() % 360) * 3.14f / 180.f;
        float speed = (rand() % 600) + 600.f;

        Vector2f direction;
```

```


        direction = Vector2f(cos(angle) * speed,
                               sin(angle) * speed);

        m_Particles.push_back(Particle(direction));
    }
}

```

After the necessary includes, we have the definition of the `init` function. We call `setPrimitiveType` with `Points` as the argument so that `m_VertexArray` knows what type of primitives it will be dealing with. We resize `m_Vertices` with `numParticles`, which was passed in to the `init` function when it was called.

The `for` loop creates random values for speed and angle. It then uses trigonometric functions to convert those values into a vector which is stored in the `Vector2f`, `direction`.

 If you want to know more about how the trigonometric functions (`cos` and `sin`) convert angles and speeds into a vector, then you can take a look at this article series: <http://gamecodeschool.com/essentials/calculating-heading-in-2d-games-using-trigonometric-functions-part-1/>.

The last thing that happens in the `for` loop (and the `init` function) is that the vector is passed into the `Particle` constructor. The new `Particle` instance is stored in `m_Particles` using the `push_back` function. Therefore, a call to `init` with a value of 1000 would mean we have 1,000 instances of `Particle`, with random velocity, stashed away in `m_Particles`, just waiting to blow!

Next, add the update function to `ParticleSystem.cpp`:

```

void ParticleSystem::update(float dt)
{
    m_Duration -= dt;
    vector<Particle>::iterator i;
    int currentVertex = 0;

    for (i = m_Particles.begin(); i != m_Particles.end(); i++)
    {
        // Move the particle
        (*i).update(dt);
    }
}

```

```
        // Update the vertex array
        m_Vertices[currentVertex++].position = i->getPosition();
    }

    if (m_Duration < 0)
    {
        m_IsRunning = false;
    }
}
```

The update function is simpler than it looks at first glance. First of all, `m_Duration` is reduced by the passed in time, `dt`. This is so we know when the two seconds have elapsed. A vector iterator, `i`, is declared for use with `m_Particles`.

The `for` loop goes through each of the `Particle` instances in `m_Particles`. For each one, it calls its update function and passes in `dt`. Each particle will update its position. After the particle has updated itself, the appropriate vertex in `m_Vertices` is updated by using the particle's `getPosition` function. At the end of each pass through the `for` loop, `currentVertex` is incremented, ready for the next vertex.

After the `for` loop has completed the code, `if (m_Duration < 0)` checks whether it is time to switch off the effect. If two seconds have elapsed, `m_IsRunning` is set to `false`.

Next, add the `emitParticles` function:

```
void ParticleSystem::emitParticles(Vector2f startPosition)
{
    m_IsRunning = true;
    m_Duration = 2;

    int currentVertex = 0;

    for (auto it = m_Particles.begin();
         it != m_Particles.end();
         it++)
    {
        m_Vertices[currentVertex++].color = Color::Yellow;
        it->setPosition(startPosition);
    }
}
```

This is the function we will call to start the particle system. So predictably, we set `m_IsRunning` to true and `m_Duration` to 2. We declare an iterator, `i`, to iterate through all the `Particle` objects in `m_Particles` and then we do so in a `for` loop.

Inside the `for` loop, we set each particle in the vertex array to yellow and set each position to `startPosition`, which was passed in as a parameter. Remember that each particle starts life in the same position, but they are each assigned a different velocity.

Next, add the pure virtual draw function definition:

```
void ParticleSystem::
    draw(RenderTarget& target,
         RenderStates states) const
{
    target.draw(m_Vertices, states);
}
```

In the preceding code, we simply use `target` to call `draw`, passing `m_Vertices` and `states` as parameters. Remember that we will never call this function directly! Shortly, when we declare an instance of `ParticleSystem`, we will pass that instance to the `RenderWindow` draw function. The draw function we have just coded will be called internally from there.

Finally, add the running function:

```
bool ParticleSystem::running()
{
    return m_IsRunning;
}
```

The running function is a simple getter function that returns the value of `m_IsRunning`. We will see where this is useful in this chapter, so that we can determine the current state of the particle system.

Using the ParticleSystem object

Putting our particle system to work is very straightforward, especially because we inherited from `Drawable`.

Adding a ParticleSystem object to the Engine class

Open `Engine.h` and add a `ParticleSystem` object, as shown in the following highlighted code:

```
#pragma once
#include <SFML/Graphics.hpp>
```

```
#include "TextureHolder.h"
#include "Thomas.h"
#include "Bob.h"
#include "LevelManager.h"
#include "SoundManager.h"
#include "HUD.h"
#include "ParticleSystem.h"

using namespace sf;

class Engine
{
private:
    // The texture holder
    TextureHolder th;

    // create a particle system
ParticleSystem m_PS;

    // Thomas and his friend, Bob
    Thomas m_Thomas;
    Bob m_Bob;
```

Now, we need to initialize the system.

Initializing ParticleSystem

Open the `Engine.cpp` file and add the short highlighted code right at the end of the `Engine` constructor:

```
Engine::Engine()
{
    // Get the screen resolution and create an SFML window and View
    Vector2f resolution;
    resolution.x = VideoMode::getDesktopMode().width;
    resolution.y = VideoMode::getDesktopMode().height;

    m_Window.create(VideoMode(resolution.x, resolution.y),
        "Thomas was late",
        Style::Fullscreen);

    // Initialize the full screen view
    m_MainView.setSize(resolution);
    m_HudView.reset(
```

```
        FloatRect(0, 0, resolution.x, resolution.y));

    // Initialize the split-screen Views
    m_LeftView.setViewport(
        FloatRect(0.001f, 0.001f, 0.498f, 0.998f));

    m_RightView.setViewport(
        FloatRect(0.5f, 0.001f, 0.499f, 0.998f));

    m_BGLeftView.setViewport(
        FloatRect(0.001f, 0.001f, 0.498f, 0.998f));

    m_BGRightView.setViewport(
        FloatRect(0.5f, 0.001f, 0.499f, 0.998f));

    // Can this graphics card use shaders?
    if (!sf::Shader::isAvailable())
    {
        // Time to get a new PC
        m_Window.close();
    }

    m_BackgroundTexture = TextureHolder::GetTexture(
        "graphics/background.png");

    // Associate the sprite with the texture
    m_BackgroundSprite.setTexture(m_BackgroundTexture);

    // Load the texture for the background vertex array
    m_TextureTiles = TextureHolder::GetTexture(
        "graphics/tiles_sheet.png");

    // Initialize the particle system
    m_PS.init(1000);

} // End Engine constructor
```

The VertexArray and the vector of Particle instances are ready for action.

Updating the particle system each frame

Open the `Update.cpp` file and add the following highlighted code. It can go right at the end of the update function:

```
// Update the HUD every m_TargetFramesPerHUDUpdate frames
if (m_FramesSinceLastHUDUpdate > m_TargetFramesPerHUDUpdate)
{
    // Update game HUD text
    stringstream ssTime;
    stringstream ssLevel;

    // Update the time text
    ssTime << (int)m_TimeRemaining;
    m_Hud.setTime(ssTime.str());

    // Update the level text
    ssLevel << "Level:" << m_LM.getCurrentLevel();
    m_Hud.setLevel(ssLevel.str());

    m_FramesSinceLastHUDUpdate = 0;
}

// Update the particles
if (m_PS.running())
{
    m_PS.update(dtAsSeconds);
}

} // End of update function
```

All that is needed in the previous code is the call to `update`. Notice that it is wrapped in a check to make sure the system is currently running. If it isn't running, there is no point updating it.

Starting the particle system

Open the `DetectCollisions.cpp` file, which has the `detectCollisions` function in it. We left a comment in it when we originally coded it.

Identify the correct place from the context and add the following highlighted code:

```
// Is character colliding with a regular block
if (m_ArrayLevel[y][x] == 1)
{
```

```

        if (character.getRight().intersects(block))
        {
            character.stopRight(block.left);
        }
        else if (character.getLeft().intersects(block))
        {
            character.stopLeft(block.left);
        }

        if (character.getFeet().intersects(block))
        {
            character.stopFalling(block.top);
        }
        else if (character.getHead().intersects(block))
        {
            character.stopJump();
        }
    }

    // More collision detection here once
    // we have learned about particle effects

    // Have the characters' feet touched fire or water?
    // If so, start a particle effect
    // Make sure this is the first time we have detected this
    // by seeing if an effect is already running
    if (!m_PS.running()) {
        if (m_ArrayLevel[y][x] == 2 || m_ArrayLevel[y][x] == 3)
        {
            if (character.getFeet().intersects(block))
            {
                // position and start the particle system
                m_PS.emitParticles(character.getCenter());
            }
        }
    }

    // Has the character reached the goal?
    if (m_ArrayLevel[y][x] == 4)
    {
        // Character has reached the goal
        reachedGoal = true;
    }

```

First, the code checks if the particle system is already running. If it isn't, it checks if the current tile being checked is either a water or fire tile. If either is the case, it checks whether the character's feet are in contact with it. When each of these `if` statements are true, the particle system is started by calling the `emitParticles` function and passing in the location of the center of the character as the coordinates to start the effect.

Drawing the particle system

This is the best bit. See how easy it is to draw `ParticleSystem`. We pass our instance directly to the `m_Window.draw` function, after checking that the particle system is running.

Open the `Draw.cpp` file and add the following highlighted code in all the necessary places:

```
void Engine::draw()
{
    // Rub out the last frame
    m_Window.clear(Color::White);

    if (!m_SplitScreen)
    {
        // Switch to background view
        m_Window.setView(m_BGMainView);
        // Draw the background
        m_Window.draw(m_BackgroundSprite);
        // Switch to m_MainView
        m_Window.setView(m_MainView);

        // Draw the Level
        m_Window.draw(m_VAlevel, &m_TextureTiles);

        // Draw thomas
        m_Window.draw(m_Thomas.getSprite());

        // Draw bob
        m_Window.draw(m_Bob.getSprite());

        // Draw the particle system
        if (m_PS.running())
        {
            m_Window.draw(m_PS);
        }
    }
}
```

```
}
else
{
    // Split-screen view is active

    // First draw Thomas' side of the screen

    // Switch to background view
    m_Window.setView(m_BGLeftView);
    // Draw the background
    m_Window.draw(m_BackgroundSprite);
    // Switch to m_LeftView
    m_Window.setView(m_LeftView);

    // Draw the Level
    m_Window.draw(m_VAlevel, &m_TextureTiles);

    // Draw bob
    m_Window.draw(m_Bob.getSprite());

    // Draw thomas
    m_Window.draw(m_Thomas.getSprite());

    // Draw the particle system
    if (m_PS.running())
    {
        m_Window.draw(m_PS);
    }

    // Now draw Bob's side of the screen

    // Switch to background view
    m_Window.setView(m_BGRightView);
    // Draw the background
    m_Window.draw(m_BackgroundSprite);
    // Switch to m_RightView
    m_Window.setView(m_RightView);

    // Draw the Level
    m_Window.draw(m_VAlevel, &m_TextureTiles);

    // Draw thomas
    m_Window.draw(m_Thomas.getSprite());
```

```
        // Draw bob
        m_Window.draw(m_Bob.getSprite());

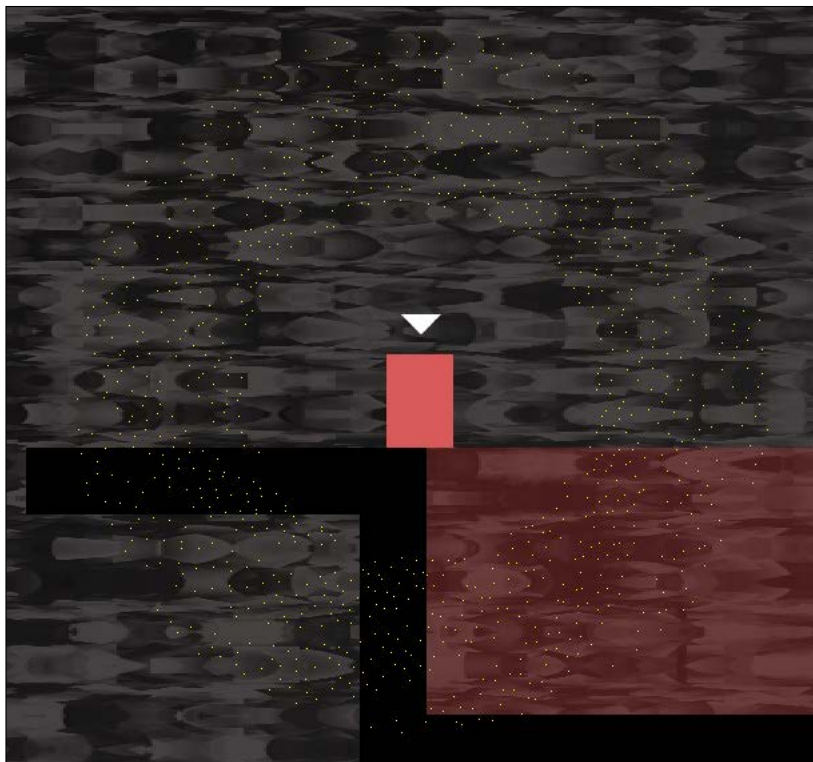
        // Draw the particle system
        if (m_PS.running())
        {
            m_Window.draw(m_PS);
        }
    }

    // Draw the HUD
    // Switch to m_HudView
    m_Window.setView(m_HudView);
    m_Window.draw(m_Hud.getLevel());
    m_Window.draw(m_Hud.getTime());
    if (!m_Playing)
    {
        m_Window.draw(m_Hud.getMessage());
    }

    // Show everything we have just drawn
    m_Window.display();
}
```

Note that we must draw the particle system in all of the left, right, and full-screen code blocks.

Run the game and move one of the character's feet over the edge of a fire tile. Notice the particle system burst into life:



Now, it's time for something else that's new.

OpenGL, Shaders, and GLSL

The **Open Graphics Library (OpenGL)** is a programming library that handles 2D as well as 3D graphics. OpenGL works on all major desktop operating systems and there is also a version that works on mobile devices, known as OpenGL ES.

OpenGL was originally released in 1992. It has been refined and improved over more than twenty years. Furthermore, graphics card manufacturers design their hardware to make it work well with OpenGL. The point of mentioning this is not for the history lesson but to explain that it would be a fool's errand to try and improve upon OpenGL and use it in 2D (and 3D games) on the desktop, especially if we want our game to run on more than just Windows, which is the obvious choice. We are already using OpenGL because SFML uses OpenGL. Shaders are programs that run on the GPU itself. We'll find out more about them in the following section.

The programmable pipeline and shaders

Through OpenGL, we have access to what is called a **programmable pipeline**. We can send our graphics off to be drawn, each frame, with the `RenderWindow` instance's `draw` function. We can also write code that runs on the GPU that can manipulate each and every pixel independently, after the call to `draw`. This is a very powerful feature.

This extra code that runs on the GPU is called a **shader program**. We can write code to manipulate the geometry (position) of our graphics in a **vertex shader**. We can also write code that manipulates the appearance of every pixel individually in code. This is known as a **fragment shader**.

Although we will not be exploring shaders in any great depth, we will write some shader code using the **GL Shader Language (GLSL)** and we will get a glimpse of the possibilities that it offers.

In OpenGL, everything is a point, a line, or a triangle. In addition, we can attach colors and textures to this basic geometry, and we can also combine these elements to make the complex graphics that we see in today's modern games. These are collectively known as **primitives**. We have access to OpenGL primitives through the SFML `primitives` and `VertexArray`, as well as the `Sprite` and `Shape` classes.

In addition to primitives, OpenGL uses matrices. Matrices are a method and structure for performing arithmetic. This arithmetic can range from extremely simple high school-level calculations such as moving (translating) a coordinate or it can be quite complex, such as performing more advanced mathematics, for example, to convert our game world coordinates into OpenGL screen coordinates that the GPU can use. Fortunately, it is this complexity that SFML handles for us behind the scenes. SFML also allows us to handle OpenGL directly.



If you want to find out more about OpenGL, you can get started here: <http://learnopengl.com/#!Introduction>. If you want to use OpenGL directly, alongside SFML, you can read this article to find out more: <https://www.sfm1-dev.org/tutorials/2.5/window-opengl.php>.

An application can have many shaders. We can then *attach* different shaders to different game objects to create the desired effects. We will only have one vertex and one fragment shader in this game. We will apply it to every frame, as well as to the background.

However, when you see how to attach a shader to a draw call, it will be plain that it is trivial to have more shaders.

We will follow these steps:

1. First, we need the code for the shader that will be executed on the GPU.
2. Then, we need to compile that code.
3. Finally, we need to attach the shader to the appropriate draw function call in the draw function of our game engine.

GLSL is a language and it also has its own types, and variables of those types, which can be declared and utilized. Furthermore, we can interact with the shader program's variables from our C++ code.

As we will see, GLSL has some syntax similarities to C++.

Coding a fragment shader

Here is the code from the `rippleShader.frag` file in the `shaders` folder. We don't need to code this because it is in the assets that we added back in *Chapter 14, Abstraction and Code Management – Making Better Use of OOP*:

```
// attributes from vertShader.vert
varying vec4 vColor;
varying vec2 vTexCoord;

// uniforms
uniform sampler2D uTexture;
uniform float uTime;

void main() {
    float coef = sin(gl_FragCoord.y * 0.1 + 1 * uTime);
    vTexCoord.y += coef * 0.03;
    gl_FragColor = vColor * texture2D(uTexture, vTexCoord);
}
```

The first four lines (excluding comments) are the variables that the fragment shader will use, but they are not ordinary variables. The first type we can see is `varying`. These are variables which are in scope between both shaders. Next, we have the uniform variables. These variables can be manipulated directly from our C++ code. We will see how we do this soon.

In addition to the `varying` and `uniform` types, each of the variables also has a more conventional type that defines the actual data, as follows:

- `vec4` is a vector with four values.
- `vec2` is a vector with two values.
- `sampler2d` will hold a texture.
- `float` is just like a `float data type` in C++.

The code inside the `main` function is executed. If we look closely at the code in `main`, we will see each of the variables in use. Exactly what this code does is beyond the scope of the book. In summary, however, the texture coordinates (`vTexCoord`) and the color of the pixels/fragments (`glFragColor`) are manipulated by several mathematical functions and operations. Remember that this executes for each pixel involved in the `draw` function that's called on each frame of our game. Furthermore, be aware that `uTime` is passed in as a different value for each frame. The result, as we will soon see, will be a rippling effect.

Coding a vertex shader

Here is the code from the `vertShader.vert` file. You don't need to code this. It was in the assets we added back in *Chapter 14, Abstraction and Code Management – Making Better Use of OOP*:

```
//varying "out" variables to be used in the fragment shader
varying vec4 vColor;
varying vec2 vTexCoord;

void main() {
    vColor = gl_Color;
    vTexCoord = (gl_TextureMatrix[0] * gl_MultiTexCoord0).xy;
    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}
```

First of all, notice the two `varying` variables. These are the very same variables that we manipulated back in the fragment shader. In the `main` function, the code manipulates the position of each and every vertex. How the code works is beyond the scope of this book, but there is some quite in-depth mathematics going on behind the scenes. If it interests you, then exploring GLSL further will be fascinating.

Now that we have two shaders (one fragment and one vertex), we can use them in our game.

Adding shaders to the engine class

Open the `Engine.h` file. Add the following highlighted line of code, which adds an SFML Shader instance called `m_RippleShader` to the `Engine` class:

```
// Three views for the background
View m_BGMainView;
View m_BGLeftView;
View m_BGRightView;

View m_HudView;

// Declare a sprite and a Texture for the background
Sprite m_BackgroundSprite;
Texture m_BackgroundTexture;

// Declare a shader for the background
Shader m_RippleShader;


// Is the game currently playing?
bool m_Playing = false;

// Is character 1 or 2 the current focus?
bool m_Character1 = true;
```

The engine object and all its functions now have access to `m_RippleShader`. Note that an SFML Shader object will be comprised of both shader code files.

Loading the shaders

Add the following code, which checks whether the player's GPU can handle shaders. The game will quit if it can't.

 You will have to have an exceptionally old PC for this not to work. If you do have a GPU that doesn't handle shaders, please accept my apologies.

Next, we will add an `else` clause that loads the shaders if the system can handle them. Open the `Engine.cpp` file and add this code to the constructor:

```
// Can this graphics card use shaders?
if (!sf::Shader::isAvailable())
{
    // Time to get a new PC
}
```

```
        // Or remove all the shader related code ☹️
        m_Window.close();
    }
    else
    {
        // Load two shaders (1 vertex, 1 fragment)
        m_RippleShader.loadFromFile("shaders/vertShader.vert",
                                   "shaders/rippleShader.frag");
    }

    m_BackgroundTexture = TextureHolder::GetTexture(
        "graphics/background.png");
```

We are nearly ready to see our ripple effect in action.

Updating and drawing the shader

Open the `Draw.cpp` file. As we already discussed when we coded the shaders, we will update the `uTime` variable directly from our C++ code each frame. We will do so with the `setParameter` function.

Add the following highlighted code to update the shader's `uTime` variable and change the call to draw for `m_BackgroundSprite`, in each of the possible drawing scenarios:

```
void Engine::draw()
{
    // Rub out the last frame
    m_Window.clear(Color::White);

    // Update the shader parameters
    m_RippleShader.setUniform("uTime",
                              m_GameTimeTotal.asSeconds());

    if (!m_SplitScreen)
    {
        // Switch to background view
        m_Window.setView(m_BGMainView);
        // Draw the background
        //m_Window.draw(m_BackgroundSprite);

        // Draw the background, complete with shader effect
        m_Window.draw(m_BackgroundSprite, &m_RippleShader);
    }
}
```

```
// Switch to m_MainView
m_Window.setView(m_MainView);

// Draw the Level
m_Window.draw(m_VAlevel, &m_TextureTiles);

// Draw thomas
m_Window.draw(m_Thomas.getSprite());

// Draw thomas
m_Window.draw(m_Bob.getSprite());

// Draw the particle system
if (m_PS.running())
{
    m_Window.draw(m_PS);
}
else
{
    // Split-screen view is active

    // First draw Thomas' side of the screen

    // Switch to background view
    m_Window.setView(m_BGLeftView);
    // Draw the background
    m_Window.draw(m_BackgroundSprite);

    // Draw the background, complete with shader effect
    m_Window.draw(m_BackgroundSprite, &m_RippleShader);

    // Switch to m_LeftView
    m_Window.setView(m_LeftView);

    // Draw the Level
    m_Window.draw(m_VAlevel, &m_TextureTiles);

    // Draw thomas
    m_Window.draw(m_Bob.getSprite());

    // Draw thomas
    m_Window.draw(m_Thomas.getSprite());
}
```

```
// Draw the particle system
if (m_PS.running())
{
    m_Window.draw(m_PS);
}

// Now draw Bob's side of the screen

// Switch to background view
m_Window.setView(m_BGRightView);
// Draw the background
//m_Window.draw(m_BackgroundSprite);

// Draw the background, complete with shader effect
m_Window.draw(m_BackgroundSprite, &m_RippleShader);

// Switch to m_RightView
m_Window.setView(m_RightView);

// Draw the Level
m_Window.draw(m_VAlevel, &m_TextureTiles);

// Draw thomas
m_Window.draw(m_Thomas.getSprite());

// Draw bob
m_Window.draw(m_Bob.getSprite());

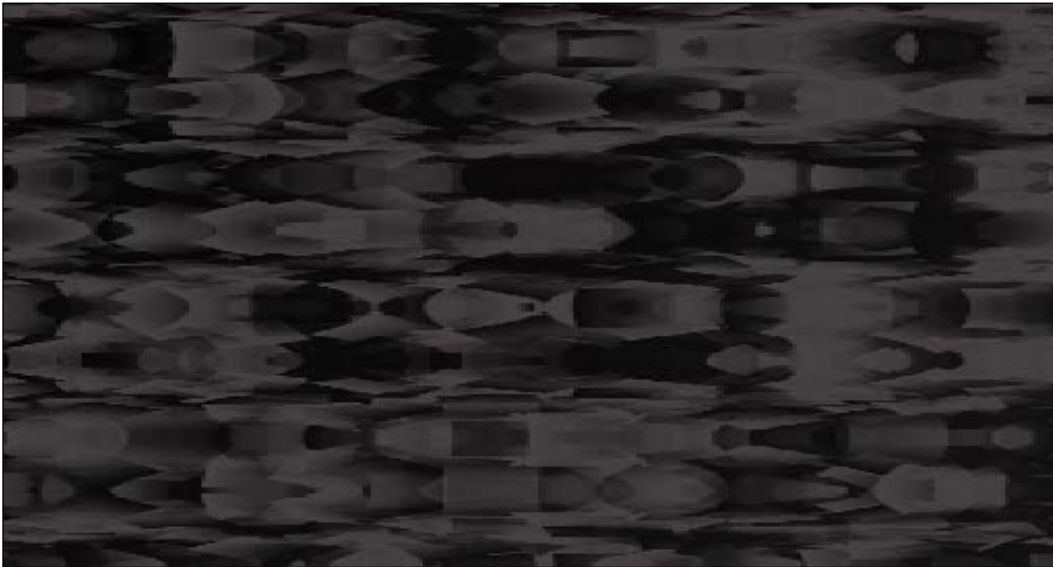
// Draw the particle system
if (m_PS.running())
{
    m_Window.draw(m_PS);
}
}

// Draw the HUD
// Switch to m_HudView
m_Window.setView(m_HudView);
m_Window.draw(m_Hud.getLevel());
m_Window.draw(m_Hud.getTime());
if (!m_Playing)
```

```
{  
    m_Window.draw(m_Hud.getMessage());  
}  
  
// Show everything we have just drawn  
m_Window.display();  
}
```

It would be best to delete the lines of code that were commented out.

Run the game and you will get an eerie kind of molten rock. Experiment with changing the background image to have some fun:



That's it! Our fourth game is done.

Summary

In this chapter, we explored the concepts of particle systems and shaders. Although we looked at probably the simplest possible case for each, we still managed to create a simple explosion and an eerie molten rock effect.

In the next four chapters, we will look at more ways that we can improve our code using design patterns at the same time as building a Space Invaders game.

19

Game Programming Design Patterns – Starting the Space Invaders ++ Game

Welcome to the final project. As you have come to expect by now, this project will take a significant step forward in terms of learning new C++ techniques. The next four chapters will look at topics such as **smart pointers**, C++ **assertions**, using a gamepad controller, debugging using Visual Studio, **casting** pointers of a base class to become pointers of a specific derived class, debugging, and a first look at **design patterns**.

It is my guess that if you are going to make deep, large-scale games in C++, then design patterns are going to be a big part of your learning agenda in the months and years ahead. In order to introduce this vital topic, I have chosen a relatively simple but fun game to serve as an example. In this chapter, we'll find out a bit more about the Space Invaders ++ game, and then we can get on to the topic of design patterns and why we need them.

In this hefty chapter, we will cover the following topics:

- Find out about Space Invaders ++ and why we chose it for the final project.
- Learn what design patterns are and why they matter to game developers.
- Study the design patterns in the Space Invaders ++ project that will be used over the next four chapters.
- We will get started on the Space Invaders ++ project.
- Code numerous classes to start fleshing out the game.

Let's talk about the game itself.

Space Invaders ++

Have a look at the following three screenshots, which visually explain most of what we need to know about Space Invaders ++. Just in case you don't know already, Space Invaders is one of the earliest arcade games and was released in 1978. If you like a bit of history, you can read the Wikipedia Space Invaders game page here: https://en.wikipedia.org/wiki/Space_Invaders.

This first screenshot shows the simple starting screen of our game. For the purposes of discussing screens, which we'll do next, we will call this the **select screen**. The player has two choices to select from: quit or play. However, by the end of this chapter, you will know how to add and switch between as many screens as you like:



As you can see, in the preceding screenshot, there is a new feature we have not implemented before: clickable buttons. We will talk more about buttons and their counterparts, such as UI panels and screens, shortly.

The following screenshot shows the game in action. It is quite simple to play. For the purposes of discussing screens, which we'll do next, we will call the following screenshot the **play screen**. The invaders move from left to right while shooting bullets at the player. When they reach the edge of the screen, they drop a little lower, speed up, and head back to the left:

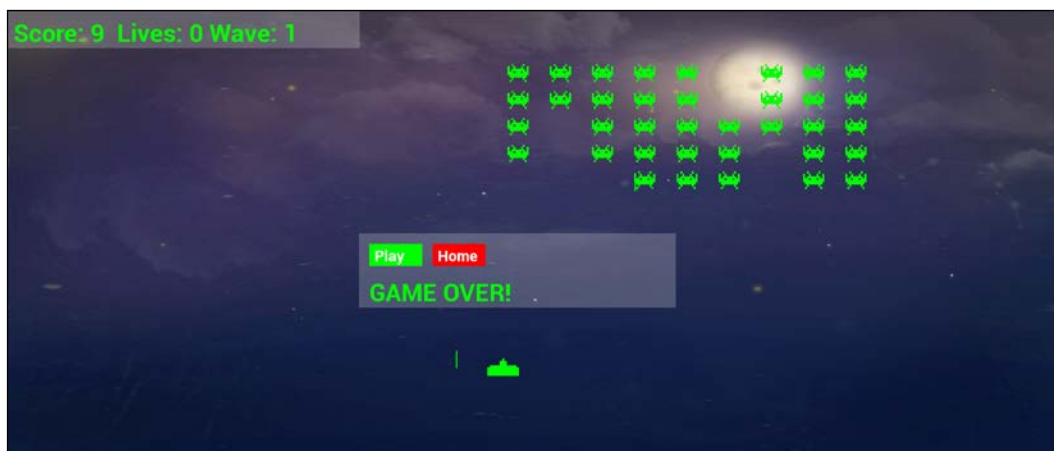


The player can move left and right as well as up and down, but the vertical movement is restricted to the bottom half of the screen.



The original Space Invaders game just allowed horizontal movement.

The following screenshot shows the options the player is presented with when they have lost three lives. They can choose to play again or quit and go back to the select screen:



While Space Invaders ++ does allow us to introduce lots of new C++ topics that I have already mentioned in the introduction to the chapter, as well as some more game-related topics such as using a gamepad controller, it is true that this isn't really a step up in terms of complexity compared to the previous project. So, why choose this as the final project?



In this project, there's lots of code. Most of it we have seen before, either in the same context or a different context. It is not possible to explain every single line as a new book would be required to do so. I have very carefully chosen which code to explain in full, which code to just mention, and which code I am guessing you will be able to work out for yourself. I recommend studying all the code in this book and in the download bundle as you progress. I will, however, go into the structure of the code in full detail as that is the real learning objective of this project. Furthermore, all the C++ code is shown in this book, so nothing is missing, although only an overview of the `level1.txt` file is shown.

Why Space Invaders ++?

To begin this discussion, please consider my two objectives for this book:

1. The first objective of this book is to introduce you to C++ programming using the learning material of video games. I have already admitted on several occasions and several topics that this is just an introduction. C++ and game development are too big to fit into this book alone.
2. The second objective of this book is to leave you in a position to continue your study while still using games as the learning material.

The problem is, as we have seen, each time we build a game with more features than the last, we end up with a more complicated code structure and the code files get longer and longer too. Throughout this book, we have learned new ways to improve the structure of our code and at each stage, we have succeeded, but the increasing complexity of the games always seems to outweigh the code improvements we learn about.

This project is designed to address this complexity issue and to take back control of our source code. Despite this game being less deep than the previous project, there will be far more classes to deal with.

This obviously implies quite a complicated structure. It will also mean, however, that once you get to grips with this structure, you will be able to reuse it for much more complicated games without any of the code files going beyond a few hundred lines of code.

What this project is designed to do is allow you to come up with your own game ideas, even complex ones, and get started on them right away, using the design patterns we'll discuss in the following section.



Note, however, that I am definitely not suggesting the code structure (design patterns) we will learn about here are the ultimate solution to your game development future; in fact, they are far from it. What you will learn are solutions that allow you to get started with your dream project without the complexity stopping you in your tracks. You will still need to study more about design patterns, C++, and game development along the way.

So, what are design patterns?

Design patterns

A **design pattern** is a reusable solution to a coding problem. In fact, most games (including this one) will use multiple design patterns. The key point about design patterns is this: they are already proven to provide a good solution to a common problem. We are not going to invent any design patterns – we are just going to use some that already exist to solve the problem of our ever-expanding code.

Many design patterns are quite complicated and require further study beyond the level of this book if you want to even begin learning them. What follows is a simplification of a few key game development-related patterns that will help fulfill the second objective of this book. You're urged to continue your study to implement them more comprehensively and alongside even more patterns than will be discussed here.

Let's look at the design patterns that are used in the Space Invaders ++ project.

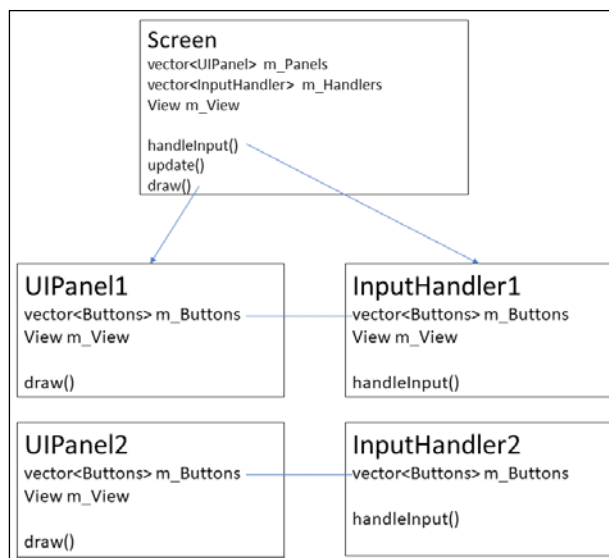
Screen, InputHandler, UIPanel, and Button

This project will abstract some concepts further than any of the other projects. Space Invaders ++ will introduce the concept of a **screen**. The concept of a screen is most easily understood by giving some examples. A game could have a menu screen, a settings screen, a high score screen, and a game screen. A **screen** is a logical division of the parts of the game. Every screen has some things in common with all the other screens, yet each screen also needs its own unique features as well. For example, a menu screen might have buttons that enable the player to transition to another screen, as well as a neat graphical image or even a dynamic scene. The high score screen will, of course, have a list of all the high scores and perhaps a button to return to the menu screen. Each screen will have a different layout, different buttons to click, and different responses to different keyboard presses, but they will all need to be drawn at 60 FPS and interact in the same way with the game engine.

In the previous projects, we crammed this concept of screens into one place. This meant we had sprawling long `if`, `else`, and `else if` blocks of code that handled updating, drawing, and responding to user interaction. Our code was getting quite challenging to handle already. If we are going to build more complicated games, we need to improve on this. The concept of screens means that we can create a class that handles all the stuff that happens for every screen, such as updating, drawing, and user interaction, and then create a derived class for each type of screen, that is, menu, game, high score, and so on, which handles the unique ways that a specific screen needs to update, draw, and respond to the user.

In Space Invaders ++, we will have a `Screen` class. We will then inherit from `Screen` to handle two screens, `SelectScreen` and `GameScreen`. Furthermore, we will have a `Button` class that knows how to display a button, a `UIPanel` class that knows how to draw text, and `Button` instances as well as an `InputHandler` class that knows how to detect keyboard and gamepad interaction. We will then be able to derive from `UIPanel` and `InputHandler` to let all the different `Screen` instances behave exactly as required without coding the basics of a screen, a UI panel, an input handler, or a button more than once. The bigger your game gets and the more screens it has, the bigger the benefit of doing things this way. It also means that the specifics of each screen will not be crammed into long `if`, `else`, and `else if` structures as we have been doing so far.

This is a bit like how we coded the `PlayableCharacter` class and derived `Thomas` and `Bob` from it. As we will see, however, we go much further with the abstraction this time. Look at the following diagram, which shows a representation of this idea and shows just one screen:

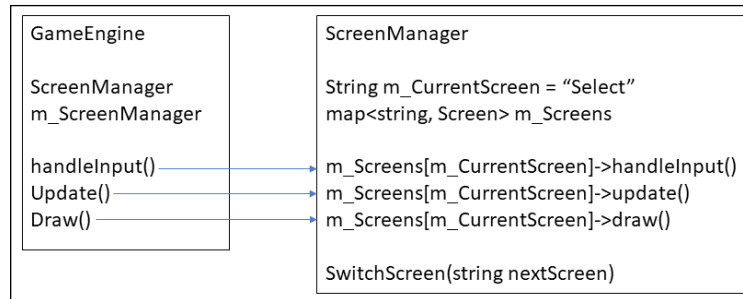


In the preceding diagram, we can see that a screen has one or more `UIPanel` instances that it can display selectively and that `UIPanel` instances can have zero or more `Button` instances. Each `UIPanel` will have a related `InputHandler` because each `UIPanel` will have different combinations and layouts of buttons. The buttons are shared via pointers between `UIPanel` and `InputHandler` instances.

If you are wondering which class handles the update stage of the game loop, the answer is the `Screen` class. However, once you get your head around how this pattern works, it will be simple to add the ability to let `UIPanel` instances act in the update phase, too. This could be useful if, say, the panel needed to move or maybe show a loading progress bar.

A screen will decide which `UIPanel` (and therefore, `InputHandler`) instances are currently visible and responding. However, only one screen at a time will be visible to the player. We will code a `ScreenManager` class that will be a fundamental part of the game engine to handle calling the key functions of the appropriate (current) screen. The `ScreenManager` class will also provide a way for the `InputHandler` instances to notify us when a change of screen is required, for example, when the player clicks the **Play** button on the select screen to go to the play screen.

`ScreenManager` will hold an instance of every screen, remember the current screen the player is on, and call `update`, `draw`, and `handleInput` on the correct screen, as well as switch between screens when required. The following diagram will hopefully help you visualize this concept, which we will also be coding soon:



Note that the diagrams and explanation are a simplification of the solution we will be coding, but they give a good overview.

Should you want to add a high score screen or another `UIPanel` instance to an existing screen, you will know how to do so by the end of *Chapter 22, Using Game Objects and Building a Game*. Of course, it's likely that you will want to get started on your very own game. You will be able to divide up your next game into as many screens with their dedicated layouts and input handling as you need.

Entity-Component pattern

We will now spend five minutes wallowing in the misery of an apparently unsolvable muddle. Then, we will see how the entity-component pattern comes to the rescue.

Why lots of diverse object types are hard to manage

In the previous projects, we coded a class for each object. We had classes such as `Bat`, `Ball`, `Crawler`, and `Thomas`. Then, in the `update` function, we would update them, and in the `draw` function, we would draw them. Each object decides how updating and drawing takes place.

We could just get started and use this same structure for `Space Invaders ++`. It would work, but we are trying to learn something more manageable so that our games can grow in complexity.

Another problem with this approach is that we cannot take advantage of inheritance. For example, all the invaders, the bullets, and the player draw themselves in an identical way, but unless we change how we do things, we will end up with three draw functions with nearly identical code. If we make a change to how we call the draw function or the way we handle graphics, we will need to update all three classes.

There must be a better way.

Using a generic `GameObject` for better code structure

If every object, player, alien, and all the bullets were one generic type, then we could pack them away in a vector instance and loop through each of their update functions, followed by each of their draw functions.

We already know one way of doing this – inheritance. At first glance, inheritance might seem like a perfect solution. We could create an abstract `GameObject` class and then extend it with the `Player`, `Invader`, and `Bullet` classes.

The draw function, which is identical in all three classes, could remain in the parent class, and we won't have the problem of all that wasted duplicate code. Great!

The problem with this approach is how varied – in some respects – the game objects are. Diversity is not a strength; it is just diverse. For example, all the object types move differently. The bullets go up or down, the invaders go left and right and drop down occasionally, and the player's ship responds to inputs.

How would we put this kind of diversity into the `update` so that it could control this movement? Maybe we could use something like this:

```
update() {
    switch(objectType) {
        case 1:
            // All the player's logic
            break;
        case 2:
            // All the invader's logic here
            Break;
        case 3:
            // All the bullet's logic here
            break;
    }
}
```

The update function alone would be bigger than the whole `GameEngine` class!

As you may remember from *Chapter 15, Advanced OOP – Inheritance and Polymorphism*, when we inherit from a class, we can also override specific functions. This means we could have a different version of the `update` function for each object type. Unfortunately, however, there is also a problem with this approach as well.

The `GameEngine` engine would have to "know" which type of object it was updating or, at the very least, be able to query the `GameObject` instance it was updating in order to call the correct version of the `update` function. What is really needed is for the `GameObject` to somehow internally choose which version of the `update` function is required.

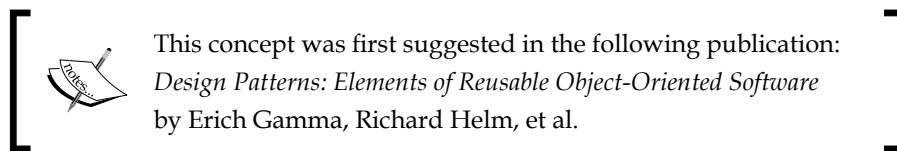
Unfortunately, even the part of the solution which did seem to work falls apart on closer inspection. I said that the code in the `draw` function was the same for all three of the objects, and therefore the `draw` function could be part of the parent class and used by all the sub-classes, instead of us having to code three separate `draw` functions. Well, what happens when we introduce a new object that needs to be drawn differently, such as an animated UFO that flies across the top of the screen? In this scenario, the `draw` solution falls apart too.

Now that we have seen the problems that occur when objects are different from each other and yet cry out to be from the same parent class, it is time to look at the solution we will use in the Space Invaders ++ project.

What we need is a new way of thinking about constructing all our game objects.

Prefer composition over inheritance

Preferring composition over inheritance refers to the idea of composing objects with other objects.

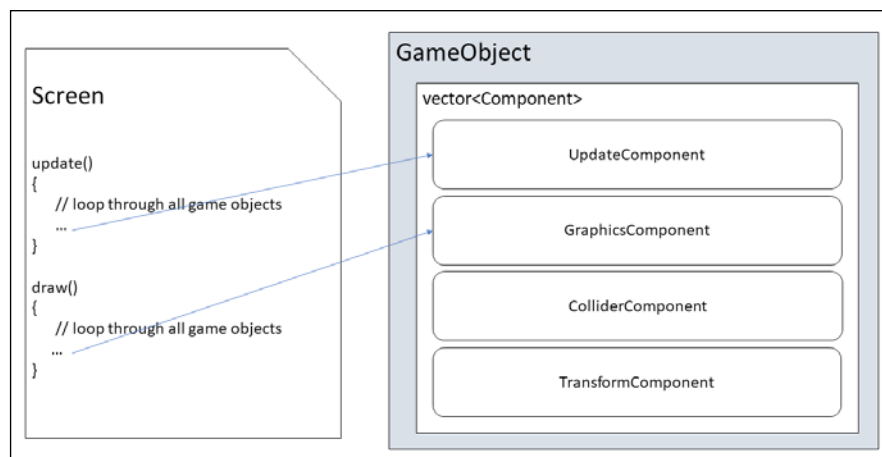


What if we could code a class (as opposed to a function) that handled how an object was drawn? Then for all the classes that draw themselves in the same way, we could instantiate one of these special drawing classes within the `GameObject`, and any objects that need to be drawn differently could have a different drawing object. Then, when a `GameObject` does something differently, we simply compose it with a different drawing or updating related class to suit it. All the similarities in all our objects can benefit from using the same code, while all the differences can benefit from not only being encapsulated but also abstracted (taken out of) the base class.

Note that the heading of this section is composition over inheritance, not composition instead of inheritance. Composition doesn't replace inheritance and everything you learned in *Chapter 15, Advanced OOP – Inheritance and Polymorphism*, still holds true. However, where possible, compose instead of inheriting.

The `GameObject` class is the entity, while the classes it will be composed of that do things such as update its position and draw it to the screen are the components, which is why it's called the Entity-Component pattern.

Have a look at the following diagram, which represents the Entity-Component pattern in the form we will implement it in this project:



In the preceding diagram, we can see that a `GameObject` instance is composed of multiple `Component` instances. There will be multiple different classes derived from the `Component` class, including `UpdateComponent` and `GraphicsComponent`. Furthermore, there can be further specific classes derived from them. For example, the `BulletUpdateComponent` and `InvaderUpdateComponent` classes will be derived from the `UpdateComponent` class. These classes will handle how a bullet and an invader (respectively) update themselves each frame of the game. This is great for encapsulation because we don't need the big `switch` blocks to distinguish between different objects.

When we use composition over inheritance to create a group of classes that represent behavior/algorithms, as we will here, this is known as the **Strategy** pattern. You could use everything you have learned here and refer to it as the Strategy pattern. Entity-Component is a lesser known but more specific implementation, and that is why we call it this. The difference is academic, but feel free to turn to Google if you want to explore things further. In *Chapter 23, Before You Go...*, I will show you some good resources for this kind of detailed research.

The Entity-Component pattern, along with using composition in preference to inheritance, sounds great at first glance but brings with it some problems of its own. It would mean that our new `GameObject` class would need to know about all the different types of component and every single type of object in the game. How would it add all the correct components to itself?

Let's have a look at the solution.

Factory pattern

It is true that if we are to have this universal `GameObject` class that can be anything we want it to be, whether that be a bullet, player, invader, or whatever else, then we are going to have to code some logic that "knows" about constructing these super-flexible `GameObject` instances and composes them with the correct components. But adding all this code into the class itself would make it exceptionally unwieldy and defeat the entire reason for using the Entity-Component pattern in the first place.

We would need a constructor that did something like this hypothetical `GameObject` code:

```
class GameObject
{
    UpdateComponent* m_UpdateComponent;
    GraphicsComponent* m_GraphicsComponent;
    // More components

    // The constructor
    GameObject(string type){
        if(type == "invader")
        {
            m_UpdateComp = new InvaderUpdateComponent();
            m_GraphicsComponent = new StdGraphicsComponent();
        }
        else if(type == "ufo")
        {
            m_UpdateComponent = new
                UFOUpdateComponentComponent();
            m_GraphicsComponent = new AnimGraphicsComponent();
        }
        // etc.
        ...
    }
};
```

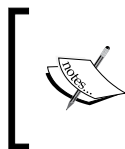
The `GameObject` class would need to know not just which components go with which `GameObject` instance, but also which didn't need certain components, such as input-related components for controlling the player. For the Space Invaders ++ project, we could do this and just about survive the complexity, but just about surviving is not the objective; we want to take complete control.

The `GameObject` class would also need to understand all this logic. Any benefit or efficiency gained from using composition over inheritance with the Entity-Component pattern would be mainly lost.

Furthermore, what if we decide we want a new type of invader, perhaps a "Cloaker" alien that teleports near to the player, takes a shot, and then teleports away again? It is fine to code a new `GraphicsComponent` class, perhaps a `CloakingGraphicsComponent` that "knows" when it is visible and invisible, along with a new `UpdateComponent`, perhaps a `CloakerUpdateComponent` that teleports instead of moving in the conventional manner, but what is not fine is we are going to have to add a whole bunch of new `if` statements to the `GameObject` class constructor.

In fact, the situation is even worse than this. What if we decide that regular invaders can now cloak? Invaders now need not just a different type of `GraphicsComponent` class. We would have to go back into the `GameObject` class to edit all of those `if` statements again.

In fact, there are even more scenarios that can be imagined, and they all end up with a bigger and bigger `GameObject` class. The **Factory** pattern is the solution to these `GameObject` class-related woes and the perfect partner to the Entity-Component pattern.



This implementation of the Factory pattern is an easier way to begin to learn about the Factory pattern. Why not do a web search for the Factory pattern once you have completed this project and see how it can be improved?

The game designer will provide a specification for each and every type of object in the game, and the programmer will provide a factory class that builds `GameObject` instances from the game designer's specifications. When the game designer comes up with new ideas for entities, then all we need to do is ask for a new specification. Sometimes, that will involve adding a new production line to the factory that uses existing components and, sometimes, it will mean coding new components or perhaps updating existing components. The point is that it won't matter how inventive the game designer is – the `GameObject` and `GameEngine` classes remain unchanged.

In the Factory code, the current object type is checked and the appropriate components (classes) are added to it. The bullet, player, and the invader have the same graphics component, but all have different update components.

When we use composition, it can be less clear which class is responsible for the memory. Is it the class that creates it, the class that uses it, or some other class? Let's learn some more C++ to help us manage memory a little more simply.

C++ smart pointers

Smart pointers are classes that we can use to get the same functionality as a regular pointer but with an extra feature – the feature being that they take care of their own deletion. In the limited way we have used pointers so far, it has not been a problem for us to delete our own memory, but as your code becomes more complex, and when you are allocating the new memory in one class but using it in another class, it becomes much less clear which class is responsible for deleting the memory when we are done with it. And how can a class or function know whether a different class or function has finished with some allocated memory?

The solution is smart pointers. There are a few types of smart pointer; we will look at the two of the most commonly used ones here. The key to success with smart pointers is using the correct type.

The first type we will consider is **shared pointers**.

Shared pointers

The way that a shared pointer can safely delete the memory it points to is by keeping a count of the number of different references there are to an area of memory. If you pass a pointer to a function, the count is increased by one. If you pass a pointer into a vector, the count is increased by one. If the function returns, the count is decreased by one. If the vector goes out of scope or has the `clear` function called on it, the smart pointer will reduce the reference count by one. When the reference count is zero, nothing points to the area of memory anymore and the smart pointer class calls `delete`. All the smart pointer classes are implemented using regular pointers behind the scenes. We just get the benefit of not having to concern ourselves about where or when to call `delete`. Let's look at the code for using a shared smart pointer.

The following code creates a new shared smart pointer called `myPointer` that will point to an instance of `MyClass`:

```
shared_ptr<MyClass> myPointer;
```

`shared_ptr<MyClass>` is the type while `myPointer` is its name. The following code is how we might initialize `myPointer`:

```
myPointer = make_shared<MyClass>();
```

The call to `make_shared` internally calls `new` to allocate the memory. The parentheses `()` is the constructor parentheses. If the `MyClass` class constructor took an `int` parameter, for example, the preceding code might look like this:

```
myPointer = make_shared<MyClass>(3);
```

The 3 in the preceding code is an arbitrary example.

Of course, you can declare and initialize your shared smart pointers in a single line of code if required, as shown in the following code:

```
shared_ptr<MyClass> myPointer = make_shared<MyClass>();
```

It is because `myPointer` is a `shared_ptr` that it has an internal reference count that keeps track of how many references point to the area of memory that it created. If we make a copy of the pointer, that reference count is increased.

Making a copy of the pointer includes passing the pointer to another function, placing it in a vector, map, or other structure, or simply copying it.

We can use a smart pointer using the same syntax as a regular pointer. It is quite easy to forget sometimes that it isn't a regular pointer. The following code calls the `myFunction` function on `myPointer`:

```
myPointer->myFunction();
```

By using a shared smart pointer, there is some performance and memory **overhead**. By overhead, I mean that our code runs slower and uses more memory. After all, the smart pointer needs a variable to keep track of the reference count, and it must check the value of the reference count every time a reference goes out of scope. However, this overhead is tiny and only an issue in the most extreme situations since most of the overhead happens while the smart pointers are being created. Typically, we will create smart pointers outside of the game loop. Calling a function on a smart pointer is as efficient as a regular pointer.

Sometimes, we know that we will only ever want one reference to a smart pointer and in this situation, **unique pointers** are the best option.

Unique pointers

When we know that we only want a single reference to an area of memory, we can use a unique smart pointer. Unique pointers lose much of the overhead that I mentioned shared pointers have. In addition, if you try and make a copy of a unique pointer, the compiler will warn us, and the code will either not compile or it will crash, giving us a clear error. This is a very useful feature that can prevent us from accidentally copying a pointer that was not meant to be copied. You might be wondering if this no copying rule means we can never pass it to a function or even put it in a data structure such as a `vector`. To find out, let's look at some code for unique smart pointers and explore how they work.

The following code creates a unique smart pointer called `myPointer` that points to an instance of `MyClass`:

```
unique_ptr<MyClass> myPointer = make_unique<MyClass>();
```

Now, let's suppose we want to add a `unique_ptr` to a `vector`. The first thing to note is that `vector` must be of the correct type. The following code declares a `vector` that holds unique pointers to `MyClass` instances:

```
vector<unique_ptr<MyClass>> myVector;
```

The `vector` is called `myVector` and anything you put into it must be of the unique pointer type to `MyClass`. But didn't I say that unique pointers can't be copied? When we know that we will only ever want a single reference to an area of memory, we should use `unique_ptr`. This doesn't mean, however, that the reference can't be moved. Here is an example:

```
// Use move() because otherwise
// the vector has a COPY which is not allowed
myVector.push_back(move(myPointer));
// myVector.push_back(myPointer); // Won't compile!
```

In the preceding code, we can see that the `move` function can be used to put a unique smart pointer into a `vector`. Note that when you use the `move` function, you are not giving the compiler permission to break the rules and copy a unique pointer – you are moving responsibility from the `myPointer` variable to the `myVector` instance. If you attempt to use the `myPointer` variable after this point, the code will execute and the game will crash, giving you a **Null pointer access violation error**. The following code will cause a crash:

```
unique_ptr<MyClass> myPointer = make_unique<MyClass>();
vector<unique_ptr<MyClass>> myVector;
// Use move() because otherwise
// the vector has a COPY which is not allowed
```

```
mVector.push_back(move(myPointer));
// mVector.push_back(myPointer); // Won't compile!

myPointer->myFunction();// CRASH!!
```

The exact same rules apply when passing a unique pointer to a function; use the `move` function to pass responsibility on. We will look at all these scenarios again, as well as some more when we get to the project in a few pages time.

Casting smart pointers

We will often want to pack the smart pointers of derived classes into data structures or function parameters of the base class such as all the different derived `Component` classes. This is the essence of polymorphism. Smart pointers can achieve this using casting. But what happens when we later need to access the functionality or data of the derived class?

A good example of where this will regularly be necessary is when we deal with components inside our game objects. There will be an abstract `Component` class and derived from that there will be `GraphicsComponent`, `UpdateComponent`, and more besides.

As an example, we will want to call the `update` function on all the `UpdateComponent` instances each frame of the game loop. But if all the components are stored as base class `Component` instances, then it might seem that we can't do this. Casting from the base class to a derived class solves this problem.

The following code casts `myComponent`, which is a base class `Component` instance to an `UpdateComponent` class instance, which we can then call the `update` function on:

```
shared_ptr<UpdateComponent> myUpdateComponent =
    static_pointer_cast<UpdateComponent>(MyComponent);
```

Before the equals sign, a new `shared_ptr` to an `UpdateComponent` instance is declared. After the equals sign, the `static_pointer_cast` function specifies the type to cast to in the angle brackets, `<UpdateComponent>`, and the instance to cast from in parentheses, `(MyComponent)`.

We can now use all the functions of the `UpdateComponent` class, which in our project includes the `update` function. We would call the `update` function as follows:

```
myUpdateComponent->update(fps);
```


There are two ways we can cast a class smart pointer to another class smart pointer. One is by using `static_pointer_cast`, as we have just seen, and the other is to use `dynamic_pointer_cast`. The difference is that `dynamic_pointer_cast` can be used if you are uncertain whether the cast will work. When you use `dynamic_pointer_cast`, you can then check to see if it worked by testing if the result is a null pointer. You use `static_pointer_cast` when you are certain the result is the type you are casting to. We will use `static_pointer_cast` throughout the Space Invaders ++ project.

We will regularly be casting `Component` instances to different derived types. How we will be sure the type we are casting to is the correct type will become apparent as we progress with the project.

C++ assertions

In this project, we will be using C++ **assertions**. As usual, there is more to this topic than we will discuss here, but we can still do some useful things with just an introduction.

We can use the `#define` preprocessor statement in a class to define a value for the entire project. We do so with the following code:

```
#define debuggingOnConsole
```

This code would be written at the top of a header file. Now, throughout the project, we can write code like the following:

```
#ifdef debuggingOnConsole
    // C++ code goes here
#endif
```

The `#ifdef debuggingOnConsole` statement checks whether the `#define debuggingOnConsole` statement is present. If it is, then any C++ code up to the `#endif` statement will be included in the game. We can then choose to comment out the `#define` statement to switch our debugging code on or off.

Typically, we will include code such as the following in the `#ifdef` blocks:

```
#ifdef debuggingOnConsole
    cout <<
        "Problem x occurred and caused a crash!"
        << endl;
#endif
```

The preceding code uses the `cout` statement to print debugging information to the console window.

What these assertions amount to is a way to get feedback from the game during development and then with a quick `//` in front of the `#define` statement, strip out all the debugging code from the game when we are done.

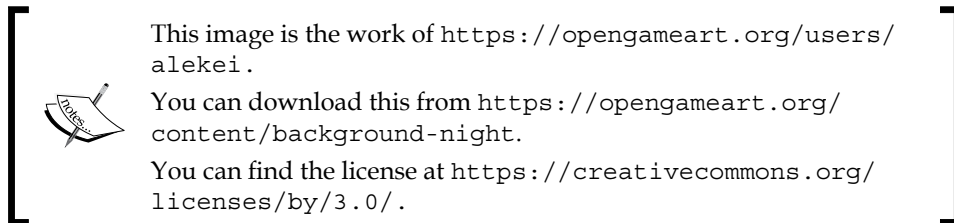
Creating the Space Invaders ++ project

You can find the runnable code that represents the project at the end of this chapter in the `Space Invaders ++` folder. It will take all of chapters 20, 21, and 22 to complete and make the project runnable again. The completed code that is runnable and represents the project at the end of *Chapter 22, Using Game Objects and Building a Game*, can be found in the `Space Invaders ++ 2` folder.

Create a new project in Visual Studio with the same settings that we used in the previous four projects. Call the new project `Space Invaders ++`.

Inside the `Space Invaders ++` folder, copy and paste the `fonts`, `graphics`, and `sound` folders and their contents from the download bundle. The `fonts`, `graphics`, and `sound` folders, as you would expect, contain the font and graphical and audio assets we will use in this game.

In addition, you will need to download the background file from <https://opengameart.org/content/background-night>.



Rename the file you just downloaded to `background.png` and place it in the `graphics` folder of your project.

Now, add the `world` folder, including the `level1.txt` file. This file contains the layout of all the game objects, and we will discuss it further in *Chapter 21, File I/O and the Game Object Factory*.

Organizing code files with filters

Next, we will do something new. As there are more class files in this project than our previous projects, we will be a bit more organized within Visual Studio. We will create a series of **filters**. These are logical organizers we use to create a structure for our files. This will allow us to view all our header and source files in a more organized way.

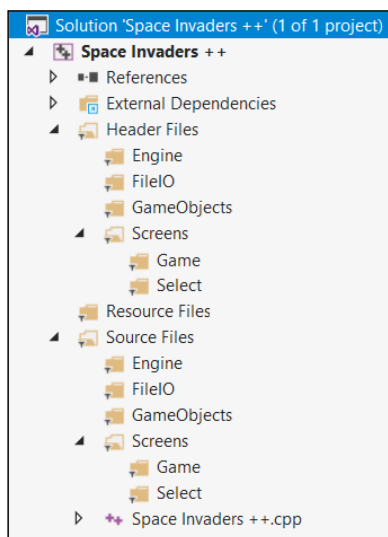
Right-click on the **Header Files** folder in the **Solution Explorer** window and select **New Filter**. Give the filter the name of **Engine**. We will add all the core header files to this filter.

Right-click on **Header Files** again and add another filter called **FileIO**. We will add all the files that read text to and from `level1.txt`, as well as some supporting classes.

Make another new filter in **Header Files** called **GameObjects**. Everything related to all the game objects, including the `GameObject` class and all the `Component` class-related header files, will go here.

Add yet another filter called **Screens**. Right-click on the **Screens** filter you just added and create a filter within **Screens** called **Select**. Now, create another filter within **Screens** called **Game**. We will place all the derived versions of `Screen`, `InputHandler`, and `UIPanel` in **Game** or **Select** (as appropriate) and all the base classes in **Screens**.

Now, repeat all the previous steps of creating filters to create the exact same structure in the **Source Files** folder. You should now have a Solution Explorer layout that looks as follows:



Note that the preceding layout is just for our organizational benefit; it has no effect on the code or the finished game. In fact, if you look in the `Space Invaders` ++ folder using your operating system's file browser, you will see there are no additional folders. As we progress with this project and add new classes, we will add them within specific filters to make them more organized and less cluttered.

Adding a DevelopState file

In order to output debugging data to the console, we will create the `DevelopState` class, which does nothing but define `debuggingOnConsole`.

Create the `DevelopState.h` file in the `Header Files/Engine` filter and add the following code:

```
#pragma once
#define debuggingOnConsole
class DevelopState {};
```

We can comment out `#define debuggingOnConsole` when the game is working but, when we have unexplained crashes, we can uncomment it. If we then add assertions at parts throughout our code, we can see if these parts are causing the game to crash.

Coding SpaceInvaders ++.cpp

Next, drag and drop the `SpaceInvaders ++.cpp` file that was autogenerated when we created the project into the `Source Files/Engine` filter. This isn't required – it is just to keep things organized. This file is the entry point to the game and is therefore a core file, albeit a very short one.

Edit `SpaceInvaders ++.cpp` so that it just has the following code:

```
#include "GameEngine.h"

int main()
{
    GameEngine m_GameEngine;
    m_GameEngine.run();
    return 0;
}
```

The preceding code creates an instance of `GameEngine` and calls its `run` function. There will be errors until we code the `GameEngine` class. We will do that next. Note that, throughout this project, there will usually be one, more, or even many errors. This is due to the interdependent nature of the classes. I will usually mention when there are errors and when they will be dealt with, but perhaps not every single one. By the end of this chapter, we will have an error-free, executable project, but, after that, it will take until *Chapter 22, Using Game Objects and Building a Game*, until the project is error-free and executable again.

Coding the GameEngine class

Create a new header file in the Header Files/Engine filter called `GameEngine.h` and add the following code:

```
#pragma once
#include <SFML/Graphics.hpp>
#include "ScreenManager.h"
#include "SoundEngine.h"

using namespace sf;

class GameEngine {
private:
    Clock m_Clock;
    Time m_DT;
    RenderWindow m_Window;

    unique_ptr<ScreenManager> m_ScreenManager;

    float m_FPS = 0;
    Vector2f m_Resolution;

    void handleInput();
    void update();
    void draw();

public:
    SoundEngine m_SoundEngine;

    GameEngine();
    void run();
};
```

Study the preceding code to get familiar with it. What's new is that we get to see smart pointers in action for the first time. We have a unique pointer of the `ScreenManager` Type. This implies that this pointer will not be passed to any other classes but, if it is, then ownership will also be passed.

Other than the smart pointers, there is nothing we haven't seen before. There is a `Clock` instance, a `Time` instance, a `RenderWindow` instance, as well as variables to keep track of the frame rate and the screen resolution. Furthermore, we have functions for handling input, updating, and drawing each frame. This is also nothing new. What we do within these functions, however, will be new. We also have a `SoundEngine` instance, which will be nearly identical to how we handled sound in our other projects. We also have the `run` function, which is public, and will kickstart all the private functions.

There are errors because we need to implement the `ScreenManager` and `SoundEngine` classes. We will get to them very soon.

Create a new source file in the `Source Files/Engine` filter called `GameEngine.cpp` and add the following code:

```
#include "GameEngine.h"

GameEngine::GameEngine()
{
    m_Resolution.x = VideoMode::getDesktopMode().width;
    m_Resolution.y = VideoMode::getDesktopMode().height;
    m_Window.create(VideoMode(m_Resolution.x, m_Resolution.y),
        "Space Invaders++", Style::Fullscreen);

    m_ScreenManager = unique_ptr<ScreenManager>(new ScreenManager(
        Vector2i(m_Resolution.x, m_Resolution.y)));
}

void GameEngine::run()
{
    while (m_Window.isOpen())
    {
        m_DT = m_Clock.restart();
        m_FPS = m_DT.asSeconds();
        handleInput();
        update();
        draw();
    }
}
```

```
void GameEngine::handleInput ()
{
    m_ScreenManager->handleInput (m_Window) ;
}

void GameEngine::update ()
{
    m_ScreenManager->update (m_FPS) ;
}

void GameEngine::draw ()
{
    m_Window.clear (Color::Black) ;
    m_ScreenManager->draw (m_Window) ;
    m_Window.display () ;
}
```

In the `GameEngine` constructor, the `RenderWindow` instance is initialized and the unique smart pointer to a `ScreenManager` instance is initialized using `new`, which passes in the resolution to the `ScreenManager` constructor.



This is an alternative to calling the `make_unique` function.

The `run` function should look very familiar; it restarts the clock and stores the time like we have done in every project so far. It then calls the `handleInput`, `update`, and `draw` functions.

In the `handleInput` function, the `handleInput` function of the `ScreenManager` instance is called. In the `update` function, the `update` function of the `ScreenManager` instance is called. Finally, in the `draw` function, the `RenderWindow` is cleared, the `draw` function of the `ScreenManager` instance is called, and the contents of the `RenderWindow` instance are displayed.

We have successfully passed full responsibility to the `ScreenManager` class for handling input, updating, and drawing each frame. As we will see in the *Coding the ScreenManager* section, the `ScreenManager` class will further delegate responsibility for all these tasks to the appropriate class that's derived from the `Screen` class.

Like the related `GameEngine.h` header file, there are errors because we need to implement the `ScreenManager` and `SoundEngine` classes.

Coding the SoundEngine class

Create a new header file in the Header Files/Engine filter called `SoundEngine.h` and add the following code:

```
#pragma once
#ifndef SOUND_ENGINE_H
#define SOUND_ENGINE_H

#include <SFML/Audio.hpp>

using namespace sf;

class SoundEngine
{
private:
    SoundBuffer m_ShootBuffer;
    SoundBuffer m_PlayerExplodeBuffer;
    SoundBuffer m_InvaderExplodeBuffer;
    SoundBuffer m_ClickBuffer;

    Sound m_ShootSound;
    Sound m_PlayerExplodeSound;
    Sound m_InvaderExplodeSound;
    Sound m_UhSound;
    Sound m_OhSound;
    Sound m_ClickSound;

public:
    SoundEngine();

    static void playShoot();
    static void playPlayerExplode();
    static void playInvaderExplode();
    static void playClick();

    static SoundEngine* m_s_Instance;
};
#endif
```

Create a new source file in the Source Files/Engine filter called `SoundEngine.cpp` and add the following code:

```
#include <SFML/Audio.hpp>
#include <assert.h>
#include "SoundEngine.h"
```



```
using namespace std;
using namespace sf;

SoundEngine* SoundEngine::m_s_Instance = nullptr;

SoundEngine::SoundEngine()
{
    assert(m_s_Instance == nullptr);
    m_s_Instance = this;

    // Load the sound into the buffers
    m_ShootBuffer.loadFromFile("sound/shoot.ogg");
    m_PlayerExplodeBuffer.loadFromFile("sound/playerexplode.ogg");
    m_InvaderExplodeBuffer.loadFromFile("sound/invaderexplode.ogg");
    m_ClickBuffer.loadFromFile("sound/click.ogg");

    // Associate the sounds with the buffers
    m_ShootSound.setBuffer(m_ShootBuffer);
    m_PlayerExplodeSound.setBuffer(m_PlayerExplodeBuffer);
    m_InvaderExplodeSound.setBuffer(m_InvaderExplodeBuffer);
    m_ClickSound.setBuffer(m_ClickBuffer);
}

void SoundEngine::playShoot()
{
    m_s_Instance->m_ShootSound.play();
}

void SoundEngine::playPlayerExplode()
{
    m_s_Instance->m_PlayerExplodeSound.play();
}

void SoundEngine::playInvaderExplode()
{
    m_s_Instance->m_InvaderExplodeSound.play();
}

void SoundEngine::playClick()
{
    m_s_Instance->m_ClickSound.play();
}
```

The `SoundEngine` class uses the exact same strategy as the previous `SoundManager` class from the previous projects. In fact, `SoundEngine` is slightly simpler than `SoundManager` because we are not using spatialization features. For a refresher of how the `SoundEngine` class works, refer to *Chapter 17, Sound Spatialization and the HUD*.

Now, we can move on to the `ScreenManager` class.

Coding the `ScreenManager` class

Create a new header file in the Header Files/Engine filter called `ScreenManager.h` and add the following code:

```
#pragma once
#include <SFML/Graphics.hpp>
#include <map>
#include "GameScreen.h"
#include "ScreenManagerRemoteControl.h"
#include "SelectScreen.h"
// #include "LevelManager.h"
#include "BitmapStore.h"
#include <iostream>

using namespace sf;
using namespace std;

class ScreenManager : public ScreenManagerRemoteControl {
private:
    map <string, unique_ptr<Screen>> m_Screens;
    //LevelManager m_LevelManager;

protected:
    string m_CurrentScreen = "Select";

public:
    BitmapStore m_BS;

    ScreenManager(Vector2i res);
    void update(float fps);
    void draw(RenderWindow& window);
    void handleInput(RenderWindow& window);

    /*****
    *****/
    From ScreenManagerRemoteControl interface
```

```
*****
*****/
void ScreenManagerRemoteControl::
    SwitchScreens(string screenToSwitchTo)
{
    m_CurrentScreen = "" + screenToSwitchTo;
    m_Screens[m_CurrentScreen]->initialise();
}

void ScreenManagerRemoteControl::
    loadLevelInPlayMode(string screenToLoad)
{
    //m_LevelManager.getGameObjects().clear();
    //m_LevelManager.
        //loadGameObjectsForPlayMode(screenToLoad);
    SwitchScreens("Game");
}

//vector<GameObject>&
//ScreenManagerRemoteControl::getGameObjects()
//{
//    //return m_LevelManager.getGameObjects();
//}

//GameObjectSharer& shareGameObjectSharer()
//{
//    //return m_LevelManager;
//}
};
```

In the previous code, there are some `#include` statements and some functions that have been commented out. This is because we will not be coding the `LevelManager` class until *Chapter 21, File I/O and the Game Object Factory*.

The next thing to notice is that `ScreenManager` inherits from `ScreenManagerRemoteControl`. More on this class shortly.

We have coded a map with a key-value pair of string and a unique pointer to `Screen`. This will allow us to grab the functionality of a specific `Screen` instance by using the corresponding string. Next, we declare the string called `m_CurrentScreen` and initialize it to `Select`.

Next, we declare an instance of `BitmapStore` called `m_BS`. This will be a slightly reworked version of the `TextureHolder` class that we saw in the two preceding projects. We will code the `BitmapStore` class next.

Notice that the constructor for `ScreenManager` takes a `Vector2i` instance, which is what we should expect from when we initialized a `ScreenManager` instance in the `GameEngine` class.

What follows is the `update`, `draw`, and `handleInput` function prototypes, which are called from the `GameEngine` class.

The next two functions are the most interesting. Note that they are from the `ScreenManagerRemoteControl` class, which `ScreenManager` inherits from. These are pure virtual functions in `ScreenManagerRemoteControl` and we do things this way so that we can share some of the functionality of the `ScreenManager` class with other classes. We will code the `ScreenManagerRemoteControl` class in a couple of sections time. Remember that, when you inherit from a class that has pure virtual functions, you must implement the functions if you want to create an instance. Furthermore, the implementations should be contained in the same file as where the class is declared. There are four functions, two of which have been commented out for now. The two functions of immediate interest are `SwitchScreens` and `loadLevelInPlayMode`.

The `SwitchScreen` function changes the value of `m_CurrentScreen`, while the `loadLevelInPlayMode` function has some temporarily commented out code and a single line of active code which calls `SwitchScreens` with the value of `Game`.

Let's move on to the `ScreenManager.cpp` file so that we can look at all the function definitions.

Create a new source file in the `Source Files/Engine` filter called `ScreenManager.cpp` and add the following code:

```
#include "ScreenManager.h"

ScreenManager::ScreenManager(Vector2i res)
{
    m_Screens["Game"] = unique_ptr<GameScreen>(
        new GameScreen(this, res));

    m_Screens["Select"] = unique_ptr<SelectScreen>(
        new SelectScreen(this, res));
}

void ScreenManager::handleInput(RenderWindow& window)
{
    m_Screens[m_CurrentScreen] ->handleInput(window);
}
```

```
void ScreenManager::update(float fps)
{
    m_Screens[m_CurrentScreen] ->update(fps);
}

void ScreenManager::draw(RenderWindow& window)
{
    m_Screens[m_CurrentScreen] ->draw(window);
}
```

In the preceding code, the constructor adds two Screen instances to the map instance – first, a GameScreen instance with a key of "Game" and then a SelectScreen instance with a key of "Select". The three functions, handleInput, update, and draw, use whatever the current screen is, use the corresponding Screen instance, and call its handleInput, update, and draw functions.

When the game is executed for the first time, the versions of these functions from SelectScreen will be called, but if the ChangeScreen or loadLevelInPlayMode function was called, then then handleInput, update, and draw could be called on the GameScreen instance from the map. You can add as many different types of Screen instance to the map as you like. I recommend that you complete the Space Invaders ++ project before you start doing your own customizations or start your own game, however.

Coding the BitmapStore class

Create a new header file in the Header Files/Engine filter called BitmapStore.h and add the following code:

```
#pragma once
#ifndef BITMAP_STORE_H
#define BITMAP_STORE_H

#include <SFML/Graphics.hpp>
#include <map>

class BitmapStore
{
private:
    std::map<std::string, sf::Texture> m_BitmapsMap;
    static BitmapStore* m_s_Instance;

public:
    BitmapStore();
```

```

        static sf::Texture& getBitmap(std::string const& filename);
        static void addBitmap(std::string const& filename);
    };
#endif

```

Create a new source file in the Source Files/Engine filter called `BitmapStore.cpp` and add the following code:

```

#include "BitmapStore.h"
#include <assert.h>

using namespace sf;
using namespace std;

BitmapStore* BitmapStore::m_s_Instance = nullptr;

BitmapStore::BitmapStore()
{
    assert(m_s_Instance == nullptr);
    m_s_Instance = this;
}

void BitmapStore::addBitmap(std::string const& filename)
{
    // Get a reference to m_Textures using m_S_Instance
    auto& bitmapsMap = m_s_Instance->m_BitmapsMap;
    // auto is the equivalent of map<string, Texture>

    // Create an iterator to hold a key-value-pair (kvp)
    // and search for the required kvp
    // using the passed in file name
    auto keyValuePair = bitmapsMap.find(filename);
    // auto is equivalent of map<string, Texture>::iterator

    // No match found so save the texture in the map
    if (keyValuePair == bitmapsMap.end())
    {
        // Create a new key value pair using the filename
        auto& texture = bitmapsMap[filename];
        // Load the texture from file in the usual way
        texture.loadFromFile(filename);
    }
}

```

```
sf::Texture& BitmapStore::getBitmap(std::string const& filename)
{
    // Get a reference to m_Textures using m_S_Instance
    auto& m = m_s_Instance->m_BitmapMap;
    // auto is the equivalent of map<string, Texture>

    // Create an iterator to hold a key-value-pair (kvp)
    // and search for the required kvp
    // using the passed in file name
    auto keyValuePair = m.find(filename);
    // auto is equivalent of map<string, Texture>::iterator

    // Did we find a match?
    if (keyValuePair != m.end())
    {
        return keyValuePair->second;
    }
    else
    {
#ifdef debuggingOnConsole
        cout <<
            "BitmapStore::getBitmap()Texture not found Crrrashh!"
            << endl;
#endif
        return keyValuePair->second;
    }
}
```

The preceding code is almost a copy and paste from the `BitmapStore` class from the previous two projects, except for the final `else` block. Inside the final `else` block, we use C++ assertions for the first time to output the name of the requested texture to the console in the event that the texture isn't found. This only happens when `debuggingOnConsole` is defined. Note that this would also crash the game.

Coding the ScreenManagerRemoteControl class

Create a new header file in the Header Files/Screens filter called `ScreenManagerRemoteControl.h` and add the following code:

```
#pragma once
#include <string>
#include <vector>
// #include "GameObject.h"
```

```
//#include "GameObjectSharer.h"

using namespace std;

class ScreenManagerRemoteControl
{
public:
    virtual void SwitchScreens(string screenToSwitchTo) = 0;
    virtual void loadLevelInPlayMode(string screenToLoad) = 0;
    //virtual vector<GameObject>& getGameObjects() = 0;
    //virtual GameObjectSharer& shareGameObjectSharer() = 0;
};
```

Note in the previous code, that there are some `#include` statements and some functions that have been commented out. This is because we will not be coding the `GameObject` and `GameObjectSharer` classes until the next chapter.

The rest of the code is for the prototypes that match the definitions we saw previously in the `ScreenManager.h` file. As you have come to expect, all the functions are pure virtual and therefore must be implemented by any class we wish to have an instance of.

Create a new source file in the `Source Files/Screens` filter called `ScreenManagerRemoteControl.cpp` and add the following code:

```
/*
*****THIS IS AN INTERFACE*****
*/
```

This code file is empty because all the code is in the `.h` file. In fact, you don't need to create this file, but I always find it a handy reminder in case I forget that all the functions for the class are pure virtual and waste time looking for the `.cpp` file, which doesn't exist.

Where are we now?

At this stage, the only remaining errors in the code are the errors that refer to the `SelectScreen` class and the `GameScreen` class. It is going to take quite a bit of work to get rid of these errors and have a runnable program. The reason for this is that `SelectScreen` and `GameScreen` are derived from `Screen` and, in turn, the `Screen` class is also dependent on `InputHandler`, `UIPanel`, and `Button`. We will get to them next.

Coding the Screen class and its dependents

What we will do now is code all the screen-related classes. In addition, each of the screens from our game will have their own specific implementation of all these classes.

Next, we will code all the base classes; `Screen`, `InputHandler`, `UIPanel`, and `Button`. Following that, we will do the full implementation of the `SelectScreen` derivations of these classes and a partial implementation of the `GameScreen` derivations. At this point, we will be able to run the game and see our screens, UI panels, and buttons in action, and also be able to switch between screens. In the next chapter, we will work on the game properly and implement `GameObject` and `LevelManager`. In *Chapter 22, Using Game Objects and Building a Game*, we will see how we can use them all in the `GameScreen` class.

Coding the Button class

Create a new header file in the Header Files/Screens filter called `Button.h` and add the following code:

```
#pragma once
#include <SFML/Graphics.hpp>

using namespace sf;

class Button
{
private:
    RectangleShape m_Button;
    Text m_ButtonText;
    Font m_Font;

public:
    std::string m_Text;
    FloatRect m_Collider;

    Button(Vector2f position,
           float width, float height,
           int red, int green, int blue,
           std::string text);

    void draw(RenderWindow& window);
};
```

As you can see from the preceding code, a button will be visually represented by an SFML `RectangleShape` instance and an SFML `Text` instance. Also note that there is a `FloatRect` instance named `m_Collider` that will be used to detect mouse clicks on the button. The constructor will receive arguments to configure the position, size, color, and text of the button. The button will draw itself once each frame of the game loop and has a `draw` function that receives a `RenderWindow` reference to enable this.

Create a new source file in the `Source Files/Screens` filter called `Button.cpp` and add the following code:

```
#include "Button.h"

Button::Button(Vector2f position,
               float width, float height,
               int red, int green, int blue,
               std::string text)
{
    m_Button.setPosition(position);
    m_Button.setFillColor(sf::Color(red, green, blue));
    m_Button.setSize(Vector2f(width, height));

    m_Text = "" + text;

    float textPaddingX = width / 10;
    float textPaddingY = height / 10;
    m_ButtonText.setCharacterSize(height * .7f);
    m_ButtonText.setString(text);
    m_Font.loadFromFile("fonts/Roboto-Bold.ttf");
    m_ButtonText.setFont(m_Font);
    m_ButtonText.setPosition(Vector2f((position.x + textPaddingX),
                                      (position.y + textPaddingY)));

    m_Collider = FloatRect(position, Vector2f(width, height));
}

void Button::draw(RenderWindow& window)
{
    window.draw(m_Button);
    window.draw(m_ButtonText);
}
```

Most of the action takes place in the constructor, and there is nothing we haven't seen already on numerous occasions in all the other projects. The button is prepared to be drawn using all the values that are passed into the constructor.

The draw function uses the `RenderWindow` reference to draw the previously configured `Text` instance on top of the previously configured `RectangleShape` instance.

Coding the `UIPanel` class

Create a new header file in the `Header Files/Screens` filter called `UIPanel.h` and add the following code:

```
#pragma once
#include <SFML/Graphics.hpp>
#include "Button.h"

using namespace std;

class UIPanel {
private:
    RectangleShape m_UIPanel;
    bool m_Hidden = false;
    vector<shared_ptr<Button>> m_Buttons;

protected:
    float m_ButtonWidth = 0;
    float m_ButtonHeight = 0;
    float m_ButtonPadding = 0;

    Font m_Font;
    Text m_Text;

    void addButton(float x, float y, int width, int height,
        int red, int green, int blue,
        string label);

public:
    View m_View;

    UIPanel(Vector2i res, int x, int y,
        float width, float height,
        int alpha, int red, int green, int blue);

    vector<shared_ptr<Button>> getButtons();
    virtual void draw(RenderWindow& window);
    void show();
    void hide();
};
```

The `private` section of the `UIPanel` class consists of a `RectangleShape` that will visually represent the background of the panel, a `Boolean` to keep track of whether the panel is currently visible to the player, and a `vector` of smart pointers to hold all the `Button` instances for this panel. Note that the smart pointers are of the `shared` variety so that we can pass them around and let the `shared_pointer` class take care of counting the references and deleting the memory when necessary.

In the `protected` section, there are member variables for remembering the size and spacing of the buttons, as well as a `Text` and a `Font` instance for drawing text on the panel. All the panels in this project have just one `Text` instance, but specific derived classes are free to add extra members as they need. For example, a `HighScoreUIPanel` class might need a `vector` full of `Text` instances to draw a list of the highest scores.

There is also an `addButton` function, and it is this function that will call the `Button` class constructor and add the instances to the `vector`.

In the `public` section, we can see that every `UIPanel` instance will have its own `View` instance. This enables every panel and screen to configure its `View` however it likes. All the `View` instances will be drawn to and added to `RenderWindow` in layers.

The `UIPanel` constructor receives all the necessary sizes and colors to configure its `RectangleShape`. The `getButtons` function shares the `vector` of `Button` instances so that other classes can interact with the buttons. For example, the `InputHandler` class will need the buttons to detect mouse clicks on them. This is why we used `shared` smart pointers.

The `draw` function, of course, is called once each frame of the game loop and is `virtual`, so it can be optionally overridden and customized by derived classes. The `show` and `hide` functions will toggle the value of `m_Hidden` to keep track of whether this panel is currently visible to the player.

Create a new source file in the `Source Files/Screens` filter called `UIPanel.cpp` and add the following code:

```
#include "UIPanel.h"

UIPanel::UIPanel(Vector2i res, int x, int y,
    float width, float height,
    int alpha, int red, int green, int blue)
{
    m_UIPanel.setFillColor(sf::Color(red, green, blue, alpha));

    // How big in pixels is the UI panel
```

```
m_UIPanel.setSize(Vector2f(width, height));

// How big in pixels is the view
m_View.setSize(Vector2f(width, height));

// Where in pixels does the center of the view focus
// This is most relevant when drawing a portion
// of the game world
// width/2, height/2 ensures it is exactly centered around the
// RectangleShape, m_UIPanel
m_View.setCenter(width / 2, height / 2);

// Where in the window is the view positioned?
float viewportStartX = 1.f / (res.x / x);
float viewportStartY = 1.f / (res.y / y);
float viewportSizeX = 1.f / (res.x / width);
float viewportSizeY = 1.f / (res.y / height);

// Params from left to right
// StartX as a fraction of 1, startY as a fraction of 1
// SizeX as a fraction of 1
// SizeY as a fraction of 1
m_View.setViewport(FloatRect(viewportStartX, viewportStartY,
    viewportSizeX, viewportSizeY));
}

vector<shared_ptr<Button>> UIPanel::getButtons()
{
    return m_Buttons;
}

void UIPanel::addButton(float x, float y,
    int width, int height,
    int red, int green, int blue,
    string label)
{
    m_Buttons.push_back(make_shared<Button>(Vector2f(x, y),
        width, height,
        red, green, blue,
        label));
}
```

```

void UIPanel::draw(RenderWindow & window)
{
    window.setView(m_View);
    if (!m_Hidden) {
        window.draw(m_UIPanel);
        for (auto it = m_Buttons.begin();
             it != m_Buttons.end(); ++it)
        {
            (*it)->draw(window);
        }
    }
}

void UIPanel::show()
{
    m_Hidden = false;
}

void UIPanel::hide()
{
    m_Hidden = true;
}

```

In the constructor, the `RectangleShape` instance is scaled, colored, and positioned. The `View` instance is scaled to the size of the panel as well. The `setViewport` function of the `View` class is used along with some extra calculations to make sure the `View` takes up the correct proportion of the screen relative to the resolution and will therefore look approximately the same on screens of varying resolutions.

The `getButtons` function simply returns the vector of buttons to the calling code. The `addButtons` function uses the `make_shared` function to allocate new `Button` instances on the heap and place them into the vector.

The `draw` function uses the `setView` function to make the specific `View` instance of this panel the one that is drawn upon. Next, there's `RectangleShape`, which represents that this panel is drawn. Then, each of the buttons in the vector are looped through and drawn on top of the `RectangleShape`. All this drawing will only happen if `m_Hidden` is false.

The `show` and `hide` functions allow users of the class to toggle `m_Hidden`.

Coding the InputHandler class

Create a new header file in the Header Files/Screens filter called InputHandler.h and add the following code:

```
#pragma once
#include <SFML/Graphics.hpp>
#include <vector>
#include "Button.h"
#include "Screen.h"
#include "ScreenManagerRemoteControl.h"

using namespace sf;
using namespace std;

class Screen;

class InputHandler
{
private:
    Screen* m_ParentScreen;
    vector<shared_ptr<Button>> m_Buttons;
    View* m_PointerToUIPanelView;
    ScreenManagerRemoteControl* m_ScreenManagerRemoteControl;

public:
    void initialiseInputHandler(
        ScreenManagerRemoteControl* sw,
        vector<shared_ptr<Button>>,
        View* pointerToUIView,
        Screen* parentScreen);

    void handleInput(RenderWindow& window, Event& event);

    virtual void handleGamepad();
    virtual void handleKeyPressed(Event& event,
        RenderWindow& window);

    virtual void handleKeyReleased(Event& event,
        RenderWindow& window);

    virtual void handleLeftClick(string& buttonInteractedWith,
        RenderWindow& window);

    View* getPointerToUIView();
```

```

    ScreenManagerRemoteControl*
        getPointerToScreenManagerRemoteControl();

    Screen* getmParentScreen();
};

```

There is an error in this file because the `Screen` class doesn't exist yet.

First, study the `private` section of this header file. Each `InputHandler` instance will hold a pointer to the screen that holds it. This will be useful in a few situations we will come across as the project continues. There is also a vector of shared smart pointers to `Button` instances. These are the same `Button` instances that are in the `UIPanel` we just coded. Each derived `UIPanel` will have a matching derived `InputHandler` with which it shares a vector of buttons.

The `InputHandler` class also holds a pointer to the `View` instance in the `UIPanel`. When we code the function definitions in `InputHandler.cpp`, we will see how we get this pointer and how it is useful.

There is also a pointer to `ScreenManagerRemoteControl`. Remember from the `ScreenManager` class that we have implemented some functions from `ScreenManagerRemoteControl`. This is what will give us access to functions such as `SwitchScreen`. This is very useful when you consider that `InputHandler` is the class where we will be detecting button clicks. Of course, we need to see how we can initialize this pointer to make it usable. We will see how in the `InputHandler.cpp` file soon.

In the `public` section, there is an `initialiseInputHandler` function. This is where the `private` members we have just talked about will be prepared for use. Look at the parameters; they match the types of the `private` members exactly.

Next is the `handleInput` function. Remember that this is called once per frame by the `GameEngine` class; the `ScreenManager` calls it on the current screen and the `Screen` class (coded next), in turn, will call it on all `InputHandler` instances that it holds. It receives a `RenderWindow` and an `Event` instance.

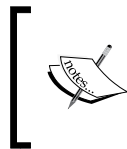
Next, there are four `virtual` functions which each derived from the `InputHandler` class that it can optionally override if it needs to. They are as follows:

- `handleGamepad`
- `handleKeyPressed`
- `handleKeyReleased`
- `handleLeftClick`

As we will see shortly, in the `InputHandler.cpp` file, the `handleInput` function will loop through the data in `Event`, just as we have done so often before. But then, instead of handling all the events directly as we have done in the past, it will delegate a response to one of the four virtual functions. The derived classes will then receive only the events and data they have decided they want to handle. Default and empty definitions of the four virtual functions are provided in the `InputHandler.cpp` file.

The `getPointerToUIView` function will return the pointer to the panels `View` that this `InputHandler` instance holds. We will see shortly that we need `View` in order to do mouse click collision detection on the buttons.

`getPointerToScreenManagerRemoteControl` and `getmParentScreen` return pointers to the member variables that are suggested by the names of the functions.



Note that, if you make the private data protected, then the derived `InputHandler` classes can access the data without going through the functions we have just discussed. When the project is complete, feel free to revisit this section and change this if you wish.

Now, we can code all the function definitions.

Create a new source file in the `Source Files/Screens` filter called `InputHandler.cpp` and add the following code:

```
#include <sstream>
#include "InputHandler.h"

using namespace sf;
using namespace std;

void InputHandler::initialiseInputHandler(
    ScreenManagerRemoteControl* sw,
    vector<shared_ptr<Button>> buttons,
    View* pointerToUIView,
    Screen* parentScreen)
{
    m_ScreenManagerRemoteControl = sw;
    m_Buttons = buttons;
    m_PointerToUIPanelView = pointerToUIView;
    m_ParentScreen = parentScreen;
}

void InputHandler::handleInput(RenderWindow& window,
    Event& event)
```

```

{
    // Handle any key presses
    if (event.type == Event::KeyPressed)
    {
        handleKeyPressed(event, window);
    }

    if (event.type == Event::KeyReleased)
    {
        handleKeyReleased(event, window);
    }

    // Handle any left mouse click released
    if (event.type == Event::MouseButtonReleased)
    {
        auto end = m_Buttons.end();

        for (auto i = m_Buttons.begin();
             i != end;
             ++i) {

            if ((*i)->m_Collider.contains(
                window.mapPixelToCoords(Mouse::getPosition(),
                    (*getPointerToUIView()))))
            {
                // Capture the text of the button that was interacted
                // with and pass it to the specialised version
                // of this class if implemented
                handleLeftClick((*i)->m_Text, window);
                break;
            }
        }
    }

    handleGamepad();
}

void InputHandler::handleGamepad()
{} // Do nothing unless handled by a derived class

void InputHandler::handleKeyPressed(Event& event,
    RenderWindow& window)
{} // Do nothing unless handled by a derived class

```

```
void InputHandler::handleKeyReleased(Event& event,
    RenderWindow& window)
{// Do nothing unless handled by a derived class

void InputHandler::handleLeftClick(std::
    string& buttonInteractedWith,
    RenderWindow& window)
{// Do nothing unless handled by a derived class

View* InputHandler::getPointerToUIView()
{
    return m_PointerToUIPanelView;
}

ScreenManagerRemoteControl*
    InputHandler::getPointerToScreenManagerRemoteControl()
{
    return m_ScreenManagerRemoteControl;
}

Screen* InputHandler::getmParentScreen() {
    return m_ParentScreen;
}
```

The `initialiseInputHandler` function initializes the private data, as we have already discussed, the four virtual functions are empty, as expected, and the getter functions return pointers to the private members, just like we said they would.

The interesting function definition is the `handleInput` function, so let's go through it.

There is a series of `if` statements, which should look familiar from previous projects. Each `if` statement tests for a different type of event, such as a key being pressed or a key being released. Instead of handling the event, however, the appropriate virtual function is called. If the derived `InputHandler` class overrides the virtual function, it will receive the data and get to handle the event. If it doesn't, then the empty default function definition is called, and nothing happens.

When the `MouseButtonReleased` event occurs, each of the `Button` instances in the vector is tested to see if the click occurred within the button. This is achieved using the `contains` function on the collider in each button and passing in the position of the mouse click. Note that the button coordinates are relative to the panels `View` and not the screen coordinates. For this reason, the `mapPixelToCoords` function is used to convert the screen coordinates of the mouse click into the corresponding coordinates of the `View`.

When a collision is detected, the `handleLeftClick` virtual function is called and the text from the button is passed in. The derived `InputHandler` classes will handle what happens on a button click based on the text of the button.

The final line of code in the `handleInput` function calls the final virtual function called `handleGamepad`. Any derived `InputHandler` classes that implement this function will get a chance to respond to the player's actions with the gamepad. In this project, only `GameInputHandler` will be concerned with what the gamepad is doing. You could adapt the project to allow the player to use the gamepad to navigate the menus of the other screen if you want to.

Coding the Screen class

Create a new header file in the Header Files/Screens filter called `Screen.h` and add the following code:

```
#pragma once
#include <SFML/Graphics.hpp>
#include <vector>
#include "InputHandler.h"
#include "UIPanel.h"
#include "ScreenManagerRemoteControl.h"

class InputHandler;

class Screen {
private:
    vector<shared_ptr<InputHandler>> m_InputHandlers;
    vector<unique_ptr<UIPanel>> m_Panels;

protected:
    void addPanel(unique_ptr<UIPanel> p,
                 ScreenManagerRemoteControl* smrc,
                 shared_ptr<InputHandler> ih);

public:
    virtual void initialise();
    void virtual update(float fps);
    void virtual draw(RenderWindow& window);
    void handleInput(RenderWindow& window);

    View m_View;
};
```

In the `private` section of the preceding code, there is a vector of shared smart pointers to `InputHandler` instances. This is where we will store all the derived `InputHandler` instances. `SelectScreen` will actually only have one `InputHandler`, while `GameScreen` will have two, but you can have as many as you like. Consider, for example, a hypothetical settings screen where you might have options for graphics, sound, controller, gameplay, and so on. Each of these options could then be clicked to reveal a unique `UIPanel` instance with a related `InputHandler`. So, we could have avoided using a vector for this project, but any significant project would almost certainly need a vector eventually. The smart pointers are of the shared variety, indicating we will be passing the contents via a function at some point.

The next member is a vector of unique smart pointers to `UIPanel` instances. This is where all the derived `UIPanel` instances will go. The unique variety of pointer indicates we will not share the pointers; if we do, we will have to transfer responsibility.

In the protected section is the `addPanel` function, which is where a `Screen` will pass in all the details of a new `UIPanel` instance, including its related `InputHandler`. Note the parameter to receive a `ScreenManagerRemoteControl` pointer; remember that this is required for passing to `InputHandler`.

There is an `initialise` function as well, which we will see the purpose of shortly. The final three functions are the virtual functions, that is, `update`, `draw` and `handleInput`, which the derived `Screen` classes can override as they see fit.

Finally, take note of the `view` instance. Every `Screen` instance will also have its own `view` instance to draw to, just like each `UIPanel` does.

Let's take a look at the implementation of the functions we have just discussed.

Create a new source file in the `Source Files/Screens` filter called `Screen.cpp` and add the following code:

```
#include "Screen.h"

void Screen::initialise() {}

void Screen::addPanel(unique_ptr<UIPanel> uip,
    ScreenManagerRemoteControl* smrc,
    shared_ptr<InputHandler> ih)
{
    ih->initialiseInputHandler(smrc,
        uip->getButtons(), &uip->m_View, this);
    // Use move() because otherwise
    // the vector has a COPY which is not allowed
```

```

        m_Panels.push_back(move(ui));
        m_InputHandlers.push_back(ih);
    }

void Screen::handleInput(RenderWindow& window)
{
    Event event;
    auto itr = m_InputHandlers.begin();
    auto end = m_InputHandlers.end();
    while (window.pollEvent(event))
    {
        for (itr;
            itr != end;
            ++itr)
        {
            (*itr)->handleInput(window, event);
        }
    }
}

void Screen::update(float fps){}

void Screen::draw(RenderWindow& window)
{
    auto itr = m_Panels.begin();
    auto end = m_Panels.end();
    for (itr;
        itr != end;
        ++itr)
    {
        (*itr)->draw(window);
    }
}

```

The `initialise` function is empty. It is designed to be overridden.

The `addPanel` function, as we already know, stores the `InputHandler` and `UIPanel` instances that are passed to it. When an `InputHandler` is passed in, the `initialiseInputHandler` function is called and three things are passed in. First is the vector of `Button` instances, next is the `View` instance from the related `UIPanel` instance, and third is the `this` argument. In the current context, `this` is a pointer to the `Screen` instance itself. Why not refer to the `InputHandler` class and verify that these arguments are correct and what happens to them?

Next, the panel and the input handler are added to the appropriate vector. Something interesting happens, however, if you look closely. Take another look at the line of code which adds the `UIPanel` instance called `uip` to the `m_Panels` vector:

```
m_Panels.push_back(move(uip));
```

The argument that's passed to `push_back` is encased in a call to `move`. This transfers responsibility for the unique pointer to the `UIPanel` in the vector. Any attempt to use `uip` after this point will result in a read access violation because `uip` is now a null pointer. The pointer in `m_Panels`, however, is good to go. You will probably agree that this is simpler than using a regular pointer and working out where to delete it.

The `handleInput` function loops through every event, passing it in to each `InputHandler` in turn.

The `update` function has no functionality in the base class and is empty.

The `draw` function loops through every `UIPanel` instance and calls their `draw` functions.

Now, we are ready to code all the derived classes. We will start with the select screen (`SelectScreen`) and then move on to the game screen (`GameScreen`). We will add one more quick class first, though.

Adding the `WorldState.h` file

Create a new header file in the `Header Files/Engine` filter called `WorldState.h` and add the following code:

```
#pragma once

class WorldState
{
public:
    static const int WORLD_WIDTH = 100;
    static int WORLD_HEIGHT;
    static int SCORE;
    static int LIVES;
    static int NUM_INVADERS_AT_START;
    static int NUM_INVADERS;
    static int WAVE_NUMBER;
};
```

These variables are public and static. As a result, they will be accessible throughout the project and are guaranteed to have only a single instance.

Coding the derived classes for the select screen

So far, we have coded the fundamental classes that represent the user interface, as well as the logical division of our game into screens. Next, we will code specific implementations of each of them. Remember that *Space Invaders ++* will have two screens: select and game. The select screen will be represented by the `SelectScreen` class and will have a single `UIPanel` instance, a single `InputHandler` instance, and two buttons. The play screen will be represented by the `GameScreen` class and it will have two `UIPanel` instances. One is called `GameUIPanel` and will display the score, lives, and invader wave number. The other is called `GameOverUIPanel` and will display two buttons, giving the player the option to go back to the select screen or play again. As the `GameScreen` class is composed of two `UIPanel` instances, it will also be composed of two `InputHandler` instances.

Coding the `SelectScreen` class

Create a new header file in the Header Files/Screens/Select filter called `SelectScreen.h` and add the following code:

```
#pragma once
#include "Screen.h"

class SelectScreen : public Screen
{
private:
    ScreenManagerRemoteControl* m_ScreenManagerRemoteControl;

    Texture m_BackgroundTexture;
    Sprite m_BackgroundSprite;

public:
    SelectScreen(ScreenManagerRemoteControl* smrc, Vector2i res);
    void virtual draw(RenderWindow& window);
};
```


The `SelectScreen` class inherits from `Screen`. In the private section of the preceding code, there is a `ScreenManagerRemoteControl` pointer for switching screens, as well as a `Texture` instance and `Sprite` instance for drawing a background.

In the public section, we can see the constructor and the prototype that overrides the `draw` function. The `SelectScreen` class does not need to override the `update` function.

Create a new source file in the `Source Files/Screens/Select` filter called `SelectScreen.cpp` and add the following code:

```
#include "SelectScreen.h"
#include "SelectUIPanel.h"
#include "SelectInputHandler.h"

SelectScreen::SelectScreen(
    ScreenManagerRemoteControl* smrc, Vector2i res)
{
    auto suip = make_unique<SelectUIPanel>(res);
    auto sih = make_shared<SelectInputHandler>();
    addPanel(move(suip), smrc, sih);
    m_ScreenManagerRemoteControl = smrc;

    m_BackgroundTexture.loadFromFile("graphics/background.png");

    m_BackgroundSprite.setTexture(m_BackgroundTexture);
    auto textureSize = m_BackgroundSprite.
        getTexture()->getSize();

    m_BackgroundSprite.setScale(float(
        m_View.getSize().x) / textureSize.x,
        float(m_View.getSize().y) / textureSize.y);
}

void SelectScreen::draw(RenderWindow& window)
{
    // Change to this screen's view to draw
    window.setView(m_View);
    window.draw(m_BackgroundSprite);

    // Draw the UIPanel view(s)
    Screen::draw(window);
}
```

In the constructor, the purpose of all the coding so far begins to come together. The `make_unique` function is used to create a unique smart pointer to a `SelectUIPanel` instance. We will code `SelectUIPanel` in a couple of sections time. Next, the `make_shared` function is used to create a shared smart pointer to a `SelectInputHandler` instance. We will code the `SelectInputHandler` class next. Now that we have a `UIPanel` and an `InputHandler` in the appropriate forms, we can call the `addPanel` function and pass them both in. Note that, in the call to `addPanel`, `suip` is wrapped in a call to `move`. Any use of `suip` after this point is not possible without crashing the program because it is now a null pointer, since ownership has been moved to the function argument. Remember that, inside the `Screen` class `addPanel` function, ownership is moved again when the unique pointer to `UIPanel` is stashed away in the vector of `UIPanel` instances.

Following this, the `ScreenManagerRemoteControl` pointer is initialized and can now be used to switch to another screen when required.

The final few lines of code in the constructor create and scale a `Sprite` instance that uses the `background.png` image, which will fill the entire screen.

In the `draw` function, the call to the `setView` function makes this panel's `View` instance the one to draw on, and then the `Sprite` instance is drawn to the `RenderWindow` instance.

Finally, the `draw` function is called on the base `Screen` class, which draws all the panels and their related buttons. In this specific case, it draws just a single panel, `SelectUIPanel`, which we will code right after we have coded `SelectInputHandler`.

Coding the `SelectInputHandler` class

Create a new header file in the `Header Files/Screens/Select` folder called `SelectInputHandler.h` and add the following code:

```
#pragma once
#include "InputHandler.h"

class SelectInputHandler : public InputHandler
{
public:
    void handleKeyPressed(Event& event,
        RenderWindow& window) override;

    void handleLeftClick(std::string& buttonInteractedWith,
        RenderWindow& window) override;
};
```

The `SelectInputHandler` class inherits from `InputHandler` and overrides the `handleKeyPressed` and `handleLeftClick` functions. Let's see how these functions are implemented.

Create a new source file in the `Source Files/Screens/Select` filter called `SelectInputHandler.cpp` and add the following code:

```
#include "SelectInputHandler.h"
#include "SoundEngine.h"
#include "WorldState.h"
#include <iostream>

int WorldState::WAVE_NUMBER;

void SelectInputHandler::handleKeyPressed(
    Event& event, RenderWindow& window)
{
    // Quit the game
    if (Keyboard::isKeyPressed(Keyboard::Escape))
    {
        window.close();
    }
}

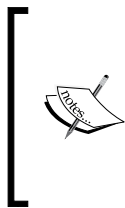
void SelectInputHandler::handleLeftClick(
    std::string& buttonInteractedWith, RenderWindow& window)
{
    if (buttonInteractedWith == "Play") {
        SoundEngine::playClick();
        WorldState::WAVE_NUMBER = 0;
        getPointerToScreenManagerRemoteControl()
            ->loadLevelInPlayMode("level1");
    }

    if (buttonInteractedWith == "Quit") {
        SoundEngine::playClick();
        window.close();
    }
}
```

The `handleKeyPressed` function interacts with just one keyboard key. When the *Escape* key is pressed, the game quits.

In the `handleLeftClick` function, there are two `if` statements. Remember that the `handleInputFunction` of the `InputHandler` class passes in the text of the button that was clicked, along with a reference to `RenderWindow`. If the **Play** button is clicked, then a click sound is played, the `WAVE_NUMBER` variable is set to zero, and the `ScreenManagerRemoteControl` pointer calls the `loadLevelInPlayMode` function. The `loadLevelInPlayMode` function has its definition in `ScreenManagerClass`. Eventually, this function will indeed load a level from the passed in file name, but for now, it simply changes screen to the play screen.

If the **Quit** button is clicked, then the game is exited.



At this stage, despite including `WorldState.h`, you might have an error using `WorldState::WaveNumber`. This is fine; this is happening because of the order in which the classes are parsed by Visual Studio. When we add all the game screen-related classes that also use `WorldState.h`, which is parsed before this file, the error will be gone.

Let's code `SelectUIPanel`. Then, we can move on to the `GameScreen` class.

Coding the `SelectUIPanel` class

Create a new header file in the `Header Files/Screens/Select` filter called `SelectUIPanel.h` and add the following code:

```
#pragma once
#include "UIPanel.h"
class SelectUIPanel : public UIPanel
{
private:
    void initialiseButtons();

public:
    SelectUIPanel(Vector2i res);
    void virtual draw(RenderWindow& window);
};
```

The `SelectUIPanel` class inherits from `UIPanel` and overrides the `draw` function. In the preceding header file, you can also see that there is a function called `initialiseButtons`, as well as a constructor. Let's code the definitions.

Create a new source file in the Source Files/Screens/Select filter called `SelectUIPanel.cpp` and add the following code:

```
#include "SelectUIPanel.h"
#include <iostream>

SelectUIPanel::SelectUIPanel(Vector2i res) :
    // Create a new UIPanel
    // by calling the super-class constructor
    UIPanel(res,
        (res.x / 10) * 2, // Start 2/10 across
        res.y / 3, // 1/3 of the resolution from the top
        (res.x / 10) * 6, // as wide as 6/10 of the resolution
        res.y / 3, // and as tall as 1/3 of the resolution
        50, 255, 255, 255) // a, r, g, b
{
    m_ButtonWidth = res.x / 20;
    m_ButtonHeight = res.y / 20;
    m_ButtonPadding = res.x / 100;

    m_Text.setFillColor(sf::Color(0, 255, 0, 255));
    m_Text.setString("SPACE INVADERS ++");

    //https://www.dafont.com/roboto.font
    m_Font.loadFromFile("fonts/Roboto-Bold.ttf");
    m_Text.setFont(m_Font);

    m_Text.setPosition(Vector2f(m_ButtonPadding,
        m_ButtonHeight + (m_ButtonPadding * 2)));

    m_Text.setCharacterSize(160);

    initialiseButtons();
}

void SelectUIPanel::initialiseButtons()
{
    // Buttons are positioned relative to the top left
    // corner of the UI panel(m_View in UIPanel)
    addButton(m_ButtonPadding,
        m_ButtonPadding,
        m_ButtonWidth,
        m_ButtonHeight,
```

```

        0, 255, 0,
        "Play");

    addButton(m_ButtonWidth + (m_ButtonPadding * 2),
        m_ButtonPadding,
        m_ButtonWidth,
        m_ButtonHeight,
        255, 0, 0,
        "Quit");
}

void SelectUIPanel::draw(RenderWindow& window)
{
    show();
    UIPanel::draw(window);
    window.draw(m_Text);
}

```

The constructor receives the screen resolution and immediately uses that data to call the superclass constructor. By doing calculations with the values stored in `res`, the starting position and size of the panel is calculated. It is important that this calculation is done here and not in the `UIPanel` class because every `UIPanel` will be a different size and in a different position. Take a look at the comments in the preceding code if you are interested in the effect of each of the specific calculations. The color is also passed in using alpha, red, green, and blue values.

Next, the member variables from the base class that determine button size and spacing are initialized. The value of 20 is just an arbitrary value that works, but the important part is that all the values are based on the resolution of the screen, so they will scale well over different screen resolutions.

The next few lines of code prepare a `Text` instance, ready to be shown in the `draw` function. Finally, in the constructor, the `initialiseButtons` function is called.

In the `initialiseButtons` function, the `addButton` function is called twice, creating a green button with "Play" on it and a red button with "Quit" on it.

There might be some errors because of the use of the `WorldState.h` file. These can be ignored as they will correct themselves as we proceed with the next few classes.

Now, we can code all the game screen-related classes.

Coding the derived classes for the game screen

The structure of all these classes is the same as the select screen-related classes. I will be sure to point out where they vary, however. Most of the significant differences will be discussed across the next three chapters, however, because that is when we will code all the game objects and components and then put them to work in the `GameScreen` class.

The first difference is that the `GameScreen` class has two `UIPanel` instances and two `InputHandler` instances.

Coding the `GameScreen` class

Create a new header file in the `Header Files/Screens/Game` filter called `GameScreen.h` and add the following code:

```
#pragma once
#include "Screen.h"
#include "GameInputHandler.h"
#include "GameOverInputHandler.h"

class GameScreen : public Screen
{
private:
    ScreenManagerRemoteControl* m_ScreenManagerRemoteControl;
    shared_ptr<GameInputHandler> m_GIH;

    Texture m_BackgroundTexture;
    Sprite m_BackgroundSprite;
public:
    static bool m_GameOver;

    GameScreen(ScreenManagerRemoteControl* smrc, Vector2i res);
    void initialise() override;
    void virtual update(float fps);
    void virtual draw(RenderWindow& window);
};
```

Note that this is not the finished code – we will add more features to this file in the next chapter. This is just enough code so that we can run the game and see some basic functionality at the end of this chapter.

The code is familiar to the `SelectScreen` class. We also override the `initialise` and `update` functions. Furthermore, we have added a Boolean called `m_GameOver`, which will keep track of whether the game is currently playing.

Let's move on to the function implementations.

Create a new source file in the `Source Files/Screens/Game` filter called `GameScreen.cpp` and add the following code:

```
#include "GameScreen.h"
#include "GameUIPanel.h"
#include "GameInputHandler.h"
#include "GameOverUIPanel.h"
#include "WorldState.h"

int WorldState::WORLD_HEIGHT;
int WorldState::NUM_INVADERS;
int WorldState::NUM_INVADERS_AT_START;

GameScreen::GameScreen(ScreenManagerRemoteControl* smrc,
    Vector2i res)
{
    m_GIH = make_shared<GameInputHandler>();
    auto guip = make_unique<GameUIPanel>(res);
    addPanel(move(guip), smrc, m_GIH);

    auto m_GOIH = make_shared<GameOverInputHandler>();
    auto gouip = make_unique<GameOverUIPanel>(res);
    addPanel(move(gouip), smrc, m_GOIH);

    m_ScreenManagerRemoteControl = smrc;
    float screenRatio = VideoMode::getDesktopMode().width /
        VideoMode::getDesktopMode().height;

    WorldState::WORLD_HEIGHT = WorldState::WORLD_WIDTH /
        screenRatio;

    m_View.setSize(
        WorldState::WORLD_WIDTH, WorldState::WORLD_HEIGHT);

    m_View.setCenter(Vector2f(WorldState::WORLD_WIDTH /
        2, WorldState::WORLD_HEIGHT / 2));

    m_BackgroundTexture.loadFromFile("graphics/background.png");
```



```
        m_BackgroundSprite.setTexture(m_BackgroundTexture);
        auto textureSize = m_BackgroundSprite.getTexture()->getSize();
        m_BackgroundSprite.setScale(float(m_View.getSize().x) /
            textureSize.x,
            float(m_View.getSize().y) / textureSize.y);
    }

    void GameScreen::initialise()
    {
        m_GIH->initialize();

        WorldState::NUM_INVADERS = 0;

        m_GameOver = false;

        if (WorldState::WAVE_NUMBER == 0)
        {
            WorldState::NUM_INVADERS_AT_START =
                WorldState::NUM_INVADERS;

            WorldState::WAVE_NUMBER = 1;
            WorldState::LIVES = 3;
            WorldState::SCORE = 0;
        }
    }

    void GameScreen::update(float fps)
    {
        Screen::update(fps);

        if (!m_GameOver)
        {
            if (WorldState::NUM_INVADERS <= 0)
            {
                WorldState::WAVE_NUMBER++;
                m_ScreenManagerRemoteControl->
                    loadLevelInPlayMode("level1");
            }

            if (WorldState::LIVES <= 0)
            {
                m_GameOver = true;
            }
        }
    }
}
```

```

    }
}

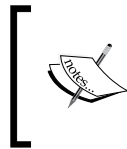
void GameScreen::draw(RenderWindow& window)
{
    // Change to this screen's view to draw
    window.setView(m_View);
    window.draw(m_BackgroundSprite);

    // Draw the UIPanel view(s)
    Screen::draw(window);
}

```

Everything that happened in the `SelectScreen` class happens here too, but for two `UIPanel` instances and two `InputHandler` instances. The next difference is that `GameScreen` does implement the update function. This is where all the game objects will be updated each frame of the game.

The next difference is that we have added some basic logic for the game into the `initialise` and `update` functions.



I apologize for the inconsistent spelling of the `initialise` and `initialize` functions. Changing them at this current stage of production is more likely to introduce errors into the book than help you out.

In the `initialize` function, the code calls the `initialize` function of the `GameInputHandler` class that we will code next. The `NUM_INVADERS` variable is set to zero, while `m_GameOver` is set to false. Next, the `WAVE_NUMBER` variable is tested and, if it equals zero, then the `WorldState` class has its static variables initialized, ready for a new game.

In the `update` function, the `m_GameOver` variable is used to determine whether the game is running and, if it is, two more tests are done. The first tests whether all the invaders have been destroyed. At this stage of development, because there aren't any invaders, this has the effect of constantly incrementing the wave number.

The second test checks whether the player has run out of lives and if they have, then `m_GameOver` is set to true.

Coding the GameInputHandler class

Create a new header file in the Header Files/Screens/Game filter called `GameInputHandler.h` and add the following code:

```
#pragma once
#include "InputHandler.h"

class GameScreen;

class GameInputHandler : public InputHandler
{
public:

    void initialize();
    void handleGamepad() override;
    void handleKeyPressed(Event& event,
        RenderWindow& window) override;

    void handleKeyReleased(Event& event,
        RenderWindow& window) override;
};
```

This class works the same way as `SelectInputHandler` does, but we need to override more of the functions. We will add code to the `initialize`, `handleGamepad`, `handleKeyPressed`, and `handleKeyReleased` functions here.

This is not the finished code – we will add lots more features to this file in the next chapter. This is just enough code so that we can run the game and see some basic functionality at the end of the chapter.

Create a new source file in the Source Files/Screens/Game filter called `GameInputHandler.cpp` and add the following code:

```
#include "GameInputHandler.h"
#include "SoundEngine.h"
#include "GameScreen.h"

void GameInputHandler::initialize() {
}

void GameInputHandler::handleGamepad()
{
}

void GameInputHandler::handleKeyPressed(
```

```

        Event& event, RenderWindow& window)
    {
        // Handle key presses
        if (event.key.code == Keyboard::Escape)
        {
            SoundEngine::playClick();
            getPointerToScreenManagerRemoteControl()->
                SwitchScreens("Select");
        }
    }

    void GameInputHandler::handleKeyReleased(
        Event& event, RenderWindow& window)
    {
    }

```

For now, we only want to add code to the `handleKeyPressed` function, but why not add the other empty functions that are shown in the preceding code? When the player presses the *Escape* key, the `ScreenMangerRemoteControl` pointer calls the `SwitchScreen` function to go back to the select screen.

This is not the finished code – we will add lots more features to this file in the next chapter. This is just enough code so that we can run the game and see some basic functionality at the end of the chapter.

Coding the GameUIPanel class

Create a new header file in the Header Files/Screens/Game filter called `GameUIPanel.h` and add the following code:

```

#pragma once
#include "UIPanel.h"

class GameUIPanel : public UIPanel
{
public:
    GameUIPanel(Vector2i res);
    void draw(RenderWindow& window) override;
};

```

Like the previous `UIPanel` child class, we'll override the `draw` function and also implement the constructor. Let's code these functions now.

Create a new source file in the Source Files/Screens/Game filter called `GameUIPanel.cpp` and add the following code:

```
#include "GameUIPanel.h"
#include <sstream>
#include "WorldState.h"

int WorldState::SCORE;
int WorldState::LIVES;

GameUIPanel::GameUIPanel(Vector2i res) :
    UIPanel(res,
        1, // The left
        1, // The top
        res.x / 3, // 1/3 width screen
        res.y / 12,
        50, 255, 255, 255) // a, r, g, b
{
    m_Text.setFill(sf::Color(0, 255, 0, 255));
    m_Text.setString("Score: 0 Lives: 3 Wave: 1");
    m_Font.loadFromFile("fonts/Roboto-Bold.ttf");
    m_Text.setFont(m_Font);
    m_Text.setPosition(Vector2f(15,15));
    m_Text.setCharacterSize(60);
}

void GameUIPanel::draw(RenderWindow& window)
{
    UIPanel::draw(window);

    std::stringstream ss;
    ss << "Score: " << WorldState::SCORE << " Lives: "
        << WorldState::LIVES << " Wave: "
        << WorldState::WAVE_NUMBER;
    m_Text.setString(ss.str());

    window.draw(m_Text);
}
```

The constructor, like the `SelectUIPanel` class, calls the base class constructor to configure the position, size, and color of the panel. Also, in the constructor, a `Text` instance is prepared for drawing to the screen.

In the draw function, a `stringstream` instance is used to concatenate a `String` of text that displays the player's score, lives remaining, and number of waves cleared. The `RenderWindow` instance then passes the `Text` instance to its draw function.

Coding the `GameOverInputHandler` class

Remember that the game screen will have two panels and two input handling classes. When the player loses their last life, the game over panel will be shown. This is what we will code now.

Create a new header file in the Header Files/Screens/Game filter called `GameOverInputHandler.h` and add the following code:

```
#pragma once
#include "InputHandler.h"

class GameOverInputHandler :
    public InputHandler
{
public:
    void handleKeyPressed(Event& event,
        RenderWindow& window) override;

    void handleLeftClick(std::string&
        buttonInteractedWith, RenderWindow& window) override;
};
```

There is nothing different in the preceding code compared to the header files of the previous two `InputHandler` derived classes.

Create a new source file in the Source Files/Screens/Game filter called `GameOverInputHandler.cpp` and add the following code:

```
#include "GameOverInputHandler.h"
#include "SoundEngine.h"
#include "WorldState.h"
#include <iostream>

void GameOverInputHandler::handleKeyPressed(Event& event,
    RenderWindow& window)
{
    if (event.key.code == Keyboard::Escape)
    {
        SoundEngine::playClick();
        getPointerToScreenManagerRemoteControl()->
```

```
        SwitchScreens("Select");
    }
}

void GameOverInputHandler::handleLeftClick(
    std::string& buttonInteractedWith, RenderWindow& window)
{
    if (buttonInteractedWith == "Play") {
        SoundEngine::playClick();
        WorldState::WAVE_NUMBER = 0;
        getPointerToScreenManagerRemoteControl()->
            loadLevelInPlayMode("level1");
    }

    else if (buttonInteractedWith == "Home") {
        SoundEngine::playClick();
        getPointerToScreenManagerRemoteControl()->
            SwitchScreens("Select");
    }
}
```

The preceding code handles two types of event. First, if the *Escape* keyboard key is pressed, the game switches to the select screen.

In the `handleLeftClick` function, there are two different buttons that are handled. If the **Play** button is clicked, a new game is started by calling `loadLevelInPlayMode`, while, if the **Home** button is clicked, then the select screen will be shown.

Coding the GameOverUIPanel class

Create a new header file in the Header Files/Screens/Game filter called `GameOverUIPanel.h` and add the following code:

```
#pragma once
#include "UIPanel.h"
class GameOverUIPanel :
    public UIPanel
{
private:
    void initialiseButtons();

public:
    GameOverUIPanel(Vector2i res);
    void virtual draw(RenderWindow& window);
};
```

There's nothing new in the preceding header file, so let's look at the function implementations

Create a new source file in the Source Files/Screens/Game filter called `GameOverUIPanel.cpp` and add the following code:

```
#include "GameOverUIPanel.h"
#include "GameScreen.h"

bool GameScreen::m_GameOver;

GameOverUIPanel::GameOverUIPanel(Vector2i res) :
    UIPanel(res,
        (res.x / 10) * 3,
        res.y / 2, // 50% of the resolution from the top
        (res.x / 10) * 3, // as wide as 1/3 of the resolution
        res.y / 6, // and as tall as 1/6 of the resolution
        50, 255, 255, 255) // a, r, g, b
{
    m_ButtonWidth = res.x / 20;
    m_ButtonHeight = res.y / 20;
    m_ButtonPadding = res.x / 100;

    m_Text.setFill(sf::Color(0, 255, 0, 255)); // Green
    m_Text.setString("GAME OVER!");

    m_Font.loadFromFile("fonts/Roboto-Bold.ttf");
    m_Text.setFont(m_Font);

    m_Text.setPosition(Vector2f(m_ButtonPadding,
        (m_ButtonPadding * 2) + m_ButtonHeight));

    m_Text.setCharacterSize(60);

    initialiseButtons();
}

void GameOverUIPanel::initialiseButtons()
{
    addButton(m_ButtonPadding,
        m_ButtonPadding,
        m_ButtonWidth,
        m_ButtonHeight,
        0, 255, 0,
```



```
        "Play");

    addButton(m_ButtonWidth + (m_ButtonPadding * 2),
        m_ButtonPadding,
        m_ButtonWidth,
        m_ButtonHeight,
        255, 0, 0,
        "Home");
}

void GameOverUIPanel::draw(RenderWindow& window)
{
    if (GameScreen::m_GameOver)
    {
        show();
        UIPanel::draw(window);
        window.draw(m_Text);
    }
    else
    {
        hide();
    }
}
```

The preceding code configures a panel in the middle of the screen with the text **Game Over!** and two buttons that will allow the player to restart the game or quit, and go back to the starting screen (home/select).

Running the game

If you run the game, you will see the select screen, as shown in the following screenshot:



Press **Play** to transition to the game screen:



Press **Escape** to quit, and go back to the select screen.

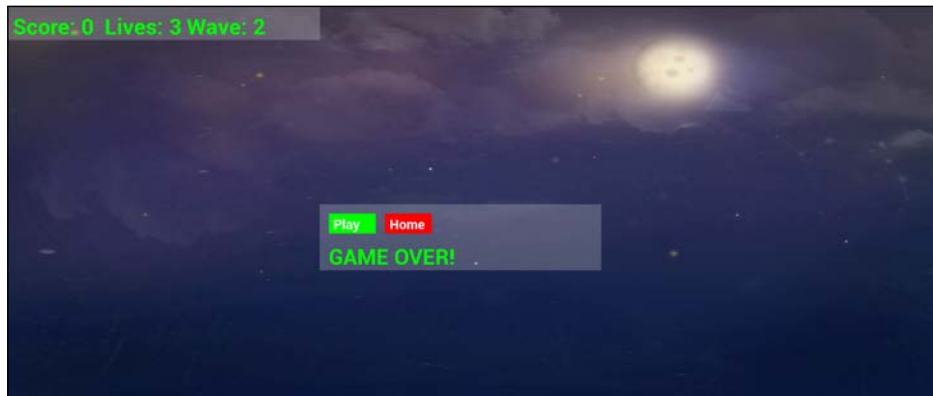
Quit the game and find the following line of code in the `GameScreen` class:

```
if (WorldState::LIVES <= 0)
```

Change it to the following:

```
if (true)
```

Now, run the game again and select the **Play** button. The game over panel will be displayed and can be interacted with:



Now, change back `if (true)` in the `GameScreen` class back to `if (WorldState::LIVES <= 0)`.

Let's take a break; that was a long chapter.

Summary

You have achieved a lot in this chapter. You have built a solid foundation for the Space Invaders ++ game and you have also coded a reusable system that can be used for almost any game that is divided up into different "screens".

We now have an input handling system in place that can detect keyboard presses and mouse clicks and route the responsibility to handle them to a specific panel that is part of a specific screen. Furthermore, the abstraction of the concept of a screen allows us to set up as many different game loops as we like. The `GameScreen` class will be the main class to handle the logic of this game but, once you see how over the next few chapters, you could easily code another screen to play a completely different game. Of course, the most likely thing you will do is get started with your own ideas.

In the next chapter, we will code the game objects and components which are the basis of our entity-component pattern implementation.

20

Game Objects and Components

In this chapter, we will be doing all the coding related to the Entity-Component pattern we discussed at the beginning of the previous chapter. This means we will code the base `Component` class, which all the other components will be derived from. We will also put our new knowledge of smart pointers to good use so that we don't have to concern ourselves with keeping track of the memory we allocate for these components. We will also code the `GameObject` class in this chapter.

We will cover the following topics in this chapter:

- Preparing to code the components
- Coding the `Component` base class
- Coding the collider components
- Coding the graphics components
- Coding the update components
- Coding the `GameObject` class

Let's discuss the components a bit more before we start coding. Please note that, in this chapter, I will try and reinforce how the Entity-Component system fits together and how all the components compose a game object. I will not be explaining each and every line or even block of logic or SFML-related code that we have seen many times already. It is up to you to study these details.

Preparing to code the components

As you work through this chapter, there will be lots of errors, and some of them won't seem logical. For example, you will get errors saying that a class doesn't exist when it is one of the classes you have already coded. The reason for this is that, when a class has an error in it, other classes can't reliably use it without getting errors as well. It is because of the interconnected nature of all the classes that we will not get rid of all the errors and have executable code again until near the end of the next chapter. It would have been possible to add code in smaller chunks to the various classes and the project would have been error-free more frequently. Doing things that gradually, however, would have meant constantly dipping in and out of classes. When you are building your own projects, this is sometimes a good way to do it, but I thought the most instructive thing to do for this project would be to help you get it built as quickly as possible.

Coding the Component base class

Create a new header file in the Header Files/GameObjects filter called `Component.h` and add the following code:

```
#pragma once
#include "GameObjectSharer.h"
#include <string>

using namespace std;

class GameObject;

class Component {
public:
    virtual string getType() = 0;
    virtual string getSpecificType() = 0;
    virtual void disableComponent() = 0;
    virtual void enableComponent() = 0;
    virtual bool enabled() = 0;
    virtual void start(GameObjectSharer* gos, GameObject* self) = 0;
};
```

This is the base class of every component in every game object. The pure virtual functions mean that a component can never be instantiated and must always be inherited from first. Functions allow the type and specific type of a component to be accessed. Component types include collider, graphics, transform, and update, but more types could be added in line with the requirements of the game. Specific types include standard graphics, invader update, player update, and more besides.

There are two functions that allow the component to be enabled and disabled. This is useful because a component can then be tested for whether it is currently enabled before it is used. For example, you could call the `enabled` function to test whether a component's update component was enabled before calling its update function or that a graphics component is enabled before calling its draw function.

The `start` function is probably the most interesting function because it has a new class type as one of its parameters. The `GameObjectSharer` class will give access to all the game objects after they have been instantiated with all their components. This will give every component in every game object the opportunity to query details and even obtain a pointer to a specific piece of data in another game object. As an example, all an invader's update components will need to know the location of the player's transform component so that it knows when to fire a bullet. Absolutely any part of any object can be accessed in the `start` function. The point is that each specific component will decide what they need and there is no requirement during the critical game loop to start querying for the details of another game object.

The `GameObject` that the component is contained in is also passed to the `start` function so that any component can find out more about itself as well. For example, a graphics component needs to know about the transform component so that it knows where to draw itself. As a second example, the update components of the invaders and the player's ship will need a pointer to their own collider component so that they can update its location whenever they move.

We will see more use cases for the `start` function as we progress.

Create a new source file in the `Source Files/GameObjects` filter called `Component.cpp` and add the following code:

```

/*****
*****THIS IS AN INTERFACE*****
*****/

```

As the `Component` class can never be instantiated, I have put the preceding comments in `Component.cpp` as a reminder.

Coding the collider components

The Space Invaders ++ game will only have one simple type of collider. It will be a rectangular box around the object, just like those we had in the *Zombie Apocalypse* and *Pong* games. However, it is easily conceivable that you might need other types of collider; perhaps a circle-shaped collider or a non-encompassing collider such as those we used for the head, feet, and sides of Thomas and Bob back in the *Thomas Was Late* game.

For this reason, there will be a base `ColliderComponent` class (that inherits from `Component`) which will handle the basic functionality of all the colliders, as well as `RectColliderComponent`, which will add the specific functionality of an all-encompassing rectangle-shaped collider. New collider types can then be added as required for the game being developed.

What follows is the base class to the specific collider, `ColliderComponent`.

Coding the ColliderComponent class

Create a new header file in the Header Files/GameObjects filter called `ColliderComponent.h` and add the following code:

```
#pragma once
#include "Component.h"
#include <iostream>

class ColliderComponent : public Component
{
private:
    string m_Type = "collider";
    bool m_Enabled = false;

public:

    /*****
    *****/
    From Component interface
    *****/
    *****/

    string Component::getType() {
        return m_Type;
    }

    void Component::disableComponent() {
        m_Enabled = false;
    }

    void Component::enableComponent() {
        m_Enabled = true;
    }
}
```

```

        bool Component::enabled() {
            return m_Enabled;
        }

        void Component::start(GameObjectSharer* gos, GameObject* self)
        {

        }

    };

```

The `ColliderComponent` class inherits from the `Component` class. In the preceding code, you can see that the `m_Type` member variable is initialized to "collider" and that `m_Enabled` is initialized to `false`.

In the public section, the code overrides the pure virtual functions of the `Component` class. Study them to become familiar with them because they work in a very similar way in all the component classes. The `getType` function returns `m_Type`. The `disableComponent` function sets `m_Enabled` to `false`. The `enableComponent` function sets `m_Enabled` to `true`. The `enabled` function returns the value of `m_Enabled`. The `start` function has no code but will be overridden by many of the more specific component-based classes.

Create a new source file in the `Source Files/GameObjects` filter called `ColliderComponent.cpp` and add the following code:

```

/*
All Functionality in ColliderComponent.h
*/

```

I added the preceding comments to `ColliderComponent.cpp` to remind myself that all the functionality is in the header file.

Coding the RectColliderComponent class

Create a new header file in the `Header Files/GameObjects` filter called `RectColliderComponent.h` and add the following code:

```

#pragma once
#include "ColliderComponent.h"
#include <SFML/Graphics.hpp>

using namespace sf;

class RectColliderComponent : public ColliderComponent
{

```



```
private:
    string m_SpecificType = "rect";
    FloatRect m_Collider;
    string m_Tag = "";

public:
    RectColliderComponent(string name);
    string getColliderTag();
    void setOrMoveCollider(
        float x, float y, float width, float height);

    FloatRect& getColliderRectF();

    /*****
    *****/
    From Component interface base class
    *****/

    string getSpecificType() {
        return m_SpecificType;
    }

    void Component::start(
        GameObjectSharer* gos, GameObject* self) {}
};
```

The RectColliderComponent class inherits from the ColliderComponent class. It has a m_SpecificType variable initialized to "rect". It is now possible to query any RectColliderComponent instance in a vector of generic Component instances and determine that it has a type of "collider" and a specific type of "rect". All component-based classes will have this functionality because of the pure virtual functions of the Component class.

There is also a FloatRect instance called m_Collider that will store the coordinates of this collider.

In the public section, we can view the constructor. Notice that it receives a string. The value that's passed in will be text that identifies the type of game object this RectColliderComponent is attached to, such as an invader, a bullet, or the player's ship. It will then be possible to determine what type of objects have collided with each other.

There are three more functions before the overridden functions; make a note of their names and parameters and then we will discuss them in a moment when we code their definitions.

Note that the `getSpecificType` function definition returns `m_SpecificType`.

Create a new source file in the `Source Files/GameObjects` filter called `RectColliderComponent.cpp` and add the following code:

```
#include "RectColliderComponent.h"

RectColliderComponent::RectColliderComponent(string name) {
    m_Tag = "" + name;
}

string RectColliderComponent::getColliderTag() {
    return m_Tag;
}

void RectColliderComponent::setOrMoveCollider(
    float x, float y, float width, float height) {

    m_Collider.left = x;
    m_Collider.top = y;
    m_Collider.width = width;
    m_Collider.height = height;
}

FloatRect& RectColliderComponent::getColliderRectF() {
    return m_Collider;
}
```

In the constructor, the passed-in `string` value is assigned to the `m_Tag` variable and the `getColliderTag` function makes that value available via the instance of the class.

The `setOrMoveCollider` function positions `m_Collider` at the coordinates passed in as arguments.

The `getColliderRectF` function returns a reference to `m_Collider`. This is ideal for carrying out a collision test with another collider using the `intersects` function of the `FloatRect` class.

Our colliders are now complete and we can move on to the graphics.

Coding the graphics components

The Space Invaders ++ game will only have one specific type of graphics component. It is called `StandardGraphicsComponent`. As with the collider components, we will implement a base `GraphicsComponent` class to make it easy to add other graphics-related components, should we wish. For example, in the classic arcade version of Space Invaders, the invaders flapped their arms up and down with two frames of animation. Once you see how `StandardGraphicsComponent` works, you will be able to easily code another class (perhaps `AnimatedGraphicsComponent`) that draws itself with a different `Sprite` instance every half a second or so. You could also have a graphics component that has a shader (perhaps `ShaderGraphicsComponent`) for fast and cool effects. There are more possibilities besides these.

Coding the `GraphicsComponent` class

Create a new header file in the Header Files/`GameObjects` filter called `GraphicsComponent.h` and add the following code:

```
#pragma once
#include "Component.h"
#include "TransformComponent.h"
#include <string>
#include <SFML/Graphics.hpp>
#include "GameObjectSharer.h"
#include <iostream>

using namespace sf;
using namespace std;

class GraphicsComponent : public Component {
private:
    string m_Type = "graphics";
    bool m_Enabled = false;

public:
    virtual void draw(
        RenderWindow& window,
        shared_ptr<TransformComponent> t) = 0;

    virtual void initializeGraphics(
        string bitmapName,
        Vector2f objectSize) = 0;
```

```

/*****
*****
From Component interface
*****
*****/

string Component::getType() {
    return m_Type;
}

void Component::disableComponent() {
    m_Enabled = false;
}

void Component::enableComponent() {
    m_Enabled = true;
}

bool Component::enabled() {
    return m_Enabled;
}

void Component::start(
    GameObjectSharer* gos, GameObject* self) {}

};

```

Most of the preceding code implements the `Component` class's pure virtual functions. What's new to the `GraphicsComponent` class is the `draw` function, which has two parameters. The first parameter is a reference to the `RenderWindow` instance so that the component can draw itself, while the second is a shared smart pointer to the `TransformComponent` instance of the `GameObject` so that vital data such as position and scale can be accessed each frame of the game.

What's also new in the `GraphicsComponent` class is the `initializeGraphics` function, which also has two parameters. The first is a `string` value that represents the file name of the graphics file to be used, while the second is a `Vector2f` instance that will represent the size of the object in the game world.

Both preceding functions are pure virtual, which makes the `GraphicsComponent` class abstract. Any class that inherits from `GraphicsComponent` will need to implement these functions. In the next section, we will see how `StandardGraphicsComponent` does so.

Create a new source file in the `Source Files/GameObjects` filter called `GraphicsComponent.cpp` and add the following code:

```
/*
All Functionality in GraphicsComponent.h
*/
```

The preceding comment is a reminder that the code is all within the related header file.

Coding the StandardGraphicsComponent class

Create a new header file in the `Header Files/GameObjects` filter called `StandardGraphicsComponent.h` and add the following code:

```
#pragma once
#include "Component.h"
#include "GraphicsComponent.h"
#include <string>

class Component;

class StandardGraphicsComponent : public GraphicsComponent {
private:
    sf::Sprite m_Sprite;
    string m_SpecificType = "standard";

public:

    /*****
    *****/
    From Component interface base class
    *****/
    *****/

    string Component::getSpecificType() {
        return m_SpecificType;
    }

    void Component::start(
        GameObjectSharer* gos, GameObject* self) {
    }
```

```

/*****
*****
From GraphicsComponent
*****
*****/

void draw(
    RenderWindow& window,
    shared_ptr<TransformComponent> t) override;

void initializeGraphics(
    string bitmapName,
    Vector2f objectSize) override;
};

```

The `StandardGraphicsComponent` class has a `Sprite` member. It doesn't need a `Texture` instance because that will be obtained each frame from the `BitmapStore` class. This class also overrides the required functions from both the `Component` and `GraphicsComponent` classes.

Let's code the implementation of the two pure virtual functions, `draw` and `initializeGraphics`.

Create a new source file in the `Source Files/GameObjects` filter called `StandardGraphicsComponent.cpp` and add the following code:

```

#include "StandardGraphicsComponent.h"
#include "BitmapStore.h"
#include <iostream>

void StandardGraphicsComponent::initializeGraphics(
    string bitmapName,
    Vector2f objectSize)
{
    BitmapStore::addBitmap("graphics/" + bitmapName + ".png");
    m_Sprite.setTexture(BitmapStore::getBitmap(
        "graphics/" + bitmapName + ".png"));

    auto textureSize = m_Sprite.getTexture()->getSize();
    m_Sprite.setScale(float(objectSize.x) / textureSize.x,
        float(objectSize.y) / textureSize.y);

    m_Sprite.setColor(sf::Color(0, 255, 0));
}

```

```
void StandardGraphicsComponent::draw(
    RenderWindow& window,
    shared_ptr<TransformComponent> t)
{
    m_Sprite.setPosition(t->getLocation());
    window.draw(m_Sprite);
}
```

In the `initializeGraphics` function, the `addBitmap` function of the `BitmapStore` class is called and the file path of the image, along with the size of the object in the game world, is passed in.

Next, the `Texture` instance that was just added to the `BitmapStore` class is retrieved and set as the image for the `Sprite`. Following on, two functions, `getTexture` and `getSize`, are chained together to get the size of the texture.

The next line of code uses the `setScale` function to make the `Sprite` the same size as the texture, which in turn was set to the size of this object in the game world.

The `setColor` function then applies a green tint to the `Sprite`. This gives it a bit more of a retro feel.

In the `draw` function, the `Sprite` is moved into position using `setPosition` and the `getLocation` function of `TransformComponent`. We'll code the `TransformComponent` class next.

The final line of code draws the `Sprite` to `RenderWindow`.

Coding the TransformComponent class

Create a new header file in the `Header Files/GameObjects` filter called `TransformComponent.h` and add the following code:

```
#pragma once
#include "Component.h"
#include<SFML/Graphics.hpp>

using namespace sf;

class Component;

class TransformComponent : public Component {
private:
    const string m_Type = "transform";
    Vector2f m_Location;
```

```

        float m_Height;
        float m_Width;

public:
    TransformComponent(
        float width, float height, Vector2f location);

    Vector2f& getLocation();

    Vector2f getSize();

    /*****
    *****/
    From Component interface
    *****/
    *****/

    string Component::getType()
    {
        return m_Type;
    }

    string Component::getSpecificType()
    {
        // Only one type of Transform so just return m_Type
        return m_Type;
    }

    void Component::disableComponent() {}

    void Component::enableComponent() {}

    bool Component::enabled()
    {
        return false;
    }

    void Component::start(GameObjectSharer* gos, GameObject* self)
    {}

};

```

This class has a `Vector2f` to store the position of the object in the game world, a `float` to store the height, and another `float` to store the width.

In the public section, there is a constructor we will use to set up the instances of this class, as well as two functions, `getLocation` and `getSize`, we'll use to share the location and size of the object. We used these functions already when we coded the `StandardGraphicsComponent` class.

The remaining code in the `TransformComponent.h` file is the implementation of the `Component` class.

Create a new source file in the `Source Files/GameObjects` filter called `TransformComponent.cpp` and add the following code:

```
#include "TransformComponent.h"

TransformComponent::TransformComponent(
    float width, float height, Vector2f location)
{
    m_Height = height;
    m_Width = width;
    m_Location = location;
}

Vector2f& TransformComponent::getLocation()
{
    return m_Location;
}

Vector2f TransformComponent::getSize()
{
    return Vector2f(m_Width, m_Height);
}
```

Implementing the three functions of this class is straightforward. The constructor receives a size and location and initializes the appropriate member variables. The `getLocation` and `getSize` functions return this data when it is requested. Notice that the values are returned by reference, so they will be modifiable by the calling code.

Next, we will code all update-related components.

Coding update components

As you might expect by now, we will code an `UpdateComponent` class that will inherit from the `Component` class. It will have all the functionality that every `UpdateComponent` will need and then we will code classes derived from `UpdateComponent`. These will contain functionality specific to individual objects in the game. For this game, we will have `BulletUpdateComponent`, `InvaderUpdateComponent`, and `PlayerUpdateComponent`. When you work on your own project and you want an object in the game that behaves in a specific unique manner, just code a new update-based component for it and you'll be good-to-go. Update-based components define behavior.

Coding the `UpdateComponent` class

Create a new header file in the Header Files/`GameObjects` filter called `UpdateComponent.h` and add the following code:

```
#pragma once
#include "Component.h"

class UpdateComponent : public Component
{
private:
    string m_Type = "update";
    bool m_Enabled = false;

public:
    virtual void update(float fps) = 0;

    /*****
    *****/
    From Component interface
    *****/

    string Component::getType() {
        return m_Type;
    }

    void Component::disableComponent() {
        m_Enabled = false;
    }

    void Component::enableComponent() {
```

```
        m_Enabled = true;
    }

    bool Component::enabled() {
        return m_Enabled;
    }

    void Component::start(
        GameObjectSharer* gos, GameObject* self) {
    }
};
```

UpdateComponent only brings one piece of functionality: the update function. This function is pure virtual so it must be implemented by any class that aspires to be a usable instance of UpdateComponent.

Create a new source file in the Source Files/GameObjects filter called UpdateComponent.cpp and add the following code:

```
/*
All Functionality in UpdateComponent.h
*/
```

This is a helpful comment to remind us that this class has all its code in the related header file.

Coding the BulletUpdateComponent class

Create a new header file in the Header Files/GameObjects filter called BulletUpdateComponent.h and add the following code:

```
#pragma once
#include "UpdateComponent.h"
#include "TransformComponent.h"
#include "GameObjectSharer.h"
#include "RectColliderComponent.h"
#include "GameObject.h"

class BulletUpdateComponent : public UpdateComponent
{
private:
    string m_SpecificType = "bullet";
```

```

shared_ptr<TransformComponent> m_TC;
shared_ptr<RectColliderComponent> m_RCC;

float m_Speed = 75.0f;

int m_AlienBulletSpeedModifier;
int m_ModifierRandomComponent = 5;
int m_MinimumAdditionalModifier = 5;

bool m_MovingUp = true;

public:
    bool m_BelongsToPlayer = false;
    bool m_IsSpawned = false;

    void spawnForPlayer(Vector2f spawnPosition);
    void spawnForInvader(Vector2f spawnPosition);
    void deSpawn();
    bool isMovingUp();

    /*****
    *****/
    From Component interface base class
    *****/

    string Component::getSpecificType() {
        return m_SpecificType;
    }

    void Component::start(
        GameObjectSharer* gos, GameObject* self) {
        // Where is this specific invader
        m_TC = static_pointer_cast<TransformComponent>(
            self->getComponentByTypeAndSpecificType(
                "transform", "transform"));

        m_RCC = static_pointer_cast<RectColliderComponent>(
            self->getComponentByTypeAndSpecificType(
                "collider", "rect"));
    }

```

```

    /*****
    *****/
    From UpdateComponent
    *****/
    *****/

    void update(float fps) override;
};
```

If you want to understand the behavior/logic of a bullet, you will need to spend some time learning the names and types of the member variables as I won't be explaining precisely how a bullet behaves; we have covered these topics many times. I will, however, point out that there are variables to cover basics such as movement, variables to help randomize the speed of each bullet within a certain range, and Booleans that identify whether the bullet belongs to the player or an invader.

The key thing which you don't yet know but will have to learn here is that each `BulletUpdateComponent` instance will hold a shared pointer to the owning game object's `TransformComponent` instance and a shared pointer to the owning game object's `RectColliderComponent` instance.

Now, look closely at the overridden `start` function. In the `start` function, the aforementioned shared pointers are initialized. The code achieves this by using the `getComponentByTypeAndSpecificType` function of the owning game object (`self`), which is a pointer to the owning game object. We will code the `GameObject` class, including this function, in a later section.

Create a new source file in the `Source Files/GameObjects` filter called `BulletUpdate.cpp` and add the following code:

```
#include "BulletUpdateComponent.h"
#include "WorldState.h"

void BulletUpdateComponent::spawnForPlayer(
    Vector2f spawnPosition)
{
    m_MovingUp = true;
    m_BelongsToPlayer = true;
    m_IsSpawned = true;

    m_TC->getLocation().x = spawnPosition.x;
    // Tweak the y location based on the height of the bullet
    // The x location is already tweaked to the center of the player
    m_TC->getLocation().y = spawnPosition.y - m_TC->getSize().y;
    // Update the collider
```

```

        m_RCC->setOrMoveCollider(m_TC->getLocation().x,
                                m_TC->getLocation().y,
                                m_TC->getSize().x, m_TC->getSize().y);
    }

void BulletUpdateComponent::spawnForInvader(
    Vector2f spawnPosition)
{
    m_MovingUp = false;
    m_BelongsToPlayer = false;
    m_IsSpawned = true;

    srand((int)time(0));
    m_AlienBulletSpeedModifier = (
        ((rand() % m_ModifierRandomComponent)))
        + m_MinimumAdditionalModifier;

    m_TC->getLocation().x = spawnPosition.x;
    // Tweak the y location based on the height of the bullet
    // The x location already tweaked to the center of the invader
    m_TC->getLocation().y = spawnPosition.y;
    // Update the collider
    m_RCC->setOrMoveCollider(
        m_TC->getLocation().x, m_TC->
        getLocation().y, m_TC->getSize().x, m_TC->getSize().y);
}

void BulletUpdateComponent::deSpawn()
{
    m_IsSpawned = false;
}

bool BulletUpdateComponent::isMovingUp()
{
    return m_MovingUp;
}

void BulletUpdateComponent::update(float fps)
{
    if (m_IsSpawned)
    {
        if (m_MovingUp)
        {
            m_TC->getLocation().y -= m_Speed * fps;

```

```
    }
    else
    {
        m_TC->getLocation().y += m_Speed /
            m_AlienBulletSpeedModifier * fps;
    }

    if (m_TC->getLocation().y > WorldState::WORLD_HEIGHT
        || m_TC->getLocation().y < -2)
    {
        deSpawn();
    }

    // Update the collider
    m_RCC->setOrMoveCollider(m_TC->getLocation().x,
        m_TC->getLocation().y,
        m_TC->getSize().x, m_TC->getSize().y);
}
}
```

The first two functions are unique to the `BulletUpdateComponent` class; they are `spawnForPlayer` and `spawnForInvader`. Both of these functions prepare the member variables, transform component and collider component for action. Each one does so in a slightly different way. For example, for a player-owned bullet, it is prepared to move up the screen from the top of the player's ship, while a bullet is prepared for an invader to move down the screen from the underside of an invader. The key thing to notice is that all this is achievable via the shared pointers to the transform component and the collider component. Also, note that the `m_IsSpawned` Boolean is set to true, making this update component's update function ready to call each frame of the game.

In the update function, the bullet is moved up or down the screen at the appropriate speed. It is tested to see if it has disappeared off the top or bottom of the screen, and the collider is updated to wrap around the current location so that we can test for collisions.

This is the same logic we have seen throughout this book; what's new is the shared pointers we are using to communicate with the other components that make up this game object.

The bullets just need to be spawned and tested for collisions; we will see how to do that in the next two chapters. Now, we will code the behavior of the invaders.

Coding the InvaderUpdateComponent class

Create a new header file in the Header Files/GameObjects filter called `InvaderUpdateComponent.h` and add the following code:

```
#pragma once
#include "UpdateComponent.h"
#include "TransformComponent.h"
#include "GameObjectSharer.h"
#include "RectColliderComponent.h"
#include "GameObject.h"

class BulletSpawner;

class InvaderUpdateComponent : public UpdateComponent
{
private:
    string m_SpecificType = "invader";

    shared_ptr<TransformComponent> m_TC;
    shared_ptr< RectColliderComponent> m_RCC;
    shared_ptr< TransformComponent> m_PlayerTC;
    shared_ptr< RectColliderComponent> m_PlayerRCC;

    BulletSpawner* m_BulletSpawner;

    float m_Speed = 10.0f;
    bool m_MovingRight = true;
    float m_TimeSinceLastShot;
    float m_TimeBetweenShots = 5.0f;
    float m_AccuracyModifier;
    float m_SpeedModifier = 0.05;
    int m_RandSeed;

public:
    void dropDownAndReverse();
    bool isMovingRight();
    void initializeBulletSpawner(BulletSpawner*
        bulletSpawner, int randSeed);
```



```

/*****
*****
From Component interface base class
*****
*****/

string Component::getSpecificType() {
    return m_SpecificType;
}

void Component::start(GameObjectSharer* gos,
    GameObject* self) {

    // Where is the player?
    m_PlayerTC = static_pointer_cast<TransformComponent>(
        gos->findFirstObjectWithTag("Player")
        .getComponentByTypeAndSpecificType(
            "transform", "transform"));

    m_PlayerRCC = static_pointer_cast<RectColliderComponent>(
        gos->findFirstObjectWithTag("Player")
        .getComponentByTypeAndSpecificType(
            "collider", "rect"));

    // Where is this specific invader
    m_TC = static_pointer_cast<TransformComponent>(
        self->getComponentByTypeAndSpecificType(
            "transform", "transform"));

    m_RCC = static_pointer_cast<RectColliderComponent>(
        self->getComponentByTypeAndSpecificType(
            "collider", "rect"));
}

/*****
*****
From UpdateComponent
*****
*****/

void update(float fps) override;
};
```

In the class declaration, we can see all the features that we need in order to code the behavior of an invader. There is a pointer to the transform component so that the invader can move, as well as a pointer to the collider component so that it can update its location and be collided with:

```
shared_ptr<TransformComponent> m_TC;
shared_ptr < RectColliderComponent> m_RCC;
```

There are pointers to the player's transform and collider so that an invader can query the position of the player and make decisions about when to shoot bullets:

```
shared_ptr < TransformComponent> m_PlayerTC;
shared_ptr < RectColliderComponent> m_PlayerRCC;
```

Next, there is a `BulletSpawner` instance, which we will code in the next chapter. The `BulletSpawner` class will allow an invader or the player to spawn a bullet.

What follows is a whole bunch of variables that we will use to control the speed, direction, rate of fire, the precision with which the invader aims, and the speed of bullets that are fired. Familiarize yourself with them as they will be used in fairly in-depth logic in the function definitions:

```
float m_Speed = 10.0f;
bool m_MovingRight = true;
float m_TimeSinceLastShot;
float m_TimeBetweenShots = 5.0f;
float m_AccuracyModifier;
float m_SpeedModifier = 0.05;
int m_RandSeed;
```

Next, we can see three new public functions that different parts of the system can call to make the invaders move down a little and head in the other direction, test the direction of travel, and pass in a pointer to the aforementioned `BulletSpawner` class, respectively:

```
void dropDownAndReverse();
bool isMovingRight();
void initializeBulletSpawner(BulletSpawner*
    bulletSpawner, int randSeed);
```

Be sure to study the `start` function where the smart pointers to the invader and the player are initialized. Now, we will code the function definitions.

Create a new source file in the Source Files/GameObjects filter called `InvaderUpdate.cpp` and add the following code:

```
#include "InvaderUpdateComponent.h"
#include "BulletSpawner.h"
#include "WorldState.h"
#include "SoundEngine.h"

void InvaderUpdateComponent::update(float fps)
{
    if (m_MovingRight)
    {
        m_TC->getLocation().x += m_Speed * fps;
    }
    else
    {
        m_TC->getLocation().x -= m_Speed * fps;
    }

    // Update the collider
    m_RCC->setOrMoveCollider(m_TC->getLocation().x,
        m_TC->getLocation().y, m_TC->getSize().x, m_TC->
        >getSize().y);

    m_TimeSinceLastShot += fps;

    // Is the middle of the invader above the
    // player +- 1 world units
    if ((m_TC->getLocation().x + (m_TC->getSize().x / 2)) >
        (m_PlayerTC->getLocation().x - m_AccuracyModifier) &&
        (m_TC->getLocation().x + (m_TC->getSize().x / 2)) <
        (m_PlayerTC->getLocation().x +
        (m_PlayerTC->getSize().x + m_AccuracyModifier)))
    {
        // Has the invader waited long enough since the last shot
        if (m_TimeSinceLastShot > m_TimeBetweenShots)
        {
            SoundEngine::playShoot();
            Vector2f spawnLocation;
            spawnLocation.x = m_TC->getLocation().x +
                m_TC->getSize().x / 2;
```

```

        spawnLocation.y = m_TC->getLocation().y +
            m_TC->getSize().y;

        m_BulletSpawner->spawnBullet(spawnLocation, false);
        srand(m_RandSeed);
        int mTimeBetweenShots = (((rand() % 10))+1) /
            WorldState::WAVE_NUMBER;

        m_TimeSinceLastShot = 0;
    }
}

void InvaderUpdateComponent::dropDownAndReverse()
{
    m_MovingRight = !m_MovingRight;
    m_TC->getLocation().y += m_TC->getSize().y;
    m_Speed += (WorldState::WAVE_NUMBER) +
        (WorldState::NUM_INVADERS_AT_START
        - WorldState::NUM_INVADERS)
        * m_SpeedModifier;
}

bool InvaderUpdateComponent::isMovingRight()
{
    return m_MovingRight;
}

void InvaderUpdateComponent::initializeBulletSpawner(
    BulletSpawner* bulletSpawner, int randSeed)
{
    m_BulletSpawner = bulletSpawner;
    m_RandSeed = randSeed;
    srand(m_RandSeed);
    m_TimeBetweenShots = (rand() % 15 + m_RandSeed);

    m_AccuracyModifier = (rand() % 2);
    m_AccuracyModifier += 0 + static_cast <float> (
        rand()) / (static_cast <float> (RAND_MAX / (10)));
}

```

That was a lot of code. Actually, there's no C++ code in there that we haven't seen before. It is all just logic to control the behavior of an invader. Let's get an overview of what it all does, with parts of the code reprinted for convenience.

Explaining the update function

The first `if` and `else` blocks move the invader right or left each frame, as appropriate:

```
void InvaderUpdateComponent::update(float fps)
{
    if (m_MovingRight)
    {
        m_TC->getLocation().x += m_Speed * fps;
    }
    else
    {
        m_TC->getLocation().x -= m_Speed * fps;
    }
}
```

Next, the collider is updated to the new position:

```
// Update the collider
m_RCC->setOrMoveCollider(m_TC->getLocation().x,
    m_TC->getLocation().y, m_TC->getSize().x, m_TC
->getSize().y);
```

This code tracks how long it's been since this invader last fired a shot and then tests to see if the player is one world unit to the left or right of the invader (+ or - for the random accuracy modifier, so that each invader is a little bit different):

```
m_TimeSinceLastShot += fps;

// Is the middle of the invader above the
// player +- 1 world units
if ((m_TC->getLocation().x + (m_TC->getSize().x / 2)) >
    (m_PlayerTC->getLocation().x - m_AccuracyModifier) &&
    (m_TC->getLocation().x + (m_TC->getSize().x / 2)) <
    (m_PlayerTC->getLocation().x +
    (m_PlayerTC->getSize().x + m_AccuracyModifier)))
{
```

Inside the preceding `if` test, another test makes sure that the invader has waited long enough since the last shot it took. If it has, then a shot is taken. A sound is played, a spawn location for the bullet is calculated, the `spawnBullet` function of the `BulletSpawner` instance is called, and a new random time to wait before another shot can be taken is calculated:

```
// Has the invader waited long enough since the last shot
if (m_TimeSinceLastShot > m_TimeBetweenShots)
{
```

```

        SoundEngine::playShoot();
        Vector2f spawnLocation;
        spawnLocation.x = m_TC->getLocation().x +
            m_TC->getSize().x / 2;

        spawnLocation.y = m_TC->getLocation().y +
            m_TC->getSize().y;

        m_BulletSpawner->spawnBullet(spawnLocation, false);
        srand(m_RandSeed);
        int mTimeBetweenShots = (((rand() % 10))+1) /
            WorldState::WAVE_NUMBER;

        m_TimeSinceLastShot = 0;
    }
}

```

The details of the `BulletSpawner` class will be revealed in the next chapter, but as a glimpse into the future, it will be an abstract class with one function called `spawnBullet` and will be inherited from by the `GameScreen` class.

Explaining the `dropDownAndReverse` function

In the `dropDownAndReverse` function, the direction is reversed and the vertical location is increased by the height of an invader. In addition, the speed of the invader is increased relative to how many waves the player has cleared and how many invaders remain to be destroyed. The more waves that are cleared and the fewer invaders remaining, the faster the invaders will move:

```

void InvaderUpdateComponent::dropDownAndReverse()
{
    m_MovingRight = !m_MovingRight;
    m_TC->getLocation().y += m_TC->getSize().y;
    m_Speed += (WorldState::WAVE_NUMBER) +
        (WorldState::NUM_INVADERS_AT_START
        - WorldState::NUM_INVADERS)
        * m_SpeedModifier;
}

```

The next function is simple but included for the sake of completeness.

Explaining the isMovingRight function

This code simply provides access to the current direction of travel:

```
bool InvaderUpdateComponent::isMovingRight()
{
    return m_MovingRight;
}
```

It will be used to test whether to look out for collisions with the left of the screen (when moving left) or the right of the screen (when moving right) and will allow the collision to trigger a call to the `dropDownAndReverse` function.

Explaining the initializeBulletSpawner function

I have already mentioned that the `BulletSpawner` class is abstract and will be implemented by the `GameScreen` class. When the `GameScreen` class' `initialize` function is called, this `initializeBulletSpawner` function will be called on each of the invaders. As you can see, the first parameter is a pointer to a `BulletSpawner` instance. This gives every `InvaderUpdateComponent` the ability to call the `spawnBullet` function:

```
void InvaderUpdateComponent::initializeBulletSpawner(
    BulletSpawner* bulletSpawner, int randSeed)
{
    m_BulletSpawner = bulletSpawner;
    m_RandSeed = randSeed;
    srand(m_RandSeed);
    m_TimeBetweenShots = (rand() % 15 + m_RandSeed);

    m_AccuracyModifier = (rand() % 2);
    m_AccuracyModifier += 0 + static_cast <float> (
        rand() / (static_cast <float> (RAND_MAX / (10))));
}
```

The rest of the code in the `initializeBulletSpawner` function sets up the random values that make each invader behave slightly differently from the others.

Coding the PlayerUpdateComponent class

Create a new header file in the Header Files/GameObjects filter called `PlayerUpdateComponent.h` and add the following code:

```
#pragma once
#include "UpdateComponent.h"
#include "TransformComponent.h"
```

```

#include "GameObjectSharer.h"
#include "RectColliderComponent.h"
#include "GameObject.h"

class PlayerUpdateComponent : public UpdateComponent
{
private:
    string m_SpecificType = "player";

    shared_ptr<TransformComponent> m_TC;
    shared_ptr<RectColliderComponent> m_RCC;

    float m_Speed = 50.0f;
    float m_XExtent = 0;
    float m_YExtent = 0;

    bool m_IsHoldingLeft = false;
    bool m_IsHoldingRight = false;
    bool m_IsHoldingUp = false;
    bool m_IsHoldingDown = false;

public:
    void updateShipTravelWithController(float x, float y);
    void moveLeft();
    void moveRight();
    void moveUp();
    void moveDown();
    void stopLeft();
    void stopRight();
    void stopUp();
    void stopDown();

    /*****
    *****/
    From Component interface base class
    *****/
    string Component::getSpecificType() {
        return m_SpecificType;
    }

    void Component::start(GameObjectSharer* gos, GameObject* self) {
        m_TC = static_pointer_cast<TransformComponent>(self->

```



```
        GetComponentByTypeAndSpecificType(
            "transform", "transform"));

    m_RCC = static_pointer_cast<RectColliderComponent>(self->
        GetComponentByTypeAndSpecificType(
            "collider", "rect"));
}

/*****
*****
From UpdateComponent
*****
*****/

void update(float fps) override;
};
```

In the `PlayerUpdateComponent` class, we have all the Boolean variables needed to keep track of whether the player is holding down a keyboard key, as well as functions that can toggle these Boolean values. We haven't seen anything like the `m_XExtent` and `m_YExtent` float type variables before and we will explain them when we look at their usage in the function definitions.

Note, just like the `BulletUpdateComponent` and the `InvaderUpdateComponent` classes, that we have shared pointers to this game object's transform and collider components. These shared pointers, as we are coming to expect, are initialized in the start function.

Create a new source file in the `Source Files/GameObjects` filter called `PlayerUpdate.cpp` and add the following code:

```
#include "PlayerUpdateComponent.h"
#include "WorldState.h"

void PlayerUpdateComponent::update(float fps)
{
    if (sf::Joystick::isConnected(0))
    {
        m_TC->getLocation().x += ((m_Speed / 100)
            * m_XExtent) * fps;

        m_TC->getLocation().y += ((m_Speed / 100)
            * m_YExtent) * fps;
    }
}
```

```
// Left and right
if (m_IsHoldingLeft)
{
    m_TC->getLocation().x -= m_Speed * fps;
}
else if (m_IsHoldingRight)
{
    m_TC->getLocation().x += m_Speed * fps;
}

// Up and down
if (m_IsHoldingUp)
{
    m_TC->getLocation().y -= m_Speed * fps;
}
else if (m_IsHoldingDown)
{
    m_TC->getLocation().y += m_Speed * fps;
}

// Update the collider
m_RCC->setOrMoveCollider(m_TC->getLocation().x,
    m_TC->getLocation().y, m_TC->getSize().x,
    m_TC->getSize().y);

// Make sure the ship doesn't go outside the allowed area
if (m_TC->getLocation().x >
    WorldState::WORLD_WIDTH - m_TC->getSize().x)
{
    m_TC->getLocation().x =
        WorldState::WORLD_WIDTH - m_TC->getSize().x;
}
else if (m_TC->getLocation().x < 0)
{
    m_TC->getLocation().x = 0;
}
if (m_TC->getLocation().y >
    WorldState::WORLD_HEIGHT - m_TC->getSize().y)
{
    m_TC->getLocation().y =
        WorldState::WORLD_HEIGHT - m_TC->getSize().y;
}
else if (m_TC->getLocation().y <
    WorldState::WORLD_HEIGHT / 2)
```

```
        {
            m_TC->getLocation().y =
                WorldState::WORLD_HEIGHT / 2;
        }
    }

void PlayerUpdateComponent::
    updateShipTravelWithController(float x, float y)
{
    m_XExtent = x;
    m_YExtent = y;
}

void PlayerUpdateComponent::moveLeft()
{
    m_IsHoldingLeft = true;
    stopRight();
}

void PlayerUpdateComponent::moveRight()
{
    m_IsHoldingRight = true;
    stopLeft();
}

void PlayerUpdateComponent::moveUp()
{
    m_IsHoldingUp = true;
    stopDown();
}

void PlayerUpdateComponent::moveDown()
{
    m_IsHoldingDown = true;
    stopUp();
}

void PlayerUpdateComponent::stopLeft()
{
    m_IsHoldingLeft = false;
}

void PlayerUpdateComponent::stopRight()
{

```

```

        m_IsHoldingRight = false;
    }

    void PlayerUpdateComponent::stopUp()
    {
        m_IsHoldingUp = false;
    }

    void PlayerUpdateComponent::stopDown()
    {
        m_IsHoldingDown = false;
    }

```

In the first `if` block of the update function, the condition is `sf::Joystick::isConnected(0)`. This condition returns true when the player has a gamepad plugged in to a USB port. Inside the `if` block, the location of both the horizontal and vertical positions of the transform component are altered:

```
...((m_Speed / 100) * m_YExtent) * fps;
```

The preceding code divides the target speed by 100 before multiplying it by `m_YExtent`. The `m_XExtent` and `m_YExtent` variables will be updated each frame to hold values that represent the extent to which the player has moved their gamepad thumbstick in a horizontal and vertical direction. The range of values is from -100 to 100, and so the preceding code has the effect of moving the transform component at full speed in any direction when the thumbstick is positioned at any of its full extents or a fraction of that speed when it is partially positioned between the center (not moving at all) and its full extent. This means that the player will have finer control over the speed of the ship should they opt to use a gamepad instead of the keyboard.

We will see more details about the operation of the gamepad in *Chapter 22, Using Game Objects and Building a Game*.

The rest of the `update` function responds to the Boolean variables, which represent the keyboard keys that the player is holding down or has released.

After the gamepad and keyboard handling, the collider component is moved into the new position and a series of `if` blocks ensures the player ship can't move outside of the screen or above the half-way-up point on the screen.

The next function is the `updateShipTravelWithController` function; when a controller is plugged in, it will update the extent to which the thumbstick is moved or at rest for each frame.

The remaining functions update the Boolean values that indicate whether keyboard keys are being used to move the ship. Notice that the update component does not handle firing a bullet. We could have handled it from here, and some games might have a good reason to do so. In this game, it was slightly more direct to handle shooting a bullet from the `GameInputHandler` class. The `GameInputHandler` class, as we will see in *Chapter 22, Using Game Objects and Building a Game*, will call all the functions that let the `PlayerUpdateComponent` class know what is happening with the gamepad and keyboard. We coded the basics of keyboard responses in the `GameInputHandler` class in the previous chapter.

Now, let's code the `GameObject` class, which will hold all the various component instances.

Coding the `GameObject` class

I am going to go through the code in this class in quite a lot of detail because it is key to how all the other classes work. I think you will benefit, however, from seeing the code in its entirety and studying it first. With this in mind, create a new header file in the `Header Files/GameObjects` filter called `GameObject.h` and add the following code:

```
#pragma once
#include <SFML/Graphics.hpp>
#include <vector>
#include <string>
#include "Component.h"
#include "GraphicsComponent.h"
#include "GameObjectSharer.h"
#include "UpdateComponent.h"

class GameObject {
private:
    vector<shared_ptr<Component>> m_Components;

    string m_Tag;
    bool m_Active = false;
    int m_NumberUpdateComponents = 0;
    bool m_HasUpdateComponent = false;
    int m_FirstUpdateComponentLocation = -1;
    int m_GraphicsComponentLocation = -1;
    bool m_HasGraphicsComponent = false;
    int m_TransformComponentLocation = -1;
    int m_NumberRectColliderComponents = 0;
```

```

    int m_FirstRectColliderComponentLocation = -1;
    bool m_HasCollider = false;

public:
    void update(float fps);
    void draw(RenderWindow& window);
    void addComponent(shared_ptr<Component> component);

    void setActive();
    void setInactive();
    bool isActive();
    void setTag(String tag);
    string getTag();

    void start(GameObjectSharer* gos);

    // Slow only use in init and start
    shared_ptr<Component> getComponentByTypeAndSpecificType(
        string type, string specificType);

    FloatRect& getEncompassingRectCollider();
    bool hasCollider();
    bool hasUpdateComponent();
    string getEncompassingRectColliderTag();

    shared_ptr<GraphicsComponent> getGraphicsComponent();
    shared_ptr<TransformComponent> getTransformComponent();
    shared_ptr<UpdateComponent> getFirstUpdateComponent();
};

```

In the preceding code, be sure to closely examine the variables, types, function names, and their parameters.

Create a new source file in the `Source Files/GameObjects` filter called `GameObject.cpp` and then study and add the following code:

```

#include "DevelopState.h"
#include "GameObject.h"
#include <iostream>
#include "UpdateComponent.h"
#include "RectColliderComponent.h"

void GameObject::update(float fps)
{

```

```
        if (m_Active && m_HasUpdateComponent)
        {
            for (int i = m_FirstUpdateComponentLocation; i <
                m_FirstUpdateComponentLocation +
                m_NumberUpdateComponents; i++)
            {

                shared_ptr<UpdateComponent> tempUpdate =
                    static_pointer_cast<UpdateComponent>(
                        m_Components[i]);

                if (tempUpdate->enabled())
                {
                    tempUpdate->update(fps);
                }
            }
        }
    }

void GameObject::draw(RenderWindow& window)
{
    if (m_Active && m_HasGraphicsComponent)
    {
        if (m_Components[m_GraphicsComponentLocation]->enabled())
        {
            getGraphicsComponent()->draw(window,
                getTransformComponent());
        }
    }
}

shared_ptr<GraphicsComponent> GameObject::getGraphicsComponent()
{
    return static_pointer_cast<GraphicsComponent>(
        m_Components[m_GraphicsComponentLocation]);
}

shared_ptr<TransformComponent> GameObject::getTransformComponent()
{
    return static_pointer_cast<TransformComponent>(
        m_Components[m_TransformComponentLocation]);
}

void GameObject::addComponent(shared_ptr<Component> component)
```

```

{
    m_Components.push_back(component);
    component->enableComponent();

    if (component->getType() == "update")
    {
        m_HasUpdateComponent = true;
        m_NumberUpdateComponents++;
        if (m_NumberUpdateComponents == 1)
        {
            m_FirstUpdateComponentLocation =
                m_Components.size() - 1;
        }
    }
    else if (component->getType() == "graphics")
    {
        // No iteration in the draw method required
        m_HasGraphicsComponent = true;
        m_GraphicsComponentLocation = m_Components.size() - 1;
    }
    else if (component->getType() == "transform")
    {
        // Remember where the Transform component is
        m_TransformComponentLocation = m_Components.size() - 1;
    }
    else if (component->getType() == "collider" &&
        component->getSpecificType() == "rect")
    {
        // Remember where the collider component(s) is
        m_HasCollider = true;
        m_NumberRectColliderComponents++;
        if (m_NumberRectColliderComponents == 1)
        {
            m_FirstRectColliderComponentLocation =
                m_Components.size() - 1;
        }
    }
}

void GameObject::setActive()
{
    m_Active = true;
}

```



```
    }

    void GameObject::setInactive()
    {
        m_Active = false;
    }

    bool GameObject::isActive()
    {
        return m_Active;
    }

    void GameObject::setTag(String tag)
    {
        m_Tag = "" + tag;
    }

    std::string GameObject::getTag()
    {
        return m_Tag;
    }

    void GameObject::start(GameObjectSharer* gos)
    {
        auto it = m_Components.begin();
        auto end = m_Components.end();
        for (it;
            it != end;
            ++it)
        {
            (*it)->start(gos, this);
        }
    }

    // Slow - only use in start function
    shared_ptr<Component> GameObject::
    GetComponentByTypeAndSpecificType(
        string type, string specificType) {

        auto it = m_Components.begin();
        auto end = m_Components.end();
        for (it;
            it != end;
            ++it)
```

```

        {
            if ((*it)->getType() == type)
            {
                if ((*it)->getSpecificType() == specificType)
                {
                    return (*it);
                }
            }
        }

#ifdef debuggingErrors
        cout <<
            "GameObject.cpp::getComponentByTypeAndSpecificType-"
            << "COMPONENT NOT FOUND ERROR!"
            << endl;
#endif
        return m_Components[0];
    }

FloatRect& GameObject::getEncompassingRectCollider()
{
    if (m_HasCollider)
    {
        return (static_pointer_cast<RectColliderComponent>(
            m_Components[m_FirstRectColliderComponentLocation]))->getColliderRectF();
    }
}

string GameObject::getEncompassingRectColliderTag()
{
    return static_pointer_cast<RectColliderComponent>(
        m_Components[m_FirstRectColliderComponentLocation])->
        getColliderTag();
}

shared_ptr<UpdateComponent> GameObject::getFirstUpdateComponent()
{
    return static_pointer_cast<UpdateComponent>(
        m_Components[m_FirstUpdateComponentLocation]);
}

bool GameObject::hasCollider()

```

```
{
    return m_HasCollider;
}

bool GameObject::hasUpdateComponent()
{
    return m_HasUpdateComponent;
}
```



Be sure to study the preceding code before moving on. The explanations that follow assume that you have a basic awareness of variable names and types, as well as function names, parameters, and return types.

Explaining the GameObject class

Let's go through the `GameObject` class one function at a time and reprint the code to make it easy to discuss.

Explaining the update function

The update function is called once for each frame of the game loop for each game object. Like most of our other projects, the current frame rate is required. Inside the update function, a test is done to see if this `GameObject` instance is active and has an update component. A game object does not have to have an update component, although it is true that all the game objects in this project do.

Next, the update function loops through all the components it has, starting from `m_FirstUpdateComponent` through to `m_FirstUpdateComponent + m_NumberUpdateComponents`. This code implies that a game object can have multiple update components. This is so that you can design game objects with layers of behavior. This layering of behavior is discussed further in *Chapter 22, Using Game Objects and Building a Game*. All the game objects in this project have just one update component, so you could simplify (and speed up) the logic in the update function, but I suggest leaving it as it is until you have read *Chapter 22, Using Game Objects and Building a Game*.

It is because a component could be one of many types that we create a temporary update-related component (`tempUpdate`), cast the component from the vector of components to `UpdateComponent`, and call the update function. It doesn't matter about the specific derivation of the `UpdateComponent` class; it will have the update function implemented, so the `UpdateComponent` type is specific enough:

```
void GameObject::update(float fps)
{
```

```

    if (m_Active && m_HasUpdateComponent)
    {
        for (int i = m_FirstUpdateComponentLocation; i <
            m_FirstUpdateComponentLocation +
            m_NumberUpdateComponents; i++)
        {

            shared_ptr<UpdateComponent> tempUpdate =
                static_pointer_cast<UpdateComponent>(
                    m_Components[i]);

            if (tempUpdate->enabled())
            {
                tempUpdate->update(fps);
            }
        }
    }
}

```

When we get to the `addComponent` function in a later section, we will see how we can initialize the various control variables, such as `m_FirstUpdateComponentLocation` and `m_NumberOfUpdateComponents`.

Explaining the draw function

The `draw` function checks whether the game object is active and that it has a graphics component. If it does, then a check is done to see if the graphics component is enabled. If all these tests succeed, then the `draw` function is called:

```

void GameObject::draw(RenderWindow& window)
{
    if (m_Active && m_HasGraphicsComponent)
    {
        if (m_Components[m_GraphicsComponentLocation]->enabled())
        {
            getGraphicsComponent()->draw(window,
                getTransformComponent());
        }
    }
}

```

The structure of the `draw` function implies that not every game object has to draw itself. I mentioned in *Chapter 19, Game Programming Design Patterns – Starting the Space Invaders ++ Game*, that you might want game objects that can never be seen to act as invisible trigger regions (with no graphics component) that respond when the player passes over them or game objects that remain invisible temporarily (temporarily disabled but with a graphics component). In this project, all game objects have a permanently enabled graphics component.

Explaining the `getGraphicsComponent` function

This function returns a shared pointer to the graphics component:

```
shared_ptr<GraphicsComponent> GameObject::getGraphicsComponent()
{
    return static_pointer_cast<GraphicsComponent>(
        m_Components[m_GraphicsComponentLocation]);
}
```

The `getGraphicsComponent` function gives any code that has an instance of the contained game object access to the graphics component.

Explaining the `getTransformComponent` function

This function returns a shared pointer to the transform component:

```
shared_ptr<TransformComponent> GameObject::getTransformComponent()
{
    return static_pointer_cast<TransformComponent>(
        m_Components[m_TransformComponentLocation]);
}
```

The `getTransformComponent` function gives any code that has an instance of the contained game object access to the transform component.

Explaining the `addComponent` function

The `addComponent` function will be used by a factory pattern class we will code in the next chapter. The function receives a shared pointer to a `Component` instance. The first thing that happens inside the function is that the `Component` instance is added to the `m_Components` vector. Next, the component is enabled using the `enabled` function.

What follows is a series of `if` and `else if` statements that deal with each possible type of component. When the type of a component is identified, the various control variables are initialized to enable the logic in the rest of the class to work correctly.

For example, if an update component is detected, then the `m_HasUpdateComponent`, `m_NumberUpdateComponents`, and `m_FirstUpdateComponentLocation` variables are initialized.

As another example, if a collider component is detected along with the rect specific type, then the `m_HasCollider`, `m_NumberRectColliderComponents`, and `m_FirstRectColliderComponent` variables are initialized:

```
void GameObject::addComponent(shared_ptr<Component> component)
{
    m_Components.push_back(component);
    component->enableComponent();

    if (component->getType() == "update")
    {
        m_HasUpdateComponent = true;
        m_NumberUpdateComponents++;
        if (m_NumberUpdateComponents == 1)
        {
            m_FirstUpdateComponentLocation =
                m_Components.size() - 1;
        }
    }
    else if (component->getType() == "graphics")
    {
        // No iteration in the draw method required
        m_HasGraphicsComponent = true;
        m_GraphicsComponentLocation = m_Components.size() - 1;
    }
    else if (component->getType() == "transform")
    {
        // Remember where the Transform component is
        m_TransformComponentLocation = m_Components.size() - 1;
    }
    else if (component->getType() == "collider" &&
        component->getSpecificType() == "rect")
    {
        // Remember where the collider component(s) is
        m_HasCollider = true;
        m_NumberRectColliderComponents++;
        if (m_NumberRectColliderComponents == 1)
        {
            m_FirstRectColliderComponentLocation =
```

```
        m_Components.size() - 1;
    }
}
```

Note that the `GameObject` class plays no part in configuring or setting up the actual components themselves. It is all handled in the factory pattern class we will code in the next chapter.

Explaining the getter and setter functions

The following code is a series of very simple getters and setters:

```
void GameObject::setActive()
{
    m_Active = true;
}

void GameObject::setInactive()
{
    m_Active = false;
}

bool GameObject::isActive()
{
    return m_Active;
}

void GameObject::setTag(String tag)
{
    m_Tag = "" + tag;
}

std::string GameObject::getTag()
{
    return m_Tag;
}
```

The preceding getters and setters provide information about a game object, such as whether it is active and what its tag is. They also allow you to set the tag and tell us whether or not the game object is active.

Explaining the start function

The `start` function is an important one. As we saw when we coded all the components, the `start` function gives access to any component in any game object the components of any other game object. The `start` function is called once all the `GameObject` instances have been composed from all their components. In the next chapter, we will see how this happens, as well as when the `start` function is called on every `GameObject` instance. As we can see, in the `start` function, it loops through every component and shares a new class instance, a `GameObjectSharer` instance. This `GameObjectSharer` class will be coded in the next chapter and will give access to any component from any class. We saw how the invaders need to know where the player is and how the `GameObjectSharer` parameter is used when we coded the various components. When `start` is called on each component, the `this` pointer is also passed in to give each component easy access to its contained `GameObject` instance:

```
void GameObject::start(GameObjectSharer* gos)
{
    auto it = m_Components.begin();
    auto end = m_Components.end();
    for (it;
        it != end;
        ++it)
    {
        (*it)->start(gos, this);
    }
}
```

Let's move on to the `GetComponentByTypeAndSpecificType` function.

Explaining the `GetComponentByTypeAndSpecificType` function

The `GetComponentByTypeAndSpecificType` function has a nested `for` loop that looks for a match of a component type to the first `string` parameter and then looks for a match of the specific component type in the second `string` parameter. It returns a shared pointer to a base class `Component` instance. This implies that the calling code needs to know exactly what derived `Component` type is being returned so that it can cast it to the required type. This shouldn't be a problem because, of course, they are requesting both a type and a specific type:

```
// Slow only use in start
shared_ptr<Component> GameObject::GetComponentByTypeAndSpecificType(
```



```
string type, string specificType) {

    auto it = m_Components.begin();
    auto end = m_Components.end();
    for (it;
        it != end;
        ++it)
    {
        if ((*it)->getType() == type)
        {
            if ((*it)->getSpecificType() == specificType)
            {
                return (*it);
            }
        }
    }

#ifdef debuggingErrors
    cout <<
        "GameObject.cpp::getComponentByTypeAndSpecificType-"
        << "COMPONENT NOT FOUND ERROR!"
        << endl;
#endif
    return m_Components[0];
}
```

The code in this function is quite slow and is therefore intended for use outside of the main game loop. At the end of this function, the code writes an error message to the console if `debuggingErrors` has been defined. The reason for this is because, if execution reaches this point, it means that no matching component was found, and the game will crash. The output to the console should make the error easy to find. The cause of the crash would be that the function was called for an invalid type or specific type.

Explaining the `getEncompassingRectCollider` function

The `getEncompassingRectCollider` function checks whether the game object has a collider and, if it has, returns it to the calling code:

```
FloatRect& GameObject::getEncompassingRectCollider()
{
    if (m_HasCollider)
    {
```

```

        return (static_pointer_cast<RectColliderComponent>(
            m_Components[m_FirstRectColliderComponentLocation]))
            ->getColliderRectF();
    }
}

```

It is worth noting that, if you extend this project to handle more than one type of collider, then this code would need adapting too.

Explaining the `getEncompassingRectColliderTag` function

This simple function returns the tag of the collider. This will be useful for determining what type of object is being tested for collision:

```

string GameObject::getEncompassingRectColliderTag()
{
    return static_pointer_cast<RectColliderComponent>(
        m_Components[m_FirstRectColliderComponentLocation]) ->
        getColliderTag();
}

```

We have just a few more functions to discuss.

Explaining the `getFirstUpdateComponent` function

`getFirstUpdateComponent` uses the `m_FirstUpdateComponent` variable to locate the update component and then returns it to the calling code:

```

shared_ptr<UpdateComponent> GameObject::getFirstUpdateComponent()
{
    return static_pointer_cast<UpdateComponent>(
        m_Components[m_FirstUpdateComponentLocation]);
}

```

Now we're just going to go over a couple of getters, and then we are done.

Explaining the final getter functions

These two remaining functions return a Boolean (each) to tell the calling code whether the game object has a collider and/or an update component:

```

bool GameObject::hasCollider()
{
    return m_HasCollider;
}

```

```
    }

    bool GameObject::hasUpdateComponent ()
    {
        return m_HasUpdateComponent;
    }
```

We have coded the `GameObject` class in full. We can now look at putting it (and all the components it will be composed of) to work.

Summary

In this chapter, we have completed all the code that will draw our game objects to the screen, control their behavior, and let them interact with other classes through collisions. The most important thing to take away from this chapter is not how any of the specific component-based classes work but how flexible the Entity-Component system is. If you want a game object that behaves in a certain way, create a new update component. If it needs to know about other objects in the game, get a pointer to the appropriate component in the `start` function. If it needs to be drawn in a fancy manner, perhaps with a shader or an animation, code a graphics component that performs the actions in the `draw` function. If you need multiple colliders, like we did for Thomas and Bob in the *Thomas Was Late* project, this is no problem: code a new collider-based component.

In the next chapter, we will code the file input and output system, as well as the class that will be the factory that builds all the game objects and composes them with components.

21

File I/O and the Game Object Factory

This chapter handles how a `GameObject` gets into the `m_GameObjects` vector that's used in the game. We will look at how we can describe individual objects and an entire level in a text file. We will write code to interpret the text and then load up values into a class that will be a blueprint for a game object. We will also code a class called `LevelManager` that oversees the whole process, starting from the initial request to load a level sent from an `InputHandler` via the `ScreenManager`, right through to the factory pattern class that assembles a game object from components and delivers it to the `LevelManager`, neatly packed away in the `m_GameObjects` vector.

The following are the steps we will go through in this chapter:

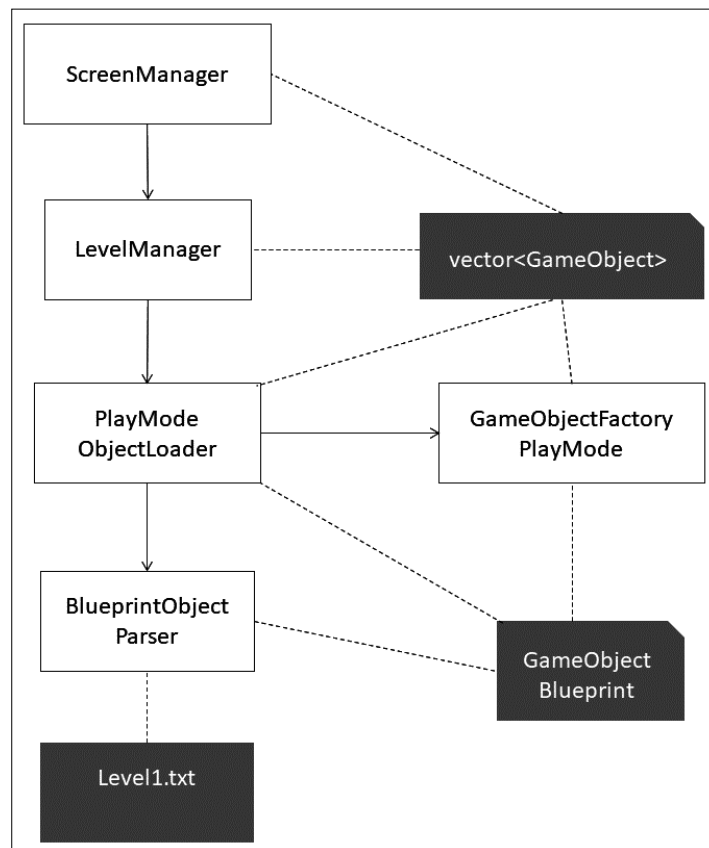
- Examine how we will describe game objects and their components in a text file
- Code the `GameObjectBlueprint` class where the data from the text file will be temporarily stored
- Code the `ObjectTags` class to help describe game objects consistently and without errors
- Code `BlueprintObjectParser`, which will be responsible for loading the data from a game object description in the text file into a `GameObjectBlueprint` instance
- Code `PlayModeObjectLoader`, which will open the text file and receive the `GameObjectBlueprint` instances one at a time from `BlueprintObjectParser`
- Code the `GameObjectFactoryPlayMode` class, which will construct `GameObject` instances from `GameObjectBlueprint` instances

- Code the `LevelManager` class, which oversees this entire process after receiving an instruction from the `ScreenManager` class
- Add the code to the `ScreenManager` class so that we can begin using the new system that we will code in this chapter

Let's start by examining exactly how we describe a game object such as a space invader or a bullet in a text file, let alone a whole wave of them.

The structure of the file I/O and factory classes

Have a look at the following diagram, which gives an overview of the classes we will code in this chapter and how the vector of `GameObject` instances will be shared with the `ScreenManager` class that we coded in *Chapter 19, Game Programming Design Patterns – Starting the Space Invaders ++ Game*:



The preceding diagram shows that there is a vector of `GameObject` instances that's shared between four classes. This is achieved by passing the vector between the functions of the classes by reference. Each class can then carry out its role with the vector and its contents. The `ScreenManager` class will trigger the `LevelManager` class when a new level needs to be loaded into the vector. The individual `Screen` classes and their `InputHandler`-derived classes, as we saw in *Chapter 19, Game Programming Design Patterns – Starting the Space Invaders ++ Game*, have access to `ScreenManager` via `ScreenManagerRemoteControl`.

The `LevelManager` class is ultimately responsible for creating and sharing the vector. `PlayModeObjectLoader` will use `BlueprintObjectParser` to create `GameObjectBlueprint` instances.

The `GameObjectFactoryPlayMode` class will complete the `GameObject` creation process using these `GameObjectBlueprint` instances and pack `GameObject` instances into the vector when prompted to do so by `PlayModeObjectLoader`.

So, where do the different component, position, size, and appearance configurations for each `GameObject` instance come from?

We can also see that three classes have access to a `GameObjectBlueprint` instance. This instance is created by the `LevelManager` class and passed around by reference. `BlueprintObjectParser` will read the `level1.txt` file, which has all the details of each of the game objects. It will initialize all the variables of the `GameObjectBlueprint` class. `PlayModeObjectLoader` will then pass a reference to the vector of `GameObject` instances, and also pass a reference to the fully configured `GameObjectBlueprint` instance to the `GameObjectFactoryPlayMode` class. This is repeated until all the `GameObject` instances are packed away in the vector.

You might be wondering why I have used slightly cumbersome class names such as `GameObjectFactoryPlayMode` and `PlayModeObjectLoader`. The reason is that, once you see how convenient this system is, you might like to build tools that allow you to design your levels in a visual way by dragging and dropping them where required and then have the text file auto-generated rather than typed. This is not especially complicated, but I had to stop adding features to the game at some point. Therefore, you might well end up with a `GameObjectFactoryDesignMode` and a `DesignModeObjectLoader`.

Describing an object in the world

We have already added the `level1.txt` file in the `world` folder in *Chapter 19, Game Programming Design Patterns – Starting the Space Invaders ++ Game*. Let's discuss its uses, future intended uses, and its contents.

First, I would like to point out that a shooter game is not the best way to demonstrate how to describe a game world in a text file like this. The reason for this is that there are only a few types of game object and the most common one, invaders, are all lined up uniformly like soldiers on parade. They would actually be more efficiently described programmatically, perhaps in a nested `for` loop. However, the intention of this project was to show the ideas, rather than learn how to make a Space Invaders clone.

Take a look at the following text, which is a sample from the `level1.txt` file in the `world` folder:

```
[START OBJECT]
[NAME] invader [-NAME]
[COMPONENT] Standard Graphics [-COMPONENT]
[COMPONENT] Invader Update [-COMPONENT]
[COMPONENT] Transform [-COMPONENT]
[LOCATION X] 0 [-LOCATION X]
[LOCATION Y] 0 [-LOCATION Y]
[WIDTH] 2 [-WIDTH]
[HEIGHT] 2 [-HEIGHT]
[BITMAP NAME] invader1 [-BITMAP NAME]
[ENCOMPASSING RECT COLLIDER] invader [-ENCOMPASSING_RECT COLLIDER]
[END OBJECT]
```

The preceding text describes a single object in the game; in this case, an invader. The object begins with the following text:

```
[START OBJECT]
```

That will inform our code we'll write that a new object is being described. Next in the text, we can see the following:

```
[NAME] invader [-NAME]
```

This informs the code that the type of object is an invader. This will eventually be set as the `m_Tag` of the `ColliderComponent` class. The invader will be identifiable for what it is. The text that comes next is as follows:

```
[COMPONENT] Standard Graphics [-COMPONENT]
[COMPONENT] Invader Update [-COMPONENT]
[COMPONENT] Transform [-COMPONENT]
```

This tells our system that this object will have three components added to it: a `StandardGraphicsComponent` instance, an `InvaderUpdateComponent` instance, and a `TransformComponent` instance. This means the object will be drawn in the standard way and will behave according to the rules we coded for an invader. It will also mean it has a location and scale in the game world. It is possible to have objects that don't have any components or fewer components. An object that takes no action and doesn't move will not need an update component, an object that is invisible will not need a graphics component (perhaps just an invisible collider which triggers some action), and an object that has no position in the world (perhaps a debugging object) will not need a transform component.

The position and scale of an object are determined by the following four lines of text:

```
[LOCATION X] 0 [-LOCATION X]
[LOCATION Y] 0 [-LOCATION Y]
[WIDTH] 2 [-WIDTH]
[HEIGHT] 2 [-HEIGHT]
```

The following line of text determines what graphics file will be used for the texture of this object:

```
[BITMAP NAME] invader1 [-BITMAP NAME]
```

The following line means that the object can be collided with. A decorative object, perhaps floating clouds (or a bee), would not need a collider:

```
[ENCOMPASSING RECT COLLIDER] invader [-ENCOMPASSING_RECT COLLIDER]
```

The final line of text will inform our system that the object has finished describing itself:

```
[END OBJECT]
```

Now, let's have a look at how we describe a bullet object:

```
[START OBJECT]
[NAME] bullet [-NAME]
[COMPONENT] Standard Graphics [-COMPONENT]
[COMPONENT] Transform [-COMPONENT]
[COMPONENT] Bullet Update [-COMPONENT]
[LOCATION X] -1 [-LOCATION X]
[LOCATION Y] -1 [-LOCATION Y]
[WIDTH] 0.1 [-WIDTH]
[HEIGHT] 2.0 [-HEIGHT]
[BITMAP NAME] bullet [-BITMAP NAME]
[ENCOMPASSING RECT COLLIDER] bullet [-ENCOMPASSING_RECT COLLIDER]
[SPEED] 75.0 [-SPEED]
[END OBJECT]
```


This is very similar but not the same as an invader. A bullet object has additional data, such as a set speed. The invader's speed is set in the logic of the `InvaderUpdateComponent` class. We could have done this for the bullet's speed as well, but this demonstrates that you can describe the object in as much or as little detail as the specific game design requires. Also, as we would expect, a bullet has a `BulletUpdateComponent` and a different value for the `[BITMAP NAME]` element. Notice that the location of the bullet is set to -1, -1. This means the bullets are outside of the playable area at the start of the game. In the next chapter, we will see how an invader, or the player, can spawn them into action when required.

Now, study the following text, which describes the player's ship:

```
[START OBJECT]
[NAME] Player [-NAME]
[COMPONENT] Standard Graphics [-COMPONENT]
[COMPONENT] Transform [-COMPONENT]
[COMPONENT] Player Update [-COMPONENT]
[LOCATION X] 50 [-LOCATION X]
[LOCATION Y] 40 [-LOCATION Y]
[WIDTH] 3.0 [-WIDTH]
[HEIGHT] 2.0 [-HEIGHT]
[BITMAP NAME] playership [-BITMAP NAME]
[ENCOMPASSING RECT COLLIDER] player [-ENCOMPASSING_RECT COLLIDER]
[SPEED] 10.0 [-SPEED]
[END OBJECT]
```

The preceding text was probably quite predictable based on our discussion so far. Now that we've gone through this, we can get to work on coding the system that will interpret these object descriptions and convert them into usable `GameObject` instances.

Coding the `GameObjectBlueprint` class

Create a new header file in the `Header Files/FileIO` filter called `GameObjectBlueprint.h` and add the following code:

```
#pragma once
#include<vector>
#include<string>
#include<map>

using namespace std;
```

```

class GameObjectBlueprint {

private:
    string m_Name = "";
    vector<string> m_ComponentList;
    string m_BitmapName = "";
    float m_Width;
    float m_Height;
    float m_LocationX;
    float m_LocationY;
    float m_Speed;
    bool m_EncompassingRectCollider = false;
    string m_EncompassingRectColliderLabel = "";

public:
    float getWidth();
    void setWidth(float width);
    float getHeight();
    void setHeight(float height);
    float getLocationX();
    void setLocationX(float locationX);
    float getLocationY();
    void setLocationY(float locationY);
    void setName(string name);
    string getName();
    vector<string>& getComponentList();
    void addToComponentList(string newComponent);
    string getBitmapName();
    void setBitmapName(string bitmapName);
    string getEncompassingRectColliderLabel();
    bool getEncompassingRectCollider();
    void setEncompassingRectCollider(string label);
};

```

GameObjectBlueprint has a member variable for every possible property that could go into a game object. Note that it does not compartmentalize the properties by component. For example, it just has variables for things such as width, height, and location; it doesn't go to the trouble of identifying these as part of the transform component. These details are handled in the factory. It also provides getters and setters so that the BlueprintObjectParser class can pack away all the values from the `level1.txt` file and the `GameObjectFactoryPlayMode` class can extract all the values, instantiate the appropriate components, and add them to an instance of `GameObject`.

Create a new source file in the Source Files/FileIO filter called `GameObjectBlueprint.cpp` and add the following code, which is for the definitions of the functions we have just declared:

```
#include "GameObjectBlueprint.h"

float GameObjectBlueprint::getWidth()
{
    return m_Width;
}

void GameObjectBlueprint::setWidth(float width)
{
    m_Width = width;
}

float GameObjectBlueprint::getHeight()
{
    return m_Height;
}

void GameObjectBlueprint::setHeight(float height)
{
    m_Height = height;
}

float GameObjectBlueprint::getLocationX()
{
    return m_LocationX;
}

void GameObjectBlueprint::setLocationX(float locationX)
{
    m_LocationX = locationX;
}

float GameObjectBlueprint::getLocationY()
{
    return m_LocationY;
}

void GameObjectBlueprint::setLocationY(float locationY)
{
    m_LocationY = locationY;
}
```

```
}

void GameObjectBlueprint::setName(string name)
{
    m_Name = "" + name;
}

string GameObjectBlueprint::getName()
{
    return m_Name;
}

vector<string>& GameObjectBlueprint::getComponentList()
{
    return m_ComponentList;
}

void GameObjectBlueprint::addToComponentList(string newComponent)
{
    m_ComponentList.push_back(newComponent);
}

string GameObjectBlueprint::getBitmapName()
{
    return m_BitmapName;
}

void GameObjectBlueprint::setBitmapName(string bitmapName)
{
    m_BitmapName = "" + bitmapName;
}

string GameObjectBlueprint::getEncompassingRectColliderLabel()
{
    return m_EncompassingRectColliderLabel;
}

bool GameObjectBlueprint::getEncompassingRectCollider()
{
    return m_EncompassingRectCollider;
}

void GameObjectBlueprint::setEncompassingRectCollider(
```

```
        string label)
    {
        m_EncompassingRectCollider = true;
        m_EncompassingRectColliderLabel = "" + label;
    }
```

Although this is a long class, there is nothing here we haven't seen before. The setter functions receive values which are copied into a vector or a variable, while the getters allow access to these values.

Coding the ObjectTags class

The way in which we describe the game objects in the `level1.txt` file needs to be precise because the `BlueprintObjectParser` class we will code after this class will be reading the text from the file and looking for matches. For example, the `[START OBJECT]` tag will trigger the start of a new object. If that tag is misspelled as, say, `[START OBJECR]`, then the whole system falls apart and there will be all kinds of bugs, and even crashes when we run the game. To avoid this happening, we will define constant (programmatically unchangeable) `string` variables for all the tags we need to describe the game objects. We can use these `string` variables instead of typing something such as `[START OBJECT]` and have much less chance of making a mistake.

Create a new header file in the Header Files/FileIO filter called `ObjectTags.h` and add the following code:

```
#pragma once
#include <string>

using namespace std;

static class ObjectTags {
public:
    static const string START_OF_OBJECT;
    static const string END_OF_OBJECT;
    static const string COMPONENT;
    static const string COMPONENT_END;
    static const string NAME;
    static const string NAME_END;
    static const string WIDTH;
    static const string WIDTH_END;
    static const string HEIGHT;
    static const string HEIGHT_END;
    static const string LOCATION_X;
```

```

static const string LOCATION_X_END;
static const string LOCATION_Y;
static const string LOCATION_Y_END;
static const string BITMAP_NAME;
static const string BITMAP_NAME_END;
static const string ENCOMPASSING_RECT_COLLIDER;
static const string ENCOMPASSING_RECT_COLLIDER_END;
};

```

We have declared a `const string` for every tag we will use to describe the game objects. Now, we can initialize them.

Create a new source file in the Source Files/FileIO filter called `ObjectTags.cpp` and add the following code:

```

#include "DevelopState.h"
#include "objectTags.h"

const string ObjectTags::START_OF_OBJECT = "[START OBJECT] ";
const string ObjectTags::END_OF_OBJECT = "[END OBJECT] ";
const string ObjectTags::COMPONENT = "[COMPONENT] ";
const string ObjectTags::COMPONENT_END = "[-COMPONENT] ";
const string ObjectTags::NAME = "[NAME] ";
const string ObjectTags::NAME_END = "[-NAME] ";
const string ObjectTags::WIDTH = "[WIDTH] ";
const string ObjectTags::WIDTH_END = "[-WIDTH] ";
const string ObjectTags::HEIGHT = "[HEIGHT] ";
const string ObjectTags::HEIGHT_END = "[-HEIGHT] ";
const string ObjectTags::LOCATION_X = "[LOCATION X] ";
const string ObjectTags::LOCATION_X_END = "[-LOCATION X] ";
const string ObjectTags::LOCATION_Y = "[LOCATION Y] ";
const string ObjectTags::LOCATION_Y_END = "[-LOCATION Y] ";
const string ObjectTags::BITMAP_NAME = "[BITMAP NAME] ";
const string ObjectTags::BITMAP_NAME_END = "[-BITMAP NAME] ";
const string ObjectTags::ENCOMPASSING_RECT_COLLIDER =
    "[ENCOMPASSING RECT COLLIDER] ";

const string ObjectTags::ENCOMPASSING_RECT_COLLIDER_END
    = "[-ENCOMPASSING RECT COLLIDER] ";

```

That's all the string variables initialized. We can now use them in the next class and be sure we are describing the game objects consistently.

Coding the BlueprintObjectParser class

This class will have the code that actually reads the text from the `level1.txt` file we have discussed. It will parse one object at a time, as identified by the start and end tags we saw previously.

Create a new header file in the Header Files/FileIO filter called `BlueprintObjectParser.h` and add the following code:

```
#pragma once
#include "GameObjectBlueprint.h"
#include <string>

using namespace std;

class BlueprintObjectParser {
private:
    string extractStringBetweenTags(
        string stringToSearch, string startTag, string endTag);

public:
    void parseNextObjectForBlueprint(
        ifstream& reader, GameObjectBlueprint& bp);
};
```

The `extractStringBetweenTags` private function will capture the content between two tags. The parameters are three `string` instances. The first `string` is a full line of text from `level1.txt`, while the second and third are the start and end tags, which need to be discarded. The text between the two tags is then returned to the calling code.

The `parseNextObjectForBlueprint` function receives an `ifstream` reader, just like the one we used in the Zombie shooter and the Thomas Was Late games. It is used to read from the file. The second parameter is a reference to a `GameObjectBlueprint` instance. The function will populate the `GameObjectBlueprint` instance with the values that were read from the `level1.txt` file, which can then be used back in the calling code to create an actual `GameObject`. We will see how that happens when we code the `PlayModeObjectLoader` class next and the `GameObjectFactoryPlayMode` class after that.

Let's code the definitions we have just discussed.

Create a new source file in the Source Files/FileIO filter called `BlueprintObjectParser.cpp` and add the following code:

```
#include "BlueprintObjectParser.h"
#include "ObjectTags.h"
#include <iostream>
#include <fstream>

void BlueprintObjectParser::parseNextObjectForBlueprint(
    ifstream& reader, GameObjectBlueprint& bp)
{
    string lineFromFile;
    string value = "";

    while (getline(reader, lineFromFile))
    {
        if (lineFromFile.find(ObjectTags::COMPONENT)
            != string::npos)
        {
            value = extractStringBetweenTags(lineFromFile,
                ObjectTags::COMPONENT,
                ObjectTags::COMPONENT_END);

            bp.addToComponentList(value);
        }

        else if (lineFromFile.find(ObjectTags::NAME)
            != string::npos)
        {
            value = extractStringBetweenTags(lineFromFile,
                ObjectTags::NAME, ObjectTags::NAME_END);

            bp.setName(value);
        }

        else if (lineFromFile.find(ObjectTags::WIDTH)
            != string::npos)
        {

```



```
        value = extractStringBetweenTags(lineFromFile,
            ObjectTags::WIDTH, ObjectTags::WIDTH_END);

        bp.setWidth(stof(value));
    }

    else if (lineFromFile.find(ObjectTags::HEIGHT)
        != string::npos)
    {

        value = extractStringBetweenTags(lineFromFile,
            ObjectTags::HEIGHT, ObjectTags::HEIGHT_END);

        bp.setHeight(stof(value));
    }

    else if (lineFromFile.find(ObjectTags::LOCATION_X)
        != string::npos)
    {

        value = extractStringBetweenTags(lineFromFile,
            ObjectTags::LOCATION_X,
            ObjectTags::LOCATION_X_END);

        bp.setLocationX(stof(value));
    }

    else if (lineFromFile.find(ObjectTags::LOCATION_Y)
        != string::npos)
    {

        value = extractStringBetweenTags(
            lineFromFile,
            ObjectTags::LOCATION_Y,
            ObjectTags::LOCATION_Y_END);

        bp.setLocationY(stof(value));
    }

    else if (lineFromFile.find(ObjectTags::BITMAP_NAME)
        != string::npos)
    {
```

```

        value = extractStringBetweenTags(lineFromFile,
            ObjectTags::BITMAP_NAME,
            ObjectTags::BITMAP_NAME_END);

        bp.setBitmapName(value);
    }

    else if (lineFromFile.find(
        ObjectTags::ENCOMPASSING_RECT_COLLIDER)
        != string::npos)
    {

        value = extractStringBetweenTags(lineFromFile,
            ObjectTags::ENCOMPASSING_RECT_COLLIDER,
            ObjectTags::ENCOMPASSING_RECT_COLLIDER_END);

        bp.setEncompassingRectCollider(value);
    }

    else if (lineFromFile.find(ObjectTags::END_OF_OBJECT)
        != string::npos)
    {

        return;
    }
}

}

string BlueprintObjectParser::extractStringBetweenTags(
    string stringToSearch, string startTag, string endTag)
{
    int start = startTag.length();
    int count = stringToSearch.length() - startTag.length()
        - endTag.length();

    string stringBetweenTags = stringToSearch.substr(
        start, count);

    return stringBetweenTags;
}

```

The code in `parseNextObjectForBlueprint` is lengthy but straightforward. The series of `if` statements identifies the starting tag at the beginning of the line of text and then passes the line of text to the `extractStringBetweenTags` function, which returns the value that is then loaded into the `GameObjectBlueprint` reference in the appropriate place. Notice that the function exits when `GameObjectBlueprint` has had all the data loaded into it. This point is recognized when `ObjectTags::END_OF_OBJECT` is found.

Coding the `PlayModeObjectLoader` class

This is the class that will pass `GameObjectBlueprint` instances to `BlueprintObjectParser`. When it gets the completed blueprint back, it will pass them to the `GameObjectFactoryPlayMode` class, which will construct the `GameObject` instance and pack it away in the vector instance. Once all the `GameObject` instances have been built and stored, responsibility will be handed to the `LevelManager` class, which will control access to the vector for other parts of the game engine. This is a very small class with just one function, but it links many other classes together. Refer to the diagram at the start of this chapter for clarification.

Create a new header file in the Header Files/`FileIO` filter called `PlayModeObjectLoader.h` and add the following code:

```
#pragma once
#include <vector>
#include <string>
#include "GameObject.h"
#include "BlueprintObjectParser.h"
#include "GameObjectFactoryPlayMode.h"

using namespace std;

class PlayModeObjectLoader {
private:
    BlueprintObjectParser m_BOP;
    GameObjectFactoryPlayMode m_GameObjectFactoryPlayMode;

public:
    void loadGameObjectsForPlayMode(
        string pathToFile, vector<GameObject>& mGameObjects);
};
```

The `PlayModeObjectLoader` class has an instance of the previous class we coded, that is, the `BlueprintObjectParser` class. It also has an instance of the class we will code next, that is, the `GameObjectFactoryPlayMode` class. It has a single public function, which receives a reference to a vector that holds `GameObject` instances.

Now, we will code the definition of the `loadGameObjectsForPlayMode` function. Create a new source file in the `Source Files/FileIO` filter called `PlayModeObjectLoader.cpp` and add the following code:

```
#include "PlayModeObjectLoader.h"
#include "ObjectTags.h"
#include <iostream>
#include <fstream>

void PlayModeObjectLoader::
    loadGameObjectsForPlayMode(
        string pathToFile, vector<GameObject>& gameObjects)
{
    ifstream reader(pathToFile);
    string lineFromFile;

    float x = 0, y = 0, width = 0, height = 0;
    string value = "";
    while (getline(reader, lineFromFile)) {
        if (lineFromFile.find(
            ObjectTags::START_OF_OBJECT) != string::npos) {

            GameObjectBlueprint bp;
            m_BOP.parseNextObjectForBlueprint(reader, bp);
            m_GameObjectFactoryPlayMode.buildGameObject(
                bp, gameObjects);
        }
    }
}
```

The function receives a `string`, which is the path to the file that needs to be loaded. This game only has one such file, but you could add more files with different layouts, varying numbers of invaders, or totally different game objects if you wanted to.

An `ifstream` instance is used to read one line at a time from the file. In the `while` loop, the start tag is identified using `ObjectTags::START_OF_OBJECT`, and the `parseNextObjectForBlueprint` function of `BlueprintObjectParser` is called. You probably remember from the `BlueprintObjectParser` class that the completed blueprint is returned when `ObjectTags::END_OF_OBJECT` is reached.

The next line of code calls the `buildGameObject` of the `GameObjectFactoryPlayMode` class and passes in the `GameObjectBlueprint` instance. We will code the `GameObjectFactory` class now.

Coding the `GameObjectFactoryPlayMode` class

Now, we will code our factory, which will construct working game objects from the `GameObject` class and all the component related classes that we coded in the previous chapter. We will make extensive use of smart pointers, so we don't have to worry about deleting memory when we have finished with it.

Create a new header file in the Header Files/FileIO filter called `GameObjectFactoryPlayMode.h` and add the following code:

```
#pragma once
#include "GameObjectBlueprint.h"
#include "GameObject.h"
#include <vector>

class GameObjectFactoryPlayMode {
public:
    void buildGameObject(GameObjectBlueprint& bp,
        std::vector<GameObject*>& gameObjects);
};
```

The factory class has just one function, `buildGameObject`. We have already seen the code that calls this function in the previous code we wrote for the `PlayModeObjectLoader` class. The function receives a reference to the blueprint, as well as a reference to the vector of `GameObject` instances.

Create a new source file in the Source Files/FileIO filter called `GameObjectFactoryPlayMode.cpp` and add the following code:

```
#include "GameObjectFactoryPlayMode.h"
#include <iostream>
#include "TransformComponent.h"
#include "StandardGraphicsComponent.h"
#include "PlayerUpdateComponent.h"
#include "RectColliderComponent.h"
#include "InvaderUpdateComponent.h"
#include "BulletUpdateComponent.h"
```

```

void GameObjectFactoryPlayMode::buildGameObject(
    GameObjectBlueprint& bp,
    std::vector<GameObject>& gameObjects)
{
    GameObject gameObject;
    gameObject.setTag(bp.getName());

    auto it = bp.getComponentList().begin();
    auto end = bp.getComponentList().end();
    for (it;
        it != end;
        ++it)
    {
        if (*it == "Transform")
        {
            gameObject.addComponent(
                make_shared<TransformComponent>(
                    bp.getWidth(),
                    bp.getHeight(),
                    Vector2f(bp.getLocationX(),
                        bp.getLocationY())));
        }
        else if (*it == "Player Update")
        {
            gameObject.addComponent(make_shared
                <PlayerUpdateComponent>());
        }
        else if (*it == "Invader Update")
        {
            gameObject.addComponent(make_shared
                <InvaderUpdateComponent>());
        }
        else if (*it == "Bullet Update")
        {
            gameObject.addComponent(make_shared
                <BulletUpdateComponent>());
        }
        else if (*it == "Standard Graphics")
        {
            shared_ptr<StandardGraphicsComponent> sgp =
                make_shared<StandardGraphicsComponent>();

            gameObject.addComponent(sgp);
            sgp->initializeGraphics(

```

```
        bp.getBitmapName(),
        Vector2f(bp.getWidth(),
        bp.getHeight()));
    }
}

if (bp.getEncompassingRectCollider()) {
    shared_ptr<RectColliderComponent> rcc =
        make_shared<RectColliderComponent>(
        bp.getEncompassingRectColliderLabel());

    gameObject.addComponent(rcc);
    rcc->setOrMoveCollider(bp.getLocationX(),
        bp.getLocationY(),
        bp.getWidth(),
        bp.getHeight());
}

gameObjects.push_back(gameObject);
}
```

The first thing that happens in the `buildGameObject` function is that a new `GameObject` instance is created and the `setTag` function of the `GameObject` class is used to pass in the name of the current object being built:

```
GameObject gameObject;
gameObject.setTag(bp.getName());
```

Next, a `for` loop loops through all the components in the `m_Components` vector. For each component that is found, a different `if` statement creates a component of the appropriate type. The way that each component is created varies, as you would expect since the way they are coded varies.

The following code creates a shared pointer to a `TransformComponent` instance. You can see the necessary arguments being passed to the constructor, that is, width, height, and location. The result of creating the new shared pointer to a `TransformComponent` instance is passed to the `addComponent` function of the `GameObject` class. The `GameObject` instance now has its size and place in the world:

```
if (*it == "Transform")
{
    gameObject.addComponent(make_shared<TransformComponent>(
        bp.getWidth(),
        bp.getHeight(),
        Vector2f(bp.getLocationX(), bp.getLocationY())));
}
```

The following code executes when a `PlayerUpdateComponent` is required. Again, the code creates a new shared pointer to the appropriate class and passes it in to the `addComponent` function of the `GameObject` instance:

```
else if (*it == "Player Update")
{
    gameObject.addComponent(make_shared
        <PlayerUpdateComponent>());
}
```

The following three blocks of code use exactly the same technique to add either an `InvaderUpdateComponent`, `BulletUpdateComponent`, or `StandardGraphicsComponent` instance. Notice the extra line of code after adding a `StandardGraphicsComponent` instance that calls the `initialize` function, which adds a `Texture` instance (if required) to the `BitmapStore` singleton and prepares the component to be drawn:

```
else if (*it == "Invader Update")
{
    gameObject.addComponent(make_shared
        <InvaderUpdateComponent>());
}

else if (*it == "Bullet Update")
{
    gameObject.addComponent(make_shared
        <BulletUpdateComponent>());
}

else if (*it == "Standard Graphics")
{
    shared_ptr<StandardGraphicsComponent> sgp =
        make_shared<StandardGraphicsComponent>();

    gameObject.addComponent(sgp);
    sgp->initializeGraphics(
        bp.getBitmapName(),
        Vector2f(bp.getWidth(),
            bp.getHeight()));
}
```


The final `if` block, as shown in the following code, handles adding a `RectColliderComponent` instance. The first line of code creates the shared pointer, while the second line of code calls the `addComponent` function to add the instance to the `GameObject` instance. The third line of code calls the `setOrMoveCollider` and passes in the location and size of the object. At this stage, the object is ready to be collided with. Obviously, we still need to write the code that tests for collisions. We will do so in the next chapter:

```
if (bp.getEncompassingRectCollider()) {
    shared_ptr<RectColliderComponent> rcc =
        make_shared<RectColliderComponent>(
            bp.getEncompassingRectColliderLabel());

    gameObject.addComponent(rcc);

    rcc->setOrMoveCollider(bp.getLocationX(),
        bp.getLocationY(),
        bp.getWidth(),
        bp.getHeight());
}
```

The following line of code in the class adds the just-constructed `GameObject` instance to the vector that will be shared with the `GameScreen` class and used to make the game come to life:

```
gameObjects.push_back(gameObject);
```

The next class we will write makes it easy to share the vector we have just filled with `GameObject` instances around the various classes of the project.

Coding the `GameObjectSharer` class

This class will have two pure virtual functions that share `GameObject` instances with other classes.

Create a new header file in the `Header Files/FileIO` filter called `GameObjectSharer.h` and add the following code:

```
#pragma once
#include<vector>
#include<string>

class GameObject;
class GameObjectSharer {
public:
```

```

        virtual std::vector<GameObject>& getGameObjectsWithGOS() = 0;
        virtual GameObject& findFirstObjectWithTag(
            std::string tag) = 0;
    };

```

The `getGameObjectsWithGOS` function returns a reference to the entire vector of `GameObject` instances. The `findFirstObjectWithTag` function returns just a single `GameObject` reference. We will see how we implement these functions when we inherit from `GameObjectSharer` when we code the `LevelManager` class next.

Briefly, before the `LevelManager` class, create a new source file in the Source Files/FileIO filter called `GameObjectSharer.cpp` and add the following code:

```

/*****
*****THIS IS AN INTERFACE*****
*****/

```

Again, this is just a placeholder file and the full functionality goes in any of the classes that inherit from `GameObjectSharer`; in this case, the `LevelManager` class.

Coding the LevelManager class

The `LevelManager` class is the connection between what we coded in *Chapter 19, Game Programming Design Patterns – Starting the Space Invaders ++ Game*, and everything we coded in this chapter. The `ScreenManager` class will have an instance of the `LevelManager` class, and the `LevelManager` class will instigate loading levels (using all the classes we have just coded) and share `GameObject` instances with any classes that need them.

Create a new header file in the Header Files/Engine filter called `LevelManager.h` and add the following code:

```

#pragma once
#include "GameObject.h"
#include <vector>
#include <string>
#include "GameObjectSharer.h"

using namespace std;

class LevelManager : public GameObjectSharer {
private:
    vector<GameObject> m_GameObjects;

    const std::string WORLD_FOLDER = "world";

```

```
    const std::string SLASH = "/";

    void runStartPhase();
    void activateAllGameObjects();

public:
    vector<GameObject>& getGameObjects();
    void loadGameObjectsForPlayMode(string screenToLoad);

    /*****
    *****/
    From GameObjectSharer interface
    /*****/

    vector<GameObject>& GameObjectSharer::getGameObjectsWithGOS()
    {
        return m_GameObjects;
    }

    GameObject& GameObjectSharer::findFirstObjectWithTag(
        string tag)
    {
        auto it = m_GameObjects.begin();
        auto end = m_GameObjects.end();
        for (it;
            it != end;
            ++it)
        {
            if ((*it).getTag() == tag)
            {
                return (*it);
            }
        }
    }

#ifdef debuggingErrors
    cout <<
        "LevelManager.h findFirstGameObjectWithTag() "
        << "- TAG NOT FOUND ERROR!"
        << endl;
#endif
    return m_GameObjects[0];
}

};
```

This class provides two different ways to get the vector full of the game objects. One way is via a simple call to `getGameObjects`, but another is via the `getGameObjectsWithGOS` function. The latter is the implementation of a pure virtual function from the `GameObjectSharer` class and will be a way to pass access to each and every game object so that it has access to all the other game objects. You may recall from *Chapter 20, Game Objects and Components*, that a `GameObjectSharer` instance is passed in during the start function call of the `GameObject` class. It was in this function that, among other things, the invaders could get access to the location of the player.

There are also two private functions: `runStartPhase`, which loops through all the `GameObject` instances calling `start`, and `activateAllGameObjects`, which loops through and sets all the `GameObject` instances to the active status.

Also, part of the `LevelManager` class is the `loadGameObjectsForPlayMode` function, which will trigger the entire game object creation process that the rest of this chapter has described.

The final function in the `LevelManager.h` file is the implementation of the other `GameObjectSharer` pure virtual function, `findFirstObjectWithTag`. This allows any class with a `GameObjectSharer` instance to track down a specific game object using its tag. The code loops through all the `GameObject` instances in the vector and returns the first match. Note, that if no match is found, a null pointer will be returned and crash the game. We use an `#ifdef` statement to output some text to the console to tell us what caused the crash so that we won't be scratching our heads for hours should we accidentally search for a tag that doesn't exist.

We can now code the implementations of the functions.

Create a new source file in the `Source Files/Engine` filter called `LevelManager.cpp` and add the following code:

```
#include "LevelManager.h"
#include "PlayModeObjectLoader.h"
#include <iostream>

void LevelManager::
    loadGameObjectsForPlayMode(string screenToLoad)
{
    m_GameObjects.clear();
    string levelToLoad = ""
        + WORLD_FOLDER + SLASH + screenToLoad;

    PlayModeObjectLoader pmol;
    pmol.loadGameObjectsForPlayMode(
```

```
        levelToLoad, m_GameObjects);

    runStartPhase();
}

vector<GameObject>& LevelManager::getGameObjects()
{
    return m_GameObjects;
}

void LevelManager::runStartPhase()
{
    auto it = m_GameObjects.begin();
    auto end = m_GameObjects.end();
    for (it;
        it != end;
        ++it)
    {
        (*it).start(this);
    }

    activateAllGameObjects();
}

void LevelManager::activateAllGameObjects()
{
    auto it = m_GameObjects.begin();
    auto end = m_GameObjects.end();
    for (it;
        it != end;
        ++it)
    {
        (*it).setActive();
    }
}
```

The `loadLevelForPlayMode` function clears the vector, instantiates a `PlayModeObjectLoader` instance that does all the file reading, and packs the `GameObject` instances in the vector. Finally, the `runStartPhase` function is called. In the `runStartPhase` function, all the `GameObject` instances are passed a `GameObjectSharer` (`this`) and given the opportunity to set themselves up, ready to be played. Remember that, inside the `GameObject` class in the `start` function, each of the derived `Component` instances is given access to `GameObjectSharer`. Refer to *Chapter 20, Game Objects and Components*, to see what we did with this when we coded the `Component` classes.

The `runStartPhase` function concludes by calling `activateAllGameObjects`, which loops through the vector, calling `setActive` on every `GameObject` instance.

The `getGameObjects` function passes a reference to the vector of `GameObject` instances.

Now that we have coded the `LevelManager` class, we can update the `ScreenManager` and the `ScreenManagerRemoteControl` classes that it implements.

Updating the `ScreenManager` and `ScreenManagerRemoteControl` classes

Open the `ScreenManagerRemoteControl.h` file and uncomment everything so that the code is the same as the following. I have highlighted the lines that have been uncommented:

```
#pragma once
#include <string>
#include <vector>
#include "GameObject.h"
#include "GameObjectSharer.h"

using namespace std;

class ScreenManagerRemoteControl
{
public:
    virtual void SwitchScreens(string screenToSwitchTo) = 0;
    virtual void loadLevelInPlayMode(string screenToLoad) = 0;
    virtual vector<GameObject>& getGameObjects() = 0;
    virtual GameObjectSharer& shareGameObjectSharer() = 0;
};
```

Next, open `ScreenManager.h`, which implements this interface and uncomments all the commented-out code. The code in question is abbreviated and highlighted as follows:

```
...
#include "SelectScreen.h"
// #include "LevelManager.h"
#include "BitmapStore.h"
...
...
private:
```

```
map <string, unique_ptr<Screen>> m_Screens;
//LevelManager m_LevelManager;

protected:
    ...
    ...

/*****
*****
From ScreenManagerRemoteControl interface
*****
*****/

    ...
    ...

//vector<GameObject>&
//ScreenManagerRemoteControl::getGameObjects()
//{
//    //return m_LevelManager.getGameObjects();
//}

//GameObjectSharer& shareGameObjectSharer()
//{
//    //return m_LevelManager;
//}
    ...
    ...
```

Be sure to uncomment the include directive, the `m_LevelManager` instance, as well as the two functions.

The `ScreenManager` and `ScreenManagerRemoteControl` classes are now fully functional and the `getGameObjects` and `shareGameObjectSharer` functions are usable by any class with a reference to the `ScreenManager` class.

Where are we now?

At this point, all the errors in our `GameObject` class, as well as all component-related classes, are gone. We are making good progress.

Furthermore, we can revisit the `ScreenManager.h` file and uncomment all the commented-out code.

Open `ScreenManager.h` and uncomment the `#include` directive, as follows:

```
//#include "LevelManager.h"
```

Change it to this:

```
#include "LevelManager.h"
```

Do the same for the functions from the `ScreenManagerRemoteControl` interface that are implemented in `ScreenManager.h`. They look like the following:

```
void ScreenManagerRemoteControl::
    loadLevelInPlayMode(string screenToLoad)
{
    //m_LevelManager.getGameObjects().clear();
    //m_LevelManager.
        //loadGameObjectsForPlayMode(screenToLoad);
    SwitchScreens("Game");
}

//vector<GameObject>&
    //ScreenManagerRemoteControl::getGameObjects()
//{
    //return m_LevelManager.getGameObjects();
//}
```

Change them as follows:

```
void ScreenManagerRemoteControl::
    loadLevelInPlayMode(string screenToLoad)
{
    m_LevelManager.getGameObjects().clear();
    m_LevelManager.
        loadGameObjectsForPlayMode(screenToLoad);

    SwitchScreens("Game");
}

vector<GameObject>&
    ScreenManagerRemoteControl::getGameObjects()
{
    return m_LevelManager.getGameObjects();
}
```

We aren't quite ready to run the game, however, because there are still some missing classes that are used in the code, such as `BulletSpawner` in the `InvaderUpdateComponent` class.

Summary

In this chapter, we have put in place a way to describe a level in a game and a system to interpret the description and build usable `GameObject` instances. The Factory pattern is used in many types of programming, not just game development. The implementation we have used is the simplest possible implementation and I encourage you to put the Factory pattern on your list of patterns to research and develop further. The implementation we have used should serve you well if you wish to build some deep and interesting games, however.

In the next chapter, we will finally make the game come to life by adding collision detection, bullet spawning, and the logic of the game itself.

22

Using Game Objects and Building a Game

This chapter is the final stage of the Space Invaders ++ project. We will learn how to receive input from a gamepad using SFML to do all the hard work and we will also code a class that will handle communication between the invaders and the `GameScreen` class, as well as the player and the `GameScreen` class. The class will allow the player and the invaders to spawn bullets, but the exact same technique could be used for any kind of communication that you need between different parts of your own game, so it is useful to know. The final part of the game (as usual) will be the collision detection and the logic of the game itself. Once Space Invaders ++ is up and running, we will learn how to use the Visual Studio debugger, which will be invaluable when you are designing your own logic because it allows you to step through your code a line at a time and see the value of variables. It is also a useful tool for studying the execution flow of the patterns we have assembled over the course of this project.

Here is what we will do in this chapter:

- Code a solution for spawning bullets
- Handle the player's input, including with a gamepad
- Detect collisions between all the necessary objects
- Code the main logic of the game
- Learn about debugging and understand the execution flow

Let's start by spawning bullets.

Spawning bullets

We need a way to spawn bullets from both the player and each of the invaders. The solutions to both are very similar but not identical. We need a way to allow `GameInputHandler` to spawn bullets when a keyboard key or gamepad button is pressed, and we need `InvaderUpdateComponent` to use its already existing logic to spawn bullets.

The `GameScreen` class has a vector holding all the `GameObject` instances, so `GameScreen` is the ideal candidate to move a bullet into position and set it moving up or down the screen, depending on who or what triggered the shot. We need a way for the `GameInputHandler` class and `InvaderUpdateComponent` to communicate with the `GameScreen` class, but we also need to restrict the communication to just spawning bullets; we don't want them to be able to take control of any other part of the `GameScreen` class.

Let's code an abstract class that `GameScreen` can inherit from.

Coding the BulletSpawner class

Create a new header file in the Header Files/`GameObjects` filter called `BulletSpawner.h` and add the following code:

```
#include <SFML/Graphics.hpp>

class BulletSpawner
{
public:
    virtual void spawnBullet(
        sf::Vector2f spawnLocation, bool forPlayer) = 0;
};
```

The preceding code creates a new class called `BulletSpawner` with a single pure virtual function called `spawnBullet`. The `spawnBullet` function has two parameters. The first is a `Vector2f` instance that will determine the spawn location. Actually, as we will see soon, when the bullet is spawned, this position will be tweaked slightly, depending on whether the bullet is going up the screen (as a player bullet) or down the screen (as an invader bullet). The second parameter is a Boolean that will be true if the bullet belongs to the player or false if it belongs to an invader.

Create a new source file in the `Source Files/GameObject` filter called `BulletSpawner.cpp` and add the following code:

```

/*****
*****THIS IS AN INTERFACE*****
*****/

```



As usual, this .cpp file is optional. I just wanted to bring balance to the source.

Now, go to `GameScreen.h`, since this is where we will implement the function of this class.

Updating GameScreen.h

First, update the include directives and the class declaration, as highlighted in the following code, to make the `GameScreen` class inherit from `BulletSpawner`:

```

#pragma once
#include "Screen.h"
#include "GameInputHandler.h"
#include "GameOverInputHandler.h"
#include "BulletSpawner.h"

class GameScreen : public Screen, public BulletSpawner
{
    ...
    ...
}

```

Next, add some extra functions and variable declarations, as highlighted in the following code, to `GameScreen.h`:

```

private:
    ScreenManagerRemoteControl* m_ScreenManagerRemoteControl;
    shared_ptr<GameInputHandler> m_GIH;

    int m_NumberInvadersInWorldFile = 0;

    vector<int> m_BulletObjectLocations;
    int m_NextBullet = 0;
    bool m_WaitingToSpawnBulletForPlayer = false;
    bool m_WaitingToSpawnBulletForInvader = false;
    Vector2f m_PlayerBulletSpawnLocation;
    Vector2f m_InvaderBulletSpawnLocation;

```

```
    Clock m_BulletClock;

    Texture m_BackgroundTexture;
    Sprite m_BackgroundSprite;

public:
    static bool m_GameOver;

    GameScreen(ScreenManagerRemoteControl* smrc, Vector2i res);
    void initialise() override;
    void virtual update(float fps);
    void virtual draw(RenderWindow& window);

    BulletSpawner* getBulletSpawner();
```

The new variables include a vector of int values that will hold the locations of all the bullets in the vector, which holds all the game objects. It also has a few control variables so that we can keep track of the next bullet to use, whether the bullet is for the player or an invader, and the position to spawn the bullet in. We have also declared a new `sf::Clock` instance because we want to limit the fire rate of the player. Finally, we have the `getBulletSpawner` function, which will return a pointer to this class in the form of a `BulletSpawner`. This will give the recipient access to the `spawnBullet` function, but nothing else.

Now, we can add the implementation of the `spawnBullet` function. Add the following code to `GameScreen.h` at the end of all the other code, but inside the closing curly brace of the `GameScreen` class:

```
/*
*****
*****
From BulletSpawner interface
*****
*****
*/

void BulletSpawner::spawnBullet(Vector2f spawnLocation,
    bool forPlayer)
{
    if (forPlayer)
    {
        Time elapsedTime = m_BulletClock.getElapsedTime();
        if (elapsedTime.asMilliseconds() > 500) {
            m_PlayerBulletSpawnLocation.x = spawnLocation.x;
            m_PlayerBulletSpawnLocation.y = spawnLocation.y;
            m_WaitingToSpawnBulletForPlayer = true;
        }
    }
}
```

```

        m_BulletClock.restart();
    }
}
else
{
    m_InvaderBulletSpawnLocation.x = spawnLocation.x;
    m_InvaderBulletSpawnLocation.y = spawnLocation.y;
    m_WaitingToSpawnBulletForInvader = true;
}
}

```

The implementation of the `spawnBullet` function is a simple `if - else` structure. The `if` block executes if a bullet is requested for the player and the `else` block executes if a bullet is requested for an invader.

The `if` block checks that at least half a second has passed since the last bullet was requested and, if it has, the `m_WaitingToSpawnBulletForPlayer` variable is set to `true`, the location to spawn the bullet at is copied, and the clock is restarted, ready to test the player's next request.

The `else` block records the spawn location for an invader's bullet and sets `m_WaitingToSpawnBulletForInvader` to `true`. No interaction with the `Clock` instance is necessary as the rate of fire for the invaders is controlled in the `InvaderUpdateComponent` class.

The last part of the `BulletSpawner` puzzle, before we get to actually spawning the bullets, is to add the definition of `getBulletSpawner` to the end of `GameScreen.cpp`. Here is the code to add:

```

BulletSpawner* GameScreen::getBulletSpawner()
{
    return this;
}

```

This returns a pointer to `GameScreen`, which gives us access to the `spawnBullet` function.

Handling the player's input

Add some more declarations to the `GameInputHandler.h` file so that your code matches what follows. I have highlighted the new code to add:

```

#pragma once
#include "InputHandler.h"
#include "PlayerUpdateComponent.h"

```

```
#include "TransformComponent.h"

class GameScreen;

class GameInputHandler : public InputHandler
{
private:
    shared_ptr<PlayerUpdateComponent> m_PUC;
    shared_ptr<TransformComponent> m_PTC;

    bool mBButtonPressed = false;
public:

    void initialize();
    void handleGamepad() override;
    void handleKeyPressed(Event& event,
        RenderWindow& window) override;

    void handleKeyReleased(Event& event,
        RenderWindow& window) override;
};
```

The `GameInputHandler` class now has access to the player's update component and the player's transform component. This is very useful because it means we can tell the `PlayerUpdateComponent` instance and the player's `TransformComponent` instance what keyboard keys and gamepad controls the player is manipulating. What we haven't seen yet is how exactly these two shared pointers will be initialized – after all, aren't the `GameObject` instances and all their components packed away in a vector? You can probably guess the solution has something to do with `GameObjectSharer`. Let's keep coding to find out more.

In the `GameInputHandler.cpp` file, add a forward declaration of the `BulletSpawner` class after the include directives but before the initialize function, as highlighted in the following code:

```
#include "GameInputHandler.h"
#include "SoundEngine.h"
#include "GameScreen.h"

class BulletSpawner;

void GameInputHandler::initialize() {
...
}
```

In the `GameInputHandler.cpp` file, add the following highlighted code to the `handleKeyPressed` function:

```
void GameInputHandler::handleKeyPressed(
    Event& event, RenderWindow& window)
{
    // Handle key presses
    if (event.key.code == Keyboard::Escape)
    {
        SoundEngine::playClick();
        getPointerToScreenManagerRemoteControl()->
            SwitchScreens("Select");
    }

    if (event.key.code == Keyboard::Left)
    {
        m_PUC->moveLeft();
    }

    if (event.key.code == Keyboard::Right)
    {
        m_PUC->moveRight();
    }

    if (event.key.code == Keyboard::Up)
    {
        m_PUC->moveUp();
    }

    if (event.key.code == Keyboard::Down)
    {
        m_PUC->moveDown();
    }
}
```

Notice that we are responding to keyboard presses just like we have been doing throughout this book. Here, however, we are calling the functions from the `PlayerUpdateComponent` class that we coded in *Chapter 20, Game Objects and Components*, in order to take the required actions.

In the `GameInputHandler.cpp` file, add the following highlighted code to the `handleKeyReleased` function:

```
void GameInputHandler::handleKeyReleased(
    Event& event, RenderWindow& window)
{
    if (event.key.code == Keyboard::Left)
    {
        m_PUC->stopLeft();
    }

    else if (event.key.code == Keyboard::Right)
    {
        m_PUC->stopRight();
    }

    else if (event.key.code == Keyboard::Up)
    {
        m_PUC->stopUp();
    }

    else if (event.key.code == Keyboard::Down)
    {
        m_PUC->stopDown();
    }

    else if (event.key.code == Keyboard::Space)
    {
        // Shoot a bullet
        SoundEngine::playShoot();
        Vector2f spawnLocation;
        spawnLocation.x = m_PTC->getLocation().x +
            m_PTC->getSize().x / 2;

        spawnLocation.y = m_PTC->getLocation().y;

        static_cast<GameScreen*>(getmParentScreen())->
            spawnBullet(spawnLocation, true);
    }
}
```

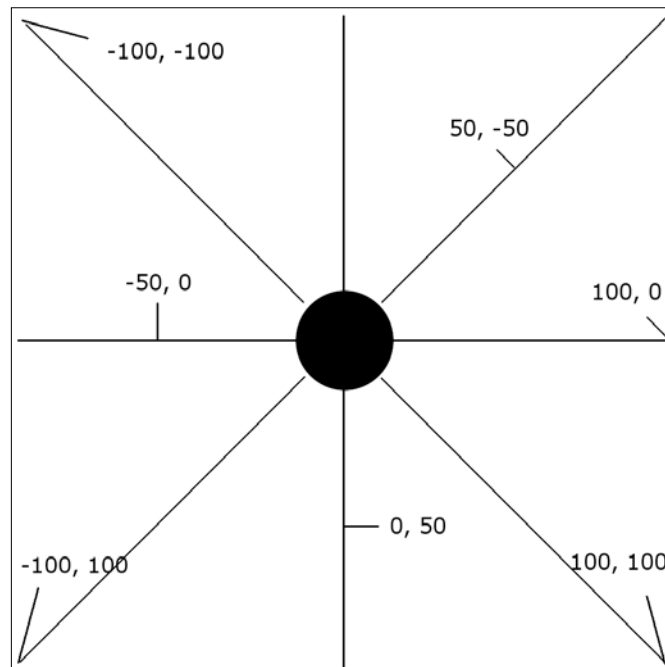
The preceding code also relies on calling functions from the `PlayerUpdateComponent` class to handle what happens when the player releases a keyboard key. The `PlayerUpdateComponent` class can then stop movement in the appropriate direction, depending on which keyboard key has just been released. When the *space* key is released, the `getParentScreen` function is chained with the `spawnBullet` function to trigger a bullet being spawned. Notice that the spawn coordinates (`spawnLocation`) are calculated using the shared pointer to the `PlayerTransformComponent` instance.

Let's learn about how SFML helps us interact with a gamepad and then we can return to the `PlayerInputHandler` class to add some more functionality.

Using a gamepad

Handling gamepad input is made exceptionally easy by SFML. Gamepad (or joystick) input is handled by the `sf::Joystick` class. SFML can handle input from up to eight gamepads, but this tutorial will stick to just one.

You can think of the position of a thumbstick/joystick as a 2D graph that starts at -100, -100 at the top left corner and goes to 100, 100 at the bottom right corner. The position of the thumbstick can, therefore, be represented by a 2D coordinate. The following diagram illustrates this with a few example coordinates:



All we need to do is grab the value and report it to the `PlayerUpdateComponent` class for each frame of the game loop. Capturing the position is as simple as the following two lines of code:

```
float x = Joystick::getAxisPosition(0, sf::Joystick::X);
float y = Joystick::getAxisPosition(0, sf::Joystick::Y);
```

The zero parameter requests data from the primary gamepad. You can use values 0 through 7 to get input from eight gamepads.

There is something else we need to consider as well. Most gamepads, especially thumbsticks, are mechanically imperfect and will register small values even when they are not being touched. If we send these values to the `PlayerUpdateComponent` class, then the ship will aimlessly drift around the screen. For this reason, we will create a **dead zone**. This is a range of movement where we will ignore any values. 10 percent of the range of movement works quite well. Therefore, if the values that are retrieved from the `getAxisPosition` function are between -10 and 10 on either axis, we will ignore them.

To get input from the B button of the gamepad, we use the following line of code:

```
// Has the player pressed the B button?

if (Joystick::isButtonPressed(0, 1))
{
    // Take action here
}
```

The preceding code detects when the B button on an Xbox One gamepad is pressed. Other controllers will vary. The 0, 1 parameters refer to the primary gamepad and button number 1. To detect when a button is released, we will need to code a bit of our own logic. As we want to shoot a bullet on release and not when it is pressed, we will use a simple Boolean to track this. Let's code the rest of the `GameInputHandler` class and see how we can put what we have just learned into action.

In the `GameInputHandler.cpp` file, add the following highlighted code to the `handleGamepad` function:

```
void GameInputHandler::handleGamepad()
{
    float deadZone = 10.0f;
    float x = Joystick::getAxisPosition(0, sf::Joystick::X);
    float y = Joystick::getAxisPosition(0, sf::Joystick::Y);

    if (x < deadZone && x > -deadZone)
    {
```

```

        x = 0;
    }

    if (y < deadZone && y > -deadZone)
    {
        y = 0;
    }

    m_PUC->updateShipTravelWithController(x, y);

    // Has the player pressed the B button?
    if (Joystick::isButtonPressed(0, 1))
    {
        mBButtonPressed = true;
    }

    // Has player just released the B button?
    if (!Joystick::isButtonPressed(0, 1) && mBButtonPressed)
    {
        mBButtonPressed = false;

        // Shoot a bullet
        SoundEngine::playShoot();
        Vector2f spawnLocation;
        spawnLocation.x = m_PTC->getLocation().x +
            m_PTC->getSize().x / 2;

        spawnLocation.y = m_PTC->getLocation().y;

        static_cast<GameScreen*>(getmParentScreen())->
            getBulletSpawner()->spawnBullet(
                spawnLocation, true);
    }
}

```

We begin by defining a dead zone of 10 and then proceed to capture the position of the thumbstick. The next two `if` blocks test whether the thumbstick position is within the dead zone. If it is, then the appropriate value is set to zero to avoid the ship drifting. Then, we can call the `updateShipTravelWithController` function on the `PlayerUpdateComponent` instance. That is the thumbstick dealt with.

The next `if` statement sets a Boolean to `true` if the B button on the gamepad is pressed. The next `if` statement detects when the B button is not pressed, and the Boolean is set to `true`. This indicates that the B button has just been released.

Inside the `if` block, we set the Boolean to `false`, ready to handle the next button release, play a shooting sound, get the location to spawn the bullet, and call the `spawnBullet` function by chaining the `getmParentScreen` and `getBulletSpawner` functions.

Coding the `PhysicsEnginePlayMode` class

This is the class that will do all the collision detection. In this game, there are several collision events we want to watch out for:

- Has an invader reached the left- or right-hand side of the screen? If so, all the invaders need to drop down one row and head back in the other direction.
- Has an invader collided with the player? As the invaders get lower, we want them to be able to bump into the player and cause a life to be lost.
- Has an invader bullet hit the player? Each time an invader bullet hits the player, we need to hide the bullet, ready for reuse, and deduct a life from the player.
- Has a player bullet hit an invader? Each time the player hits an invader, the invader should be destroyed, the bullet hidden (ready for reuse), and the player's score increased.

This class will have an `initialize` function that the `GameScreen` class will call to prepare for detecting collisions, a `detectCollisions` function that the `GameScreen` class will call once for each frame after all the game objects have updated themselves, and three more functions which will be called from the `detectCollisions` function to separate out the work of detecting the different collisions I have just listed.

Those three functions are `detectInvaderCollisions`, `detectPlayerCollisionsAndInvaderDirection`, and `handleInvaderDirection`. Hopefully, the names of these functions make it clear what will happen in each function.

Create a new source file in the Header Files/Engine filter called `PhysicsEnginePlayMode.h` and add the following code:

```
#pragma once
#include "GameObjectSharer.h"
#include "PlayerUpdateComponent.h"

class PhysicsEnginePlayMode
{
```

```

private:
    shared_ptr<PlayerUpdateComponent> m_PUC;

    GameObject* m_Player;
    bool m_InvaderHitWallThisFrame = false;
    bool m_InvaderHitWallPreviousFrame = false;
    bool m_NeedToDropDownAndReverse = false;
    bool m_CompletedDropDownAndReverse = false;

    void detectInvaderCollisions(
        vector<GameObject>& objects,
        const vector<int>& bulletPositions);

    void detectPlayerCollisionsAndInvaderDirection(
        vector<GameObject>& objects,
        const vector<int>& bulletPositions);

    void handleInvaderDirection();

public:
    void initilize(GameObjectSharer& gos);
    void detectCollisions(
        vector<GameObject>& objects,
        const vector<int>& bulletPositions);
};

```

Study the preceding code to make a note of the parameters that are passed to each of the functions. Also take note of the four member Boolean variables that will be used throughout the class. Furthermore, notice that there is a pointer to a `GameObject` type being declared which will be a permanent reference to the player ship, so we don't need to keep finding the `GameObject` that represents the player for each frame of the game loop.

Create a new source file in the `Source Files/Engine` filter called `PhysicsEnginePlayMode.cpp` and add the following include directives and the `detectInvaderCollisions` function. Study the code and then we will discuss it:

```

#include "DevelopState.h"
#include "PhysicsEnginePlayMode.h"
#include <iostream>
#include "SoundEngine.h"
#include "WorldState.h"
#include "InvaderUpdateComponent.h"

```

```
#include "BulletUpdateComponent.h"

void PhysicsEnginePlayMode::
detectInvaderCollisions(
    vector<GameObject>& objects,
    const vector<int>& bulletPositions)
{
    Vector2f offScreen(-1, -1);

    auto invaderIt = objects.begin();
    auto invaderEnd = objects.end();
    for (invaderIt;
        invaderIt != invaderEnd;
        ++invaderIt)
    {
        if ((*invaderIt).isActive()
            && (*invaderIt).getTag() == "invader")
        {
            auto bulletIt = objects.begin();
            // Jump to the first bullet
            advance(bulletIt, bulletPositions[0]);
            auto bulletEnd = objects.end();
            for (bulletIt;
                bulletIt != bulletEnd;
                ++bulletIt)
            {
                if ((*invaderIt).getEncompassingRectCollider()
                    .intersects((*bulletIt)
                        .getEncompassingRectCollider())
                    && (*bulletIt).getTag() == "bullet"
                    && static_pointer_cast<
                        BulletUpdateComponent>(
                            (*bulletIt).getFirstUpdateComponent())
                    ->m_BelongsToPlayer)
                {
                    SoundEngine::playInvaderExplode();
                    (*invaderIt).getTransformComponent()
                        ->getLocation() = offScreen;

                    (*bulletIt).getTransformComponent()
                        ->getLocation() = offScreen;

                    WorldState::SCORE++;
                }
            }
        }
    }
}
```

```

        WorldState::NUM_INVADERS--;
        (*invaderIt).setInactive();
    }
}
}
}
}

```

The preceding code loops through all the game objects. The first `if` statement checks whether the current game object is both active and an invader:

```

if ((*invaderIt).isActive()
    && (*invaderIt).getTag() == "invader")

```

If it is an active invader, another loop is entered and each of the game objects that represents a bullet is looped through:

```

auto bulletIt = objects.begin();
// Jump to the first bullet
advance(bulletIt, bulletPositions[0]);
auto bulletEnd = objects.end();
for (bulletIt;
    bulletIt != bulletEnd;
    ++bulletIt)

```

The next `if` statement checks whether the current invader has collided with the current bullet and whether that bullet was fired by the player (we don't want invaders shooting themselves):

```

if ((*invaderIt).getEncompassingRectCollider()
    .intersects((*bulletIt)
    .getEncompassingRectCollider())
    && (*bulletIt).getTag() == "bullet"
    && static_pointer_cast<BulletUpdateComponent>(
    (*bulletIt).getFirstUpdateComponent())
    ->m_BelongsToPlayer)

```

When this test is true, a sound is played, the bullet is moved off-screen, the number of invaders is decremented, the player's score is increased, and the invader is set to inactive.

Now, we will detect player collisions and the invader's direction of travel.

Add the `detectPlayerCollisionsAndInvaderDirection` function, as follows:

```
void PhysicsEnginePlayMode::
detectPlayerCollisionsAndInvaderDirection(
    vector<GameObject>& objects,
    const vector<int>& bulletPositions)
{
    Vector2f offScreen(-1, -1);

    FloatRect playerCollider =
        m_Player->getEncompassingRectCollider();

    shared_ptr<TransformComponent> playerTransform =
        m_Player->getTransformComponent();

    Vector2f playerLocation =
        playerTransform->getLocation();

    auto it3 = objects.begin();
    auto end3 = objects.end();
    for (it3;
        it3 != end3;
        ++it3)
    {
        if ((*it3).isActive() &&
            (*it3).hasCollider() &&
            (*it3).getTag() != "Player")
        {
            // Get a reference to all the parts of
            // the current game object we might need
            FloatRect currentCollider = (*it3)
                .getEncompassingRectCollider();

            // Detect collisions between objects
            // with the player
            if (currentCollider.intersects(playerCollider))
            {
                if ((*it3).getTag() == "bullet")
                {
                    SoundEngine::playPlayerExplode();
                    WorldState::LIVES--;
                    (*it3).getTransformComponent()->
```

```

        getLocation() = offScreen;
    }

    if ((*it3).getTag() == "invader")
    {
        SoundEngine::playPlayerExplode();
        SoundEngine::playInvaderExplode();
        WorldState::LIVES--;
        (*it3).getTransformComponent()->
            getLocation() = offScreen;

        WorldState::SCORE++;
        (*it3).setInactive();
    }
}

shared_ptr<TransformComponent>
currentTransform =
    (*it3).getTransformComponent();

Vector2f currentLocation =
    currentTransform->getLocation();

string currentTag = (*it3).getTag();
Vector2f currentSize =
    currentTransform->getSize();

// Handle the direction and descent
// of the invaders
if (currentTag == "invader")
{
    // This is an invader
    if (!m_NeedToDropDownAndReverse &&
        !m_InvaderHitWallThisFrame)
    {
        // Currently no need to dropdown
        // and reverse from previous frame
        // or any hits this frame
        if (currentLocation.x >=
            WorldState::WORLD_WIDTH -
                currentSize.x)
        {

```

```
        // The invader is passed its
        // furthest right position
        if (static_pointer_cast
            <InvaderUpdateComponent>((*it3)
            .getFirstUpdateComponent())->
            isMovingRight())
        {
            // The invader is travelling
            // right so set a flag that
            // an invader has collided

            m_InvaderHitWallThisFrame
                = true;
        }
    }
    else if (currentLocation.x < 0)
    {
        // The invader is past its furthest
        // left position
        if (!static_pointer_cast
            <InvaderUpdateComponent>(
            (*it3).getFirstUpdateComponent())
            ->isMovingRight())
        {
            // The invader is travelling
            // left so set a flag that an
            // invader has collided
            m_InvaderHitWallThisFrame
                = true;
        }
    }
}
else if (m_NeedToDropDownAndReverse
    && !m_InvaderHitWallPreviousFrame)
{
    // Drop down and reverse has been set
    if ((*it3).hasUpdateComponent())
    {
        // Drop down and reverse
        static_pointer_cast<
            InvaderUpdateComponent>(
            (*it3).getFirstUpdateComponent())
            ->dropDownAndReverse();
    }
}
```

```

        }
    }
}
}

```

The preceding code is longer than the previous function because we are checking for more conditions. Before the code loops through all the game objects, it gets a reference to all the relevant player data. This is so we don't have to do this for every check:

```

FloatRect playerCollider =
    m_Player->getEncompassingRectCollider();

shared_ptr<TransformComponent> playerTransform =
    m_Player->getTransformComponent();

Vector2f playerLocation =
    playerTransform->getLocation();

```

Next, the loop goes through every game object. The first `if` test checks whether the current object is active, has a collider, and is not the player. We don't want to test the player colliding with themselves:

```

if ((*it3).isActive() &&
    (*it3).hasCollider() &&
    (*it3).getTag() != "Player")

```

The next `if` test does the actual collision detection to see if the current game object intersects with the player:

```

if (currentCollider.intersects(playerCollider))

```

Next, there are two nested `if` statements: one that handles collisions with a bullet belonging to an invader and one that handles collisions with an invader.

Next, the code checks each and every game object that is an invader to see whether it has hit the left- or right-hand side of the screen. Note that the `m_NeedToDropDownAndReverse` and `m_InvaderHitWallLastFrame` Boolean variables are used because it will not always be the first invader in the vector that hits the side of the screen. Therefore, detecting the collision and triggering dropdown and reversal are handled in consecutive frames to guarantee that all the invaders drop down and reverse, regardless of which one of them triggers it.

Finally, when both conditions are true, `handleInvaderDirection` is called.

Add the `handleInvaderDirection` function, as follows:

```
void PhysicsEnginePlayMode::handleInvaderDirection()
{
    if (m_InvaderHitWallThisFrame) {
        m_NeedToDropDownAndReverse = true;
        m_InvaderHitWallThisFrame = false;
    }
    else {
        m_NeedToDropDownAndReverse = false;
    }
}
```

This function just sets and unsets Booleans accordingly so that the next pass through the `detectPlayerCollisionAndDirection` function will actually drop-down the invaders and change their direction.

Add the `initialize` function to prepare the class for action:

```
void PhysicsEnginePlayMode::initilize(GameObjectSharer& gos) {
    m_PUC = static_pointer_cast<PlayerUpdateComponent>(
        gos.findFirstObjectWithTag("Player")
        .getComponentByTypeAndSpecificType("update", "player"));

    m_Player = &gos.findFirstObjectWithTag("Player");
}
```

In the preceding code, the pointer to `PlayerUpdateComponent` is initialized, as well as the pointer to the player `GameObject`. This will avoid calling these relatively slow functions during the game loop.

Add the `detectCollisions` function, which will be called from the `GameScreen` class once each frame:

```
void PhysicsEnginePlayMode::detectCollisions(
    vector<GameObject>& objects,
    const vector<int>& bulletPositions)
{
    detectInvaderCollisions(objects, bulletPositions);
    detectPlayerCollisionsAndInvaderDirection(
        objects, bulletPositions);

    handleInvaderDirection();
}
```

The `detectCollisions` function calls the three functions that handle the different phases of collision detection. You could have lumped all the code into this single function, but then it would be quite unwieldy. Alternatively, you could separate the three big functions into their own `.cpp` files, just like we did with the `update` and `draw` functions in the *Thomas Was Late* game.

In the next section, we will create an instance of the `PhysicsEngineGameMode` class and use it in the `GameScreen` class as we bring the game to life.

Making the game

By the end of this section, we will have a playable game. In this section, we will add code to the `GameScreen` class to bring together everything we have been coding over the last three chapters. To get started, add an instance of `PhysicsEngineGameMode` to `GameScreen.h` by adding an extra include directive, as follows:

```
#include "PhysicsEnginePlayMode.h"
```

Then, declare an instance, as highlighted in the following code:

```
private:
    ScreenManagerRemoteControl* m_ScreenManagerRemoteControl;
    shared_ptr<GameInputHandler> m_GIH;
    PhysicsEnginePlayMode m_PhysicsEnginePlayMode;
    ...
    ...
```

Now, open the `GameScreen.cpp` file, add some extra include directives, and forward-declare the `BulletSpawner` class, as highlighted in the following code:

```
#include "GameScreen.h"
#include "GameUIPanel.h"
#include "GameInputHandler.h"
#include "GameOverUIPanel.h"
#include "GameObject.h"
#include "WorldState.h"
#include "BulletUpdateComponent.h"
#include "InvaderUpdateComponent.h"

class BulletSpawner;

int WorldState::WORLD_HEIGHT;
int WorldState::NUM_INVADERS;
int WorldState::NUM_INVADERS_AT_START;
```

Next, in the `GameScreen.cpp` file, update the `initialize` function by adding the following highlighted code inside the existing code:

```
void GameScreen::initialise()
{
    m_GIH->initialize();
    m_PhysicsEnginePlayMode.initilize(
        m_ScreenManagerRemoteControl->
        shareGameObjectSharer());

    WorldState::NUM_INVADERS = 0;

    // Store all the bullet locations and
    // Initialize all the BulletSpawners in the invaders
    // Count the number of invaders
    int i = 0;
    auto it = m_ScreenManagerRemoteControl->
        getGameObjects().begin();

    auto end = m_ScreenManagerRemoteControl->
        getGameObjects().end();

    for (it;
        it != end;
        ++it)
    {
        if ((*it).getTag() == "bullet")
        {
            m_BulletObjectLocations.push_back(i);
        }
        if ((*it).getTag() == "invader")
        {
            static_pointer_cast<InvaderUpdateComponent>(
                (*it).getFirstUpdateComponent())->
                initializeBulletSpawner(
                    getBulletSpawner(), i);

            WorldState::NUM_INVADERS++;
        }
        ++i;
    }
}
```

```

    m_GameOver = false;

    if (WorldState::WAVE_NUMBER == 0)
    {
        WorldState::NUM_INVADERS_AT_START =
            WorldState::NUM_INVADERS;

        WorldState::WAVE_NUMBER = 1;
        WorldState::LIVES = 3;
        WorldState::SCORE = 0;
    }
}

```

The preceding code in the `initialize` function initializes the physics engine that will handle all the collision detection. Next, it loops through all the game objects and performs two tasks: one task in each of the `if` blocks.

The first `if` block tests whether the current game object is a bullet. If it is, then its integer location in the vector of game objects is stored in the `m_BulletObjectLocations` vector. Remember from when we coded the physics engine that this vector is useful when doing collision detection. The vector will also be used in this class to keep track of the next bullet to use when the player or an invader wants to take a shot.

The second `if` block detects whether the current game object is an invader and, if it is, calls the `initializeBulletSpawner` function on its update component and passes in a pointer to a `BulletSpawner` by calling the `getBulletSpawner` function. The invaders are now capable of spawning bullets.

Now, we need to add some code to the `update` function to handle what happens in each frame of the game during the updating phase. This is highlighted in the following code. All the new code goes inside the already existing `if (!m_GameOver)` block:

```

void GameScreen::update(float fps)
{
    Screen::update(fps);

    if (!m_GameOver)
    {
        if (m_WaitingToSpawnBulletForPlayer)
        {
            static_pointer_cast<BulletUpdateComponent>(
                m_ScreenManagerRemoteControl->
                getGameObjects())

```



```
        [m_BulletObjectLocations[m_NextBullet]].
        getFirstUpdateComponent()->
        spawnForPlayer(
            m_PlayerBulletSpawnLocation);

    m_WaitingToSpawnBulletForPlayer = false;
    m_NextBullet++;

    if (m_NextBullet == m_BulletObjectLocations
        .size())
    {
        m_NextBullet = 0;
    }
}

if (m_WaitingToSpawnBulletForInvader)
{
    static_pointer_cast<BulletUpdateComponent>(
        m_ScreenManagerRemoteControl->
        getGameObjects()
        [m_BulletObjectLocations[m_NextBullet]].
        getFirstUpdateComponent()->
        spawnForInvader(
            m_InvaderBulletSpawnLocation);

    m_WaitingToSpawnBulletForInvader = false;
    m_NextBullet++;

    if (m_NextBullet ==
        m_BulletObjectLocations.size())
    {
        m_NextBullet = 0;
    }
}

auto it = m_ScreenManagerRemoteControl->
    getGameObjects().begin();

auto end = m_ScreenManagerRemoteControl->
    getGameObjects().end();

for (it;
    it != end;
    ++it)
```

```

    {
        (*it).update(fps);
    }

    m_PhysicsEnginePlayMode.detectCollisions(
        m_ScreenManagerRemoteControl->getGameObjects(),
        m_BulletObjectLocations);

    if (WorldState::NUM_INVADERS <= 0)
    {
        WorldState::WAVE_NUMBER++;
        m_ScreenManagerRemoteControl->
            loadLevelInPlayMode("level1");
    }

    if (WorldState::LIVES <= 0)
    {
        m_GameOver = true;
    }
}
}

```

In the preceding new code, the first `if` block checks whether a new bullet is required for the player. If it is the next available bullet, the `GameObject` instance, has its `BulletUpdateComponent` instance's `spawnForPlayer` function called. The specific `GameObject` instance to use is identified using the `m_NextBulletObject` variable with the `m_BulletObjectLocations` vector. The remaining code in the first `if` block prepares for the next bullet to be fired.

The second `if` block executes if an invader is waiting for a bullet to be fired. Exactly the same technique is used to activate a bullet, except the `spawnForInvader` function is used, which sets it moving downward.

Next, there is a loop which loops through every game object. This is key to everything because, inside the loop, the `update` function is called on every `GameObject` instance.

The final line of code in the preceding new code calls the `detectCollisions` function to see if any of the `GameObject` instances (in their just-updated positions) have collided.

Finally, we will add some code to the draw function in `GameScreen.cpp`. The new code is highlighted inside the existing code in the following listing:

```
void GameScreen::draw(RenderWindow & window)
{
    // Change to this screen's view to draw
    window.setView(m_View);
    window.draw(m_BackgroundSprite);

    // Draw the GameObject instances
    auto it = m_ScreenManagerRemoteControl->
        getGameObjects().begin();

    auto end = m_ScreenManagerRemoteControl->
        getGameObjects().end();

    for (it;
        it != end;
        ++it)
    {
        (*it).draw(window);
    }

    // Draw the UIPanel view(s)
    Screen::draw(window);
}
```

The preceding code simply calls the draw function on each of the `GameObject` instances in turn. Now, you have completed the Space Invaders ++ project and can run the game. Congratulations!

Understanding the flow of execution and debugging

Much of the last four chapters has been about the code structure. It is very possible that you still have doubts and uncertainties about which class instantiates which instance or in what order the various functions are called. Wouldn't it be useful if there was a way to execute the project and follow the path of execution from `int main()` right through to `return 0;` in the `Space Invaders ++.cpp` file? It turns out we can, and the following is how to do it.

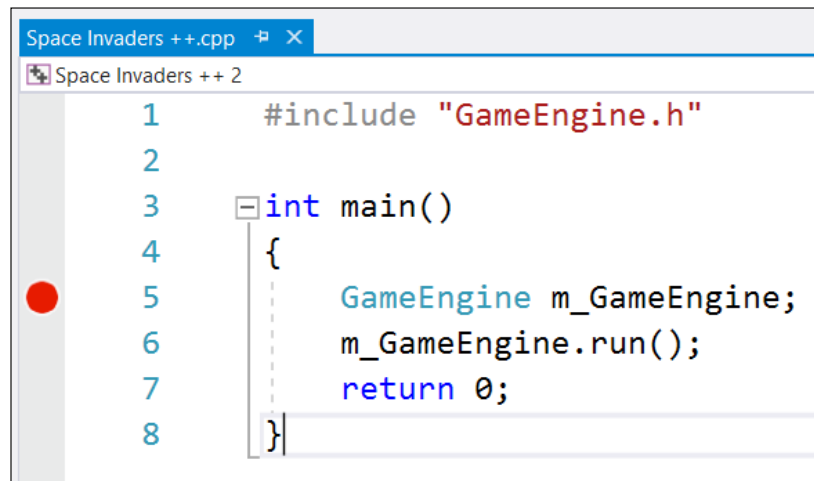
We will now explore the debugging facilities in Visual Studio while simultaneously trying to understand the structure of the project.

Open the `Space Invaders ++.cpp` file and find the first line of code, as follows:

```
GameEngine m_GameEngine;
```

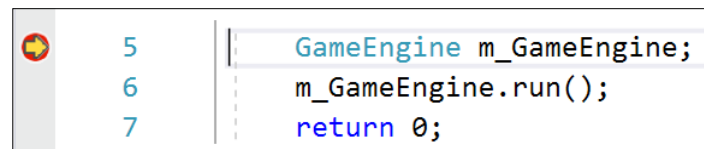
The preceding code is the first line of code that gets executed. It declares an instance of the `GameEngine` class and sets all our hard work in motion.

Right-click the preceding line of code and select **Breakpoint | Insert Breakpoint**. The following is what the screen should look like:

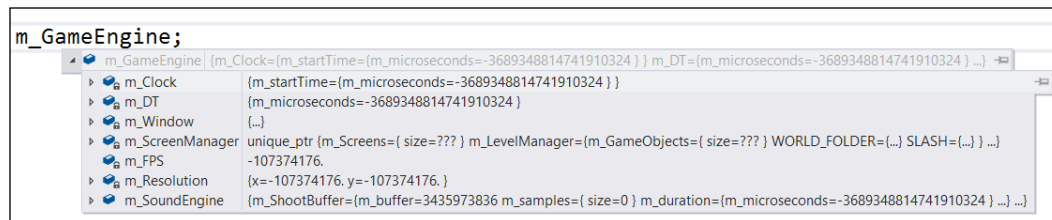


Notice that there is a red circle next to the line of code. This is a breakpoint. When you run the code, execution will pause at this point and we will have some interesting options available to us.

Run the game in the usual way. When execution pauses, an arrow indicates the current line of execution, as shown in the following screenshot:



If you hover the mouse over the `m_GameEngine` text and then click the arrow (the top-left corner in the following screenshot), you will get a preview of all the member variables and their values in the `m_GameEngine` instance:



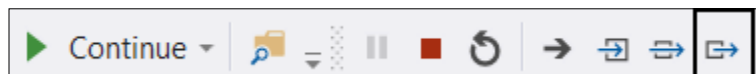
Let's progress through the code. In the main menu, look for the following set of icons:



If you click the arrow icon highlighted in the previous screenshot, it will move to the next line of code. This arrow icon is the **Step into** button. The next line of code will be the top of the `GameEngine` constructor function. You can keep clicking the **Step into** button and examine the value of any of the variables at any stage.

If you click into the initialization of `m_Resolution`, then you will see the code jumps into the `Vector2i` class provided by SFML. Keep clicking to see the code flow progress through all the steps that make up our game.

If you want to skip to the next function, you can click the **Step out** button, as shown in the following screenshot:



Follow the flow of execution for as long as it interests you. When you are done, simply click the **Stop** button, as shown in the following screenshot:



Alternatively, if you want to run the game without stepping through the code, you can click the **Continue** button shown in the following screenshot. Note, however, that if the breakpoint is placed inside a loop, it will stop each time the flow of execution reaches the breakpoint:



If you want to examine the flow of code from a different starting point and don't want to have to click through every line or function from the start, then all you need to do is set a different breakpoint.

You can delete a breakpoint by stopping debugging (with the **Stop** button), right-clicking the red circle, and selecting **Delete Breakpoint**.

You could then begin stepping through the game loop by setting a breakpoint at the first line of code in the `update` function of `GameEngine.cpp`. You can put a breakpoint anywhere, so feel free to explore the flow of execution in individual components or anywhere else. One of the key parts of the code that is worth examining is the flow of execution in the `update` function of the `GameScreen` class. Why not try it?

While what we have just explored is useful and instructive, the real purpose of these facilities provided by Visual Studio is to debug our games. Whenever you get behavior that is not as you expect, just add a breakpoint to any likely lines that might be causing the problem, step through the execution, and observe the variable values.

Reusing the code to make a different game and building a design mode

On a few occasions, we have already discussed the possibility that this system we have coded can be reused to make a totally different game. I just thought it was worth giving this fact a full hearing.

The way that you would make a different game is as follows. I have already mentioned that you could code the appearance of game objects into new components that derive from the `GraphicsComponent` class and that you could code new behaviors into classes that derive from the `UpdateComponent` class.

Suppose you wanted a set of game objects that had overlapping behaviors; consider perhaps a 2D game where the enemy hunted the player and then shot at the player at a certain distance.

Maybe you could have an enemy type that got close to the player and fired a pistol at the player and an enemy type that took long distance shots at the player, like a sniper might.

You could code an `EnemyShooterUpdateComponent` class and an `EnemySniperUpdateComponent` class. You could get a shared pointer to the player transform component during the `start` function and code an abstract class (such as `BulletSpawner`) to trigger spawning shots at the player, and you would be done.

Consider, however, that both of these game objects would have code to take a shot and code to close in on the player. Then consider that, at some stage, you might want a "brawler" enemy who tries to punch the player.

The current system can also have multiple update components. You could then have a `ChasePlayerUpdateComponent` class which closes in on the player and separate update components to punch, shoot, or snipe the player. The punching/shooting/sniping component would enforce some values on the chasing component regarding when to stop and start chasing, and then the more specific component (punch, shoot, or snipe) would attack the player when prompted that the time was right.

As we've already mentioned, the ability to call the `update` function on multiple different update components is already built into the code, although it has never been tested. If you take a look at the `update` function in `GameObject.cpp`, you will see this code:

```
    for (int i = m_FirstUpdateComponentLocation; i <
        m_FirstUpdateComponentLocation +
        m_NumberUpdateComponents; i++)
    {
        ...
    }
```

In the preceding code, the `update` function would be called on as many update components that are present. You just need to code them and add them to specific game objects in the `level1.txt` file. Using this system, a game object can have as many update components as it needs, allowing you to encapsulate very specific behaviors and share them as needed around the required game objects.

When you want to create a pool of objects, like we did for the invaders and the bullets, you can be more efficient than we were in the `Space Invaders ++` project. For the purposes of showing you how to position objects in the game world, we added all the invaders and bullets individually. In a real project, you would simply design a type that represents a pool of bullets, perhaps a magazine of bullets, like so:

```
[NAME]magazine of bullets[-NAME]
```

You could do the same for a fleet of invaders:

```
[NAME] fleet of invaders [-NAME]
```

Then, you would code the factory to handle a magazine or a fleet, probably with a `for` loop, and the slightly cumbersome text file would be improved upon. And, of course, there is no limit to the number of different levels you can design across multiple text files. More likely names for these text files are `beach_level.txt` or `urban_level.txt`.

You might have wondered about the names of some of the classes, such as `PhysicsEnginePlayMode` or `GameObjectFactoryPlayMode`. This implies that `...PlayMode` is just one option for these classes.

The suggestion I am making here is that, even if you use the fleet/magazine strategy in your level design files, they could still become cumbersome and unwieldy as they grow. It would be much better if you could view the levels and edit them on-screen and then save changes back to the file.

You would certainly need new physics engine rules (detecting clicks and drags on objects), a new screen type (that didn't update each frame), and probably new classes for interpreting and building the objects from the text files. The point is, however, that the Entity-Component/screen/UI panel/input handling systems could remain unchanged.

There isn't even anything stopping you from devising some completely new component types, for example, a scrolling background object that detects which direction the player is moving and moves accordingly, or perhaps an interactive lift object that detects when the player is standing on it and then accepts input to move up and down. We could even have a door that opens and closes, or a teleport object that detects input when the player is touching it and loads a new level from another text file. The point here is that these are all game mechanics that can be easily integrated into the same system.

I could go on about these possibilities for much longer, but you would probably rather make your own game.

Summary

In this chapter, we finally completed the Space Invaders ++ game. We coded a way for game objects to request bullets to be spawned, learned how to receive input from a gamepad, and we put in the final logic of the game to bring it to life.

Perhaps the most important thing to take from this chapter, however, is how the toil of the last four chapters will help you get started on your next project.

There is one final chapter in this slightly chunky book, and it is a short and simple one, I promise.

23

Before You Go...

When you first opened this big doorstop of a book, the back page probably seemed like a long way off. But it wasn't too tough, I hope.

The point is you are here now and, hopefully, you have a good insight into how to build games using C++.

The point of this chapter is to congratulate you on a fine achievement but also to point out that this page probably shouldn't be the end of your journey. If, like me, you get a bit of a buzz whenever you make a new game feature come to life, then you probably want to learn more.

It might surprise you to hear that, even after all these hundreds of pages, we have only dipped our toes into C++. Even the topics we did cover could be covered in more depth and there are numerous – some quite significant – topics that we haven't even mentioned. With this in mind, let's take a look at what might be next.

If you absolutely must have a formal qualification, then the only way to proceed is with a formal education. This, of course, is expensive and time-consuming, and I can't really help any further.

On the other hand, if you want to learn on the job, perhaps while starting work on a game you will eventually release, then what follows is a discussion of what you might like to do next.

Possibly the toughest decision we face with each project is how to structure our code. In my opinion, the absolute best source of information on how to structure your C++ game code is <http://gameprogrammingpatterns.com/>. Some of the discussion is around concepts that aren't covered in this book, but much of it will be completely accessible. If you understand classes, encapsulation, pure virtual functions, and singletons, dive into this website.

I have already pointed out the SFML website throughout this book. In case you haven't visited it yet, please take a look at it: <http://www.sfml-dev.org/>.

When you come across C++ topics you don't understand (or have never even heard of), the most concise and organized C++ tutorials can be found at <http://www.cplusplus.com/doc/tutorial/>.

In addition to this, there are four more SFML books you might like to look into. They are all good books but vary greatly in who they are suitable for. Here is a list of the books in ascending order from most beginner focused to most technical:

- *SFML Essentials* by Milcho G. Milchev: <https://www.packtpub.com/game-development/sfml-essentials>
- *SFML Blueprints* by Maxime Barbier: <https://www.packtpub.com/game-development/sfml-blueprints>
- *SFML Game Development By Example* by Raimondas Papius: <https://www.packtpub.com/game-development/sfml-game-development-example>
- *SFML Game Development* by Jan Haller, Henrik Vogelius Hansson, and Artur Moreira: <https://www.packtpub.com/game-development/sfml-game-development>

You also might like to consider adding life-like 2D physics to your game. SFML works perfectly with the Box2d physics engine. This URL is for the official website: <http://box2d.org/>. The following URL takes you to probably the best guide to using it with C++: <http://www.iforce2d.net/>.

Lastly, I am going to shamelessly plug my own website for beginner game programmers: <http://gamecodeschool.com>.

Thanks!

Most importantly, thanks very much for buying this book and keep making games!

Other Books You May Enjoy

If you enjoyed this book, you may be interested in these other books by Packt:

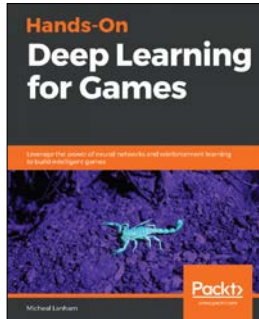


C++ Game Development By Example

Siddharth Shekar

ISBN: 978-1-78953-530-3

- Understand shaders and how to write a basic vertex and fragment shader
- Build a Visual Studio project and add SFML to it
- Discover how to create sprite animations and a game character class
- Add sound effects and background music to your game
- Grasp how to integrate Vulkan into Visual Studio
- Create shaders and convert them to the SPIR-V binary format



Hands-On Deep Learning for Games

Micheal Lanham

ISBN: 978-1-78899-407-1

- Learn the foundations of neural networks and deep learning.
- Use advanced neural network architectures in applications to create music, textures, self driving cars and chatbots.
- Understand the basics of reinforcement and DRL and how to apply it to solve a variety of problems.
- Working with Unity ML-Agents toolkit and how to install, setup and run the kit.
- Understand core concepts of DRL and the differences between discrete and continuous action environments.
- Use several advanced forms of learning in various scenarios from developing agents to testing games.

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

Symbols

<< operator 86

A

abstract class 401

addComponent function 630-632

algorithms

using 62

arithmetic operators 52, 53

array notation 107

arrays

about 79, 106, 268

declaring 107

elements, initializing 107, 108

using 108, 109

ASCII art

reference link 38

assets

adding, to project 197

exploring 196, 197

assignment operators 52, 53

attenuation 466

auto keyword 274

B

Bat class

Bat.cpp, coding 169-171

Bat.h, coding 166-168

coding 166

constructor function 168

using 172-176

bee

buzzing 56

drawing 60, 61

moving 71

preparing 57, 58

setting up 71-74

block 29

BlueprintObjectParser class

coding 648, 652

Bob

Controlling, by updating input

function 421, 422

drawing 427-430

frame, updating 424-426

game engine, updating 420

Instance, adding by updating Engine.h 420

spawning 423, 424

Bob class

Bob.cpp, coding 418-420

Bob.h, coding 417, 418

building 414

branches

drawing 126

growing 122, 123

moving 127-129

preparing 123, 124

sprites, updating 124-126

break keyword

using 104

bugs 45

Bullet class

Bullet header file, coding 302-305

Bullet source file, coding 305-310

coding 302

bullets

spawning 668

bullets fly

bullet array 311

bullet frame, drawing 316, 317

- bullet frame, updating 315
- bullet, shooting 314, 315
- control variables 311
- creating 310
- gun, reloading 311-313
- include directive, adding to Bullet class 310
- BulletSpawner class**
 - coding 668
- bullets, spawning**
 - BulletSpawner class, coding 668
 - GameScreen.h, updating 669-671
- BulletUpdateComponent class**
 - coding 604-608
- button 526**
- C**
- C++**
 - about 5
 - random numbers, generating 62, 63
 - reference link 54
- calculations**
 - using 62
- C++ assertions**
 - about 521
 - using 538, 539
- casting pointers 521**
- C++ else**
 - about 65
 - reader challenge 67, 68
 - used, for making decision 63
 - using 65, 66
- child class 399**
- C++ if**
 - about 65
 - reader challenge 67, 68
 - used, for making decision 63
 - using 65
- class**
 - about 156, 158
 - coding, for pickups 321
 - Pickup class function definition,
 - coding 325-330
 - Pickup header file, coding 321-324
- class enumerations 111, 112**
- classes 32, 33**
- class function**
 - defining 161-163
- class instance**
 - using 163
- clouds**
 - adding 56
 - drawing 60, 61
 - moving 71
 - preparing 58-60
 - setting up 75-79
- ColliderComponent class**
 - coding 592, 593
- collider components**
 - coding 591
- collision detection**
 - about 334, 462, 463
 - example 335-339
 - handling 455
- compiler program 5**
- Component base class**
 - coding 590, 591
- components**
 - code, preparing 590
- concatenation 85**
- concrete class 401**
- configuration errors 44**
- constructor function 168**
- C++ references**
 - about 236-239
 - summarizing 239
- crosshair**
 - adding to player 317-320
- C++ smart pointers**
 - about 534
 - shared pointers 534, 535
 - unique pointers 536, 537
- C++ Standard Library 31**
- C++ Strings**
 - about 84
 - declaring 84
 - manipulating 85, 86
 - value, assigning 85
- C-style code comments 38**
- C-style comment 27**
- C++ tutorials**
 - URL 700

C++ variables

- about 47, 48
- arithmetic operators 52, 53
- assignment operators 52, 53
- constant, initializing 50
- constants, declaring 50
- declaring 49
- declaring, in one step 50
- initializing 49, 50
- initializing, in one step 50
- manipulating 52
- results, obtaining with expressions 53-56
- tips, reference link 50
- types 48
- user-defined types 49
- user-defined types, declaring 51
- user-defined types, initializing 51

D

dangling pointer 266

dead zone

- creating 676

decision

- making, with C++ else 63
- making, with C++ if 63
- making, with logical operators 63, 64
- making, with switch 109-111

design mode

- building, by reusing code 695-697

design patterns 157, 521

design patterns, Space Invaders ++ project

- about 525
- button 526-528
- composition, versus inheritance 530, 531
- entity-component pattern 528
- Factory pattern 532, 533
- InputHandler 526, 527
- screen 526, 527
- UIPanel 526, 527

detectCollisions function

- coding 455-461

development environment

- Linux 7
- Mac 7
- project, creating 12-16
- project properties, configuring 16-18

- setting up 7

- SFML, setting up 10-12

- Visual Studio 2019 Community edition,
installing 8, 9

directive 31

double buffering 39

Drawable class

- exploring 495-497

draw 38

draw function 629

draw function definition

- coding 391, 393

dropDownAndReverse function 615

E

elements

- initializing, of array 107, 108

emitter 466

encapsulation 156

engine

- updating 451-455

Engine class

- about 393

- ParticleSystem object, adding to 503

- shaders, adding to 515

Engine class constructor definition

- coding 385-387

Engine.cpp

- coding 385

- draw function definition, coding 391-393

- Engine class constructor definition,

- coding 385-387

- input function definition, coding 388, 389

- run function definition, coding 387, 388

- update function definition, coding 390, 391

Engine.h

- coding 381-384

entity-component pattern

- diverse object types, managing issue 528

- generic GameObject, using 529, 530

errors

- compiling 44

- configuring 44

- handling 43

- linking 44

execution 28

expressions
using 53-56

F

factory classes
structure 638, 639
Factory pattern 532, 533
filters 540
final getter functions 635
for loops 105, 106
fragment shader
about 512
coding 513, 514
frame rate problem 68, 69
Freesound
URL 22
free store 264
function body 118
function calls 29
function gotcha 120, 121
function names 117
function parameters 118
function prototypes 119, 120
function returns 29
function return types 114-117
functions
about 28, 121
organizing 120
working with 113, 114

G

games
building 1
code clearer, creating with comments 27
coding 27
executing 30, 35
main function 28
Pong 2
presentation 28
restarting 360
Space Invaders 4, 5
syntax 28
Thomas 4
Timberman 2
values, returning from function 29, 30
Zombie Arena 3

game background
background sprite, double buffering 42
drawing 39
executing 43
Sprite preparing, Texture used 40, 41

game engine
SoundManager, adding 475
updating, for Bob usage 420
updating, for Thomas usage 420

game loop
about 35
C-style code comments 37
draw 38
executing 39
input 38
key press, detecting 38
repeat 38
scene section, clearing 39
scene section, drawing 39
update 38
while loop 37

GameObjectBlueprint class
coding 642-646

GameObject class
about 628
coding 622-628

GameObjectFactoryPlayMode class
coding 654-658

GameObjectSharer class
coding 658, 659

GameScreen.h
updating 669-671

game screen-related classes
coding, for select screen 576

game screen-related classes, select screen
coding 576
GameInputHandler class, coding 580, 581
GameOverInputHandler class,
coding 583, 584
GameOverUIPanel class, coding 584-586
GameScreen class, coding 576-579
GameUIPanel class, coding 581-583

GetComponentByTypeAndSpecificType
function 633, 634

getEncompassingRectCollider function 634
getEncompassingRectColliderTag
function 635

- getFirstUpdateComponent** function 635
- getGraphicsComponent** function 630
- getter** function 632
- getTransformComponent** function 630
- global scope** 123
- GraphicsComponent** class
 - coding 596, 597
- graphics components**
 - coding 596
- Graphics Library Shading**
 - Language (GLSL) 374, 511, 512
- graphics processing unit (GPU)** 40

H

- header file** 16, 31
- Heads Up Display (HUD)**
 - home screen, drawing 348-351
 - implementing 87-93
 - level-up screens, drawing 348-351
- heap** 264
- high score**
 - loading 353-355
 - saving 353-355
- horde**
 - adding, to game 291-297
 - creating, with **Zombie** class 287-291
- HUD class**
 - HUD.cpp file, coding 483, 484
 - HUD.h, coding 482, 483
 - implementing 482
 - using 486-489
- HUD frame**
 - updating 345-347
- HUD objects**
 - adding 341-345

I

- infinite loop** 104
- inheritance** 156, 157
 - about 397
 - class, extending 398-400
- inheriting, from **Drawable****
 - alternatives 497, 498
- initializeBulletSpawner** function 616
- input** 38
- inputFile** 354

- input function**
 - updating, for controlling Bob 421, 422
 - updating, for controlling Thomas 421, 422
- input function definition**
 - coding 388, 389
- InputHandler** 526, 527
- input/output (i/o)** 353
- instance** 156
- int** 28
- integer** 28
- integrated development environment (IDE)** 6
- internal coordinates** 24-27
- InvaderUpdateComponent** class
 - coding 609-613
- I/O file**
 - structure 638, 639
- isMovingRight** function 616

K

- key press**
 - detecting 38
- Komika Poster font**
 - download link 23

L

- leveling up** 357-360
- LevelManager** class
 - building 438, 439
 - coding 659-663
- LevelManager.cpp** file
 - coding 441-447
- LevelManager.h**
 - coding 439-441
- levels**
 - designing 434-438
- linker** 5
- Linux**
 - about 7
 - reference link 7
- listener** 466
- loadLevel** function
 - coding 448-451
- local coordinates** 25
- local variable** 121
- logical AND** 64

logical operators
 used, for making decisions 63, 64
logical OR 64
loops 102

M

Mac
 about 7
 reference link 7
main function
 coding 172-176, 394, 395
map
 about 270
 data, adding 271
 data, removing 272
 data, searching 271
 declaring 271
 iterating, through key-value pairs 273
 keys, checking 272
 looping, through key-value pairs 273
 size, checking 272
matrices 512
mechanics 19
member variables 160
Microsoft Visual Studio 6
multi-line comment 27

N

namespace 34
namespace sf
 using 33, 34
non-player characters (NPC) 32

O

object
 about 32, 33
 debugging 640-642
object-oriented programming (OOP)
 about 6, 32, 33, 155, 156, 376
 and Zombie Arena project 197, 198
 class 158
 encapsulation 156
 inheritance 157
 need for 157, 158
 polymorphism 157

ObjectTags class
 coding 646, 647
OpenGL ES 511
Open Graphics Library (OpenGL) 511
operator overloading
 reference link 86
origin 26
overhead 535

P

parent class 399
Particle class
 coding 493
Particle.cpp file
 coding 494, 495
Particle.h
 coding 493, 494
particle system
 building 491, 492
 drawing 508-510
 starting 506-508
ParticleSystem
 initializing 504, 505
ParticleSystem class
 coding 495
ParticleSystem.cpp file
 coding 500-503
particle system each frame
 updating 506
ParticleSystem.h
 coding 498-500
ParticleSystem object
 adding, to Engine class 503
 using 503
PhysicsEnginePlayMode class
 coding 678-681, 685-687
Pickup class
 using 330-333
Pickup class function definition
 coding 325-330
Pickup header file
 coding 321-324
PlayableCharacter class
 building 403
 PlayableCharacter.cpp, coding 409-414
 PlayableCharacter.h, coding 404-408

PlayableCharacter instances

updating 422

player death

handling 145, 146

player's input

axe, animating 142-144

chopped logs, animating 142-144

chopping key presses, detecting 137-141

handling 135

keyboard key released, detecting 141, 142

new game, set up handling 136, 137

player's input, Space Invaders ++ project

gamepad, using 675-677

handling 671-675

player's sprite

drawing 133, 134

preparing 131-133

PlayerUpdateComponent class

coding 616-621

play function

about 360

shooting sound, creating 361, 362

sound effects, adding on player reload 361

sound, playing on health pickup 363

sound, playing on player hit 362, 363

splat sound, creating on zombie

shot 364, 365

PlayModeObjectLoader class

coding 652, 653

pointers

about 121, 238, 257, 258, 268

declaring 260

declaring, to an object 267

dereferencing 262, 263

initializing 261

passing, to functions 266

reinitializing 262

summary 269

syntax 259

used, for addressing memory 264, 265

using, to an object 267

polymorphism 156, 157, 400, 401

Pong

about 2

project, creating 164, 165

reference link 2, 159

Pong Bat

class function, declaring 159-161

class function, defining 161-163

class instance, using 163

class variable, declaring 159-161

theory 159

Pong game

Ball class, coding 179-183

Ball class, using 183, 184

collection, detecting 185-187

collection, scoring 185-187

running 188

populateEmitters function

coding 477, 478

preprocessing 31

preprocessor 5, 31

primitives 512

primitive types

line 242

point 242

quad 242

triangle 242

private specifier 400

programmable pipeline 512

project

assets 195

assets, adding to 197

creating 12-16, 193-195

project assets

about 22

adding 23

exploring 23, 24

outsourcing 22

sound FX, creating 22

project assets, Thomas Was Late game

assets, adding to project 375

game level design 373

GLSL shaders 374

graphical assets 374

sound assets 374, 375

project properties

configuring 16-18

protected specifier 400

public specifier 400

pure virtual function 402

Q

quad sets 242

R

randomly generated scrolling background

creating 245-251

using 251-254

random numbers

about 62

generating, in C++ 62, 63

rectangle intersection 334

RectColliderComponent class

coding 593-595

references 121, 236

repeat 38

Roboto Light font

download link 375

run function definition

coding 387, 388

S

scope 120

scope resolution operator 162

screen 24-27, 526

Screen class

Button class, coding 554-556

coding 554

dependents, coding 554

InputHandler class, coding 560-565

Screen class, coding 565-568

UIPanel class, coding 556-559

ScreenManager class

updating 663, 664

ScreenManager.h file

commented-out code,

uncommenting 664, 665

ScreenManagerRemoteControl class

updating 663, 664

select screen

derived classes, coding 569

SelectInputHandler class, coding 571-573

SelectScreen class, coding 569-571

SelectUIPanel class, coding 573-575

setter function 632

Simple Fast Media Library (SFML)

about 6, 7

features 31, 32

Font class 86

RenderWindow 34, 35

setting up 10-12

Text class 86

used, for handling spatialization 466-468

used, for opening window 30, 31

VideoMode 34, 35

SFML Blueprints

URL 700

SFML DLL files 10

SFML Essentials

URL 700

SFML frame rate solution 69, 70

SFML Game Development

URL 700

SFML sound

code, adding 148-151

play function, using 148

SFML sound effects

using 147

SFML vertex arrays

about 240-242

building 242-244

using, for 244

SFML View class

reference link 351

SFML website

URL 700

shader program 512

shaders

about 374, 511

adding, to Engine class 515

drawing 516

loading 515

updating 516

shared pointers 534, 535

signature 113

singleton 274

smart pointers

about 521

casting 537, 538

sound effects

preparing 355-357

- sound emitters**
 - populating 476, 477
- sound FX**
 - adding 147
- SoundManager**
 - adding, to game engine 475
- SoundManager class**
 - building 469
 - SoundManager.cpp file, coding 471
 - SoundManager.h, coding 469, 471
- SoundManager.cpp file**
 - constructor, coding 471, 472
 - playFire function, coding 473, 474
 - SoundManager functions, coding 474
- sounds**
 - playing 479-482
- Space Invaders 4, 5**
- Space Invaders ++ project**
 - about 522-524
 - background file, download link 539
 - BitmapStore class, coding 550-552
 - bullets, spawning 668
 - code files, organizing with filters 540, 541
 - code, reusing 695-697
 - creating 539, 687-692
 - debugging 692-695
 - DevelopState file, adding 541
 - errors, handling 553
 - executing 586, 587
 - flow of execution 692-695
 - GameEngine class, coding 542-544
 - need for 524, 525
 - PhysicsEnginePlayMode class, coding 678
 - player's input, handling 671
 - play screen 522
 - reference link 522
 - ScreenManager class, coding 547-550
 - ScreenManagerRemoteControl class, coding 552
 - select screen 522
 - SoundEngine class, coding 545, 547
 - SpaceInvaders ++.cpp file, coding 541
- spatialization**
 - about 465, 466
 - handling, SFML used 466-468
- Sprite 25**
- sprite sheet 193, 240, 241**

- stack 264**
- StandardGraphicsComponent class**
 - coding 598-600
- Standard Template Library (STL)**
 - about 257, 269, 270
 - summary 274
- start function 633**
- static function 231, 257**
- STL container types**
 - about 270
 - list 270
 - map 270
 - reference link 270
 - set 270
 - vector 270
- Strategy pattern 531**
- sub-class 399**
- super-class 399**
- switch**
 - used, for making decisions 109-111
- syntax error 29**

T

- tearing 39**
- template 403**
- text objects**
 - adding 341-345
- texture coordinates 241**
- TextureHolder**
 - using 278
- TextureHolder class**
 - about 274
 - reusing 378-381
 - textures, modifying for background 297
 - textures, modifying for Player 298, 299
 - using, for textures 297
- TextureHolder function definitions**
 - coding 276-278
- TextureHolder header file**
 - coding 275, 276
- Thomas**
 - about 4
 - controlling, by updating input function 421, 422
 - drawing 427-430
 - frame, updating 424-426

- game engine, updating 420
- instance, adding by updating Engine.h 420
- reference link 4
- spawning 423, 424
- Thomas class**
 - building 414
 - Thomas.cpp, coding 415-417
 - Thomas.h, coding 415
- Thomas Was Late code**
 - structuring 376-378
- Thomas Was Late game**
 - about 368
 - engine, building 378
 - Engine class 393
 - Engine.cpp, coding 385
 - Engine.h, coding 381-384
 - features 368-371
 - project assets 373
 - project, creating 372, 373
 - TextureHolder class, reusing 378-381
- Timber!! game**
 - improving 151, 152
 - pausing 81-84
 - restarting 81-84
- Timberman**
 - about 2
 - features 20
 - planning 18-21
 - reference link 2
- time-bar**
 - adding 93-99
- timing**
 - considerations 68
- TransformComponent class**
 - coding 600-602
- tree**
 - adding 56
 - drawing 60, 61
 - preparing 56, 57
- type 28**

U

- UIPanel 526, 527**
- unique pointers 536**
- UpdateComponent class**
 - coding 603, 604
- update components**
 - coding 603
- update function**
 - about 614, 615, 628, 629
 - updating 422
- update function definition**
 - coding 390, 391
- Upwork**
 - URL 22
- user-defined types**
 - about 49
 - declaring 51
 - initializing 51

V

- variables 35**
- vertex 241**
- vertex shader**
 - about 512
 - coding 514
- virtual function 401-403**
- Visual C++ 2017 11**
- Visual Studio 2019 Community edition**
 - installing 8-10

W

- while loops**
 - about 37, 102-104
 - breaking out 104, 105
- wild pointer 266**
- window**
 - opening, SFML used 30, 31
- WorldState.h file**
 - adding 568

Z

- Zombie Arena background**
 - building, from tiles 242
- Zombie Arena game**
 - about 3
 - planning 192, 193
 - reference link 3
 - starting 192, 193
- Zombie Arena game engine**
 - starting 218-222

Zombie Arena project

- code files, managing 222, 223
- game camera, controlling with
 SFML View 215-217
- main game loop, coding 224-233
- OOP 197, 198
- player, building 198, 199
- Player class function definitions,
 coding 206-215
- Player class header file,
 coding 199-205

Zombie class

- used, for creating horde 287-291

Zombie.cpp file

- coding 282-286

Zombie.h file

- coding 279-281

zombies

- horde, building 278