

HEALTH:   

SCORE: 654243



Community Experience Distilled

Haxe Game Development Essentials

Create games on multiple platforms from a single codebase using Haxe and the HaxeFlixel engine

Jeremy McCurdy

[PACKT] open source*
PUBLISHING community experience distilled

Haxe Game Development Essentials

Create games on multiple platforms from a single codebase using Haxe and the HaxeFlixel engine

Jeremy McCurdy



BIRMINGHAM - MUMBAI

Haxe Game Development Essentials

Copyright © 2015 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, and its dealers and distributors will be held liable for any damages caused or alleged to be 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.

First published: November 2015

Production reference: 1241115

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

ISBN 978-1-78528-978-1

www.packtpub.com

Credits

Author

Jeremy McCurdy

Project Coordinator

Sanchita Mandal

Reviewers

Romuald Halasz

Nathan McDonald

Robin Poirier

Proofreader

Safis Editing

Indexer

Hemangini Bari

Acquisition Editors

Harsha Bharwani

Usha Iyer

Production Coordinator

Shantanu N. Zagade

Content Development Editor

Samantha Gonsalves

Cover Work

Shantanu N. Zagade

Technical Editor

Edwin Moses

Copy Editor

Dipti Mankame

About the Author

Jeremy McCurdy is a game developer who has been making games using ActionScript, C#, and Haxe for over four years. He has developed games targeted at iOS, Android, Windows, OS X, Flash, and HTML5. He has worked on games that have had millions of gameplay sessions, and has built games for many major North American television networks.

He is the games technical lead at REDspace, an award-winning interactive studio that has worked for some of the world's largest brands. They are located in Nova Scotia, Canada, and have been building awesome experiences for 15 years.

Acknowledgements

I would like to thank my family for constantly pushing me to reach my goals. Thanks go to my parents, Gaile Jodrey and Blaine McCurdy; sisters, Darlene and Gabrielle Therrien; grandparents, Roy and Marg Jodrey; and cousin, Tammy Lucy Prall.

I would also like to thank my college instructor, Sean Morrow, from the Nova Scotia Community College's Truro Campus. He set me on the path to being a programmer twice, taught me the fundamentals of programming, and made learning incredibly fun.

I'd also like to thank my coworkers who pushed me from being a scared little intern to a proper technical lead – Shane Kerr, Thomas Gillis, Armen Abrahamyan, Mike McGraw, Robin Poirier, and Nathan McDonald.

Lastly, I'd like to thank Mike Johnston and Wes Gould, who have given me awesome career opportunities at REDspace and made a workplace that I feel proud to be a part of.

I wouldn't be here writing this now if it weren't for all of these people. They have all made my life as a developer possible, and I honestly can't thank them enough.

About the Reviewers

Romuald Halasz began his adventure in the world of programming nearly 20 years ago. His passion for writing code was sparked by computer games. He was amazed by them and wanted to learn how they are made in order to bring his own ideas to life. It's been a wonderful experience that involved learning and great amounts of fun.

Throughout his journey, he has explored different segments of the development spectrum, such as web development, enterprise software development, and more recently, mobile development.

He has worked in different companies, from start-ups to multinationals. He has also tried his hand at freelancing. In general, he sees coding as a way of life. It is a journey of learning and giving life to projects, which start with one idea. There is also the satisfaction of people using the products that you have worked on.

Nathan McDonald is a game developer who is currently working at REDspace, where he helps build games using languages and tools such as Haxe, Flambe, and Unity.

I would like to thank Jeremy McCurdy for the opportunity to help him with this book and Robin Poirier for all the help that he's given.

Robin Poirier is a game developer from Nova Scotia, Canada. He works mainly in Haxe/Flambe and Unity3D. He has developed games that deploy for iOS, Android, Flash, and HTML5. He currently works at REDspace as part of their games division.

Specials thanks to Jeremy McCurdy for this opportunity to be part of this book and Nathan McDonald for being a good rubber duck.

www.PacktPub.com

Support files, eBooks, discount offers, and more

For support files and downloads related to your book, please visit www.PacktPub.com.

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

At www.PacktPub.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.



<https://www2.packtpub.com/books/subscription/packtlib>

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can search, access, and read Packt's entire library of books.

Why subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print, and bookmark content
- On demand and accessible via a web browser

Free access for Packt account holders

If you have an account with Packt at www.PacktPub.com, you can use this to access PacktLib today and view 9 entirely free books. Simply use your login credentials for immediate access.

Table of Contents

Preface	ix
Chapter 1: Getting Started	1
What's Haxe?	1
OpenFL	2
HaxeFlixel	2
Installation	3
Installing Haxe	3
OpenFL	3
Setting up an IDE	3
Installing HaxeFlixel	4
Setting up target platforms	5
Flash	5
Android	6
iOS	7
Desktop	8
Hello World	9
Creating a project	9
Running the project	9
Copying assets	10
Making changes	10
Summary	11
Chapter 2: Building a New Game	13
The Haxe syntax	13
Data types and structures	14
Operators	14
Access modifiers	14

Variables	14
Functions	17
The for loops	18
Classes	18
Setting up a new HaxeFlixel project	20
Creating the new project	20
The anatomy of our project	20
Changing settings	21
Adding assets	22
Adding a background and basic UI	22
Making enemies	24
Creating the enemy class	24
Adding variables	25
Setting up the display	25
Creating the spawn timer	26
Instantiating enemies and adding interaction	27
Adding new imports	27
Adding variables	27
Creating instances and adding mouse events	27
Creating the end screen	29
Creating the end screen class	29
Adding variables	30
Adding the background and score text	30
Restarting the level	32
Implementing the game timer	33
Importing the end screen	33
Adding variables	33
Creating the level timer	33
Creating the timer callback function	34
Summary	36
Chapter 3: Dealing with Menus and Screen Flow	37
Adding new assets	38
Changing the default state	38
Building the splash screen	39
Adding imports	40
Creating variables	40
Adding sprites	41
Adding instructions	42

Starting the animation sequence	43
Creating the next stage of the sequence	44
Finishing the sequence	45
Navigating back to the game	45
Updating the level end screen	47
Adding imports	47
Adding variables	47
Housekeeping	48
Adding the group and setting up the window	49
Adding scores	50
Adding the play again button	51
Showing the best score sprite	51
Creating the animation sequence	52
Creating the new game HUD	53
Creating the GameHUD class	54
Adding imports	54
Adding variables	54
Laying things out	55
Changing the score	56
Adding the new game HUD	56
Adding imports	56
Adding variables	57
Housekeeping	57
Adding the HUD and enemy group	58
Summary	59
Chapter 4: Delving into Animations and Gameplay	61
Object pooling	61
Project updates	62
Adding new assets	62
Updating the project settings	63
Creating the explosion effects	63
Creating the class	63
Adding imports	64
Creating variables	64
Setting up Texture Packer data	65
Adding sprites	66
Creating a particle emitter	67
Adding everything to the group	68

Starting the explosion sequence	68
Showing the explosion bulb	69
Showing smoke and completing the animation	70
Preparing for object pooling	72
Making muzzle flashes	72
Creating the class	72
Adding imports	73
Creating variables	73
Adding sprites	74
Playing the flash	74
Showing the rings	75
Updating enemies	75
Adding imports	76
Adding variables	76
Adding animation	77
Resetting the spawn position	78
Updating the onSpawn function	79
Adding death functionality	79
Creating projectiles	80
Creating the class	80
Adding imports	80
Creating variables	80
Displaying the sprite and setting the velocity	81
Making the player character	81
Creating the class	81
Adding imports	82
Creating variables	82
Adding animation	83
Making the projectile pool	84
Creating the muzzle flash	85
Starting the shoot timer	85
Shooting projectiles	85
Handling player input	86
Building movement functions	88
Handling player death	89
Setting health on the HUD	89
Changing hearts	89
Adding everything to PlayState	90
Adding imports	90

Adding variables	90
Cleaning things up	91
Adding the scrolling backdrop	92
Creating and adding the player	92
Creating an explosion object pool	92
Killing enemies	93
Updating the score	94
Detecting collisions	94
Handling collisions between projectiles and enemies	95
Handling collisions between the player and enemies	95
Summary	97
Chapter 5: Adding Sound	99
Project updates	99
Adding new assets	100
Updating the project settings	100
Creating sound constants	103
Creating the class	103
Creating sound ID constants	103
Creating volume constants	104
Building the sound manager class	105
Creating the class	105
Adding imports	105
Implementing the singleton pattern	105
Adding music	107
Creating playback functions	107
Calling playback functions	107
Adding menu sounds	108
Creating playback functions	108
Calling playback functions	109
Adding gameplay sounds	110
Creating variables	111
Creating functions	112
Calling playback functions	112
Summary	113
Chapter 6: Working with Configuration Files	115
JSON versus XML	116
Project updates	116
Adding new assets	117

Understanding the JSON configuration file	117
A simple example	117
Looking at the JSON format	118
Building the enemy spawner	119
Creating the enemy spawner class	119
Adding imports	120
Adding variables	120
Parsing JSON data	121
Populating the pool	121
Spawning enemies	122
Adding movement patterns to enemies	123
Adding imports	123
Adding variables	123
Removing old functionality	124
Creating the startPattern function	124
Updating onSpawn and resetSpawn	125
Updating PlayState	126
Removing old functionality	126
Using EnemySpawner	127
Understanding the XML file	128
Looking at the file's contents	128
Pulling string data from XML	129
Building the Strings class	129
Handling XML data	129
Building the StringIDs class	131
Replacing strings	132
Updating MenuState	132
Updating GameHUD	132
Updating LevelEndScreen	133
Summary	134
Chapter 7: Deploying to Multiple Platforms	135
Building for Flash	135
Deploying to the Web	136
Building for Windows	136
Building for OS X	136
Building for Android	137
Building for iOS	139
Xcode setup	139
Setting up a certificate	140
Setting up an App ID	143

Registering a device	145
Creating a provisioning profile	146
Running the game on iOS	148
Summary	149
Chapter 8: What's Next?	151
Expanding the game	151
Level-based gameplay	151
Power-ups and collectibles	152
More enemies and bosses	152
Exploring advanced features	153
Gamepads	153
Physics	153
Tilemaps	153
Continuing on with learning resources	154
Haxe	154
OpenFl	154
HaxeFlixel	154
Community resources	155
Summary	156
Index	157

Preface

Developing games that can reach a wide audience can often be a serious challenge. A big part of the problem is figuring out how to make a game that will work on a wide range of hardware and operating systems. This is where Haxe comes in.

Over the course of this book, we'll look at getting started with Haxe and the HaxeFlixel game engine, build a side-scrolling shooter game that covers the core features you need to know, and prepare the game for deployment to multiple platforms.

After completing this book, you will have the skills you need to start producing your own cross-platform Haxe-driven games!

What this book covers

Chapter 1, Getting Started, explains setting up the Haxe and HaxeFlixel development environment and doing a quick Hello World example to ensure that everything is working.

Chapter 2, Building a New Game, looks at the fundamentals of the Haxe language and building the core foundation of our game.

Chapter 3, Dealing with Menus and Screen Flow, helps to set up our game's UI, add simple UI animations, and control the flow between screens.

Chapter 4, Delving into Animations and Gameplay, explains adding sprites and scripted animations and adding the bulk of our gameplay functionality.

Chapter 5, Adding Sound, delves into the aspects of adding music and sound effects to make our animations and gameplay truly engaging.

Chapter 6, Working with Configuration Files, talks of using JSON and XML configuration files to make our game expandable and editing the text far easier.

Chapter 7, Deploying to Multiple Platforms, shows how to take our completed game and build it to work on desktop, web, and mobile platforms.

Chapter 8, What's Next?, looks at game features to expand on and at resources to continue learning.

What you need for this book

The introductory chapter of the book will walk you through getting the tools you need to develop with. However, you will need a computer to develop with. You can develop games with Haxe on Apple OS X, Microsoft Windows, or Linux operating systems.

In order to develop a game for a particular operating system, you will need to have access to a machine with it installed to create playable builds. For example, you need to be using OS X to develop a game targeted at OS X.

For mobile platforms, you can use any operating system to develop Android applications, but you will need a Mac running OS X to develop iOS applications. You will also need an iOS device in order to develop for iOS; this can be an iPhone, iPad, or iPod Touch.

Who this book is for

This book is for game developers with some experience in programming games on one or more platforms already. If you want to leverage your game development experience on one platform to develop for multiple platforms and to get up and run quickly, this book is for you.

In addition, if you want to learn how using Haxe, OpenFL, and HaxeFlixel can make multiplatform development faster and easier, this book will show you.

Having prior experience with a language similar to Haxe, such as ActionScript or JavaScript, will help, but isn't required.

Conventions

In this book, you will find a number of text styles that distinguish between different kinds of information. Here are some examples of these styles and an explanation of their meaning.

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows:
"Open `MenuState.hx` and navigate inside the `Create` function."

A block of code is set as follows:

```
override public function create():Void
{
    super.create();
}
```

Any command-line input or output is written as follows:

```
lime test flash
```

New terms and **important words** are shown in bold. Words that you see on the screen, for example, in menus or dialog boxes, appear in the text like this: "If you're using FlashDevelop, you can create a class by right-clicking on the source folder and going to **Add | New Class**."

 Warnings or important notes appear in a box like this.

 Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book – what you liked or disliked. Reader feedback is important for us as it helps us develop titles that you will really get the most out of.

To send us general feedback, simply e-mail feedback@packtpub.com, and mention the book's title in the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide at www.packtpub.com/authors.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files from your account at <http://www.packtpub.com> for all the Packt Publishing books you have purchased. If you purchased this book elsewhere, you can visit <http://www.packtpub.com/support> and register to have the files e-mailed directly to you.

Downloading the color images of this book

We also provide you with a PDF file that has color images of the screenshots/diagrams used in this book. The color images will help you better understand the changes in the output. You can download this file from http://www.packtpub.com/sites/default/files/downloads/Haxe_GameDevelopment_Essentials_ColorImages.pdf.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books – maybe a mistake in the text or the code – we would be grateful if you could report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting <http://www.packtpub.com/submit-errata>, selecting your book, clicking on the **Errata Submission Form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded to our website or added to any list of existing errata under the Errata section of that title.

To view the previously submitted errata, go to <https://www.packtpub.com/books/content/support> and enter the name of the book in the search field. The required information will appear under the **Errata** section.

Piracy

Piracy of copyrighted material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works in any form on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at copyright@packtpub.com with a link to the suspected pirated material.

We appreciate your help in protecting our authors and our ability to bring you valuable content.

Questions

If you have a problem with any aspect of this book, you can contact us at questions@packtpub.com, and we will do our best to address the problem.

1

Getting Started

In this chapter, we're going to see what Haxe is, how it will enable you to create awesome multiplatform games, and then we'll install all of the tools we'll need to develop with. After setting things up, we'll work with a quick Hello World example to make sure that everything works and we're ready to start making games!

Here's a rundown of the sections in this chapter:

- What's Haxe?
- Installation
- Setting up target platforms
- Hello World
- Summary

What's Haxe?

Haxe is an open source programming toolkit that is composed of a language, compiler, and command-line interface. It is heavily inspired by ActionScript 3, with some tweaks here and there to add functionality. It also draws some inspiration from C#. It's so similar that at a glance, Haxe code could easily be confused for ActionScript. This doesn't mean that you have to know ActionScript to learn Haxe, but it does help. Knowing an object-oriented language, such as C# or Java, will also give you a leg up.

Haxe compiles to several different platforms, allowing you to have one codebase that will work in browsers, desktop operating systems, and on mobile devices.

In case you were wondering about the pronunciation, Haxe is pronounced *Hex*, though many people pronounce it *Hacks*.

OpenFL

OpenFL is a framework built on top of Haxe; it adds additional API functionality and provides build tools to streamline your workflow. Most of the API changes it introduces are meant to help mimic the ActionScript 3 API. For example, it adds support for Flash's Stage 3D mode, text formatting, and bitmap data.

OpenFL also uses the Lime library, which is a library that helps to ensure consistent cross-platform support so that builds for different platforms don't behave radically differently. It covers things such as cross-platform audio, rendering, and asset management.

Some of our build commands will be executed using Lime, and that's pretty much all we'll be directly using it for.

HaxeFlixel

HaxeFlixel is a game engine that's built on top of OpenFL. It was originally based on the ActionScript 3 game engine named Flixel, and has since branched off on its own. It's well documented and has a solid community, making it an ideal engine for people who want to learn how to make games with Haxe.

HaxeFlixel supports the following features:

- Efficient and high-performance rendering
- Collision detection
- Particles
- Tilemaps
- Bitmap fonts
- Pathfinding
- Object pooling
- Tweening
- GPU acceleration

That's just a handful of the features of the engine; it's very robust. HaxeFlixel will help us start building high-quality games much faster than doing everything from scratch.

Installation

Now that you have a good understanding of what we'll be using, let's start installing what we'll need. Haxe, OpenFL, and HaxeFlixel are all designed to be easy to set up, so this is going to be pretty straightforward.

Installing Haxe

Go to <http://haxe.org/download/> and download the installer for your operating system. Run the installer; be sure to install both Haxe and Neko when given the option.

This will install the Haxe framework, which includes Haxelib, a useful tool for installing Haxe frameworks and libraries. You will typically use this to install all Haxe libraries, which will make the installation of OpenFL and HaxeFlixel incredibly straightforward.

OpenFL

To install OpenFL, open a command window (or terminal on OS X and Linux). Next, run the following commands:

```
haxelib install openfl  
haxelib -notimeout run openfl setup
```

These two commands will download, install, and set up the OpenFL framework and all of the libraries it needs. The `notimeout` flag prevents the setup from timing out before it completes. This is useful because the OpenFL setup will download quite a number of files, and it can time out if your internet connection is a little slow.

When asked if you want to install OpenFL, press Y. On Mac and Linux, you may be asked to enter your password. After this, OpenFL will be installed and you'll be good to go!

Setting up an IDE

Before installing HaxeFlixel, it's important to install a code-editing tool, because you will be prompted to choose one during the HaxeFlixel setup. Your operating system will be a factor in choosing the best tool for you.

Windows users are able to use FlashDevelop, which is a fully featured IDE that will work with Haxe out of the box. The best part of using FlashDevelop is that you can debug your games by targeting Flash and using the Flash debug player. This will allow you to put in breakpoints, see console logs, and inspect variables. You can download the latest version by going to <http://flashdevelop.org>. If you are using Windows, I highly recommend using it.

For Mac and Linux users, the best tool available is Sublime Text 2. It's a solid code-editing tool, though it lacks support for breakpoints, console logs, and variable inspection. You can download the latest version by going to <http://sublimetext.com>. You can also use Sublime Text on Windows if you prefer it.

To make Sublime Text a more effective Haxe editing tool, you should install the Haxe plugin. The easiest way to install it is using package control, which you can install using the steps located at <https://packagecontrol.io/installation#st2>.

After installing package control, use *Ctrl + Shift + P* (Windows or Linux) or *cmd + Shift + P* (Mac) to bring up the command palette. Start typing in *package* and then navigate down to **Install Package**. You will then see a list of available packages to install. Type in *Haxe* to find the package named Haxe. Install that and you're good to go!

Installing HaxeFlixel

To install HaxeFlixel, run these commands:

```
haxelib install flixel
haxelib install flixel-tools
haxelib run flixel-tools setup
```

You will then be prompted several times during the setup. Here's what you should do:

1. When a command appears asking if you want to set up Flixel, press *Y*.
2. When asked if you would like to download *flixel-demos* and *flixel-templates*, press *Y*.
3. When prompted to enter an author name, either type in your name or leave the field blank. It will automatically use that name when creating project templates.

4. When asked to choose your default IDE, enter the number corresponding to your development tool of choice.
5. Choose either *Y* or *N* when asked if you want to automatically open created templates and demos with your IDE of choice. This will just immediately open your project in your IDE instead of requiring you to navigate your filesystem.
6. On Macs, if you chose to have templates and demos automatically open, you will be asked whether or not you want to run a `symlink` command. Press *Y*.

If you ever need to change these settings, you can run the Flixel tools setup command again.

Setting up target platforms

To be able to deploy games on multiple platforms, a little extra work is involved. You're going to be using Flash, primarily, to develop your game, but we'll walk through what you need to download and set up for all of the major platforms that HaxeFlixel supports. We'll be going into detail about how to build each of these platforms in *Chapter 7, Deploying to Multiple Platforms*.

Flash

Haxe and OpenFL will build to Flash out of the box, so there's not really anything special you have to do in order to deploy to Flash. That said, to debug Flash content you should download the Flash debug player.

On Windows, you will be able to place breakpoints and inspect variables; on Mac, you will just get popups with stack traces when you get errors, which is still very useful. Unfortunately, Linux users will be unable to debug Flash in the browser, but you will still be able to use the content debugger application.

You can download the debug versions of the Flash player at <https://www.adobe.com/support/flashplayer/downloads.html>.



To debug Flash without a browser, download the projector content debugger application for your operating system. If you intend to primarily use the projector application, make sure that you set it up as the default application to run when loading SWF files.

To debug Flash in a browser, you will need to download different plugins based on the browser you intend to use. Here are some quick guidelines you can follow:

- For Internet Explorer, use the ActiveX plugin
- For Firefox and Safari, use the NPAPI plugin
- For Chrome, use the PPAPI plugin

Android

OpenFL makes getting the tools you need to deploy to Android much easier than it typically is. Normally, this would involve downloading some tools and installing a bunch of SDKs. OpenFL has a tool that will guide you through the process as fast as possible.

Before using OpenFL's tool, ensure that you have the Java runtime installed. You can download it from <https://java.com/en/download/>. After installing the runtime, you can begin the Android setup process.

For Windows users, run this command to get started:

```
openfl setup android
```

For Mac and Linux users, run the command using `sudo`, like so:

```
sudo openfl setup android
```

After using `sudo`, you will have to enter your password to continue.

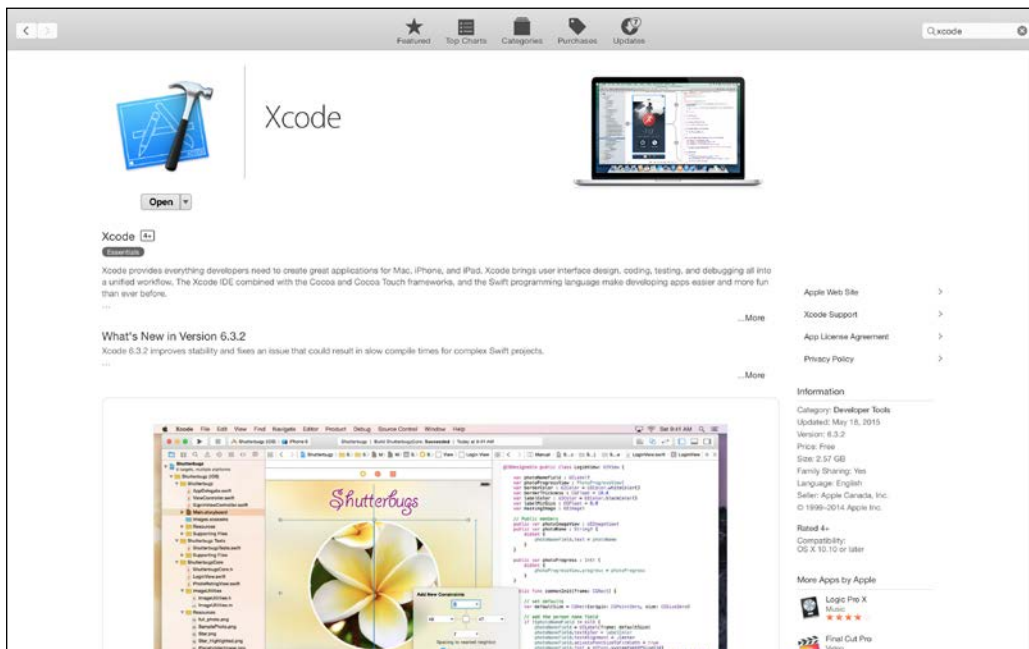
You will then be prompted several times during the setup. Here's what you should do:

- Press `Y` when asked if you want to download and install the Android SDK.
- When asked to enter an output directory for the Android SDK, either enter a new path or just hit *Enter* to choose the default directory.
- Press `Y` when asked whether or not you want to download and install the Android NDK.
- When asked to enter an output directory for the Android NDK, either enter in a new path or just hit *Enter* to choose the default directory.
- Press `Y` when asked whether or not you want to download and install Apache Ant.
- When asked to enter an output directory for Apache Ant, either enter in a new path or just hit *Enter* to choose the default directory.

- Press **Y** when asked whether or not you want to download and install the Java JDK. Press **Y** when asked to be taken to the download site of the JDK. Follow the download links on the site to download the JDK installer, and run and install it before continuing.
- When asked to enter the path to the Android SDK, either you can enter a custom path type in it and continue, or hit *Enter* to continue with the default path.
- When asked to enter the path to the Java JDK, type in the path you used to install the JDK earlier.

iOS

The setup process for iOS development is much simpler. Of course, in order to develop iOS games, you will need to use Max OS X and be a member of Apple's iOS developer program. You can sign up for the program by going here: <https://developer.apple.com/programs/ios/>. You will also need to install the latest version of Xcode, Apple's development tool, to develop iOS and OS X applications. You can download it by going to the Mac App store and searching for Xcode.



After Xcode has been installed, run this command in the terminal:

```
sudo lime setup ios
```

After using `sudo`, you will have to enter your password to continue.

When asked to download and install Apple Xcode, press *N*. This is only for users who haven't installed Xcode yet. It's easier to just install Xcode in advance like we did.

Desktop

HaxeFlixel can also be built to desktop targets. When you want to build your games for a particular desktop operating system, you will need to use that operating system to make the build. For example, you can't build a Mac game on Windows or vice versa.

When building for Windows, you will need to install a version of Visual Studio, Microsoft's application development tool that can build C++ and can target Windows 32-bit. If you don't already have a copy of Visual Studio that meets the requirements, you can download Visual Studio C++ 2010 Express at <http://go.microsoft.com/?linkid=9709949>.

After installing Visual Studio, you will also need to run this command:

```
openfl setup windows
```

When building for OS X, you will need to install Xcode. If you didn't install Xcode as part of the iOS setup, please refer to the section on how to download and install it.

Once Xcode is installed, run this command:

```
openfl setup mac
```

When asked to download and install Apple Xcode, press *N*.

When building for Linux, there are no special setup steps. You will just need to be running Linux when making builds.



It is worth noting that some more exotic Linux distributions may run into issues. If you're running into issues with developing for Linux, I suggest asking questions on OpenFL's community forum, located at <http://community.openfl.org/>.

Hello World

Now that we have everything we need installed, we should make sure that Haxe, OpenFL, and HaxeFlixel are all up and running. To do this, we'll create a new project using the command line, run it, and then add an image to the screen to ensure that we can make changes.

Creating a project

To create a project, open up a command or terminal window and navigate to the folder in which you want your game's folder to be created.

After doing this, enter this command:

```
flixel tpl -n "HelloWorld"
```

This will create a new folder using the standard HaxeFlixel project template. It will also create project files for your code-editing tool of choice, allowing for quick integration out of the box.



If HaxeFlixel doesn't automatically open your project in your code-editing tool, open it in the tool manually. For FlashDevelop users, double-click on the `.hxproj` file in the project's folder. For Sublime Text users, drag your project's folder into an empty Sublime window.

A number of Haxe classes with the file extension `.hx` will be created. These are the base files you need to get a basic HaxeFlixel project up and running. We'll be going over the project's structure throughout the book, so it's not important to know what everything means right now.

Running the project

Now that we have a project set up, we can run it to see what we get out of the box.

If you're running FlashDevelop, select **Debug** from the drop-down list next to the play button and then select **flash** from the drop-down list next to that. After this, hit the play button to run your project.

For Sublime Text users, you can press `Ctrl + Enter` to run the project using the Haxe package that was installed earlier.

To create a build using the command line, navigate to your project's folder and enter this command:

```
lime test flash
```

The project should build and then open with your operating system's default application for running `.swf` files. When the `.swf` file loads, you'll briefly see the HaxeFlixel logo and then you'll be presented with a blank screen.

Copying assets

We're going to need the image that we want to display, so let's get that now. In the assets provided for this chapter, there is an image named `HelloWorld.png`. Copy it to the `images` folder under `assets` in your project's directory.

Making changes

Now that we have the image copied to the template project, let's add the traditional *Hello World* image to the screen.

Open `MenuState.hx` and navigate to the `Create` function. It will look like this:

```
override public function create():Void
{
    super.create();
}
```

After `super.create()` ;, add the following lines:

```
var helloWorldText = new FlxSprite();
helloWorldText.loadGraphic(AssetPaths.HelloWorld__png);
add(helloWorldText);
```

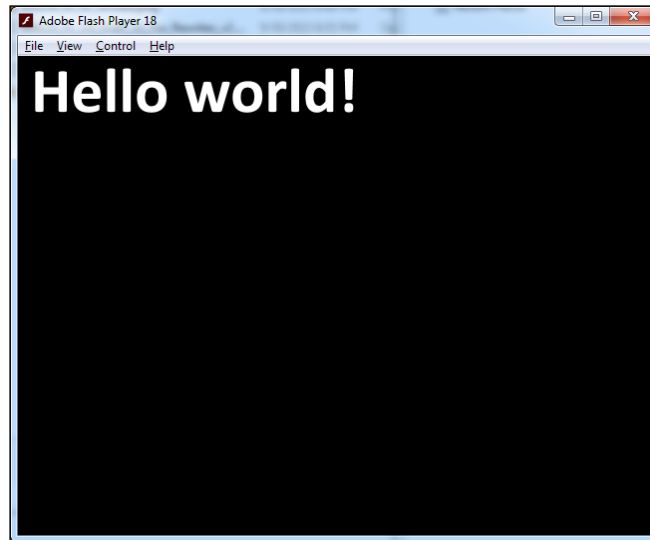
This will create a new variable of the `FlxSprite` class. It will then load in the image we just copied over, and then finally add the image to the screen. We'll go into more detail about how to use the `FlxSprite` class and work with loaded assets later, but that's a quick rundown of what's happening.

All that's left to do is to run the project to see your changes! In `FlashDevelop`, you just need to press the play button. For `Sublime Text` users, press `Ctrl + Enter`.

To create a build using the command line, enter the `test` command again:

```
lime test flash
```

That's it! You should see your SWF load and display the text **Hello world!**. If something along the way didn't work, take some time to review the chapter to see where things went wrong.



Summary

In this chapter, you learned what Haxe, OpenFL, and HaxeFlixel are, and how they will streamline multiplatform game development. You learned how to install and set up Haxe, OpenFL, HaxeFlixel, and a code-editing tool. You then learned how to set up the tools you'll need for cross-platform development. Finally, you learned how to build and run your very first Haxe project. These skills will enable you to create and set up projects as you see fit.

You now have everything you need to dive head first into Haxe game development! In the next chapter, we're going to look at the Haxe language and start building the foundation of your first HaxeFlixel game.

2

Building a New Game

In this chapter, we're going to start building our game. We'll start by exploring some of the fundamental Haxe syntaxes that we'll be using. After this, we'll make a new HaxeFlixel project, make a few changes, and work on getting some simple gameplay in place that will serve as the base to build on.

Here's a rundown of the sections in this chapter:

- The Haxe syntax
- Setting up a new HaxeFlixel project
- Adding the background and basic UI
- Making enemies
- Instantiating enemies and adding interaction
- Creating the end screen
- Restarting the level
- Implementing the game timer

The Haxe syntax

Haxe is a very easy language to learn, so we'll be go over the basic syntax now and build on what we know as the book progresses.

At its core, Haxe contains all of the same things you find in other languages: variables, data types, data structures, access modifiers, functions (also known as methods), and expressions.

Haxe is also an object-oriented language, so you will see classes, inheritance, and polymorphism.

Data types and structures

The basic data types in Haxe are `Bool`, `Float`, `Int`, and so on.

Those data types are considered basic because they are simple pieces of data that aren't represented by a class. There are many data types and structures that are important and are represented by a class. They are explained here.

Operators

Haxe has a number of operators that are used to perform math equations, concatenation, and comparison. Let's go over the most common ones.

Access modifiers

Variables, classes, and functions can all be given an access modifier to determine how they can be accessed in your codebase. These are the access modifiers you will use most often.

Variables

Here's an example of a variable:

```
var score:Int = 0;
```

Variables are defined using the `var` keyword, followed by the names you choose to give them. The colon allows you to specify a data type (in this case, it's an integer). Using the equals symbol allows you to assign a value, and the semicolon ends the line of code.

Semicolons are required to end every line of code. You will get errors if you forget to add them.



It is best practice to provide a data type; it increases the readability of your code and makes debugging much easier because you don't have to guess what data type you're using.

It's also worth stating that you aren't required to provide a data type to variables in most cases. Haxe supports type inference at compile time, which means that the compiler will do its best to determine what data type you intend to use based on the value assigned to it. For example, if a variable was set to 5 without a data type provided, the inferred type given to the variable would be `Int`.

Here's a list of data types with their descriptions:

Type	Description
Bool	This represents Boolean values and can be either <code>true</code> or <code>false</code> .
Float	This represents a double precision floating point number. This is a value that allows positive and negative numbers with decimal points (for example, 3.14 or -3.14).
Int	This represents an integer, a whole number that can be either positive or negative. You can assign an <code>Int</code> value to a <code>Float</code> variable, but you can't assign a <code>Float</code> value to an <code>Int</code> variable without doing a type conversion first.
String	This represents text data and is used whenever you need to use text. String values are wrapped in quotation marks (for example, "Hello world").
Array	This represents a collection of elements of a single type. This is used to store objects and data. Arrays in Haxe have a dynamic length, which means that you can add and remove elements whenever you want and the length of the array will change accordingly. Arrays are useful for looping through data.
Map	This is a key-value pair collection. It is used when you want a collection of data or objects that allows you to quickly access specific objects without looping through everything. For example, you could find an object by name using a <code>String</code> type.
Dynamic	This is a type of variable that can be assigned any type of value or object. This is different than using type inference because you can assign a value of a different type as needed to a variable of the <code>Dynamic</code> type. This type is useful when you're reading data of an unknown type or you want a single variable that can temporarily hang onto any type of object. This type should be used wisely. It prevents certain compiler optimizations from working and can result in code that's extremely difficult to debug.
Void	This isn't really an object type. It indicates that you don't intend to return a value. It's generally used in functions that don't return anything.
null	This is what you will see when a variable that isn't a basic type has no value. You won't be able to do anything with a <code>null</code> variable until it has a value assigned to it.

Here's a list of operators with their descriptions:

Operator	Description
+	This is the addition operator, used to add numbers together and to add parts of strings together using concatenation.
-	This is the subtraction operator, used to subtract numbers.

Operator	Description
*	This is the multiplication operator, used to multiply numbers.
++	This is the increment operator, used to increase a number by 1.
--	This is the decrement operator, used to reduce a number by 1.
=	This is the assignment operator, used to assign a value to a variable.
==	This is the equivalence operator, used to check whether two values are the same.
<	This is the less than operator, used to check whether or not one value is less than another.
>	This is the greater than operator, used to check whether or not one value is greater than another.
<=	This is the less than or equal to operator, used to check whether or not one value is less than or equal to another.
>=	This is the greater than or equal to operator, used to check if one value is greater than or equal to another.

Here's a list of access modifiers with their descriptions:

Access modifier	Description
<code>private</code>	This can only be accessed from inside the current class and its subclasses. This is the default access modifier if you don't specify one, but it's generally best to explicitly state the modifier.
<code>public</code>	This can be accessed from anywhere you can access the class instance from.
<code>static</code>	This is used to define a variable or function as something that exists at the class level. Static properties and functions don't require an instance of a class to be used, and creating multiple instances of a class will not create multiple instances of static properties or functions.
<code>inline</code>	This defines a variable or function as a fixed value at compile time. It is most often used as an equivalent to a constant property that you may see in other languages. It can also be used to optimize function calls if the values you are using are defined in your code and aren't fed in dynamically. The function with the values you provided will actually get included as part of the compiled code for faster execution.

Functions

Functions in Haxe work in pretty much the same way that functions and methods work in other languages. A function definition looks like this:

```
private function processData(inputData:String, optionalData:Int
= -1):String{
    return inputData;
}
```

This function will simply take in a piece of string data and return it immediately. We start by using an access modifier to indicate that the function is private. We then use the keyword `function` to indicate that the block of code contains a function (similar to the way we use `var` to declare variables). After this, we provide a name for the function.

Inside the parentheses, we add arguments (also referred to as parameters). These are any pieces of data you wish to pass in when calling the function. You provide a name for the property, followed by the data type. Arguments passed into functions don't have to be typed, but for the sake of clarity and debugging, they should be typed.

You'll also note that the second argument in the function has `= -1` next to it. This indicates that the second value will default to a value of `-1` if no value is passed in for the second parameter. This lets you create robust functions that can have optional arguments.

After the arguments section, you will see that we specified a `String` data type, and this means that the function will return a string value. If your functions don't need to return a value, set the return type to `Void`.

The function's logic is wrapped in curly brackets (also named curly braces); all of the function's logic must be contained between the opening and closing brackets.

Finally, we return a value using the `return` keyword followed by the returned value. Functions that have a specified return type must return data of that type. If you set the return type to `Void`, then you do not have to use the `return` keyword.

One last thing to note about functions is that you cannot overload functions in Haxe. You must either use optional arguments or create multiple functions instead.

The for loops

The `for` loops in Haxe behave a little differently from the way they behave in most other languages. Every `for` loop uses iterators — objects designed to be iterated through in sequence. This basically means that Haxe `for` loops work much like `for-each` loops do in other languages.

A `for` loop looks like this:

```
for(i in 0...10)
{
    //Loop functionality goes here.
}
```

The loop starts with the `for` keyword, followed by opening and closing parenthesis that contain the iteration parameters. The `i` parameter is a variable that will store the current iteration value, in this case, an integer from 0 to 9. After this, there is the `in` keyword, which acts to separate the iteration variable from the iterator. Finally, we have `0...10`, which uses a special range operator that Haxe uses to iterate from one value to a higher value.



You use ellipsis (...) to separate the low and high values. It's worth noting that the maximum value that `i` can be is the high value minus 1, because the maximum value is converted to a zero-based numbering system. One other thing to note is that `for` loops in Haxe can't loop in reverse, but rather they must always increment.

Classes

Classes in Haxe work in the same way that they work in most object-oriented languages. Let's have a look at an example and walk through it:

```
package examplePackage;

import packageName.ClassName;

class ExampleClass extends ParentClass
{
    public static inline var CONSTANT_VALUE:Int = 1;

    public var publicVariable:String;
```

```
private var privateVariable:Int;

public function new()
{
    super();
}

override private function ParentClassFunction(
{
    super.ParentClassFunction(data);
}

}
```

To start, we define the package that our class resides in. This typically relates to the folder path to the class file from the root of the source directory. Using packages lets you have classes with the same name in a project, which is useful for things like code libraries. The `package` line is ended with a semicolon.

After the `import` statements, you define the classes you intend to use in variables and functions. If a class you want to use is in the same package as your class, you don't need to import it.

Next, there's the class definition. You use the keyword `class` followed by the name of your class. If you have a parent class that you're extending, use the keyword `extends` followed by the name of the class you're extending. Make sure to import the parent class if it isn't in the same package as the current class.

After defining the class, you wrap all variables and functions in a set of curly brackets.

Inside the class, you'll see that we have a couple of functions. The first is the `new` function. This is the constructor function for the class and is a special function that will get called when you make a new instance of your class. If you're extending a class and overriding its functionality, you must also call the `super` function; this will call the constructor of the parent class.

The second function uses the `override` keyword. This indicates that you intend to add or change functionality in a function that's defined in the parent class. Since it's a function that belongs to the parent, you need to call the function using the `super` object. This fires the parent's function and then anything done afterwards will be exclusive to the child class and any of its children.

Setting up a new HaxeFlixel project

Now that we've gone over the basics of the Haxe language, we'll start working on the foundation of our game. The game we're going to make is a side scrolling space shooter similar to games like R-Type or Gradius. It will be much simpler than those games, but by the end of this book you will have the skills you need to take the game to that level of complexity.

To keep the game fun, we'll add a goofy twist: you'll control a walrus with a raygun and a jetpack, and your enemies will be rocket-powered space cats. We'll name this game *Star Walrus*!

To start, we're going to keep it simple and have enemies that spawn in that you click on to increase your score until a timer runs out. This will help with learning the fundamentals.

Creating the new project

Like in our Hello World example, we'll start by making a new project using the command line. So, open up a command or terminal window, navigate to the folder that you want game's folder to be in, and type in this command:

```
flixel tpl -n "StarWalrus"
```

If your code-editing tool doesn't open automatically, navigate to your project's folder and open your project.

The anatomy of our project

A HaxeFlixel project is composed of many files by default. Let's quickly go over what we have to work with. Open up the source folder.

The first file to look at is `Main.hx`. This is the main class for the game. It handles initialization and gets HaxeFlixel up and running. You can change some of HaxeFlixel's base settings here, but you typically don't write a lot of code yourself in this class.

In the `setupGame` function, you'll note that a new `FlxGame` object is added to the stage. `FlxGame` is the class that gets the HaxeFlixel framework up and running.

Next, you'll see `MenuState.hx` and `PlayState.hx`. These two classes extend the `FlxState` class. You can think of these as the main screens of the game. The `FlxState` objects will be used to represent different screens of the game and will also be used for things like menus inside other `FlxState` objects.

After this, have a look at `Reg.hx`. This is a class meant for tracking game data and progress. It's an entirely optional class to use.

Then, we have `AssetPaths.hx`. This is a handy class that will keep track of all files your game can load so that you can access them without having to keep track of file paths. We won't be changing it, but we will be using it a lot.

The last file we'll look at is `Project.xml`. This file contains most of your project's settings. This will let you pull in Haxe libraries, set up paths for files, change compilation settings, and change settings specific to each platform you intend to target.

Before moving on, have a look at the `assets` and `export` folders. The `assets` folder will be used to store any art, sound, and data assets you need to load. The `export` folder will store builds of your game for each platform. The compiler will add a new folder for each platform as you make builds for them.

Changing settings

We'll get more in-depth with changing project settings over the course of the book, but for now, we'll keep it simple. To start, open up the `Project.xml` file.

In the `window settings` section, change the `width` property to 960 and the `height` property to 640. Here's what it should look like:

```
<!--These window settings apply to all targets-->
<window width="960" height="640" fps="60"
background="#000000" hardware="true" vsync="true" />
```

We're going to use the 960 x 640 screen dimensions as our base screen size because it's a good general-purpose screen size for web and mobile games. Don't worry about being locked into those dimensions; we can change them at any time and we'll look at screen scaling options in *Chapter 7, Deploying to Multiple Platforms*.

Next, open up `Main.hx` and make the following changes to the settings at the top of the class:

```
var gameWidth:Int = 960;
var gameHeight:Int = 640;
var initialState:Class<FlxState> = PlayState;
```

The changes in the `gameWidth` and `gameHeight` variables are pretty self-explanatory; it's the same change that we made in the `Project.xml` file. We also need to make those changes in the `Main` class to ensure that everything gets scaled correctly.

The `initialState` property is the `FlxState` class that will appear when we first run the game. We've changed it from `MenuState` to `PlayState` because we will be focusing on gameplay for the time being. In the next chapter, we will introduce screen flows and then use `MenuState`.

Adding assets

Next, we need to add the assets we're going to use to start building the game. Copy the image assets provided for this chapter—`enemy01.png` and `gameBackground.png`—into the `assets/images` folder.

Adding a background and basic UI

We're going to start by adding a background, and two text fields to display the score and game time to the player. Open up `PlayState.hx`, and we'll add some variables.

Inside the class, add the following variables:

```
private var background:FlxSprite;  
private var txtScore:FlxText;  
private var txtTime:FlxText;
```

The background is a `FlxSprite` object. It is used when you want to display an art asset or a shape drawn by `HaxeFlixel`. The `txtScore` and `txtTime` variables are `FlxText` objects, and as the name implies, they're used to display text.

After this, go inside the `create` function and add the following underneath `super.create()`:

```
background = new FlxSprite();  
background.loadGraphic(AssetPaths.gameBackground__png);  
add(background);
```

In those lines, we are creating a new `FlxSprite` instance in the `background` variable, loading our background image using the `AssetPaths` class, and then using the `add` function to add the background to `PlayState`. The `add` function is inherited from the `FlxState` class, and we'll be using it frequently.

It's also worth noting that the order in which we add things to `FlxState` is important. This order determines how things will layer visually. So, since we added the background first, it will appear underneath everything else.

Next, to add our two text fields to screen, add these lines:

```
txtScore = new FlxText(10, 10, 200, "Score: 0", 20);  
txtTime = new FlxText(10, 30, 200, "Time: 0", 20);  
add(txtScore);  
add(txtTime);
```

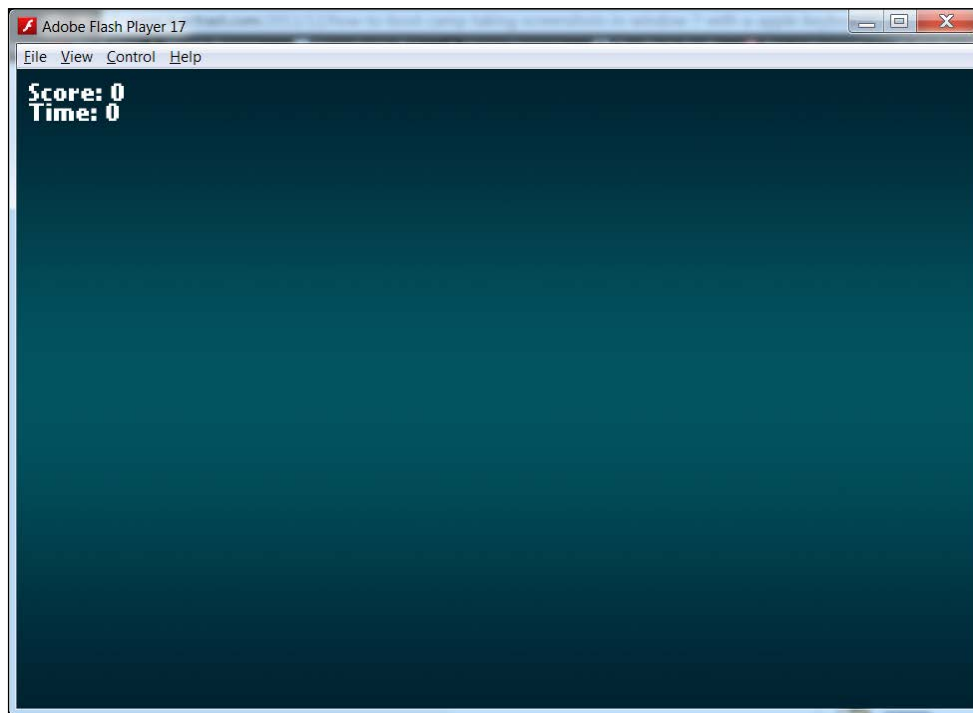
We start by creating instances of our two `FlxText` objects. The constructor for the `FlxText` object takes many arguments; the ones we're setting here are the x position, the y position, the field width, the initial starting text value, and the font size.

In this case, both fields will be placed in the top-left corner of the screen with a little bit of padding. They will have a field width of 200; they both have starting text values and a font size of 20.

After this, we add both to `PlayState`.

Now you can run the game to see what we have so far. You can hit the play button in `FlashDevelop` or use this command-line function:

```
lime test flash
```



Making enemies

Next, we'll start making our enemies, and then we'll add them to the screen. To do this, we'll make a new class that will handle displaying our enemies. They will appear at random places on the screen after random time intervals.

Creating the enemy class

If you're using FlashDevelop, you can create a class by right-clicking on the source folder and going to **Add | New Class**. Then, name it `Enemy` and set its base class to `FlxSprite`.

If you're using Sublime Text, you can make a new class by going to **File | New File** and saving the file under the source folder as `Enemy.hx`.

FlashDevelop will create part of the class for you. In Sublime Text, you'll have to add most of the class by hand. For FlashDevelop, remove `source` from the package section. We're not going to use packages for classes that sit directly under the source folder.

Your class should look like this:

```
package;

import flixel.util.FlxTimer;
import flixel.FlxSprite;
import AssetPaths;

class Enemy extends FlxSprite
{
    public function new()
    {
        super();
    }
}
```

Make note of the `import` statements and the fact that we added a call to `super()` in the constructor.

Adding variables

Next, we're going to add some variables:

```
private var spawnTimer:FlxTimer;  
private var maxSpawnTime:Float = 14;  
  
private var maxX:Int = 750;  
private var minX:Int = 0;  
private var maxY:Int = 500;  
private var minY:Int = 0;
```

The `spawnTimer` variable is a `FlxTimer` object that we will use to randomly delay the enemy appearing on screen. The `maxSpawnTime` variable will be the longest possible delay enemies can have before appearing.

The `maxX`, `minX`, `maxY`, and `minY` variables are used to determine the boundaries that enemies can be placed within. Feel free to play with these values to change how densely packed enemies will be.

Setting up the display

Now we're going to add the enemy art and position the enemy at a random position. In the constructor, add this code after the `super` call:

```
loadGraphic(AssetPaths.enemy01__png);  
var randomX:Int = (Math.floor(Math.random() * (maxX - minX + 1))  
+ minX);  
var randomY:Int = (Math.floor(Math.random() * (maxY - minY + 1))  
+ minY);  
this.setPosition(randomX, randomY);  
  
this.visible = false;
```

First, we'll load the `enemy01` image in the same way we loaded the background image earlier. It is a rocket-powered cat head. Don't think about it too hard.

Next, we're going to generate our random *x* and *y* positions. These lines may look daunting if you've never done this before, but here's how it breaks down:

1. Use the `random` function of Haxe's `Math` class; this will generate a random number between `0.0` and `1.0`.
2. Multiply that number using your maximum and minimum values to get a number within the range you need.

3. Wrap that number in the `floor` function of the `Math` class. This will round your number down to the nearest integer. If you're trying to get a random float value, you can leave this step out.

Finally, we'll set the enemy's `visible` property to `false`. This will hide the enemy until we want it to appear on screen. It's also worth noting that a `FlxSprite` object with its `visible` property set to `false` won't react to mouse clicks, so this is also an easy way to prevent players from interacting with enemies until we want them to.

Next, we'll use a timer to determine when to show the enemy.

Creating the spawn timer

This is the last step in setting up the enemy class. We'll run a timer that will last for a random duration and then set the `visible` property of our enemy to `true`.

After the previous block of code, add this:

```
var spawnTime:Float = Math.random() * (maxSpawnTime+ 1);
spawnTimer = new FlxTimer();
spawnTimer.start(spawnTime, onSpawn);
```

First, we'll generate a random number again. This one is a bit simpler. Since we only have a maximum number, we only have to use our `maxSpawnTime`. This will generate a number between 0 and our `maxSpawnTime` value (14). This is also a `Float` type instead of an `Int` type because we want enemies to appear at really varied times to keep the game interesting. This time, therefore, we don't need to round the number.

After this, we create a new `FlxTimer` object in our `spawnTimer` variable. Then, we call the `start` function on `spawnTimer`. The `start` function takes in many arguments; we passed in our `spawnTime` and `onSpawn`. The `spawnTime` variable is pretty self-explanatory, but `onSpawn` is a callback function that will trigger after the timer runs for the duration of `spawnTime`.

Finally, let's create the `onSpawn` function:

```
private function onSpawn(timer:FlxTimer):Void {
    this.visible = true;
}
```

This function is pretty simple. It takes a `FlxTimer` object as an argument; this is the timer that triggered the callback. Having this argument enables you to reuse callbacks across multiple `FlxTimer` objects. Inside the function, we set the `visible` property of the enemy to `true`, making it appear to players.

Instantiating enemies and adding interaction

Next, we'll go back to `PlayState` and create instances of our enemies, and then make enemies disappear when they're clicked. Finally, we'll then add to our score every time the player clicks on an enemy.

Adding new imports

Before adding functionality, we'll need to import some more classes we'll be using in `PlayState`:

```
import flixel.plugin.MouseEventManager;
import flixel.FlxObject;
import flixel.util.FlxTimer;
```

Adding variables

Next, we'll add the new variables we'll need to track:

```
private var numEnemies:Int = 20;
private var score:Int = 0;
private var enemyPointValue:Int = 155;
private var enemies:Array<Enemy>;
```

These variables are pretty straightforward: `numEnemies` is the number of enemies we'll spawn in the game, `score` is the player's current score, and `enemyPointValue` is the number of points that will be added to your score each time an enemy is clicked. Finally, `enemies` is an array that will store the `Enemy` objects we're going to make.

Creating instances and adding mouse events

In the `create` function, after the text fields are added to `PlayState`, add these lines:

```
enemies = new Array<Enemy>();
var enemy:Enemy;
for (i in 0...numEnemies) {
    enemy = new Enemy();
    add(enemy);

    enemies.push(enemy);
    MouseEventManager.add(enemy, onEnemyMouseDown);
}
```

First, we'll instantiate the array of the `Enemy` objects. Next, we'll create a temporary variable named `enemy` to store each enemy as it is created. After this, we'll run a `for` loop that goes from 0 to `numEnemies`. In this case, it will run 20 times.

Inside the `for` loop, we'll create a new enemy instance and add it to `PlayState`. Then, we'll push the created enemy into the array. Next, we'll use the `add` function of the `MouseEventManager` class. This is a static function that will check a `for` loop when the player clicks on an object. In this case, we pass in two arguments: our enemy instance, and `onEnemyMouseDown`, which is a callback function that will get called as soon as the player's mouse is clicked down on the enemy.

So, next let's add the `onEnemyMouseDown` callback:

```
private function onEnemyMouseDown(object:FlxObject):Void{
    object.visible = false;
    score += enemyPointValue;
    txtScore.text = "Score: " + score;
}
```

This function takes in an `FlxObject` as an argument, and this is the object that triggers the callback. In this case, that's the enemy that was clicked.

First, we'll hide the enemy by setting the object's `visible` property to `false`. This will prevent it from being interacted with again.

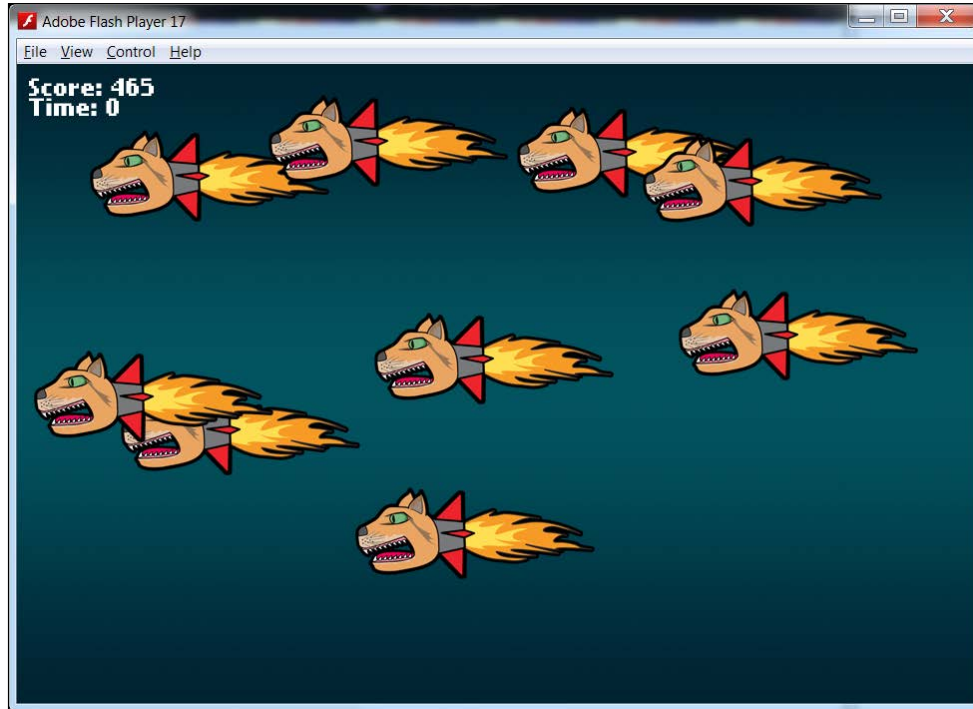
After this, we'll add our `enemyPointValue` to `score`. Finally, we'll update our `txtScore` field by setting its `text` property. Now we have some real gameplay going!

Finally, we need to clean up our mouse events so that they don't create a memory leak. To do this, go down to the `destroy` function and add the following code above the `super.destroy()` ; function call:

```
for(i in 0...enemies.length){
    MouseEventManager.remove(enemies[i]);
}
```

This will loop through all of the enemies in the `enemies` array and remove all callbacks on them when the state gets destroyed. It's extremely important to remove callbacks so that objects can be properly destroyed and the memory they're occupying released.

At this point, we can give the game a run and see how everything is working.



Creating the end screen

Now, let's make the screen that will appear when the game timer runs out. Once we have this made, we'll just need to set up the game timer, and we'll be done!

Creating the end screen class

First, let's make a folder under the source folder named `ui`. Since we'll eventually have a few user interface classes, we should organize them under a separate folder. This will put them in a different package, and these classes will need to be imported when used.

In the `ui` folder, create a class named `LevelEndScreen`. Previously, we made a class that extended `FlxSprite` and this time, we'll extend `FlxState`. This will allow us to create an object that can contain multiple visual elements instead of a single sprite.

Here's the initial class definition with its needed imports:

```
package source.ui;
import flixel.FlxSprite;
import flixel.util.FlxColor;
import flixel.FlxG;
import flixel.FlxState;
import flixel.text.FlxText;

class LevelEndScreen extends FlxState
{
    public function new(score:Int)
    {
        super();
    }
}
```

Note that we added a `score` argument to the constructor. This will be the game's final score at the end of the game.

Adding variables

Let's add the variables we'll need for this screen:

```
private var background:FlxSprite;
private var txtGameScore:FlxText;
```

So, we have a variable to store the screen's background and a `text` field to display our final score.

Adding the background and score text

In the constructor, after the `super()` call, we'll add the screen's background with the following code:

```
background = new FlxSprite();
background.makeGraphic(FlxG.stage.stageWidth,
    FlxG.stage.stageHeight, FlxColor.BLACK);
background.alpha = 0.5;
add(background);
```

We'll start by creating a `FlxSprite` object as we have before. This time, however, we'll use the `makeGraphic` function to create a solid rectangle. This function takes in three arguments: `width`, `height`, and `color`.

For the first two arguments, we use the `FlxG` class to get the `stageWidth` and `stageHeight` properties of `stage`. These are the dimensions of the game. In this case, those values are 960 and 640 because we changed the game's size to those during the setup of the project.

The color we passed in uses the `FlxColor` class, which just contains some predefined color values we can quickly make use of.

After the `makeGraphic` function, we set the background's `alpha` property to `0.5`. This gives the background 50% opacity and makes it semi-transparent. Doing this will act as an overlay, so players will still be able to see what was left on the screen in the game.

Next, we add the background to `LevelEndScreen`.

After this, let's add the `txtGameScore` field to the screen and set its text value:

```
txtGameScore = new FlxText(0, 0, 500, "Final score: " + score + "\nClick to play again", 40);
txtGameScore.setPosition(FlxG.stage.stageWidth / 2 - txtGameScore.width / 2, FlxG.stage.stageHeight / 2 - txtGameScore.height / 2);
add(txtGameScore);
```

First, we'll create a new `FlxText` object, as we have before. One thing to note is that the text value passed in contains `\n`. It is a special set of characters that will break the following text down to a new line. So, the `Click to play again` text will appear one line down from the final score.

Next, we set the position of the text field using the `setPosition` function. It works the same way as it does for `FlxSprite` objects. In this case, we take the `stageWidth` value, divide it by two and then subtract half the width of `txtGameScore`. We then do the same with `stageHeight` and the height of `txtGameScore`. What this will do is position the text in the middle of the screen.

Restarting the level

Next, we'll make it so that the level will restart when the player clicks anywhere on screen.

To do this, we'll have to add an update function:

```
override public function update():Void
{
    if (FlxG.mouse.justReleased)
    {
        onPlayAgain();
    }
    super.update();
}
```

This function will override the update function of the `FlxState` parent class, which is why we use the `override` keyword in this case.

Inside the function, we make an `if` statement to check if `FlxG.mouse.justReleased` is `true`. This is another way to handle mouse input in HaxeFlixel and we're doing it this way because we just want to know when the player clicks anywhere on screen, instead of on a particular thing.

Inside the `if` statement, we'll call an `onPlayAgain` function, which we'll make shortly. Following the `if` statement, call `super.update()` so that the functionality of the parent class will still execute.

To finish the class off, we just have to make the `onPlayAgain` function:

```
private function onPlayAgain():Void{
    FlxG.resetState();
}
```

In this function, we'll just call `FlxG.resetState()`. What this will do is reset the current top level `FlxState`, which is `PlayState` currently. The `FlxState` objects and all objects underneath `PlayState` will get removed and cleaned up too. It's a really handy feature of HaxeFlixel!

Implementing the game timer

We're on the home stretch! The last things we have to do are setting up the game timer and showing the level-end screen! Open up `PlayState` to make the last few changes.

Importing the end screen

Since the `LevelEndScreen` class is in a package, we'll need to import it to use it, so add this to your `import` section:

```
import source.ui.LevelEndScreen;
```

Adding variables

Next, we'll have to add new variables to handle the timer functionality:

```
private var levelTimer:FlxTimer;  
private var levelTime:Int = 15;  
private var ticks:Int = 0;
```

The `levelTimer` variable is a `FlxTimer` like the one we used when making enemies. The `levelTime` variable is the duration that the game will run for. The `ticks` variable is used to keep track of the number of seconds that have passed so that we know when to end the game. In other words, it keeps track of how many times the timer has *ticked*.

Creating the level timer

In the `create` function, after the section where enemies are added, add these lines:

```
levelTimer = new FlxTimer();  
levelTimer.start(1, onTimeComplete, levelTime);
```

To start, we create a new `FlxTimer` object, then we call the `start` function.

We're doing things a little differently from the `Enemy` class. In this case, we're setting the timer's duration to 1, indicating that we want it to fire after one second. The next argument is `onTimeComplete`, which is a callback that we'll create shortly.

The last argument is something we didn't use last time; it indicates the number of times the timer will repeat, so in this case, it will be using the `levelTime` variable.

What all of this means is that `onTimeComplete` will get called every second until `levelTime` has elapsed, and in this case, that happens after 15 seconds.

Creating the timer callback function

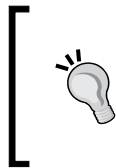
The last thing we need to write for this chapter is the `onTimeComplete` callback. Here it is:

```
private function onTimeComplete(timer:FlxTimer):Void {
    ticks++;
    txtTime.text = "Time: " + ticks;

    if(ticks >= levelTime){
        var levelEndScreen:LevelEndScreen = new
        LevelEndScreen(score);
        add(levelEndScreen);
    }
}
```

Like the last timer callback function we made, this function takes in a `FlxTimer` object as an argument.

Inside the function, we start by incrementing the `ticks` variable using the `++` operator. This increases the `ticks` variable by one. After this, we update `txtTime` using the `ticks` variable to show the player the time that's elapsed.

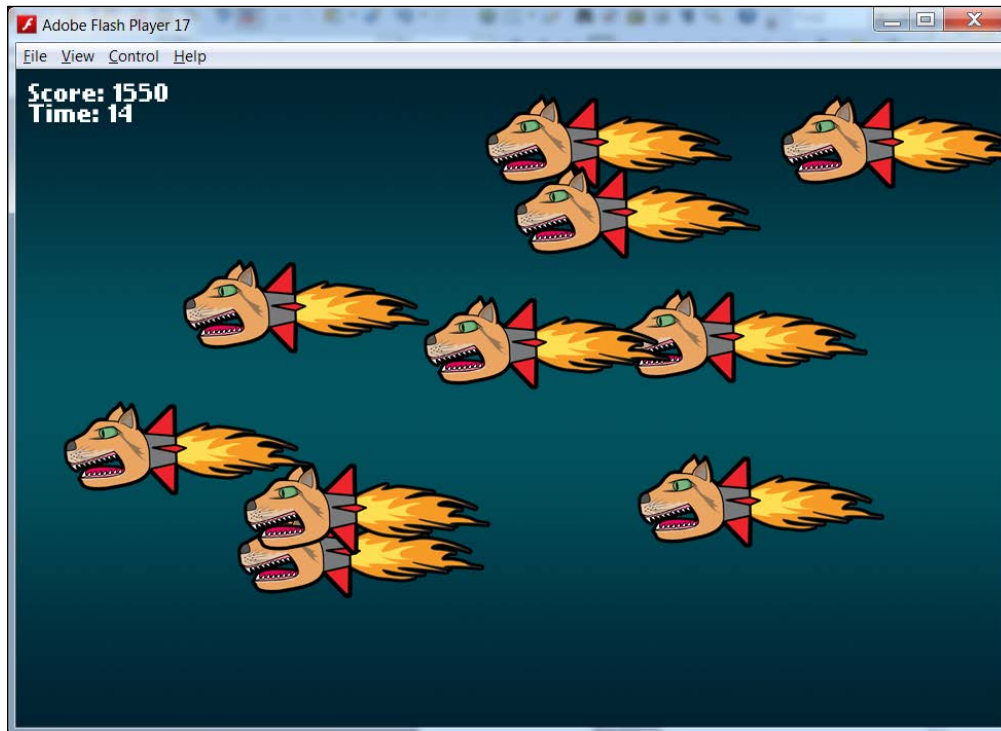


It's worth noting that we used a hardcoded text value here. It's generally considered good practice to make text configurable from a single location instead of placing it inline as we have here. In *Chapter 6, Working with Configuration Files*, we'll look at moving all of our game's text to external configuration files.

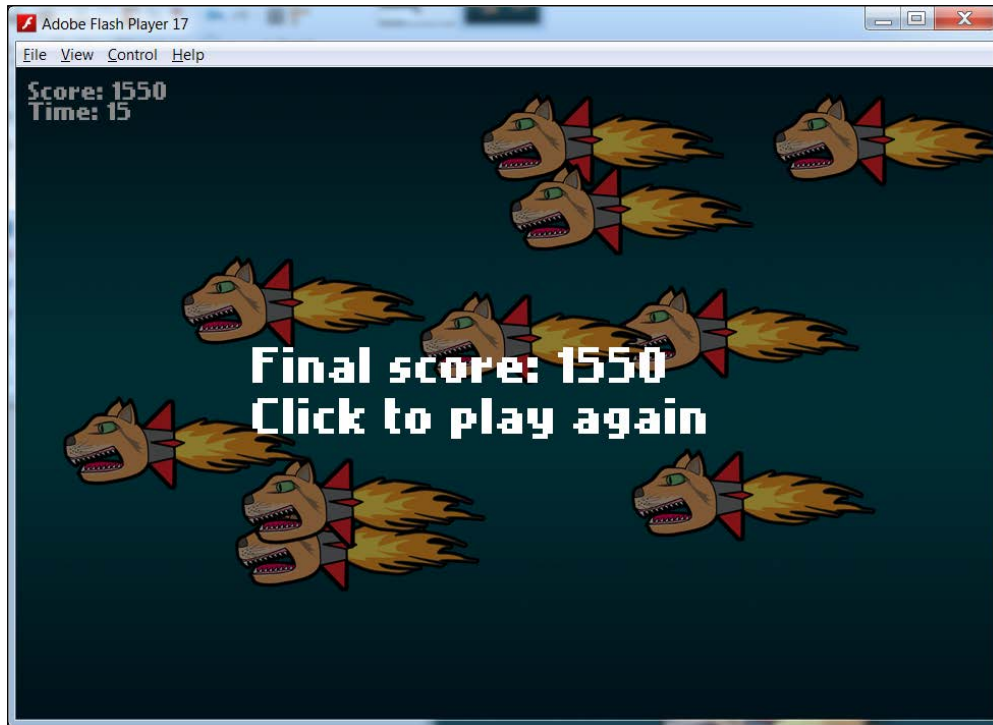
Next, we check to see whether the `ticks` variable is greater than or equal to `levelTime`. If it is, then that means the game time has run out and we need to show the end screen.

Inside the `if` statement, we create a new `LevelEndScreen` variable and pass the `score` variable into the constructor.

Finally, we add the `LevelEndScreen` variable, and we're done! You can now run the game to see everything you've made in this chapter!



The following screenshot depicts the end screen:



Summary

In this chapter, you learned the fundamentals of the Haxe language and started learning about how to build games with HaxeFlixel. You then applied those skills to build the foundation of the game you'll continue to improve over the course of this book.

If you ran into problems along the way, check out the source files provided for this chapter. Compare it with what you've done to spot differences.

To get a better understanding of how everything works, feel free to play around with the game's variables to change the gameplay experience. Even though the game is very simple, you can change things pretty drastically!

In the next chapter, we're going to set up the game's screen flow and some of the basic user interface elements that we'll need. We'll also look at some low-impact effects that can easily bring the game to life quickly.

3

Dealing with Menus and Screen Flow

In this chapter, we'll add the UI and screen flow for our game. We'll start by building the splash screen that will appear when the game first loads, then we'll revisit the level end screen and make it look cooler. Finally, we'll update our in-game **heads-up display (HUD)**.

Throughout the chapter, we'll also use some cool HaxeFlixel features that will easily make our UI feel more exciting.

Here's a rundown of the sections in this chapter:

- Project updates
- Building the splash screen
- Updating the level end screen
- Adding the new game HUD
- Adding the HUD to the game
- Project updates

To start, we'll update our project by adding the new art assets and changing our default `FlxState` class.

Adding new assets

Copy all of the images provided for this chapter into the `assets/images` folder. Here's a quick rundown of what each one is used for:

Image	Description
<code>button.png</code>	A generic button image that can be used to skin HaxeFlixel buttons
<code>healthHeartOn.png</code> and <code>healthHeartOff.png</code>	Health indicator images that we'll add to the HUD and then make use of in the next chapter
<code>hudBackground.png</code>	The background image for our in-game HUD
<code>levelEndHeader.png</code>	The new header for the level end screen
<code>levelEndWindow.png</code>	The background image for the level end screen
<code>newBestScore.png</code>	An image that will appear on the level end screen when the player reaches a new high score
<code>titleScreenBackground.png</code>	The background image for our splash screen, which has our game's main character on it
<code>titleScreenLogo.png</code>	The game's logo

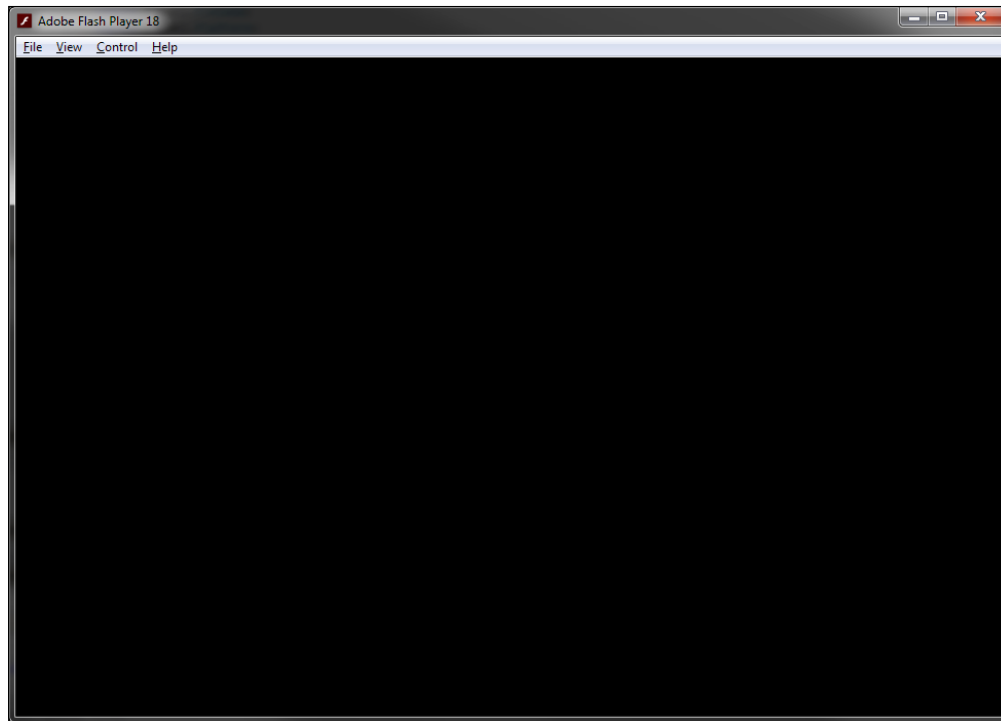
Next, create a new folder in the `assets` folder named `fonts`. After this, copy `BebasNeue.otf` into the new folder that was created. This will be the font we will use in-game instead of the default one HaxeFlixel normally uses.

Changing the default state

Now that we have the assets we need, we need to change the default `FlxState` instance that appears when the game starts.

To do this, open `Main.hx` and change the initial state variable to use `MenuState`, like this:

```
var initialState:Class<FlxState> = MenuState;
```



That's it! Now you can run the game to see the change.

It's not terribly exciting, but we'll fix that!

Building the splash screen

Let's make the first screen that players will see when the game starts. Since it is the first thing they see, we'll add some small touches that will help get players excited about playing the game. We'll add the background image with our main character, pogo, and the star graphic that will go behind it, and some text instructions to prompt players to continue the game.

We'll animate these things on screen in a very simple sequence using some tools that HaxeFlixel provides to improve the feel of games. We'll also add an old-school blinking effect to our input prompt.

Adding imports

To start, open `MenuState.hx` and add these import statements:

```
import flixel.util.FlxColor;
import flixel.effects.FlxFlicker;
import flixel.tweens.FlxTween;
import flixel.tweens.FlxEase;
import flixel.util.FlxSpriteUtil;
```

The `FlxColor` class will give us shortcuts for tinting text and screen effects. The `FlxFlicker` class provides a quick blinking effect we can use in different ways. The `FlxTween` and `FlxEase` classes allow us to animate objects around the screen and apply timing curves to the animations. The `FlxSpriteUtil` class provides utility functions for working with sprites; we'll use it to center a sprite in the middle of the screen.

Creating variables

Now we're going to make variables for all of the objects we're going to use. We'll add these above the constructor so that they're all in a central location at the top of the class.

The first three are the `FlxSprite` objects for the art we'll add to the screen:

```
private var background:FlxSprite;
private var star:FlxSprite;
private var logo:FlxSprite;
```

The names are fairly self-explanatory: `background` is our background image, `star` is the star that appears behind the logo, and `logo` is the game's logo.

Next, let's add variables for our player input prompt:

```
public var checkForInput:Bool = false;
private var txtInstructions:FlxText;
```

The `checkForInput` Boolean will be a variable we enable after all onscreen animations have completed and the user can click or tap to continue the game. The `txtInstructions` variable will be the text that instructs the player how to continue.

Adding sprites

Let's add the sprites we just made variables for. In the `create` function, just after the `super.create()` ; function call, add the following lines:

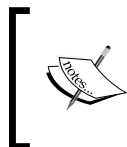
```
background = new FlxSprite();
background.loadGraphic(AssetPaths.titleScreenBackground__png);
background.visible = false;
FlxSpriteUtil.screenCenter(background);
add(background);

star = new FlxSprite();
star.loadGraphic(AssetPaths.titleScreenStar__png);
star.setPosition(3000, 276);
add(star);

logo = new FlxSprite();
logo.loadGraphic(AssetPaths.titleScreenLogo__png);
logo.setPosition(59, 360);
logo.visible = false;
add(logo);
```

At this point, you should be pretty used to adding `FlxSprite` objects to the screen, so I won't go over this in too much detail. There are a few things to note:

- First, you'll note that I set the background and logo sprites' `visible` property to `false`. We're doing this because we're going to make those sprites appear in a specific sequence based on the animation we'll be setting up.



The `visible` property prevents a sprite from being rendered entirely. This is distinct from the `alpha` property, which can make a sprite entirely invisible, but it will still continue to be rendered.

- For the background sprite, we used the `screenCenter` function of the `FlxSpriteUtil` class. This will center an object on screen. You can also use the `FlxG.width` and `FlxG.height` properties to get the screen dimensions to position things manually.
- Finally, we've set the star sprite's `X` position to `3,000`. This means that the star will be entirely off screen when the game first loads. We're going to animate the star in from the right-hand side of the screen to start off our splash screen sequence.

Adding instructions

Next, we're going to add the instructions text to the screen. After the last block of code, add this:

```
txtInstructions = new FlxText(0, 0, 500);
txtInstructions.text = "- TAP TO START -";
txtInstructions.setFormat(AssetPaths.BebasNeue__otf, 72,
    FlxColor.WHITE, "center");
txtInstructions.setBorderStyle(FlxText.BORDER_OUTLINE,
    FlxColor.BLACK, 3);

txtInstructions.setPosition(FlxG.width / 2 - txtInstructions.width
    / 2, 0);
txtInstructions.visible = false;

add(txtInstructions);
```

Some of these you've seen before, and some of these are new. In the first couple of lines, we create a new `FlxText` object and set its text value.

After this, we call the `setFormat` function on `txtInstructions`. This function lets us style the font on the `FlxText` object. Here's a breakdown of the arguments in the function call:

- The first argument is the font you wish to use, in this case the `BebasNeue` font we added earlier.
- Next is the font size, which uses point sizes like text editors, not pixels, so we're saying that this is a 72 pt font.
- After that is the color of the text. We use the `FlxColor` class to say that we want this text to be white. Later in this chapter, we'll use custom color values, so you'll get to see what that looks like.
- Finally, we define the alignment of the text. In this case, we're saying that we want the text to be center-aligned. This works the same as a text editor does; *center* alignment makes all text flow from the middle out. You can also use *left* and *right* to align text on the left-hand or right-hand side of the `FlxText` object.

After this, we call the `setBorderStyle` function, which will add a border around the `FlxText` object. First, you specify the type of border you need, in this case, an outline. Next, you specify the color of the border, and finally you pass in a border thickness in pixels.

Also, like the sprites given earlier, we'll set the `visible` property of `txtInstructions` to `false`, so we can have it appear later.

Starting the animation sequence

Now, let's create the animation sequence for the title screen. Here's a quick rundown of how it will work:

1. The game will start blank, showing a black screen.
2. The star sprite will animate on screen.
3. When that finishes, the logo will appear and the screen will flash white and shake.
4. During this, the background image with our main character will appear.
5. Finally, the instructions prompt will start blinking.

It sounds really complicated, but HaxeFlixel makes all of that incredibly simple to do. So, let's start by adding the animation that will kick things off. After the last block of code, add this:

```
FlxTween.tween(star, { x:0 }, 1.0, { ease:
  FlxEase.quadIn,complete: onStarAnimateIn} );
```

This uses the `FlxTween` object to create a tween. If you've never heard this term before, a tween is an old animation term used to describe animations between two points, hence the name. Animators used this to describe frames drawn between two keyframes; programmers use it to describe a property animating from one value to another over time.

The `tween` function takes in a number of arguments; here's a quick rundown of them:

- The first argument is the object you wish to animate; this value is `Dynamic`, so it can be used on pretty much anything with public properties.
- The second argument is used to determine the properties you wish to animate and what value you want them to animate to. In this case, we're animating the star's `x` property to 0 from its starting position of 3,000. The curly braces are used to define a dynamic object inline.
- The third argument is the duration of the tween. This is given in seconds, but it's a `Float` value, so typing in something like 1.587 is totally valid if you need that much precision.
- The last argument contains options for the tween that let you further customize it and define callbacks. In this case, we use the `FlxEase` function to specify an animation ease curve; `ease` lets you determine the speed of the animation over time. We used `quadIn`, which will slightly accelerate the animation over time.

We also defined a *complete* option, which is a callback that will be called once the animation is finished. The `onStarAnimateIn` value we passed in is a callback that we'll create next.

The `FlxTween` class is extremely powerful and lets you do really complex animations easily. It might seem like there's a lot to it, but once you use it enough, it will become second nature.

Creating the next stage of the sequence

Next, we're going to define the callback that our tween calls when it is complete:

```
private function onStarAnimateIn(tween:FlxTween):Void{
    logo.visible = true;
    FlxG.camera.flash(FlxColor.WHITE, 0.25, onFlashComplete);
    FlxG.camera.shake();
    background.visible = true;
}
```

This function takes in a `FlxTween` object as an argument; this is the tween that calls the callback in case you needed to access it. We don't need it, however.

Inside the function, we set the logo's `visible` property to `true` so that it will now appear.

After this, we call the `flash` function on the `camera` object of `FlxG`; this is a helper function that lets you achieve a screen flash effect really easily. It takes in a color, duration in seconds, and a callback that fires when the flash is complete. It's like a much simpler version of the tween we did earlier.

Then, we call the `shake` function on the `camera` object, which will shake the screen. It can take in parameters like the `flash` function, but the default functionality is fine for now.

Those two functions being called one after another will make the screen flash and shake at the same time.

Finally, we make the background visible during the chaos of the shake and flash.

Finishing the sequence

Now we're going to create the callback function that the `flash` function will call:

```
private function onFlashComplete():Void{
    FlxFlicker.flicker(txtInstructions, 0, 0.5);
    checkForInput = true;
}
```

This function takes no arguments. Inside the function, we call the `flicker` function of the `FlxFlicker` object. This is another helper object that HaxeFlixel has to speed up development. This will make our instructions text blink like the instructions you'd see in an old-school arcade of NES game.

The `flicker` function takes in a number of arguments; here are the ones we used:

- The first argument is the object that you wish to flicker, in this case, our instructions text.
- The second argument is the duration of the flicker; setting it to 0 will make it flicker forever.
- The last argument is the interval duration, which is how fast it should flicker.

You'll note that we didn't set the `visible` property of `txtInstructions`. This is because the `flicker` function will toggle the visibility of the object automatically.

Finally, we'll set `checkForInput` to `true`, since we're ready for the user to continue onto the game.

Navigating back to the game

We're going to handle the input the same way that we did on `LevelEndScreen` in the previous chapter by checking for input in the update loop. So, in the update function after the `super.update()` ; call, add this:

```
if (checkForInput && FlxG.mouse.justReleased)
{
    goToGame();
}
```

So, just like before, we're going to check for `FlxG.mouse.justReleased`, but we're also going to check for `checkForInput`. This means the function inside the `if` statement won't get called until the animation sequence is complete and `checkForInput` is set to `true`.

Finally, let's make the function that was called in the `if` statement:

```
private function goToGame():Void {  
    FlxG.camera.fade(FlxColor.BLACK,0.5, false,function() {  
        FlxG.switchState(new PlayState());  
    });  
}
```

This function will handle the transition to `PlayState` from `MenuState`. Inside it, we use one more helper function of the camera object of `FlxG`: `fade`. The `fade` function works in much the same way as the `flash` function. It takes in a color to fade to, and duration, as we've seen before. The Boolean indicates that if we're fading in or out, setting it to `false` specifies that we are fading out. The last argument is a callback function that gets called when `fade` is complete.

In this case, we've defined an inline function instead of creating a separate named function. These are useful when you have a small piece of functionality you want to happen in a callback.

Inside the callback, we call `FlxG.switchState`, and pass in a new `PlayState` object. This will move us from `MenuState` to `PlayState`, and the game will play as it did before.

That's the last step in building our splash screen. Run the game and check it out!



Updating the level end screen

Now that we've built the splash screen, we're going to build on what we know and make the level end screen more interesting:

- We're going to add art to the screen so that it looks nicer
- Then, we're going to use a button to reset the level instead of clicking anywhere
- After that, we're going to keep track of high scores to give players a goal while playing
- Finally, we're going to make another animated sequence, as we did on the splash screen

Adding imports

To start, open `LevelEndScreen.hx` and add the following imports:

```
import AssetPaths;
import Reg;

import flixel.ui.FlxBUTTON;
import flixel.effects.FlxFlicker;
import flixel.group.FlxFGroup;
import flixel.tweens.FlxFTween;
import flixel.tweens.FlxFEase;
```

Adding variables

Next, in the variables section, add these new `FlxSprite` variables:

```
private var windowHeader:FlxSprite;
private var newBestScore:FlxSprite;
private var window:FlxSprite;
```

The `windowHeader` sprite is the header of the window we'll add to the screen. It will fly in from off-screen as our logo did on the splash screen. The `newBestScore` sprite will blink when the player beats a previous high score, encouraging them to keep getting better. The `window` sprite is the window that our score information will appear in.

After this, add a new `FlxText` variable:

```
private var txtBestScore:FlxText;
```

This will appear under the current game score to show the player's best score.

Then, make the `FlxButton` variable that will contain the button we'll use to restart the level, as follows:

```
private var btnPlayAgain:FlxButton;
```

Finally, add a `FlxGroup` variable:

```
private var windowGroup:FlxGroup;
```

The `FlxGroup` class is the base class of the `FlxState` class. As the name implies, it lets you group objects together. We'll be using it to group everything but the header together so that all the elements can have their visibility toggled at the same time.



The `FlxGroup` objects are also useful for managing layers of objects so that they appear on top of or underneath other objects correctly.

Housekeeping

We're going to change a couple of things and update our mouse functionality. Before we do that, let's go through and clean up what we already have.

Inside the constructor, go to the line where the `alpha` property of `background` is set, and change it to this:

```
background.alpha = 0.75;
```

This will make the background 25% darker, a minor change, but it will make the new screen contrast better.

After this, go to the section where `txtGameScore` was created, and remove it entirely. For reference, this is the block:

```
txtGameScore = new FlxText(0, 0, 500, "Final score: " +  
score + "\nClick to play again", 40);  
txtGameScore.setPosition(FlxG.width / 2 - txtGameScore.width  
/ 2, FlxG.height / 2 - txtGameScore.height / 2);  
add(txtGameScore);
```

We're still going to be using `txtGameScore`, but we'll be styling and positioning it differently, so for now just remove it.

Finally, go down to the `update` function and remove this block:

```
if (FlxG.mouse.justReleased)
{
    onPlayAgain();
}
```

Since we're going to use a button, we no longer need to check for clicks on the screen.

Adding the group and setting up the window

Now that we've cleaned things up, let's add our `windowGroup` object. In the constructor after the background is added to the screen, add this block:

```
windowGroup = new FlxGroup();
add(windowGroup);
windowGroup.visible = false;
```

It's pretty straightforward to set up the group. Since we're setting the `visible` property to `false`, everything that gets added to it will automatically be invisible.

Next, let's add our header and window:

```
windowHeader = new FlxSprite();
windowHeader.loadGraphic(AssetPaths.levelEndHeader__png);
windowHeader.setPosition(3000, FlxG.height / 2 -
    windowHeader.height - 71);
add(windowHeader);

window = new FlxSprite();
window.loadGraphic(AssetPaths.levelEndWindow__png);
window.setPosition(FlxG.width/2 - window.width/2, FlxG.height/2
    - window.height/2);
windowGroup.add(window);
```

This is pretty standard. The main difference is that we're adding the window sprite to `windowGroup` instead of `LevelEndScreen`. The `windowHeader` object will be the only thing we don't add to `windowGroup`.



Be sure to note that the position of `windowHeader` is 3000, as was the position of our logo on the splash screen. It will work the same way here.

You might also note some numbers being added to or removed from positions. These are just offsets applied to the *x* and *y* positions of objects to get them into the correct spots on screen.

Adding scores

Next, we're going to use the `Reg` class to keep track of the game's high score:

```
var newHighScore:Bool = false;
if(score > Reg.score){
    Reg.score = score;
    newHighScore = true;
}
```

First, we create a temporary variable to indicate if the player has reached a new high score or not. This will be used to figure out if we should show the `newBestScore` sprite or not.

Next, we'll check our current score against the `score` property of the `Reg` class. If it's larger, set the `score` property of the `Reg` class to the current score. After that, set `newHighScore` to `true`.

Next, let's add our `FlxText` objects:

```
txtGameScore = new FlxText(0, 0, 500);
txtGameScore.text = "SCORE: " + score;
txtGameScore.setFormat(AssetPaths.BebasNeue__otf,
72, 0x9d4300, "left");
txtGameScore.setPosition(FlxG.width / 2 - txtGameScore.width
/ 2 + 20, FlxG.height / 2 - txtGameScore.height - 54);
windowGroup.add(txtGameScore);

txtBestScore = new FlxText(0, 0, 500);
txtBestScore.text = "BEST SCORE: " + Reg.score;
txtBestScore.setFormat(AssetPaths.BebasNeue__otf, 48,
0xffd800, "left");
txtBestScore.setPosition(FlxG.width / 2 - 210,
FlxG.height / 2 - 2);
windowGroup.add(txtBestScore);
```

First, we start by reading `txtGameScore`; this time, however, we call `setFormat` to define its font style. In this case, we also passed in a custom color value using hexadecimal notation.



Haxe converts hexadecimal values preceded by `0x` to their corresponding integer values. So, the value `0x9d4300` will be converted to an integer for us. This is incredibly useful if you're trying to match a specific design, as you can take a hexadecimal value from Photoshop or another graphics program and use it directly.

After that, we add `txtGameScore` to the `windowGroup` object.

Then, we create the `txtBestScore` object and set it to use `Reg.score` to show the player's best score. The `txtBestScore` object also gets styled with a custom color.

Adding the play again button

Now, let's add the play again button:

```
btnPlayAgain = new FlxButton(0, 0, "PLAY AGAIN", onPlayAgain);
btnPlayAgain.loadGraphic(AssetPaths.button__png);
btnPlayAgain.label.setFormat(AssetPaths.BebasNeue__otf,
47, FlxColor.WHITE, "center");
btnPlayAgain.label.setBorderStyle(FlxText.BORDER_OUTLINE,
0x9d4300, 3);
btnPlayAgain.setPosition(FlxG.width / 2 - btnPlayAgain.width/2,
FlxG.height / 2 + 51);
windowGroup.add(btnPlayAgain);
```

Buttons are basically a `FlxSprite` and a `FlxText` object grouped together with some functionality to handle mouse input. In the constructor, you pass along *x* and *y* values, followed by the text you want on the button's label and then you follow up with a callback to be called when the button is clicked. This callback will call the `onPlayAgain` function we originally were calling in the `update` loop.

After this, we call `loadGraphic` like you would on a sprite, to load up the graphic you need.

Then, we set some values on the `label` property of the button. The `label` object is the `FlxText` object, so it's styled exactly as we've been doing all along.

After everything is set up, we add the button to the window group.

Showing the best score sprite

Next, we're going to add the logic to show the `newBestScore` sprite:

```
if(newHighScore){
    newBestScore = new FlxSprite();
    newBestScore.loadGraphic(AssetPaths.newBestScore__png);
    newBestScore.setPosition(FlxG.width / 2 - 296, FlxG.height / 2
- 45);
    FlxFlicker.flicker(newBestScore, 0, 0.5);
    windowGroup.add(newBestScore);
}
```

First, we check to see if the player has reached a new high score. If not, then this block will be skipped.

Inside the `if` statement, we create a new `FlxSprite` object. After this, we use the `FlxFlicker` object to make the sprite blink as we did with the instructions text on the splash screen. Finally, we add the `newBestScore` sprite to the window group.

Creating the animation sequence

To finish off the changes to `LevelEndScreen`, we'll add the animation sequence that will play when a level ends. After the last block of code, add this:

```
FlxTween.tween(windowHeader, { x:FlxG.width / 2 -
windowHeader.width / 2 - 19}, 1.5, { ease:
FlxEase.quadIn,complete: onHeaderAnimateIn} );

FlxG.camera.flash(FlxColor.WHITE, 0.25);
```

Like the splash screen, we'll use the `FlxTween` object to kick things off. This time, we're animating the `windowHeader` object, and we'll call `onHeaderAnimateIn` when the tween completes.

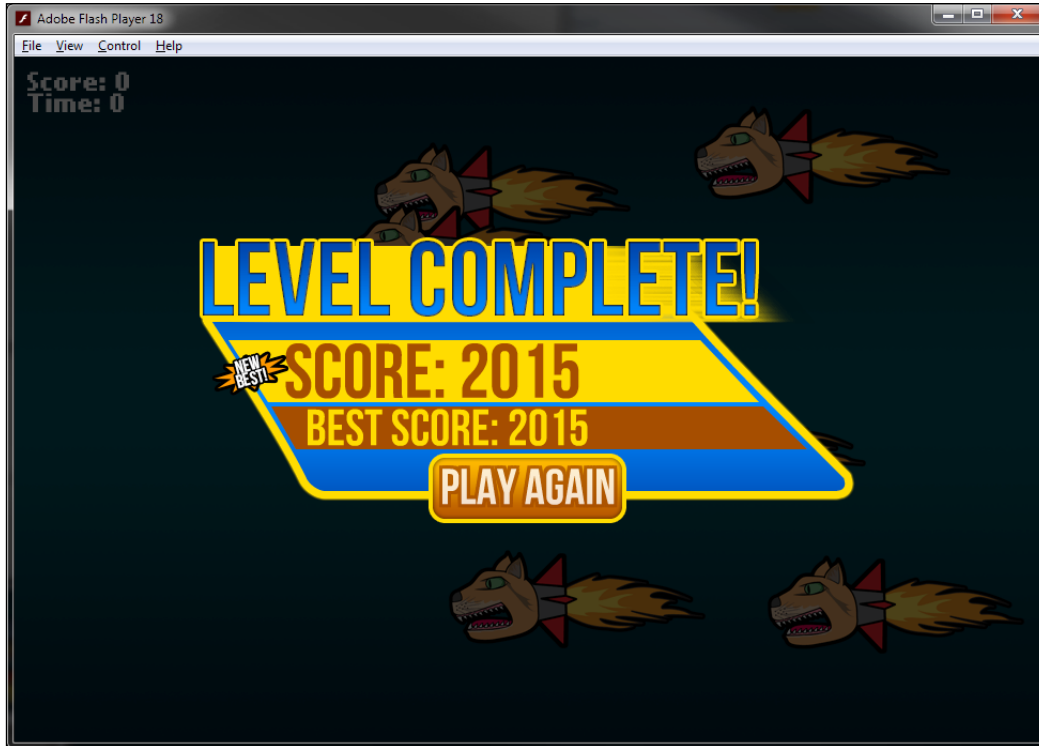
After the tween, we'll call the `flash` function on the `FlxG.camera` object. This time we won't use a callback on the `flash` function since the tween is already going to call one.

Next, let's add the callback function that the tween will call:

```
private function onHeaderAnimateIn(tween:FlxTween):Void{
    FlxG.camera.shake();
    windowGroup.visible = true;
}
```

Like before, this callback takes in a `FlxTween` object. Inside the function, we'll call `FlxG.camera.shake()`; again. Then, we'll set the `visible` property of the `windowGroup` to `true`. This will make everything we added to `windowGroup` become visible.

With this, our changes to `LevelEndScreen` are done! Run the game and check it out!



Creating the new game HUD

Now we're going to implement the new game HUD. This will display a series of hearts that indicate player health and show the player's score. Since we haven't fleshed out the rest of the gameplay, the health display won't do anything yet, but it will be ready for when we take care of that in the next chapter.

Creating the GameHUD class

To start, we're going to create a new class in the `source/ui` folder as we did with `LevelEndScreen`. So, make a class named `GameHUD`. This class will extend the `FlxGroup` class this time around.

For Sublime Text users, here's what the class should look like to start:

```
package ui;

import flixel.group.FlxGroup;

class GameHUD extends FlxGroup
{
    public function new()
    {
        super();
    }
}
```

Adding imports

Now that we have our empty class set up, let's add the imports that we'll need:

```
import flixel.FlxG;
import flixel.FlxSprite;
import flixel.text.FlxText;
import flixel.util.FlxColor;
```

Adding variables

Next, let's add our variables:

```
private var background:FlxSprite;

private var txtHealthHeader:FlxText;
private var healthHearts:Array<FlxSprite>;
private var maxHealth:Int = 3;
private var heartPadding:Int = 14;

private var txtScore:FlxText;
```

So to start, we have a background sprite as usual. Next, we have `txtHealthHeader`, which will just be a header that indicates where the health is displayed.

Then, we have an array of `FlxSprite` objects named `healthHearts`. This will contain the sprites that we will use to represent the player's health. Putting the sprites into an array gives us a lot of flexibility if we ever choose to increase the amount of health a player can have. The maximum amount of health a player can have is in the `maxHealth` variable.

After this, we have the `heartPadding` variable, and this will specify how many pixels apart the hearts should be.

Finally, we have a `FlxText` object to store our game score.

Laying things out

Now that we have the variables we need, let's add everything. To start, navigate to the constructor. We'll add the background:

```
background = new FlxSprite();
background.loadGraphic(AssetPaths.hudBackground__png);
background.setPosition(6, 18);
add(background);
```

After this, we'll add and style our health header and our score field:

```
txtHealthHeader = new FlxText(37, 14, 500);
txtHealthHeader.text = "HEALTH:";
txtHealthHeader.setFormat(AssetPaths.BebasNeue__otf, 36,
    FlxColor.WHITE, "left");
add(txtHealthHeader);

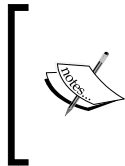
txtScore = new FlxText(62, 52, 500);
txtScore.text = "SCORE: " + 0;
txtScore.setFormat(AssetPaths.BebasNeue__otf, 30, 0x9d4300,
    "left");
add(txtScore);
```

Next, let's add the health hearts:

```
healthHearts = new Array<FlxSprite>();
var healthHeart:FlxSprite;
for (i in 0...maxHealth) {
    healthHeart = new FlxSprite();
    healthHeart.loadGraphic(AssetPaths.healthHeartOn__png);
    healthHeart.setPosition(132 + (i * (healthHeart.width +
        heartPadding)), 22);
    add(healthHeart);
    healthHearts.push(healthHeart);
}
```

Here, we're creating a new array of `FlxSprite` objects. We create a reusable variable to hold each `FlxSprite` instance before they go into the array, and then we loop from 0 to `maxHealth` (3 in this case).

Inside the loop, we create a new `FlxSprite` instance, load its graphic, and position it.



The hearts have a base position of 132, and then we're adding `i` multiplied by the width of the heart sprite plus our `heartPadding` variable. What this will do is offset each heart a specific distance from the previous one. This works whether we have three hearts or 50, it really doesn't matter.

After this, we add the `healthHeart` object to `GameHUD` and then push the object into the `healthHearts` array.

For now, that's all we're going to do with the hearts. We'll add the functionality for them when we add a character that can lose health in the next chapter.

Changing the score

Now, we just need a way to update the score on the HUD. To do this, we'll make a public function:

```
public function setScore(score:Int):Void{  
    txtScore.text = "SCORE: " + score;  
}
```

This function will take in an integer and then use that to update the `txtScore` field.

Adding the new game HUD

Finally, we need to add the new HUD to the game. We're also going to use a `FlxGroup` object to prevent our enemies appearing over the game's UI. Open up `PlayState.hx` to get started.

Adding imports

As usual, we have a few imports to add:

```
import ui.GameHUD;  
import flixel.group.FlxGroup;
```

Adding variables

We need to add a couple of new variables:

```
private var gameHud:GameHUD;  
private var enemyLayer:FlxGroup;
```

The `gameHud` variable will store our `GameHUD` class, and `enemyLayer` will store the group that will be used to layer enemies under the HUD.

Housekeeping

Like when we made updates to the `LevelEndScreen` class, we need to change a few things before we start adding new content. To start, go into the variables section and delete these two lines:

```
private var txtScore:FlxText;  
private var txtTime:FlxText;
```

We're removing the score counter because it will now be in the HUD. We're removing the time counter because we're going to change the purpose of the timer in the next chapter, and we will no longer need to display it.

In the `create` function, after the background is added, remove these lines:

```
txtScore = new FlxText(10, 10, 200, "Score: 0", 20);  
txtTime = new FlxText(10, 30, 200, "Time: 0", 20);  
add(txtScore);  
add(txtTime);
```

We need to get rid of any references to these variables since they no longer exist. Next, go to the `onEnemyMouseDown` function and remove this line:

```
txtScore.text = "Score: " + score;
```

Then, go into the `onTimeComplete` function and remove this line:

```
txtTime.text = "Time: " + ticks;
```

And with this, we've cleared out the references to the old temporary HUD and are ready to add the new one.

Adding the HUD and enemy group

Now, we're going to add the new HUD and layer our enemies under it. To start, add the following lines:

```
enemyLayer = new FlxGroup();  
add(enemyLayer);  
  
gameHud = new GameHUD();  
add(gameHud);
```

Things are pretty straightforward here we're creating and adding a new `FlxGroup` object and then creating and adding `GameHUD`. Adding `GameHUD` after `enemyLayer` means that it will be layered on top of that group.

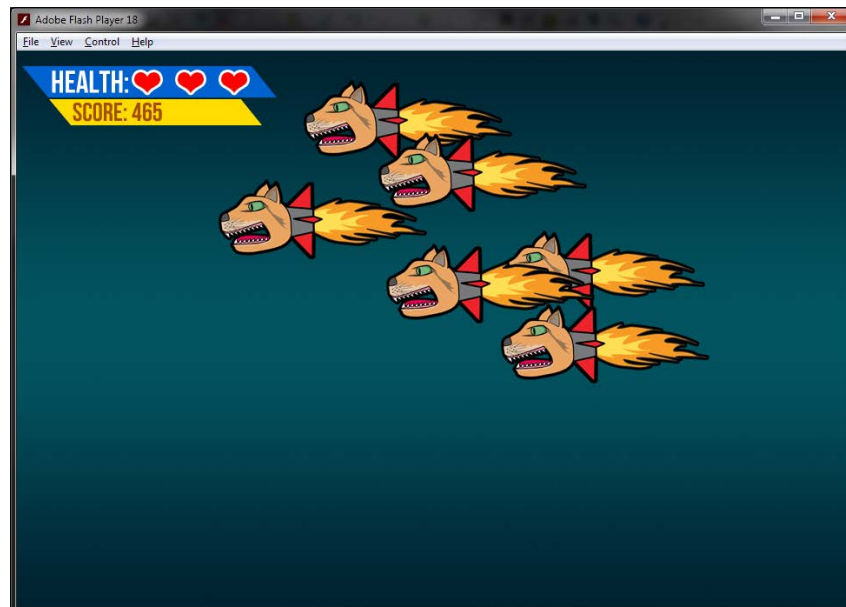
Then, we just need to make one little tweak inside the loop where enemies are created and added. Change the line where you see `add(enemy);` to this:

```
enemyLayer.add(enemy);
```

Now all enemies will appear under the HUD! We just have one last thing to add. In the `onEnemyMouseDown` function, add this line after the score is added up:

```
gameHud.setScore(score);
```

Now the score on the HUD will update when the score changes. With this, we're done! Run the game to check out the new HUD.



Summary

In this chapter, you learned how to set up screen flow between the `FlxState` objects, and how to set up and style UI components that HaxeFlixel provides. You also learned how to easily use HaxeFlixel tools to add some visual effects with very little effort.

If you ran into problems along the way, check out the provided source files for this chapter. Compare them with what you've done to spot differences.

I recommend playing around with the effects and screen layout to make sure that you have a good feel for everything. A few minor changes can make the effects look significantly different.

In the next chapter, we're going to really start fleshing out the game. We're going to add animations and most of the core game functionality.

4

Delving into Animations and Gameplay

In this chapter, we're going to finish off our core game functionality. We will add our walrus player character, make it shoot, and implement the health system. Along with this, we will make the experience truly engaging by adding explosion and muzzle flash effects. We'll also make sure that our game continues to perform well by pooling game elements that we'll be using over and over.

Here's a rundown of the sections in this chapter:

- Object pooling
- Project updates
- Creating the explosion effects
- Making muzzle flashes
- Updating enemies
- Creating projectiles
- Making the player character
- Setting health on an HUD
- Adding everything to `PlayState`

Object pooling

Before we dive into the project, it's important to discuss object pooling and how it will improve the game. Object pooling is the process of storing object instances in cases where you need several objects of a specific type. Instead of destroying objects in a pool, they are removed temporarily and reset to their starting state so that they can be reused.

Recycling objects like this saves memory because you only need to allocate resources for a fixed number of objects instead of a potentially infinite number of objects. It also prevents the garbage collector from running frequently and wasting processing time.

The general rule of thumb is that if you have a large number of similar objects that will be getting added to and removed from the screen, you should place them in an object pool and recycle them.

In HaxeFlixel, the `FlxGroup` class contains the functionality for object pooling. Objects that are added to it can be recycled for use later on, so I will refer to the `FlxGroup` objects as pools at times.

Project updates

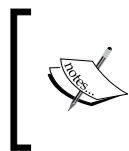
To start, we'll update our project by adding the new assets to the game, and then we'll make a quick change to our project's settings.

Adding new assets

Copy the new images provided for this chapter into the `assets/images` folder. Even though there are a lot of new art assets, you need to add only two images:

- `gameBackground.png`: This is a new background image that we will scroll to make the game look more exciting. This will overwrite the previous background.
- `ingameSprites.png`: This is a spritesheet that we'll pull art from and use to build animations.

The spritesheet contains the game's enemy art that we'll animate later on. This means that we no longer need the old enemy sprite, so delete `enemy01.png`.



In addition to the new images, there is `ingameSprites.json` also; copy this into the `assets/data` folder. This file defines the names and coordinates of each element on the spritesheet.

The spritesheet and data files were built using a tool named **Texture Packer**. HaxeFlixel has built-in support to read Texture Packer spritesheets. You can get Texture Packer from <https://www.codeandweb.com/texturepacker>.

We'll be going over the spritesheet and JSON files in greater detail once we start implementing them in-game.

Updating the project settings

Now we're going to add a new library that will add a few new features to HaxeFlixel. In this chapter, we'll be using it to really easily make a scrolling background.

Open `Project.xml` and go down to the `libraries` section. Find this line:

```
<!--haxelib name="flixel-addons" /-->
```

Uncomment the section so that it looks like this:

```
<haxelib name="flixel-addons" />
```

This will enable you to access classes in the `flixel.addons` package, which we will use to access the Texture Packer API.

Creating the explosion effects

We're going to build out the explosion effect that will play when an enemy dies. The effect will also play when an enemy makes contact with the player. The effect will comprise three parts:

- **The explosion line:** This is a line that will expand outward when the animation starts.
- **The explosion bulb:** This is a giant fireball that will swell and fade out; this is what will actually make it look like an explosion.
- **Smoke particle effect:** This is a puff of smoke that will linger on after the explosion.

Creating the class

We're going to make a new class named `ExplosionEffect` in the source folder. The class extends the `FlxGroup` object.

Here's what it should look like to start with:

```
package;  
  
import flixel.group.FlxGroup;  
  
class ExplosionEffect extends FlxGroup  
{
```

```
public function new()  
{  
    super();  
}  
}
```

Adding imports

Now let's add the imports that we'll need:

```
import flixel.effects.particles.FlxEmitter;  
import flixel.effects.particles.FlxParticle;  
import flixel.FlxCamera;  
import flixel.group.FlxCamera;  
import flixel.util.loaders.TexturePackerData;  
import flixel.tweens.FlxEase;  
import flixel.tweens.FlxTween;
```

The `FlxEmitter` and `FlxParticle` classes are used for particle effects. We'll be using the `FlxCamera` class to display sprites. The `FlxCamera` class is the class we're extending. The `TexturePackerData` class will be used to handle sprites loaded in from Texture Packer spritesheets. The `FlxEase` and `FlxTween` classes will be used for tweening.

Creating variables

Now, let's add variables:

```
private var texturePackerData:TexturePackerData;
```

This variable will store the data used to pull sprites from our spritesheet:

```
private var explosionLine:FlxCamera;  
private var lineXOffset:Float = 0;  
private var lineYOffset:Float = 30;
```

These variables contain the sprite for the explosion line, and we'll use the *x* and *y* offset to position the line over enemies that get destroyed.

```
private var explosionBulb:FlxCamera;  
private var bulbXOffset:Float = -20;  
private var bulbYOffset:Float = -120;
```

Like the variables for the line, these handle the explosion bulb and its positioning:

```
private var smoke:FlxEmitter;  
private var maxParticles:Int = 5;  
private var smokeDuration:Float = 1;
```

These variables are used to store and configure the smoke particle effect. The `FlxEmitter` class is a particle emitter.

The `maxParticles` variable will be used to instantiate particles and to limit how many can spawn.

The `smokeDuration` variable is used to determine how long the particle effect will emit for and to determine when the explosion effect has finished playing.

```
public var animationComplete:Bool = false;
```

The `animationComplete` variable is a flag that will be checked so that we know when the explosion can be added back to the object pool.

Setting up Texture Packer data

Now we're going to set up the `TexturePackerData` object. Inside the constructor, add this:

```
texturePackerData = new TexturePackerData(  
    AssetPaths.ingameSprites__json,  
    AssetPaths.ingameSprites__png);
```

The constructor of the object takes in the JSON and PNG files that define the spritesheet. This data was created in Texture Packer using the JSON export option.

Let's have a look at one of the JSON objects contained in the `ingameSprites.json` file:

```
{  
    "filename": "explosionLine.png",  
    "frame": { "x":244, "y":461, "w":240, "h":11 },  
    "rotated": false,  
    "trimmed": false,  
    "spriteSourceSize": { "x":0, "y":0, "w":240, "h":11 },  
    "sourceSize": { "w":240, "h":11 },  
    "pivot": { "x":0.5, "y":0.5 }  
},
```

The `filename` property is the name of the original image used to create the spritesheet. We will be referencing these filenames whenever a sprite is pulled from the spritesheet.

The `frame` property contains the coordinates and dimensions of the sprite on the sheet.

The `rotated` variable indicates that the sprite was rotated to fit it onto the spritesheet. The sprites on this sheet haven't been rotated.

The `trimmed` variables indicate that transparent alpha pixels around the edges of the sprite have been cut down to fit better on the sheet. The sprites on this sheet haven't been trimmed.

The `spriteSourceSize` and `sourceSize` properties indicate the original sprite in case it was shrunk to fit on the sheet. The sprites on this sheet haven't been resized.

The `pivot` property indicates the pivot point of the sprite.

For our use, we only need the `filename` property, but it's always good to know what information is provided by Texture Packer so that it can be used if needed.

Adding sprites

Now we're going to start making sprites using our `TexturePackerData` object. First is the explosion line:

```
explosionLine = new FlxSprite();
explosionLine.loadGraphicFromTexture(texturePackerData,
false, "explosionLine.png");
explosionLine.visible = false;
```

Notice that the setup of the sprite is similar to other sprites that have been made before. This time, we use the `loadGraphicFromTexture` function to add the art asset to the sprite.

This function takes in the following three arguments:

- The first is the texture data. In our case, it's the `TexturePackerData` object we made, but it will work with other formats as well.
- A Boolean is used to indicate whether the art is unique. When this is `true`, a new instance of the sprite texture will be created. If it is set to `false`, HaxeFlixel will try and find a cached copy of the texture to reuse; if it can't find one, it will make a new instance. Using a cached version of the texture will help reduce the memory usage.

- The third argument is the filename of the sprite to grab from the sheet. This is pulled from the filename properties in the JSON file.

It's also worth noting that the `visible` property of the sprite is set to `false` to start with. We'll be showing the parts of the explosion in a specific sequence, so we'll be hiding it for now.

Now let's make the explosion bulb using the following code:

```
explosionBulb = new FlxSprite();
explosionBulb.loadGraphicFromTexture(texturePackerData, false,
"explosionBulb.png");
explosionBulb.visible = false;
```

The setup for this is exactly the same as the explosion line; we pull in data from the spritesheet and set its `visible` property to `false`.

Creating a particle emitter

Now let's create a particle emitter and set some values on it.

```
smoke = new FlxEmitter(0, 0, maxParticles);
smoke.setXSpeed(-100, 100);
smoke.setYSpeed(-100, 100);
smoke.setScale(0.15, 0.5, 1, 3);
smoke.setAlpha(0.25, 1, 0, 0);
smoke.setRotation(-180, 180);
```

First, we create an instance of the `FlxEmitter` class; the constructor takes in x and y values followed by the maximum number of particles to emit. After this, we're going to set a bunch of properties on the emitter that will define how it will behave:

- `setXSpeed` and `setYSpeed` both take in two values. These are used to determine the range of velocity particles we have on the x and y axes. Setting these values to equal positive and negative values will make our particles expand outward in all directions.
- `setScale` takes in four values: a start minimum and maximum scale, and an end minimum and maximum scale. The start scales are a range used to determine the initial scale of particles when the effect starts, and the end scales are used to determine the range of scales that particles will have by the time the effect finishes.
- `setAlpha` also takes in four values like `setScale`; these are used to determine the start and end alpha values of the particles.

- `setRotation` takes in a minimum and maximum rotation value, which will rotate particles.

Now let's add the particles to our emitter:

```
var smokeParticle:FlxParticle;
for (i in 0..maxParticles) {
    smokeParticle = new FlxParticle();
    smokeParticle.loadGraphicFromTexture(texturePackerData,
    false, "smoke.png");
    smoke.add(smokeParticle);
}
```

To create the particles we'll need, we will loop until we hit `maxParticles` and create a series of `FlxParticle` objects. The `FlxParticle` objects extend `FlxSprites`, so we can still use the `loadGraphicFromTexture` function to pull art from the spritesheet.

After creating the particle, use the `add` function on the emitter to make the particle available to be used.

Adding everything to the group

Next, we just have to add everything to the group as follows:

```
add(smoke);
add(explosionLine);
add(explosionBulb);
```

The smoke will be layered on the bottom, followed by the explosion line and then the bulb.

Starting the explosion sequence

Now let's create the function that will be used when an explosion is displayed:

```
public function playExplosion(x:Float, y:Float):Void{}
```

This is a public function that will be called from `PlayState`. It takes in `x` and `y` coordinates, which we will use to position all of the elements of the effect. In this case, the `x` and `y` coordinates will be pulled from enemies that get destroyed.

Inside the function add this:

```
animationComplete = false;
```

Since the animation sequence is just starting, the `animationComplete` variable will be set to `false`.

Next, we'll position everything:

```
explosionLine.setPosition(x + lineXOffset, y + lineYOffset);
explosionBulb.setPosition(x + bulbXOffset, y + bulbYOffset);
smoke.setPosition(x, y);
smoke.visible = false;
```

The `explosionLine` and `explosionBulb` are positioned using their offsets. The smoke can be positioned using the straight *x* and *y* coordinates. We'll also hide the smoke effect until it's ready to be used.

Now let's kick the animation off:

```
explosionLine.scale.x = 0;
explosionLine.alpha = 1;
explosionLine.visible = true;
FlxTween.tween(explosionLine.scale, { x:1 }, 0.15, { ease:
    FlxEase.quadIn, complete: showBulb } );
```

We'll start by scaling `explosionLine` down, making it opaque and visible. After this we will tween the *x* property of the scale object of `explosionLine`. It will be very quick, taking place over 0.15 seconds. When complete, it will call the `showBulb` function.

Showing the explosion bulb

Now let's make the `showBulb` function to continue the sequence:

```
private function showBulb(tween:FlxTween):Void {}
```

Like other `FlxTween` callbacks, this function takes in a `FlxTween` object.

Inside the function, start by fading the explosion line out with a tween:

```
FlxTween.tween(explosionLine, { alpha:0 }, 0.15,
{ ease: FlxEase.quadIn } );
```

Next, let's get the bulb ready to animate:

```
explosionBulb.scale.x = 0;
explosionBulb.scale.y = 0;
explosionBulb.alpha = 1;
explosionBulb.visible = true;
```

We scale down the bulb on the x and y axes and make it opaque and visible. After this, we're going to introduce a little randomization to keep the effect looking interesting:

```
var flipBulb:Float = Math.random();
if(flipBulb < 0.5){
    FlxTween.tween(explosionBulb.scale, { x:1,y:1 }, 0.15, {
        ease: FlxEase.quadIn,complete: showSmoke } );
}else{
    FlxTween.tween(explosionBulb.scale, { x:1,y:-1 }, 0.15, {
        ease: FlxEase.quadIn,complete: showSmoke } );
}
```

Since the player is going to see a lot of explosions, one easy way to help them stay visually interesting is to just vertically flip the bulb randomly.

To do this, we grab a random float. Remember that the random function will give a float between 0 and 1, so we just need to check if the value is less than 0.5 to give us a 50/50 chance for the bulb to flip.

If the value is less than 0.5, we'll scale the bulb up to normal size without flipping. If it's greater, then we'll set the y scale to -1, which will be the normal sale of the sprite, but it will be vertically flipped.

At the end of either tween a callback named `showSmoke` is called.

Showing smoke and completing the animation

Now let's create the `showSmoke` function. Like before, let's create a function that takes in a `FlxTween` object:

```
private function showSmoke(tween:FlxTween):Void{}
```

Next, let's show and start the smoke particle emitter:

```
smoke.visible = true;
smoke.start(true, smokeDuration);
```

The `start` function takes in a couple of values; the first value is whether or not the emitter should explode, which means that all of the particles will emit at once. Since this is what we want, it will get set to `true`. This is the lifespan of particles, so we use the `smokeDuration` variable for that.

Next, let's make the bulb disappear:

```
if(explosionBulb.scale.y > 0){
    FlxTween.tween(explosionBulb.scale, { x:2, y:2 },
        0.15, { ease: FlxEase.quadIn } );
}else{
    FlxTween.tween(explosionBulb.scale, { x:2, y:-2 }, 0.15,
        { ease: FlxEase.quadIn } );
}
FlxTween.tween(explosionBulb, { alpha:0 }, 0.25, { ease:
    FlxEase.quadIn } );
```

Start by checking to see whether the bulb's *y*-scale value is greater than or less than 0. This will determine if it's been flipped. Once this has been determined, the scale will get tweened up to twice its normal size. This will help it look like it's dissipating as it fades out.

After this, the bulb's `alpha` property will be tweened down to zero to fade it out.

Next, we're going to use a tween that will fire a callback when the smoke has finished playing:

```
var smokeTimeOut:Float = 1;
FlxTween.num(smokeTimeOut, 0, smokeDuration,{
    complete:onAnimationComplete } );
```

We're going to start by creating a temporary float value, and then we're going to tween it down to 0 using the `num` function of the `FlxTween` object. This works similarly to the `tween` function, but it takes in a number instead of an object. The tween will last for the length of `smokeDuration` and will call the `onAnimationComplete` callback.

Let's quickly make that callback:

```
private function onAnimationComplete(tween:FlxTween):Void{
    animationComplete = true;
}
```

So, this function just sets the `animationComplete` flag to `true`. This will be checked later so we know which explosions can be reused.

Preparing for object pooling

We'll be going over how object pooling works in greater detail later in the chapter, but for now we need to make one function that will get called as explosions are being reused:

```
override public function revive():Void{
    super.revive();
    animationComplete = false;
    explosionLine.revive();
    explosionBulb.revive();
    smoke.revive();
}
```

The object pools in HaxeFlixel operate using the `kill` and `revive` functions on objects. These functions help to remove and reset objects that are pooled. Since the `ExplosionEffect` class extends `FlxGroup`, we need to add a little functionality to ensure all of the visual elements get reset properly.

To do this, we override the `revive` function and then call the `revive` function on the super class. After this, we set `animationComplete` to `false` and then call `revive` on the `explosionLine`, `explosionBulb`, and `smoke` objects.

Normally, when `revive` is called on a `FlxGroup` object, it doesn't try to revive objects that are children of it, so we have to do that ourselves.

Making muzzle flashes

Now we're going to create the muzzle flash that will play when the player fires a projectile. This effect will comprise two parts:

- **Flash:** The sparks that appear immediately when the projectile is fired.
- **Rings:** A set of red rings that radiate outward, after the flash. This will make the projectiles appear more powerful.

The muzzle flash will work like a simplified version of the explosion effect, so it'll be very easy to put together.

Creating the class

To start, make a new class named `MuzzleFlash` in the source folder. The class extends `FlxGroup`.

Here's what it should look like to start with:

```
package;

import flixel.group.FlxGroup;

class MuzzleFlash extends FlxGroup
{
    public function new()
    {
        super();
    }
}
```

Adding imports

Next, let's add the imports:

```
import flixel.FlxSprite;
import flixel.group.FlxGroup;
import flixel.util.loaders.TexturePackerData;
import flixel.tweens.FlxTween;
import flixel.tweens.FlxEase;
```

There are no real surprises here; we'll be using the `TexturePackerData` class here too.

Creating variables

Now, to create the variables, we have this:

```
private var texturePackerData:TexturePackerData;

private var flash:FlxSprite;
private var flashXOffset:Float = 330;
private var flashYOffset:Float = 90;

private var rings:FlxSprite;
private var ringsXOffset:Float = 340;
private var ringsYOffset:Float = 80;
```

Like the explosion effect, we create sprites we need and create *x* and *y* offset values that will be used for positioning. In this case, we will be offsetting the flash and rings based on the player character's position to put the muzzle flash at the end of its gun.

Adding sprites

Inside the constructor, create and add the sprites:

```
texturePackerData = new TexturePackerData(AssetPaths.ingameSprites__
json,
    AssetPaths.ingameSprites__png);

flash = new FlxSprite();
flash.loadGraphicFromTexture(texturePackerData, true,
    "muzzleFlash.png");
add(flash);
flash.visible = false;

rings = new FlxSprite();
rings.loadGraphicFromTexture(texturePackerData, true,
    "muzzleRings.png");
add(rings);
rings.visible = false;
```

As discussed before, we make a new `TexturePackerData` object and then use it to load graphics into our `FlxSprite` objects. We'll also be setting the `visible` property of the sprites to `false` so that we can play out the muzzle flash animation sequence.

Playing the flash

Let's build the function that will get called when the muzzle flash needs to get played. Like the explosion effect's `play` function, this is a public function, and it takes in *x* and *y* coordinates:

```
public function playFlash(x:Float, y:Float):Void {
    flash.setPosition(x + flashXOffset, y + flashYOffset);
    rings.setPosition(x + ringsXOffset, y + ringsYOffset);

    flash.scale.y = 0;
    flash.scale.x = 0;

    flash.alpha = 1;
    flash.visible = true;
    FlxTween.tween(flash.scale, { x:1,y:1 }, 0.1, { ease:
    FlxEase.quadIn,complete: showRings } );
}
```

Using the offsets that were defined earlier, position the flash and rings. After this, the scale x and y are set to 0 and the effect is made opaque and visible. The tween will expand the flash up to its full scale over 0.1 seconds, and it will then call the `showRings` function.

Showing the rings

Now let's move to the `showRings` callback. This takes in a `FlxTween` object.

```
private function showRings(tween:FlxTween):Void {
    flash.alpha = 0;
    flash.visible = false;

    rings.scale.y = 0;
    rings.scale.x = 0;

    rings.alpha = 1;
    rings.visible = true;

    FlxTween.tween(rings.scale, { x:1.5, y:1.5 }, 0.15, {
        ease: FlxEase.quadIn } );
    FlxTween.tween(rings, { alpha:0 }, 0.3, {
        ease: FlxEase.quadIn } );
}
```

Inside the function, immediately hide the flash, then set the scale values on the rings the same way we did with the flash. Then make the rings opaque and visible.

After this, we're going to call two tweens. The first one will scale up to 1.5 times the normal size, and the other will tween the alpha down to zero. This will make the rings expand and look like they dissipate into nothing.

Updating enemies

We're going to add some new art and functionality to our enemies. We'll add a two-frame looping animation that will make the enemies seem livelier.

The enemies will be made to spawn offscreen on the right and at a random y coordinate. They will fly across to the left side of the screen. After flying to the left side of the screen they will respawn on the right side of the screen at a new y coordinate and begin again.

They will also have a function that can be called when they are destroyed by the player. This will fade them out and then make them respawn.

To start, open `Enemy.hx` so we can make the changes we need.

Adding imports

To start, let's add the new imports we'll need:

```
import flixel.FlxG;
import flixel.tweens.FlxTween;
import flixel.tweens.FlxEase;
import flixel.util.loaders.TexturePackerData;
```

Adding variables

Next, let's add the new variables that we'll need, as follows:

```
private var texturePackerData:TexturePackerData;
private var movementTween:FlxTween;

private var spawnTimer:FlxTimer;
private var maxSpawnTime:Float = 14;

private var minSpeed:Float = 2.0;
private var maxSpeed:Float = 3.5;
```

To start, we'll need a `TexturePackerObject` to store our new enemy art.

Then, we have a `FlxTween` object. This time, we're actually hanging onto a reference to the tween we'll use. This is because we'll be tweening the enemies from right to left and we need to be able to stop the tween if the enemy hits a projectile or the player.

After this, we have a `FlxTimer` object that will be used to randomize when the enemies appear. With that is `maxSpawnTime`, which is a float that will provide us with the maximum amount of time to wait before appearing.

Following that are the `minSpeed` and `maxSpeed` variables. These provide the range of speeds that enemies can travel at.

After adding the new variables, we need to get rid of a couple, so delete the following variables:

```
private var maxX:Int = 750;
private var minX:Int = 0;
```

Since enemies will now always appear from the right side of the screen, we no longer need to give a random x coordinate.

We have one more variable change to make; change the `minY` variable's value to 40. This will ensure that enemies can always be shot by the player.

Adding animation

Inside the constructor, we'll be removing and moving some things around. To start, remove this line:

```
loadGraphic(AssetPaths.enemy01__png);
```

Since we'll be using the new enemy art, we no longer need to reference the old art. After removing that, remove this line as well:

```
var randomX:Int = (Math.floor(Math.random() *
(maxX - minX + 1)) + minX);
```

We no longer are randomly placing enemies on the x axis, so this is no longer needed. After that, remove this line:

```
this.setPosition(randomX, randomY);
```

We're going to be setting the position in a couple of different spots. So this line isn't needed any more.

After that, let's set up the sprite:

```
texturePackerData = new TexturePackerData(AssetPaths.ingameSprites__
json,
AssetPaths.ingameSprites__png);
loadGraphicFromTexture(texturePackerData);
```

You'll note that we didn't pass in a filename to the `loadGraphicFromTexture` function. This is because we're going to specify frames for the animation loop:

```
var frameNames:Array<String> = new Array<String>();
frameNames.push("RocketKitty01.png");
frameNames.push("RocketKitty02.png");

animation.addByNames("Idle", frameNames, 8);
animation.play("Idle");
To add animation frames, we create an array of strings
that contain the filenames of each frame on the sheet.
```

Then, we use the `addByNames` function to add the frames. The `addByNames` function takes in a string animation name, the array of filenames, and a frame rate to play the animation at. Since we only have two frames, the animation needs to be played at a slow frame rate to actually see the animation. By default, the animation will loop on its own.

Following this, we call the `play` function and pass in our animation name to start the animation.

Resetting the spawn position

Now we'll create the function that will be called whenever we want to reset the spawn position of the enemy. To start, define the function:

```
private function resetSpawn(tween:FlxTween = null):Void {}
```

This function takes in a `FlxTween` object as an optional argument by setting it to a default value of `null`. This is because the function needs to be called by a `FlxTween` and also elsewhere in the class.

Now we're going to pull a line from the constructor. In the constructor, find the following line:

```
visible = false;
```

Cut and paste it into the `resetSpawn` function. After this, add the following lines:

```
this.alpha = 1;  
setPosition(FlxG.width + width, 0);
```

We set the sprite's `alpha` to 1 because it will get faded out if the enemy impacts the player or a projectile. Then, we set the position of the sprite to `FlxG.width` plus the width of the sprite. This will place it just offscreen.

After this, move these lines down from the constructor:

```
var spawnTime:Float = Math.random() * (maxSpawnTime+ 1);  
spawnTimer.start(spawnTime, onSpawn);
```

The instantiation of the spawn timer will be left in the constructor, but the `spawnTime` randomization and starting the timer will be used every reset so that the enemy behaves differently after each reset.

After this, add this last line to the function:

```
this.solid = true;
```

Setting the `solid` property to `true` will make the sprite capable of registering collisions. When enemies get killed, they will have their `solid` property set to `false` so that they stop colliding with things before being reset.

Updating the `onSpawn` function

We'll now expand the `onSpawn` function; this will set the random `y` position and start the tween.

After the line where the `visible` property is set, move this line down from the constructor:

```
var randomY:Int = (Math.floor(Math.random() *  
    (maxY - minY + 1)) + minY);
```

We'll still use this for `y` position randomizing.

After this, set the `y` position like so:

```
this.y = randomY;
```

Following that, let's get the tween going:

```
var randomSpeed:Float = (Math.random() * (maxSpeed - minSpeed +  
    1) + minSpeed);  
movementTween = FlxTween.tween(this, { x:-width }, randomSpeed,  
    { ease: FlxEase.quadIn,complete: resetSpawn } );
```

First, we generate a random float using the minimum and maximum speeds. Then, we tween the enemy to `-width`. What this will do is move the enemy completely off the left side of the screen. After that, the `resetSpawn` function is called when the tween is complete.

Adding death functionality

Finally, we need to make the function for killing enemies:

```
public function killEnemy():Void {  
    this.solid = false;  
    movementTween.cancel();  
    FlxTween.tween(this, { alpha:0}, 0.15, { ease:  
        FlxEase.quadIn,complete: resetSpawn } );  
}
```

This is a public function that will get called in `PlayState`. Inside the function, we start off by setting the `solid` property to `false`; this will prevent the enemy from colliding with anything until it's reset.

After that, we call the `cancel` function on the tween. This will stop the tween completely.

Finally, we fade the alpha of the enemy out to zero, so they don't just pop out of existence when hit.

Creating projectiles

Now we'll create the `projectile` class. It will have a sprite that's pulled from the spritesheet, and it will move from left to right at a fixed velocity.

Creating the class

To start, make a new class in the source directory named `Projectile`. Here it is for reference:

```
package;  
  
class TestSprite2 extends FlxSprite  
{  
    public function new()  
    {  
        super();  
    }  
}
```

Adding imports

Next, add the imports:

```
import flixel.util.loaders.TexturePackerData;  
import flixel.FlxSprite;
```

Creating variables

Now add the variables we'll be using:

```
private var texturePackerData:TexturePackerData;  
private var speed:Float = 2000;
```

The speed value will be used to determine the velocity of the projectile.

Displaying the sprite and setting the velocity

After the super call in the constructor, add these lines:

```
texturePackerData = new TexturePackerData(AssetPaths
    .ingameSprites__json,    AssetPaths.ingameSprites__png);
loadGraphicFromTexture(texturePackerData,
    false, "projectile.png");
this.velocity.x = speed;
```

Setting up a sprite from the spritesheet is pretty familiar now. The only new bit of functionality is the `velocity` property. This will move a sprite at a continuous speed in the direction provided. It's like a very simplified way to tween a sprite.

These few lines of code are all we need for the projectile!

Making the player character

Finally, we will create our player character and the namesake of our game, the Star Walrus! This walrus will also have a two-frame animation loop like the enemies.

Since this is a game that will be playable on both desktop and touch-based platforms, we'll be keeping the walrus controls nice and simple. The walrus will automatically shoot while the player moves it around the screen using the keyboard, mouse, or touch input.

Creating the class

First, let's create the class. In the source folder, make a new class named `Player` that extends `FlxGroup`:

```
package;

import flixel.group.FlxGroup;

class Player extends FlxGroup
{
    public function new()
    {
        super();
    }
}
```

Adding imports

Next, let's add the imports:

```
import flixel.FlxSprite;
import AssetPaths;
import flixel.group.FlxGroup;

import flixel.FlxG;
import flixel.tweens.FlxTween;
import flixel.tweens.FlxEase;

import flixel.util.loaders.TexturePackerData;
import flixel.group.FlxTypedGroup;
import flixel.util.FlxTimer;
```

There are quite a few classes imported here because the `Player` class covers a lot of our core game functionality.

Creating variables

First, let's create the variables for the sprite that will show our jet pack-powered walrus:

```
private var texturePackerData:TexturePackerData;
public var sprite:FlxSprite;
```



The sprite is public in this case. We're making it public so that it can be used for collision detection in `PlayState`.

After that, let's create the variables needed for the projectiles:

```
public var projectiles:FlxTypedGroup<Projectile>;
private var maxProjectiles:Int = 20;
private var shootSpeed:Float = 0.5;
private var shootTimer:FlxTimer;
```

First, we have a new object of the type `FlxTypedGroup`. This is a special type of group that we're going to use for object pooling. These groups only accept one type of object to be added to them unlike the more general `FlxGroup` objects that can take many types of object.

Inside the two angle brackets, we pass in the type of object `FlxTypedGroup` will use; in this case, that's the `Projectile` class we created earlier.

After that, we have `maxProjectiles`, which is used to determine the maximum number of `Projectile` objects we'll instantiate. Then, we have `shootSpeed`, which is the interval between each shot. After that is `shootTimer`, which is the timer we'll use to fire projectiles.

Next, let's create a variable for our `MuzzleFlash` object:

```
private var muzzleFlash:MuzzleFlash;
```

Then, on to some variables for the player control:

```
private var movementSpeed:Float = 10;
private var mouseDeadzone:Float = 30;
```

```
private var maxY:Int = 500;
private var minY:Int = 0;
```

```
private var maxX:Int = 600;
private var minX:Int = -100;
```

The `movementSpeed` variable is used to determine how fast the walrus sprite will move around the screen. The `mouseDeadzone` variable is the number of pixels the mouse or touch input need to move before the walrus will move to that point.

The `maxY`, `minY`, `maxX`, and `minX` variables dictate the boundaries that the walrus can move within.

Adding animation

We'll start setting up the sprite as we have before:

```
texturePackerData = new TexturePackerData(
    AssetPaths.ingameSprites__json, AssetPaths.ingameSprites__png);
sprite = new FlxSprite();
sprite.loadGraphicFromTexture(texturePackerData);
```

So, like before, we set up `sprite` from the spritesheet, and since this is an animated sprite, we only need to pass in the `texturePackerData` variable to the `loadGraphicFromTexture` function.

Next, let's make it animate, and position it:

```
sprite.animation.addByNames("Loop",  
    ["player01.png", "player02.png"], 8);  
sprite.animation.play("Loop");  
sprite.setPosition(minX, FlxG.height / 2 - sprite.height/2);  
  
add(sprite);
```

Like with the `Enemy` class, we're calling the `addByNames` function to add the frames of animation and set it up. In this case, we're using an inline string array instead of defining a variable to store it. This is just another way you can handle adding the names.

After that, `play` is called with the animation name passed in. Then, we position the sprite at `minX` and center it vertically on the `y` axis. Finally, the sprite is added to the group.

Making the projectile pool

Now we'll create the pool of projectiles. To start, we need to instantiate the `FlxTypedGroup` object:

```
projectiles = new FlxTypedGroup<Projectile>(maxProjectiles);
```

In the constructor, pass in the maximum number of projectiles.

Next, we have to instantiate and populate the pool:

```
var projectile:Projectile;  
for(i in 0...maxProjectiles){  
    projectile = new Projectile();  
    projectiles.add(projectile);  
    projectile.kill();  
}  
add(projectiles);
```

We start by creating a variable to hold the projectiles we'll create, then we loop from 0 to `maxProjectiles`. In the loop, we create a new `Projectile` instance and add it to `FlxTypedGroup`.

Then, we call the `kill` function on the projectile. The `kill` function prevents the object from rendering or running any logic. It also queues the object up for recycling.

After that, we add the `projectiles` object to the group.

Creating the muzzle flash

Next, let's create the muzzle flash:

```
muzzleFlash = new MuzzleFlash();  
add(muzzleFlash);
```

For now, we just need to instantiate and add it.

Starting the shoot timer

Then, we create and set up the shootTimer object:

```
shootTimer = new FlxTimer();  
shootTimer.start(shootSpeed, onShoot, 0);
```

This will call the onShoot function every time the timer passes shootSpeed.

Shooting projectiles

Now we'll implement the functionality to shoot projectiles. To start, make the onShoot function that's called by shootTimer.

```
private function onShoot(timer:FlxTimer):Void {}
```

Next, we'll pull a projectile from the pool and position it, as follows:

```
var projectile = projectiles.recycle();  
projectile.x = sprite.x + 230;  
projectile.y = sprite.y + 100;
```

The recycle function will grab an object instance from the pool. Only objects that have had their kill function called can be used.

Once we have the object instance, we position it based on the sprite's position and offset it a little bit so that it appears at the end of the walrus's gun.

After that, we need to clean up any projectiles that have gone offscreen:

```
projectiles.forEachAlive(function(currentProjectile:Projectile) {  
    if(currentProjectile.x > (FlxG.stage.stageWidth +  
        currentProjectile.width)){  
        currentProjectile.kill();  
    }  
} );
```

The `forEachAlive` function loops through all of the objects in `FlxTypedGroup` that are currently alive. The function takes in a callback as its argument. The callback needs to take in an object of the type you specified for `FlxTypedGroup`.

Inside the callback function, we check to see if the projectile's x position will place it off the screen; if it will, then the projectile is killed and is ready to be recycled.

After that, we just need to play the muzzle flash:

```
muzzleFlash.playFlash(sprite.x, sprite.y);
```

Handling player input

Now we need to handle the player input. Since we'll be supporting both keyboard and touch input, we'll have to receive input in a couple of different ways. To start, we'll need to override the `update` function:

```
override public function update():Void
{
    super.update();
}
```

Next, let's handle the keyboard input. The walrus will be moved using the arrow and WASD keys.

After the `super` call, add this:

```
if (FlxG.keys.pressed.UP || FlxG.keys.pressed.W)
{
    moveUp();
}else if (FlxG.keys.pressed.DOWN || FlxG.keys.pressed.S) {
    moveDown();
}

if (FlxG.keys.pressed.RIGHT || FlxG.keys.pressed.D)
{
    moveRight();
}else if (FlxG.keys.pressed.LEFT || FlxG.keys.pressed.A) {
    moveLeft();
}
```

Therefore, we have a couple of statements that check the `FlxG.keys.pressed` properties. These are Booleans that will be set to `true` when specific keys are pressed.

In the first `if` statement, we check to see whether the up or *w* keys are pressed. If they are pressed, the `moveUp` function is called. In the `else if` block, we look to see whether the down or *s* keys are pressed. If they are pressed, then the `moveDown` function is called.

In the second `if` statement, we do the same with the right and *d* keys to call the `moveRight` function, and we use the left and *a* keys to call the `moveLeft` function.

Now let's handle the mouse input:

```
if (FlxG.mouse.pressed) {}
```

To start, make an `if` statement that checks `FlxG.mouse.pressed`. Inside the `if` statement, add this block:

```
if (FlxG.mouse.y - sprite.height / 2 < (sprite.y + sprite.height / 2) && Math.abs((sprite.y + sprite.height / 2) - (FlxG.mouse.y - sprite.height / 2)) > mouseDeadzone){
    moveUp();
}else if ((FlxG.mouse.y - sprite.height / 2) > (sprite.y + sprite.height / 2) && Math.abs((FlxG.mouse.y - sprite.height / 2) - (sprite.y + sprite.height / 2)) > mouseDeadzone){
    moveDown();
}
```

In this function, we're checking the mouse's *y* position against the walrus sprite's position and then moving the sprite in the direction of the mouse if the distance is large enough.

There are two conditions that are being checked in both branches of the `if` statement. The first condition checks to see if the mouse position offset by half of the sprite's height is less than or greater than the sprite's *y* position that is also offset by half of its height.

The second condition takes those same values and subtracts the lesser one from the greater one. We then wrap that in the `Math.abs` function to get its absolute value because we don't care whether it's negative or not. Then, we check to see if the resulting number is greater than our `mouseDeadzone` value. This second condition will ensure that the walrus sprite doesn't behave erratically.

If all of that passes, then we call the corresponding movement function.

After this, we're going to do the same, but this time on the *x* axis:

```
if (FlxG.mouse.x - sprite.width / 2 < (sprite.x + sprite.width / 2) && Math.abs((sprite.x + sprite.width / 2) - (FlxG.mouse.x - sprite.width / 2)) > mouseDeadzone){
```

```
        moveLeft();
    }else if ((FlxG.mouse.x - sprite.width / 2) > (sprite.x +
    sprite.width / 2) && Math.abs((FlxG.mouse.x - sprite.width / 2) -
    (sprite.x + sprite.width / 2)) > mouseDeadzone){
        moveRight();
    }
}
```

The same idea as before applies here; the only difference is that we are now checking *x* and width values instead of *y* and height values. We are also calling movement functions to go left and right instead of up and down.

Building movement functions

Now let's create the movement functions that are called in the `update` loop. They're all pretty simple and similar. Let's start with the `moveUp` function:

```
private function moveUp():Void {
    if(sprite.y > minY){
        sprite.y -= movementSpeed;
    }
}
```

This function checks to see if the sprite's *y* position is within the boundary we defined in the `variables` section. If it is, then we subtract the movement speed from the sprite's *y* property.

All of the other movement functions follow the same pattern. They check the sprite's position to ensure that it's within the defined boundaries and then move the sprite along the appropriate axis:

```
private function moveDown():Void {
    if(sprite.y < maxY){
        sprite.y += movementSpeed;
    }
}

private function moveRight():Void {
    if(sprite.x < maxX){
        sprite.x += movementSpeed;
    }
}

private function moveLeft():Void {
```

```
        if(sprite.x > minX){
            sprite.x -= movementSpeed;
        }
    }
```

Handling player death

Finally, we need to have a function that will handle the death of the player. Luckily, this is really easy:

```
public function killPlayer():Void{
    shootTimer.cancel();
    this.kill();
}
```

This is a public function so that it can be called from `PlayState`. So, we stop `shootTimer` so that projectiles won't spawn and then call the `kill` function on the `Player` class. What this will do is immediately remove the walrus sprite when this function is called.

Setting health on the HUD

Since we now have a character that will take damage, we need a function to update the health display.

Changing hearts

We just need to add one small function to the `GameHUD` class:

```
public function setHealth(health:Int):Void{
    for (i in 0...healthHearts.length) {
        if(i > (health - 1)){
            healthHearts[i].loadGraphic(AssetPaths.healthHeartOff__png);
        }
    }
}
```

This is a public function that takes in an integer health value.

Inside the function, we loop through each of the hearts on the HUD and check whether its index is greater than the amount of health that was passed in. If it is, then we load the *off* version of the heart graphic into that heart sprite.

Adding everything to PlayState

We're finally going to put everything together in `PlayState` and have the new functionalities we created work in unison. To do this, we're going to add our `Player` object, create a pool of the `ExplosionEffect` objects, and handle collisions between projectiles and enemies, and between the player and enemies.

In addition, we'll make the background scroll and use the timer to increase variability in the game's score, as mentioned in the previous chapter.

It's a lot to do, so let's get started!

Adding imports

To start, let's add the imports we'll need:

```
import flixel.group.FlxCGroup;
import flixel.addons.display.FlxBBackdrop;
import flixel.util.FlxCColor;
```

The new class of note is the `FlxBBackdrop` class. This class is used to create scrolling backgrounds. It's pulled from the addons library we enabled earlier in the chapter.

Adding variables

To start, we change the background variable to a `FlxBBackdrop` object like so:

```
private var background:FlxBBackdrop;
```

After this, we'll need to have a variable to handle the background's scroll speed:

```
private var backgroundScrollSpeed:Float = -200;
```

Next, we create variables for the `Player` object:

```
private var player:Player;
private var playerLayer:FlxCGroup;
private var health:Int = 3;
```

We have a variable to store the `Player` object, followed by a `FlxCGroup` object that we'll add it to, and then a variable to store the player's health.

Next, we need to have variables for our explosion effects:

```
private var explosions:FlxCGroup<ExplosionEffect>;
private var maxExplosions:Int = 40;
```

We'll start by creating `FlxTypedGroup` with the `ExplosionEffect` type so that the explosions can be pooled. After this, we will create a variable to store the maximum number of explosions.

Cleaning things up

Before we start adding new functionality, we need to remove and change a few things. In the constructor, delete these two lines:

```
var backgroundSprite:FlxSprite = new FlxSprite();
background.loadGraphic(AssetPaths.gameBackground__png);
```

Since we're going to handle the background differently, we need to get rid of these lines.

Also in the constructor, down in the loop where enemies are instantiated, remove this line:

```
MouseEventManager.add(enemy, onEnemyMouseDown);
```

We will no longer be clicking on enemies to kill them, so we no longer need to check for clicks on them.

At the bottom of the constructor, where the `start` function is called on `levelTimer`, change it to this:

```
levelTimer.start(1, onTimeComplete, 0);
```

We're going to use the timer to just add to the score, so we changed the last argument from `levelTime` to `0`. This will keep the timer ticking permanently until we stop it manually.

Now, move down to the `onEnemyMouseDown` function and delete the entire function. It doesn't serve a purpose now, as we don't click on enemies to kill them.

Finally, go inside the `onTimeComplete` function and delete this block:

```
if(ticks >= levelTime){
    var levelEndScreen:LevelEndScreen = new LevelEndScreen(score);
    add(levelEndScreen);
}
```

Since the game will no longer end based on time, we don't need this block.

Adding the scrolling backdrop

In the constructor, before the `add(background) ;` line, add these lines:

```
background = new FlxBackdrop(AssetPaths.gameBackground__png) ;
background.velocity.x = backgroundScrollSpeed;
```

Here, we're creating a new instance of the `FlxBackdrop` object, and it takes in the graphic that it's supposed to render. In this case, that's the new background image we added. After this, we set the `x` property on the `velocity` object of the background to `backgroundScrollSpeed`.

This will span the background to the left, and the `FlxBackdrop` object will tile it seamlessly as it moves.

Creating and adding the player

Next, in the constructor, after the `enemyLayer` is added, add these lines:

```
player = new Player();
playerLayer = new FlxGroup();

playerLayer.add(player);
add(playerLayer);
```

Here, we're creating the `Player` object and the `FlxGroup` object it will be added to. Then, we add a player to `playerLayer` and then add `playerLayer` to `PlayState`.

Creating an explosion object pool

Now, let's make the pool of `ExplosionEffect` objects:

```
explosions = new FlxTypedGroup<ExplosionEffect>(maxExplosions);
var explosionEffect:ExplosionEffect;

for(i in 0...maxExplosions){
    explosionEffect = new ExplosionEffect();
    explosionEffect.kill();
    explosions.add(explosionEffect);
}
playerLayer.add(explosions);
```

In the same way that we made the pool for projectiles, we'll make one for explosions. We create a new `FlxTypedGroup` that contains `ExplosionEffect` objects and then we set its maximum size in the constructor to `maxExplosions`.

After this, we loop through and create `ExplosionEffect` objects, kill them, and add them to the pool.

Finally, we add the explosions pool to `playerLayer`.

Killing enemies

Before we detect collisions, we'll have to create a couple of functions that we'll need. The first one is the function that will kill enemies when they collide with projectiles or the player:

```
private function killEnemy(enemy:Enemy, shakeStrength:Float =
0.005):Void{
    enemy.killEnemy();
    var explosion:ExplosionEffect = explosions.recycle();
    explosion.playExplosion(enemy.x, enemy.y);
    FlxG.camera.shake(shakeStrength, 0.5);

    explosions.forEachAlive(function(currentExplosion:
ExplosionEffect) {
        if (currentExplosion.animationComplete) {
            currentExplosion.kill();
        }
    } );
}
```

The function will take in an enemy object and a float that we'll use to control the screen shake when the enemy dies.

Inside the function, we immediately call the `killEnemy` function. After this, we'll grab an explosion from the pool and call the `playExplosion` function on it.

Then, we call `FlxG.camera.shake`, and this time, we'll pass in values to configure it. `ShakeStrength` will determine how strong the shake is. The second argument is the duration of the shake, so in this case, it'll last for half of a second.

Then, like we did with projectiles, we'll call the `forEachAlive` function on the explosion pool. Inside the callback function, we'll check whether the explosion's `animationComplete` property is `true`. If it is, then we call `kill` on the explosion.

Updating the score

Next, let's create the function that will get called when we need to increment the score. This will happen when an enemy is killed by a projectile and for every second the player survives.

```
private function updateScore(points:Int):Void{
    score += points;
    gameHud.setScore(score);
}
```

The function takes in an integer, which is the number of points received. That value is added to the score and then the score is updated on the HUD.

While we're updating the scoring, let's add that timer-incrementing score now. In the `onTimeComplete` function, add this line:

```
updateScore(pointsPerSecond);
```

This will increment the score by `pointsPerSecond`.

Detecting collisions

In the `update` function, after the `super.update()`; call, add this block:

```
if(player != null){
    FlxG.overlap(player.projectiles, enemyLayer,
        onProjectileCollision);
    FlxG.overlap(player.sprite, enemyLayer, onEnemyCollision);
}
```

Here, we check to see whether the player is not null, and if the player exists we make a couple of calls to `FlxG.overlap`.

The `overlap` function will check for overlaps between sprites. It also can check collisions for all sprites in `FlxGroup`, which enables us to easily check collisions between several objects at once.

The first overlap call checks the player's projectiles against the enemy layer. It will then call `onProjectileCollision` if an overlap is found. So, this will check for collisions between all projectiles and enemies.

The second overlap call checks the player sprite against the enemy layer and calls `onEnemyCollision` if an overlap is found.

Handling collisions between projectiles and enemies

Now, let's make the function that gets called when a projectile overlaps an enemy:

```
private function onProjectileCollision(projectile:
FlxObject, enemy: FlxObject):Void {
    projectile.kill();
    if(enemy.x < (FlxG.stage.stageWidth - 10)
    && Std.is(enemy, Enemy)){
        killEnemy(cast(enemy, Enemy));
        updateScore(enemyPointValue);
    }
}
```

The function takes in two `FlxObjects`: the first argument is the first object checked in the overlap function, and the second argument is the second object checked in the overlap function. So, in this case, the first object is `projectile`, and the second is `enemy`.

Inside the function, we call the `kill` function on the `projectile` object, which will remove it from the screen and get it ready to be recycled.

Next, we check the enemy's `x` position to make sure that it's at least 10 pixels on screen, and then we use `Std.is` to make sure that the `enemy` argument is an `Enemy` object.

The function on the `Std` class is native Haxe functionality. We use this function to ensure that when we cast the `FlxObject` to `Enemy`, the cast won't fail.

Inside the `if` statement, we'll call the `killEnemy` function.

After this, we'll call the `updateScore` function and pass in `enemyPointValue`.

Handling collisions between the player and enemies

Next, let's implement the function to handle collisions between the player and enemies. Like the previous function, this function will take in two `FlxObject` arguments:

```
private function onEnemyCollision(playerObject:
FlxObject, enemy: FlxObject):Void {
    if(Std.is(enemy, Enemy)){
```

```
        killEnemy(cast(enemy, Enemy), 0.1);
        FlxG.camera.flash(FlxColor.RED, 0.15);
        health--;
        gameHud.setHealth(health);
        if (health == 0) {
            player.killPlayer();
            levelTimer.cancel();
            var levelEndScreen:LevelEndScreen = new
            LevelEndScreen(score);
            add(levelEndScreen);
        }
    }
}
```

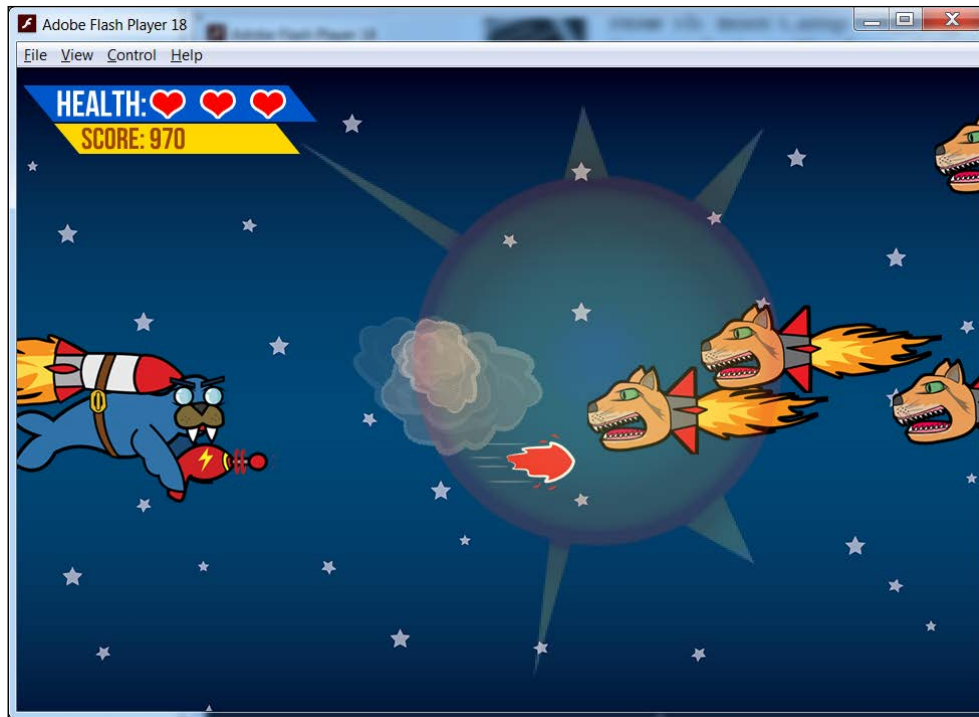
First, we check to make sure that the enemy is an `Enemy` object. After this, we call the `killEnemy` function and pass `0.1` as the shake strength; setting it higher will make impacts against the player really feel impactful.

Then, we'll flash the screen red for 0.15 seconds.

After this, we'll reduce the player's health by 1 and update the HUD.

If the player's health is 0, we'll call the `killPlayer` function, stop the level timer, and show the end screen.

Now that we're done with `PlayState`, we can finally run the game and see the awesome new gameplay and effects!



Summary

In this chapter, you learned how to use spritesheets, frame-based animations, object pools, and particle effects. That's a lot of learning! Along the way, you built some exciting new gameplay and really started to make the game visually shine.

If you ran into problems along the way, check out the source files provided for this chapter. Compare them with what you've done to spot differences.

There was a lot of functionality introduced in this chapter, and it would be worthwhile playing around with some of the properties to get a feel for how everything works. I suggest changing the timings on animations, making enemies move faster or slower, and shooting more projectiles.

In the next chapter, we're going to add music and sound effects, which will make all of the animations we've implemented feel more immersive and fun.

5

Adding Sound

In this chapter, we're going to add music and sound effects to make our new animations and gameplay really stand out. Good sound effects can make even simple animations have more punch, and music will focus players and keep them engaged. Luckily, OpenFL and HaxeFlixel make implementing music and sound incredibly easy.

Here's a rundown of the sections in this chapter:

- Project updates
- Creating sound constants
- Building the sound manager class
- Adding music
- Adding menu sounds
- Adding gameplay sounds

Project updates

To start, we'll update our project by adding the new sounds to the game. Then, we'll add a new section to our project settings XML file.

Adding new assets

In the `assets` folder, make a new folder named `audio`. Copy all of the new files provided for this chapter into the newly created `assets/audio` folder. These files are a collection of MP3 and OGG files. Each sound has an MP3 and OGG version; we'll get into why that is shortly, but here's a rundown of what each sound is:

- `music_fanfare`: This is music that will play on the title screen and level-end screen. It's got an upbeat, triumphant sound to it. This will get players excited to start playing and will keep them engaged after they eventually lose the game.
- `music_ingame`: This is the music that will play during gameplay. It's fast paced and has a sci-fi style sound to it. This will keep the gameplay feeling energetic.
- `sound_whoosh`: This is a *whoosh* sound effect that will play when the star on the title screen slides in and when the header on the level-end screen slides in. It will really help to sell those animations.
- `sound_ting`: This *ting* sound will play when the star on the title screen and the header on the level-end screen finish tweening. It will add some impact to the screen shake effects that play.
- `sound_click`: This is the sound that will play when the player clicks on the title screen and on the play again button on the level-end screen.
- `sound_damage`: This sound will play when the player takes damage. It's a deep grunt that will nail home that getting hit is a bad thing. It's a bit more human sounding than an actual walrus' grunt, but it still fits.
- `sound_explosion1` and `sound_explosion2`: These are two sound effects for explosions. If you remember from the previous chapter, we randomly flipped the explosion art to keep explosions looking interesting. We'll now take that further by playing a different sound effect for both variations.
- `sound_shoot`: This is a blaster sound effect that will play when the player shoots. Since the player is shooting constantly, it's not too high pitched or in your face, so it won't get annoying during extended play sessions. To help with that, we'll also drop the volume on the sound effect, to keep it as more of an ambient effect.

Updating the project settings

Next, open up `Project.xml`. We're now going to define asset paths for the sounds. Previously, we've relied on the `AssetPaths` class to do this for us, but sounds need a little extra effort due to the different file formats we'll be using.

As noted before, all of the sound files have an MP3 or OGG version. This is because Flash uses MP3 files, and most other platforms use OGG files. What this means is that we'll be manually defining which file to use based on the platform we're using. Luckily, OpenFL's XML project format helps make that really easy.

In `Project.xml`, go down to the path settings section and find this line:

```
<assets path="assets" />
```

After this, add the following block:

```
<assets path="assets">
</assets>
```

This will provide a block that we can manually define custom asset paths in. By setting the `path` property to `assets`, we're stating that the base folder for all of the referenced assets will be the `assets` folder.

Now let's define our first music file. Inside the opening and closing `assets` tags, add the following lines:

```
<music path="audio/music_fanfare.mp3" id="music_fanfare"
if="flash" />
<music path="audio/music_fanfare.ogg" id="music_fanfare"
unless="flash" />
```

In these first two lines, we're using the `music` tag to define our two `music_fanfare` files. The `music` tag indicates that the audio files are meant to be streamed from disk instead of being fully loaded into memory before playing. Since music files are generally larger and longer than sound effects, this provides a more efficient way to play them.

The first attribute in both tags is the path to the file. Since we already said that the files will be located under the `assets` folder, we only need to provide the path to the files from inside that directory. In this case, we're pointing to our two sound files inside of the `audio` subdirectory.

After this, set the `id` attribute. This will be a shortcut name we can use to reference the file so that we don't have to specify the full path.

After the `id` attribute, you will see the `if` and `unless` attributes. These let us define conditions for using the asset. In this case, we're saying that the MP3 version of the music should be used if the platform is `flash`, and we should use the OGG version unless if the platform is `flash`.

For more information on how these conditions work, check out OpenFL's XML format documentation at <http://www.openfl.org/learn/resources/command-line-tools/project-files/xml-format/>.

Now let's add our other piece of music:

```
<music path="audio/music_ingame.mp3" id="music_ingame"
if="flash" />
<music path="audio/music_ingame.ogg"
id="music_ingame" unless="flash" />
```

The setup for this is the same as the previous piece of music.

Next, let's add one of our sound effects:

```
<sound path="audio/sound_whoosh.mp3" id="sound_whoosh"
if="flash" />
<sound path="audio/sound_whoosh.ogg" id="sound_whoosh"
unless="flash" />
```

The sound tag has the same attributes as the music tag. The only real difference is that files using the sound tag will be fully loaded into memory before playing. This means that your sound effects should have as small of a file size as possible to prevent issues with memory usage.

Let's add our remaining sound effects:

```
<sound path="audio/sound_ting.mp3" id="sound_ting" if="flash" />
<sound path="audio/sound_ting.ogg" id="sound_ting"
unless="flash" />

<sound path="audio/sound_click.mp3" id="sound_click" if="flash" />
<sound path="audio/sound_click.ogg" id="sound_click"
unless="flash" />

<sound path="audio/sound_damage.mp3" id="sound_damage"
if="flash" />
<sound path="audio/sound_damage.ogg" id="sound_damage"
unless="flash" />

<sound path="audio/sound_explosion1.mp3"
id="sound_explosion1" if="flash" />
<sound path="audio/sound_explosion1.ogg"
id="sound_explosion1" unless="flash" />

<sound path="audio/sound_explosion2.mp3"
id="sound_explosion2" if="flash" />
```

```
<sound path="audio/sound_explosion2.ogg"
id="sound_explosion2" unless="flash" />

<sound path="audio/sound_shoot.mp3" id="sound_shoot" if="flash" />
<sound path="audio/sound_shoot.ogg"
id="sound_shoot" unless="flash" />
```

Creating sound constants

Now we're going to create a class that will contain **constants** for all of the music and the `id` attributes of sound we defined in the project file. We'll also create constants for our music and sound volume settings.

Creating the class

First, in the `source` folder, make a new folder named `audio`. Inside that folder, make a new class named `Sounds`. This class doesn't extend any other classes and doesn't need to import anything.

Here's what it should look like to start:

```
package audio;

class Sounds
{

}
```

Creating sound ID constants

Now let's add variables for each ID that we defined in the project file:

```
public static inline var MUSIC_FANFARE:String = "music_fanfare";
public static inline var MUSIC_INGAME:String = "music_ingame";
public static inline var SOUND_WHOOSH:String = "sound_whoosh";
public static inline var SOUND_TING:String = "sound_ting";
public static inline var SOUND_CLICK:String = "sound_click";
public static inline var SOUND_SHOOT:String = "sound_shoot";
public static inline var SOUND_DAMAGE:String = "sound_damage";
public static inline var SOUND_EXPLOSION1:String =
"sound_explosion1";
```

```
public static inline var SOUND_EXPLOSION2:String =  
    "sound_explosion2";
```

Each one of these variables is a `public static inline` variable. What this means is that these variables will be accessible from anywhere just by referencing the `Sounds` class. This also means that the variables cannot change at runtime, which is what makes them constant.

Naming the constants in uppercase with underscores is a fairly standard convention for constants.

These constants will give us a nice way to access the music and sounds, similar to the way that `AssetPaths` exposes other types of asset files.

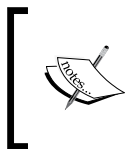
Creating volume constants

Next we'll create `float` constants for each sound that will define their volume levels:

```
public static inline var VOLUME_FANFARE:Float = 0.5;  
public static inline var VOLUME_INGAME:Float = 0.75;  
public static inline var VOLUME_WHOOSH:Float = 1;  
public static inline var VOLUME_TING:Float = 0.5;  
public static inline var VOLUME_CLICK:Float = 1;  
public static inline var VOLUME_SHOOT:Float = 0.5;  
public static inline var VOLUME_DAMAGE:Float = 1;  
public static inline var VOLUME_EXPLOSION1:Float = 1;  
public static inline var VOLUME_EXPLOSION2:Float = 0.5;
```

These are defined the same way we defined the ID constants. In `HaxeFlixel`, sound volume can range from 0 to 1. So, this means when `VOLUME_FANFARE` is set to 0.5, it means we want the volume to be at 50% of the volume that file normally defaults to.

These values are balanced to prevent repeated sounds from being too loud or in your face, and to cut down especially loud files from blowing out your eardrums. A lot of this is based on my personal opinion, so I suggest playing with these values after we implement the sounds to see if you can find a mix that suits your tastes perfectly. Having all of the values in one place like this makes balancing out the game audio very easy.



It's also worth noting that, typically, audio files are meant to be mixed by an audio engineer to be balanced in the audio file themselves, but being able to tweak volume levels after balancing is still very important.

Building the sound manager class

Now we're going to build the class we'll use to handle playback of all music and sound effects. This will keep the playback functionality centralized, which will be useful since so many things in the game need to play sounds.

To help with this, the class will implement the **singleton pattern** so that the class instance will be accessible everywhere.

Creating the class

To start, make a new class in the audio folder named `SoundManager`. The class doesn't extend anything.

Here's what it should look like to start:

```
package audio;

class SoundManager
{
    public function new()
    {
    }
}
```

Adding imports

Next, let's add the imports:

```
import flixel.FlxG;
import flixel.system.FlxSound;
import flixel.tweens.FlxTween;
```

The only new class to note is `FlxSound`. It will be used to store reusable sound effects. We'll be using the `FlxTween` class to add a delay to one effect.

Implementing the singleton pattern

The singleton pattern lets you have a single instance of a class that you can access from anywhere. It exposes a `static` property named `instance` that will create a new instance if one doesn't exist, otherwise it will grab the previously created instance.

So, above the constructor, add the following block of code:

```
public static var instance(get, null):SoundManager;

private static function get_instance():SoundManager {
    if(instance == null) {
        instance = new SoundManager();
    }
    return instance;
}
```

So, to start with, we make a public static variable, *SoundManager*, using the *get/set* syntax that is named *instance*. The *get/set* syntax lets you create functions to get or set a value.

You define *get/set* properties in a way similar to defining other variables, but you follow it with a set of parentheses that contains two arguments. The first specifies how to get the value, and the second specifies how to set the value.

By passing in *get* into the first argument, we're stating that there will be a function that is named starting with *get_* followed by the property name. In this case, that function is *get_instance*. A *get* function takes in no arguments and returns the data type that is defined in the property. A *set* function takes in an argument of the type defined in the property and doesn't return anything.

By passing *null* into the second argument, we're stating that there will be no *set* function for this property.

What all of this means is that any class accessing the *instance* property of *SoundManager* will have read-only access and cannot modify the instance.

After the property is defined, we create the *get* function. Inside the function, we check to see if we have already created an instance. If we haven't, we instantiate and return a new one, otherwise we return the one we've already created.

Finally, change the constructor to be *private* like so:

```
private function new()
{
}
```

This will ensure that the class can only be instantiated from the *get* function.

Adding music

Now we're going to add functions to play our music tracks and then add the calls to play them in the game.

Creating playback functions

In `SoundManager`, add the function to play our fanfare music:

```
public function playFanfareMusic():Void{
    FlxG.sound.playMusic(Sounds.MUSIC_FANFARE,
        Sounds.VOLUME_FANFARE);
}
```

This function will call the `playMusic` function on the `sound` object of `FlxG`. This will stream and play a music clip that will loop by default. We pass in two arguments: the first is the ID of the music to play and the second is the volume to play it at. We pull both of these from the constants that were made earlier.

Next let's make the function to play the in game music:

```
public function playInGameMusic():Void{
    FlxG.sound.playMusic(Sounds.MUSIC_INGAME,
        Sounds.VOLUME_INGAME);
}
```

The setup for this is identical to the previous function.

Calling playback functions

Now let's add the calls to these functions in the game. To start, open `MenuState` and add this import:

```
import audio.SoundManager;
```

Then, add this to the bottom of the `create` function:

```
SoundManager.instance.playFanfareMusic();
```

So, all we have to do is reference the `instance` property of `SoundManager` and call the `playFanfareMusic` function on it. This will start playing the fanfare music when the `MenuState` starts. Nice and easy.

Next, open `PlayState` and add this import:

```
import audio.SoundManager;
```

Then, add this to the bottom of its `create` function:

```
SoundManager.instance.playInGameMusic();
```

Like before, we just need to call the appropriate function on the instance of `SoundManager`. This will start playing the in-game music when `PlayState` starts.

Finally, open `LevelEndScreen` and add this import:

```
import audio.SoundManager;
```

Then, add this line to the bottom of the constructor:

```
SoundManager.instance.playFanfareMusic();
```

This will start playing the fanfare music when `LevelEndScreen` is displayed. Now you can run the game to hear all of the music playing in the game. It will already feel better with music alone!

Adding menu sounds

Next, we're going to handle playback of our *whoosh*, *ting*, and *click* sound effects. These will be used on both the title screen and the level-end screen.

Creating playback functions

To start, let's build the functions to play our sound effects. We'll start with the whoosh sound that will play when the star on the title screen and header on the level end screen tween in.

In `SoundManager`, add this function:

```
public function playWhooshSound(delay:Float):Void {
    var timeout:Float = 1;
    FlxTween.num(timeout, 0, delay, {
        complete:function(tween:FlxTween) {
            FlxG.sound.play(Sounds.SOUND_WHOOSH, Sounds.VOLUME_WHOOSH);
        }
    });
}
```

This function takes in a `float` value that will be used to delay playback of the sound effect. This will allow us to sync the sound effect up with the tweens on the title screen and level-end screen easier.

Inside the function we start by making a `float` variable that we will tween down, named `timeout`.

Next, we use a `num tween` to delay the call to play the sound effect. We create an inline function that will be called by the `num` function's complete callback.

Inside the callback, we call the `play` function on `sound` object of `FlxG`. This works similar to the `playMusic` function; it takes in the sound ID and volume as arguments. By default, sound effects don't loop and will be destroyed when they are done playing.

Next, let's add the functions to play our *ting* and *click* sounds:

```
public function playTingSound():Void{
    FlxG.sound.play(Sounds.SOUND_TING, Sounds.VOLUME_TING);
}

public function playClickSound():Void{
    FlxG.sound.play(Sounds.SOUND_CLICK, Sounds.VOLUME_CLICK);
}
```

These sounds don't require delays, so we just have to call the `play` function and pass in the sound ID and volume in both cases.

Calling playback functions

Now, let's play our sound effects. To start, open `MenuState` and add this line at the very bottom of the `create` function after the `playFanfareMusic` call:

```
SoundManager.instance.playWhooshSound(0.75);
```

This will play the whoosh sound with a 0.75 second delay. This will sync it up with the star tween.

Next, add this to the bottom of the `onStarAnimateIn` function:

```
SoundManager.instance.playTingSound();
```

This will play the *ting* sound when the screen flashes and shakes.

There's one last thing to add to `MenuState`. Add this to the bottom of the `goToGame` function:

```
SoundManager.instance.playClickSound();
```

This will play the click sound when the player clicks or taps.

Now we're going to do the same things in `LevelEndScreen`. Open that class up and add this to the bottom of the constructor:

```
SoundManager.instance.playWhooshSound(1);
```

This time, we're going to play the whoosh sound with a delay of one second; this is because the header takes longer to animate in than the star did.

Next, add this to the bottom of `onHeaderAnimateIn`:

```
SoundManager.instance.playTingSound();
```

This will play the *ting* sound when the camera shakes.

Now, go to the `onPlayAgain` function. Before adding our sound effect, we're going to wrap our `resetState` call in the `fade` object of the camera like so:

```
FlxG.camera.fade(FlxColor.BLACK, 0.5, false, function() {  
    FlxG.resetState();  
});
```

This will do two things for us: first, it will make the level-end screen behave more like the title screen, and second, it will give the click sound enough time to play.

If we just played the click sound without the extra time added by `fade`, the sound effect would get cut off because sound effects get cleaned up when states change or get reset. It's a handy functionality, but we also have to keep it in mind to make sure that everything works as intended.

So after this, add the call to play the click sound effect:

```
SoundManager.instance.playClickSound();
```

With this, you can run the game to hear the new sound effects!

Adding gameplay sounds

Finally, we're going to set up and play our gameplay sounds. Since these sounds are going to get played frequently, we're going to load and store them so that they can be reused instead of being loaded every time they're needed.

This will also prevent the game from sounding overwhelming as there will only be one of any given type of effect playing at a time.

Creating variables

Let's start by making variables to store our gameplay sounds. In `SoundManager`, add these variables before any function calls:

```
private var shootSound:FlxSound;  
private var damageSound:FlxSound;  
private var explosion1Sound:FlxSound;  
private var explosion2Sound:FlxSound;
```

These variables are of the `FlxSound` type. This class represents a sound that's loaded into memory.

Navigate into the constructor. In there, we're going to load and set up the sounds:

```
shootSound = FlxG.sound.load(Sounds.SOUND_SHOOT,  
Sounds.VOLUME_SHOOT);  
shootSound.persist = true;  
  
damageSound = FlxG.sound.load(Sounds.SOUND_DAMAGE,  
Sounds.VOLUME_DAMAGE);  
damageSound.persist = true;  
  
explosion1Sound = FlxG.sound.load(Sounds.SOUND_EXPLOSION1,  
Sounds.VOLUME_EXPLOSION1);  
explosion1Sound.persist = true;  
  
explosion2Sound = FlxG.sound.load(Sounds.SOUND_EXPLOSION2,  
Sounds.VOLUME_EXPLOSION2);  
explosion2Sound.persist = true;
```

In each case, we're calling the `load` function on the `sound` object of `FlxG`. Like the `playMusic` and `play` functions, `load` takes in the sound ID and volume.

After each `load`, we're setting each sound's `persist` property to `true`. This will prevent the sound effects from getting destroyed when the state changes or is reset.

You may be wondering why the gameplay sound effects are getting loaded in the constructor and are persisting across states. This is because our game is relatively simple and can handle those sound effects sticking around. It makes things easier and won't affect performance.

In a game that takes place across several states, it would be wise to unload at the end of a state and then load new sounds in the next state.

Creating functions

Now let's add functions for our shoot and damage sound effects:

```
public function playShootSound():Void{
    shootSound.play(true);
}

public function playDamageSound():Void{
    damageSound.play(true);
}
```

In both of these functions, we call the `play` function on each `FlxSound` object. We also pass in `true` as an argument. This will make the sound restart if it is already playing.

Since these sounds will be used frequently, it's possible that they may need to play again before they are finished playing through. If we didn't do this, there would be times where sounds wouldn't play and the game would sound broken.

We just need one more function to play our explosion sounds:

```
public function playExplosion(explosionNumber:Int):Void {
    switch(explosionNumber) {
        case 0:
            explosion1Sound.play(true);
        case 1:
            explosion2Sound.play(true);
    }
}
```

This function takes in an integer value to determine which explosion sound effect to play. If `explosionNumber` is 0, `explosion1Sound` plays. If `explosionNumber` is 1, `explosion2Sound` plays.

Calling playback functions

Now let's go and add the calls to our playback functions.

In `PlayState`, go to the `onEnemyCollision` function and add this line above the `if (health == 0)` check:

```
SoundManager.instance.playDamageSound();
```

This will play the damage sound every time the player is damaged.

Next, open the `Player` class and add this import:

```
import audio.SoundManager;
```

After this, navigate to the bottom of the `onShoot` function and add this line:

```
SoundManager.instance.playShootSound();
```

This will play the shoot sound effect every time a shot is fired.

Now navigate to the `ExplosionEffect` class and import `SoundManager`:

```
import audio.SoundManager;
```

Finally, go to the `showBulb` function and add `playExplosion` call to both conditions of the `if (flipBulb < 0.5)` check.

The `if` statement should look like this:

```
if (flipBulb < 0.5) {
    SoundManager.instance.playExplosion(0);
    FlxTween.tween(explosionBulb.scale, { x:1,y:1 }, 0.15,
        { ease: FlxEase.quadIn,complete: showSmoke} );
}else {
    SoundManager.instance.playExplosion(1);
    FlxTween.tween(explosionBulb.scale, { x:1,y:-1 }, 0.15,
        { ease: FlxEase.quadIn,complete: showSmoke} );
}
```

Note that in the first `playExplosion` call, we pass in 0, and in the second, we pass in 1. This will play the different explosion sounds based on how the explosion is flipped.

With this, all of the sounds have been implemented! Run the game and listen to everything.

To continue with what you've learned, play around with the volume mixes, and you can also try swapping out some of the audio files for some of your own. Depending on what sounds you choose, you can change the entire feel of the game without altering gameplay or art. Try it out.

Summary

In this chapter, you learned how to play sound files of different types based on the target platform. You also learned the difference between playing music and sound effects, and multiple ways to play sound in HaxeFlixel.

With sounds, the game is really starting to come together. Sounds help tie together the gameplay and art and make it feel cohesive.

In the next chapter, we're going to work with JSON and XML files to make the game more easily configurable and to liven up the gameplay by giving enemies movement patterns!

6

Working with Configuration Files

In this chapter, we're going to use a JSON configuration file to give enemies unique movement patterns and use an XML file to make our user interface text easily changeable. Adding movement patterns to enemies will make the game feel fairer to players instead of being purely random. This will also give us a way to easily add new content to the game without writing new code.

These two changes will also make the game updates and changes more accessible, allowing people who aren't programmers to work directly on the game.

Here's a rundown of the sections in this chapter:

- JSON versus XML
- Project updates
- Understanding the JSON configuration file
- Building the enemy spawner
- Adding movement patterns to enemies
- Updating `PlayState`
- Understanding the XML file
- Pulling string data from XML
- Replacing strings

JSON versus XML

We'll go into detail on implementing both JSON and XML data in this chapter, but first, we should look at why you may choose one over the other. At their core, both JSON and XML are external text-based formats that you can use to store various types of data.

The key differences you should evaluate are:

- **Ease of implementation:** How easy is the format to implement into a code base?
- **Readability:** How easy is it to read? Can a nonprogrammer understand it quickly?
- **Efficiency:** How much can the file be compressed?

On the first hand, JSON is slightly easier to implement. JSON data is parsed and turned into Haxe dynamic objects. XML uses the `Fast` class to pull out data as needed, which is still pretty easy, but does require a bit of extra effort.

On the other hand, XML is generally easier for nonprogrammers to read because you can add comments to describe the use of each property. JSON looks a bit closer to code, so programmers generally don't have problems with it. Since XML is generally easier to read for nonprogrammers, it's often used to handle text translation because translation teams generally don't have strong programming backgrounds. In this chapter, we'll use XML for string text changing because of this.

For the last point, JSON is generally considered more efficient because it reduces the number of characters needed to represent data. You can compress JSON files pretty significantly by removing whitespace. This is useful when you have large sets of data to load like complex level layouts. We'll be using JSON for our enemy movement patterns as an example of this case.

Ultimately, either format can be used in most cases, and it often comes to a judgment call as to what works best for your project.

Project updates

To start, we'll update our project by adding the two new configuration files we'll need. This time we don't need to adjust project settings.

Adding new assets

Copy the new files provided for this chapter into the `assets/data` folder. There is a JSON file and an XML file. Here's what they're used for:

- `movementPatterns.json`: This is the configuration file that will define the various movement patterns that enemies will use. We'll examine the contents of the file in the next section.
- `strings.xml`: This is the file that will contain all of the game's text data. It will provide a central location to change text and prevent the need to hunt for specific lines of text in the code base.

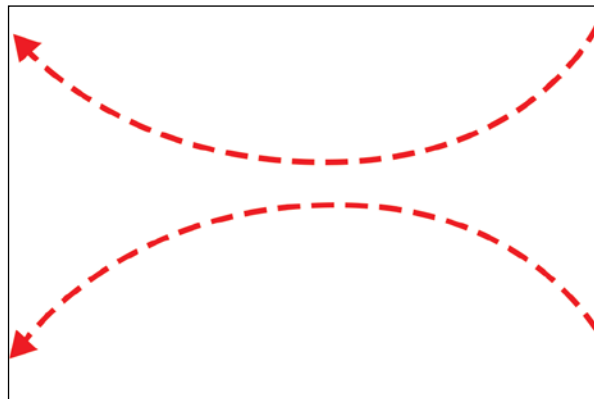
Understanding the JSON configuration file

Before we do anything with the `movementPatterns.json` file, let's understand what it will help us do and have a look at its structure. We used JSON before while importing sprite sheets, but this time, we'll be using some data designed specifically for this game.

A simple example

Previously, our enemies spawned in at random *y* coordinates and would move straight toward the left-hand side of the screen. Having completely randomized enemy movement can make the game feel unfair and chaotic. This may frustrate our players and prevents them from developing skills to improve their scores.

To change this, we're going to have enemies move in set patterns as follows:



In this example, we are going to have enemies spawn on the top and bottom of the screen, sweep into the middle of the screen, and sweep back to the top and bottom as they move off screen.

Looking at the JSON format

Now that we know what kind of pattern we want enemies to move in, let's have a look at some example JSON to see what kind of data we need to achieve this:

```
{
  "patterns": [
    {
      "speed":2,
      "timeOnScreen":2.75,
      "enemies":[
        {
          "startDelay":0,
          "startY":40,
          "points": [[1200,40],[480,320],[-240,40]]
        },
        {
          "startDelay":0,
          "startY":500,
          "points": [[1200,500],[480,320],[-240,500]]
        }
      ]
    }
  ]
}
```

At the top level, you will see that we have the word `patterns` followed by a colon and a square bracket. The text `patterns` indicates the name of the object, the colon acts like a variable assignment operator, and the square bracket defines an array.

So, in this case, we're going to have an array of patterns to choose from.

Each object in the array is wrapped in curly braces and then has a number of properties, such as `speed`, `timeOnScreen`, and `enemies`.

The `speed` property indicates the speed that each enemy will move at, `timeOnScreen` is the amount of time to wait before spawning in another pattern of enemies, and `enemies` is an array of objects that will define specific movement patterns for individual enemies.

Inside each object of the `enemies` array, we see three properties: `startDelay`, `startY`, and `points`.

The `startDelay` property indicates how long the enemy should wait before moving, `startY` is the y coordinate that the enemy should initially be at, and `points` contain an array of two elements of arrays.

Each array in the `points` array is a set of x and y coordinates that the enemy should tween to. In this example, each enemy has three sets of x/y coordinates, indicating the start, middle, and end positions the enemy should be at.

The way that we'll be moving enemies requires that we have at least three sets of coordinates, but you can have more coordinates if you so choose. The only stipulation is that you must have an odd number of coordinates. The reason for this will be explained later in this chapter.

The JSON file that has been provided with the chapter uses three movement patterns with many more enemies than this example, but this should help you understand what the data is when we go to implement it.

Building the enemy spawner

Now that we've gone over the data format, let's start implementing it. Luckily, the process to load and read this data is extremely easy due to functionality that Haxe provides. We're going to randomly choose which patterns of enemies to use, so there is still an element of chance to the game, but it will be far fairer.

Creating the enemy spawner class

To start, create a new class in the source folder named `EnemySpawner`. This class will extend `FlxTypedGroup`.

Here's what it should look like to start:

```
package;

import flixel.group.FlxTypedGroup;

class EnemySpawner extends FlxTypedGroup
{
    public function new(MaxSize:Int=0)
    {
```

```
        super(MaxSize);  
    }  
  
}
```

Before moving on, we'll make a few tweaks. Update the `extends` section so that `FlxTypedGroup` will use the `Enemy` class as follows:

```
class EnemySpawner extends FlxTypedGroup<Enemy>
```

We're extending the `FlxTypedGroup` class so that we can have a pool of `Enemy` objects similar to the way that we managed projectiles and explosions.

After this, remove the argument that's passed into the constructor and change the name of the argument passed into the super constructor to `numEnemies` as:

```
public function new()  
{  
    super(numEnemies);  
}
```

If you remember, the `FlxTypedGroup` object takes in a maximum size as an argument in its constructor. Our class will handle on its own with the `numEnemies` variable, so we'll just need to instantiate the `EnemySpawner` class when we need it and not worry about setting up properties on it.

Adding imports

Now, let's add the imports we'll need:

```
import haxe.Json;  
import openfl.Assets;  
import flixel.util.FlxTimer;  
import flixel.FlxG;
```

The one class to note here is the `haxe.Json` class. This will be used to parse our data file.

Adding variables

Next, let's add the variable we'll be using:

```
private var patternData:Dynamic;  
private var numEnemies:Int = 10;  
private var spawnTimer:FlxTimer;  
private var currentPattern:Int = 0;
```

Here's the explanation for the code:

- The `patternData` variable will store the JSON data that we'll load in. Once loaded in, we'll be able to access the data like a normal object with public properties.
- The `numEnemies` variable is used to define the maximum number of enemies to have pooled. We passed that into the super constructor earlier. This will also be used to prepopulate our enemies, so they can be recycled from the pool.
- The `spawnTimer` variable will be used to spawn patterns of enemies at timed intervals. The exact timing will be pulled from each pattern object in the JSON data.
- The `currentPattern` variable will be used to store the index of the current pattern, so we can access it easily.

Parsing JSON data

Now, let's parse our JSON data into the `patternData` variable. In the constructor, add this after the `super` call:

```
patternData = Json.parse(Assets.getText(
    AssetPaths.movementPatterns__json));
```

Here, we're loading in text information from the `movementPatterns.json` file using the `getText` function of the `Assets` class.

We then pass the `parse` function of the `Json` object to the text. Doing this will return a dynamic object that contains all of the data that was in the JSON file.

That one line is literally all we need to get JSON data into a usable state. Pretty awesome, right?

Populating the pool

Like other `FlxTypedGroup` objects, we need to initialize the objects in it. After the previous line, add this block:

```
var enemy:Enemy;
for (i in 0..numEnemies) {
    enemy = new Enemy();
    add(enemy);
    enemy.kill();
}
```

As we did before, we need to loop through, instantiate, and kill all of the objects that will be used by `FlxTypedGroup`. This will get our enemies ready to be pulled from the pool.

After this, let's create the timer and kick off the enemy spawning:

```
spawnTimer = new FlxTimer();
spawnEnemies();
```

Here, we're just creating our timer and calling the `spawnEnemies` function, which we'll create next.

Spawning enemies

Now, let's make the function that will spawn in enemies and give them their pattern data. After the constructor, add this function:

```
private function spawnEnemies(timer:FlxTimer = null):Void{}
```

This function will be called as a timer callback and in the constructor, so it takes in a `FlxTimer` object that defaults to `null`.

Inside the function, add the following line:

```
currentPattern = (Math.floor(Math.random() *
((patternData.patterns.length-1) - 0 + 1)) + 0);
```

To start, we'll decide a random pattern to be used. We do this in the same way that we've made random ranges before. The one difference here is that we're using `patternData.patterns.length` to determine the maximum value. This means that we're checking the length of the `patterns` array from the JSON data.

Just to nail home the point, we can now use the JSON data like we would with any other object. The `patterns` array from JSON gets converted to an array of Haxe dynamic objects. The `Json.parse` function will intelligently convert the data into types that Haxe understands. For another example, the `points` arrays in JSON will get converted to arrays that contain integer arrays.

Next, let's recycle and set up the enemies we'll need for the pattern:

```
var enemy:Enemy;
for(i in 0...patternData.patterns[currentPattern].enemies.length){
    enemy = recycle();
    enemy.startPattern(patternData.patterns[currentPattern].speed,
patternData.patterns[currentPattern].enemies[i]);
}
```

```
spawnTimer.start(patternData.patterns[currentPattern] .  
timeOnScreen, spawnEnemies);
```

Here, we're looping through the `enemies` array of the pattern we've randomly chosen. Inside the loop, we're recycling `Enemy` objects. We're calling a `startPattern` function on each enemy. This is a new function that will allow `Enemy` objects to use movement pattern data correctly.

This function will take in the speed we specified for the pattern and the enemy's pattern data from the current index of the `enemies` array.

After this, we start the timer using the `timeOnScreen` property for the pattern and set the `spawnEnemies` function as the timer's callback. This means that the `spawnEnemies` function will call itself forever.

Adding movement patterns to enemies

Now that we've loaded in our pattern data and have a pool for enemies to be spawned from, let's update our `Enemy` class to work with movement patterns and allow them to be recycled when they move offscreen or get shot.

Adding imports

To start, open the `Enemy` class and add this import:

```
import flixel.util.FlxFPoint;
```

The `FlxFPoint` object is an object that contains the *x* and *y* values. We'll be using this class to define each point that the enemy will move to.

Adding variables

Next, let's add the new variables we'll need.

```
private var movementSpeed:Float;  
private var movementPattern:Dynamic;  
private var movementPoints:Array<FlxFPoint>;
```

Here's the explanation of the code:

- The `movementSpeed` variable will store the speed value of the pattern. The `movementPattern` variable will store the pattern data specific to this enemy.
- The `movementPoints` array will store an array of `FlxFPoint` objects, and we'll convert the array of `points` from the JSON data to the `FlxFPoint` objects. This will allow us to use the data to tween the enemy along each point.

Removing old functionality

Before we go any further, we're going to have to remove some of the existing functionality we no longer need or will be doing in a different way.

First, delete the following variables:

```
private var maxSpawnTime:Float = 14;
private var minSpeed:Float = 2.0;
private var maxSpeed:Float = 3.5;
private var maxY:Int = 500;
private var minY:Int = 40;
```

Next, in the constructor, remove this line at the bottom:

```
resetSpawn();
```

Finally, delete everything inside the `resetSpawn` and `onSpawn` functions. We will be completely replacing the functionality in these two functions. Leave the functions there, and just empty them out for now.

Creating the `startPattern` function

Now, we'll make the `startPattern` function that gets called by the `EnemySpawner` class. To start, define the function as:

```
public function startPattern(speed:Float,pattern:Dynamic):Void {}
```

As mentioned before, this function takes in `float` for speed and a `dynamic` value for the pattern data.

Inside the function, initialize a few variables like this:

```
movementSpeed = speed;
movementPattern = pattern;
movementPoints = new Array<FlxPoint>();
```

Here, we're storing the speed and pattern values and initializing our `movementPoints` array that we will populate next.

So, after this, add this block:

```
var point:FlxPoint;
for(i in 0...movementPattern.points.length){
```

```

    point = new FlxPoint(movementPattern.points[i][0],
        movementPattern.points[i][1]);
    movementPoints.push(point);
}

```

Here, you see that we're looping through the `points` array for this enemy. It's an array that contains arrays, and this is often referred to as a **two-dimensional array**.

When instantiating a `FlxPoint` object, you pass in the `x` and `y` values to its constructor.

In this case, we know that each bottom-level array will contain two integers, so we can safely use index 0 and 1 to create the `FlxPoint` object.

After instantiating, we push the `FlxPoint` object into the `movementPoints` array.

Next, let's start our `spawnTimer` and initialize the enemy's position as follows:

```

spawnTimer.start(movementPattern.startDelay, onSpawn);
setPosition(FlxG.stage.stageWidth + width,
    movementPattern.startY);

```

We use the `startDelay` from the `movementPattern` object to set our spawn time. The `spawnTimer` object still calls the `onSpawn` function.

After this, we still set the starting `x` value to the `stageWidth` plus the width of the sprite; however, we now use the `startY` property from the `movementPattern` object.

Updating `onSpawn` and `resetSpawn`

Now that we've got our `startPattern` function made, let's update the `onSpawn` and `resetSpawn` functions with their new functionality.

First, go down to the `onSpawn` function and add this line:

```

movementTween = FlxTween.quadPath(this, movementPoints,
    movementSpeed, true, { ease: FlxEase.quadIn,
    complete: resetSpawn } );

```

Instead of using the normal `tween` function to move, we're now using the `quadPath` function to handle the tween.

The `quadPath` function works mostly the same as the `tween` function, but instead of an object detailing the properties to manipulate, we pass in an array of `FlxPoint` objects; in this case, it is the `movementPoints` array.



There are also a couple other things to note. The `quadPath` function will move enemies on a *quadratic path*, which is a curved path, so enemies won't move straight from point to point.

Using a quadratic path also requires you to use an odd number of points to move along. Therefore, there has to be a minimum of three points to move along.

Although this will create a curved path, you can still move straight along the X or Y axis as long as all of your points have the same X or Y coordinates. For an example, one of the patterns in the JSON file follows straight paths.

For more information on how quadratic paths work, check out this Wikipedia article at https://en.wikipedia.org/wiki/B%C3%A9zier_curve#Quadratic_curves.

Besides using a different type of tween, we're still using the same ease parameter and we're still calling `resetSpawn` when the tween is complete.

Next, update the `resetSpawn` function with this block:

```
this.solid = true;
this.alpha = 1;
this.kill();
```

So, we're resetting the `solid` and `alpha` properties to their starting values and then we're calling `kill` to enable the enemy to be recycled. This is all we need to do in order to reset the enemy.

Updating PlayState

Now that we have our `EnemySpawner` class and we've updated our `Enemy` class to work with it, we need to go and change a few things in `PlayState` to make it all work. Open up the `PlayState` class, and we'll get to work.

Removing old functionality

To start, we'll go through and remove things we no longer need. First, remove the following variable:

```
private var numEnemies:Int = 20;
```

Next, go into the constructor and remove the block where we're instantiating enemies:

```
var enemy:Enemy;
for (i in 0...numEnemies) {
```

```
enemy = new Enemy();  
enemyLayer.add(enemy);  
}
```

The spawner will do this for us, so `PlayState` is no longer responsible for handling enemies like this.

Using EnemySpawner

Now we just need to change our enemy layer to use our `EnemySpawner` class. First, update the variable:

```
private var enemyLayer:EnemySpawner;
```

Then in the constructor, make sure that we're instantiating the `EnemySpawner` class instead of a `FlxGroup` class:

```
enemyLayer = new EnemySpawner();
```

Since the `EnemySpawner` object is still a kind of group, all of the collision detection functionality can remain. Essentially as far as the rest of the game is concerned, enemies just move differently, but otherwise work pretty much in the same way.

And with this, you can give the game a run to see enemies flying in patterns!



Understanding the XML file

We've been working with XML many times over the course of this book. Any time we needed to change project settings, we had edited the `Project.xml` file. Now we're going to see how to use our own XML file.

The `strings.xml` file will allow us to change all text in the game from one spot, so if there's a typo somewhere, you won't have to hunt through the codebase to find it. This will also make it easier to translate the game into another language, as you can swap out the XML with a translated version very quickly.

Looking at the file's contents

Let's have a look at the `strings.xml` file to see what's in it:

```
<?xml version="1.0" encoding="utf-8" ?>
<data>
  <string id="TAP_TO_START"><![CDATA[- TAP TO START -]]></string>
  <string id="HEALTH"><![CDATA[HEALTH: ]]></string>
  <string id="SCORE"><![CDATA[SCORE: ]]></string>
  <string id="BEST_SCORE"><![CDATA[BEST SCORE: ]]></string>
  <string id="PLAY_AGAIN"><![CDATA[PLAY AGAIN]]></string>
</data>
```

The first tag at the top of the file indicates the version of XML the file uses and the character encoding that will be used in it. Generally speaking, you rarely, if ever, have to modify that tag.

After this, we have a `data` tag that has several `string` tags. This is the section that contains all of the string data for the game. Each `string` tag has an `id` property, and its contents are wrapped in a `CDATA` tag.

The `id` property is what we will use to retrieve the string from the XML. We will make a `constants` file that has a constant for each `id` in the XML file.

The sections with `CDATA` tags are the actual string values that will be seen in-game. The `CDATA` tag is a type of tag that prevents things like special characters from breaking the XML parser. For example, I could write `<![CDATA[ /> ]]>` and not worry about it pre-emptively ending the tag.

So now that we have an understanding of what's in the file, let's start using it.

Pulling string data from XML

To get string data out of XML, we're going to create a singleton class that can be accessed anywhere to get at the strings and constants for each string's ID.

Building the Strings class

To start, create a new class in the `ui` folder named `Strings`. This class won't extend anything, and it should be in the `ui` package.

Here's what it should look like to start:

```
package ui;
class Strings
{
    public function new() {}
}
```

Next, add the following imports:

```
import haxe.xml.Fast;
import openfl.Assets;
```

The new class in use here is the `Fast` class, which is Haxe's XML handling class.

After this, add the code needed to use this class like a singleton:

```
public static var instance(get, null):Strings;
private static function get_instance():Strings {
    if(instance == null) {
        instance = new Strings();
    }
    return instance;
}
```

This works exactly like the singleton setup in the `SoundManager` class does. As such, we'll also need to make our constructor private:

```
private function new() {}
```

Handling XML data

Now that we have the basic class setup, let's add the functionality to load and read the XML data. Using XML in Haxe is little bit more work than using JSON, but it's luckily still pretty easy to work with.

First, we'll need a variable to store our `Fast` object:

```
private var fast:Fast;
```

The `Fast` class will take in parsed XML data and give us quick access to it. It works a little like the way parsing a JSON file into a dynamic object works. As the name of the class implies, it's a fast way to get at XML data.

Next, let's make an `init` function that will read in the XML data and load it into the `fast` object:

```
public function init():Void{
    var stringXML = Xml.parse(Assets.getText(
        AssetPaths.strings__xml));
    fast = new Fast(stringXML.firstElement());
}
```

This `init` function needs to be called before we try to access any string data due to the way it loads everything in. In our game, we'll be calling it on the title screen.

In this function, we're creating a temporary variable named `stringXML` and then passing in data from the `Xml.parse` function. This works similarly to the `Json.parse` function, and we pass in text from our data file into it.

After parsing, we instantiate our `Fast` object and pass in the data by calling the `firstElement` function on `stringXML`. What this does is grab the root XML node and use that to pull data from. In the case of our file, the root node is the `<data>` tag.

Now, let's make a function that will return the string data we need:

```
public function getValue(id:String):String{
    for(string in fast.nodes.string){
        if (id == string.att.id) {
            return string.innerData;
        }
    }
    return "";
}
```

This function takes in a string ID and return a `String` value. The `id` will be the `id` that's on each node in our file, and the returned value will be the value that's wrapped in the `CDATA` tag.

Inside the function, we use a `for each` loop to go through every `string` node in the XML data. In the loop, we check to see if the function's ID matches the ID of the tag. If it is a match, then the data is returned; if there is no match, the loop will continue. If no matches are found at all, an empty string is returned.

When accessing nodes using the `Fast` object, you call the `nodes` property, which gives you access to the nodes in the XML data. At this point, you can refer to the node by the name of its tag, so in this case, we're referring to the actual `string` nodes.

Calling the `att` property on the individual string objects gives you access to its attributes. In this case, we're accessing the `id` attribute.

Referring to the `innerData` property on the string object lets you access the data inside the `CDATA` section of the tag. In this case, this is the string data we're looking for.

Building the StringIDs class

Now we just need to make the constants file that will contain the IDs of every tag in the XML file.

To start, make a new class in the `ui` folder named `StringIDs`. This class won't extend anything, and it should be in the `ui` package.

Here's what it should look like to start:

```
package ui;
class StringIDs
{
    public function new() {}
}
```

Delete the constructor since you won't need it. After this, add the following constants:

```
public static inline var TAP_TO_START:String = "TAP_TO_START";
public static inline var HEALTH:String = "HEALTH";
public static inline var SCORE:String = "SCORE";
public static inline var BEST_SCORE:String = "BEST_SCORE";
public static inline var PLAY_AGAIN:String = "PLAY_AGAIN";
```

And that's all we need to access any of the strings in our XML file.

Replacing strings

Now we'll go and replace the strings that we have hardcoded throughout the game. There are strings in three classes in the game: `MenuState`, `GameHUD`, and `LevelEndScreen`.

Updating MenuState

Open up `MenuState` and add the following imports:

```
import ui.Strings;
import ui.StringIDs;
```

After this, go in the `create` function and add this immediately after the `super.create` function call:

```
Strings.instance.init();
```

This will load and set up our XML data. Then, we can just get at the data whenever we need it.

Now still inside the `create` function, go down to the line where the `text` property is set on `txtInstructions` and replace it with this:

```
txtInstructions.text =
    Strings.instance.getValue(StringIDs.TAP_TO_START);
```

That's all we need to do in `MenuState`.

Updating GameHUD

Open `GameHUD` and add the imports we need:

```
import ui.Strings;
import ui.StringIDs;
```

After this, go into the constructor and replace the line that mentions the `text` property on `txtHealthHeader` with this:

```
txtHealthHeader.text =
    Strings.instance.getValue(StringIDs.HEALTH);
```

Then, go to the line where the `text` property on `txtScore` is set and replace it with this:

```
txtScore.text = Strings.instance.getValue(StringIDs.SCORE) + 0;
```

Finally, go to the `setScore` function and replace the line in it with this:

```
txtScore.text = Strings.instance.getValue(StringIDs.SCORE)+ score;
```

Updating LevelEndScreen

There's one last class to update. Open `LevelEndScreen`, and add the imports:

```
import ui.Strings;
import ui.StringIDs;
```

Go to the constructor and replace the line where the `text` property on `txtGameScore` is set with this:

```
txtGameScore.text = Strings.
instance.getValue(StringIDs.SCORE) + score;
```

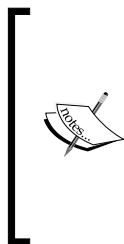
Next, go to the line where the `text` property on `txtBestScore` is set and replace it with:

```
txtBestScore.text =
Strings.instance.getValue(StringIDs.BEST_SCORE) + Reg.score;
```

Finally, go to the line where `btnPlayAgain` is instantiated and replace it with this:

```
btnPlayAgain = new FlxButton(0,
0,Strings.instance.getValue(StringIDs.PLAY_AGAIN), onPlayAgain);
```

Now with this, we're done! If you run the game now, it should look exactly as it did before, but now all of the text is customizable. It may be a little anticlimactic, but it will save you some serious headaches if the text in the game needs to change.



It's also worth noting that there is some text in the game that can't be changed through this method. In our case, those are the `levelEndHeader.png`, `newBestScore.png`, and `titleScreenLogo.png` images.

These are images with specific designs on them and therefore aren't easily changeable. In cases like this, the game's art assets will need to change if the text needs to change.

Summary

In this chapter, you learned how to load and use both the XML and JSON data to make your game more modular and configurable. This is also the last bit of functionality we'll be adding before deployment. The game is effectively feature-complete now!

I suggest you to play with both data files to get a better understanding of them. You can add tons of new enemy movement patterns to add challenges to the game. You can also swap out all text in the game with whatever you feel like writing in. Give it a shot!

In the next chapter, we'll take our feature-complete game and build it so that it can be played on desktop, mobile, and web platforms!

7

Deploying to Multiple Platforms

Now that we have our game feature complete, it's time to start deploying our game to our target platforms. In this chapter, we're going to build and run our game on five different platforms: Flash, Windows, OS X, Android, and iOS. Since we did a lot of the setup for these platforms in *Chapter 1, Getting Started*, many of the platforms will be very simple to run.

Here's a rundown of the sections in this chapter:

- Building for Flash
- Building for Windows
- Building for OS X
- Building for Android
- Building for iOS

Building for Flash

Flash is by far the easiest platform to target when building most things in Haxe. We've also been using it for development all along. The only difference now is that we'll be building in release mode, which will strip out debug information.

If you're using FlashDevelop on Windows, you can just run the game by selecting the release configuration and then choosing the Flash target build.

If you're not using FlashDevelop, you can just run this command:

```
lime test flash -release
```

The only difference from what we've been doing all along is the `-release` flag, which specifies that you want a release build.

Deploying to the Web

Since Flash is typically considered as a web platform, it's natural to want to deploy your game to the Web. All you need to do is use some simple HTML to embed the `.swf` file that Haxe compiles in a web page.

I've included a sample HTML file in the assets for this chapter. To use the HTML file, copy `index.html` from the provided assets to the `bin` folder under `export/flash`. Ensure that the `.swf` file in the folder is named `StarWalrus.swf` and then open the HTML file in your browser of choice.

If the game doesn't load, make sure that you have the Flash plugin installed in your browser, and it is enabled. Some browsers will disable the Flash plugin if it hasn't been updated.

If you want to put the build-up on the Web, you just need to upload the `.swf` and `.html` files to a web server.

Building for Windows

Building for Windows is also very simple. In *Chapter 1, Getting Started*, you installed Visual Studio, which is the only thing you need to build for Windows. Keep in mind that you do need to run the Windows operating system in order to build the game.

If you're using FlashDevelop on Windows, you can just run the game by selecting the release configuration and then choosing the Windows target build.

If you're not using FlashDevelop, you can just run this command:

```
lime test windows -release
```

The build files are located in the `bin` folder `export/windows/cpp`. All of the files in that folder are needed to run the game.

Building for OS X

Targeting OS X is also very simple. It requires that you have Xcode installed, which we covered in *Chapter 1, Getting Started*. You do need to have a Mac computer running OS X in order to build the game for OS X.

To build your game, run this command:

```
lime test mac -release
```

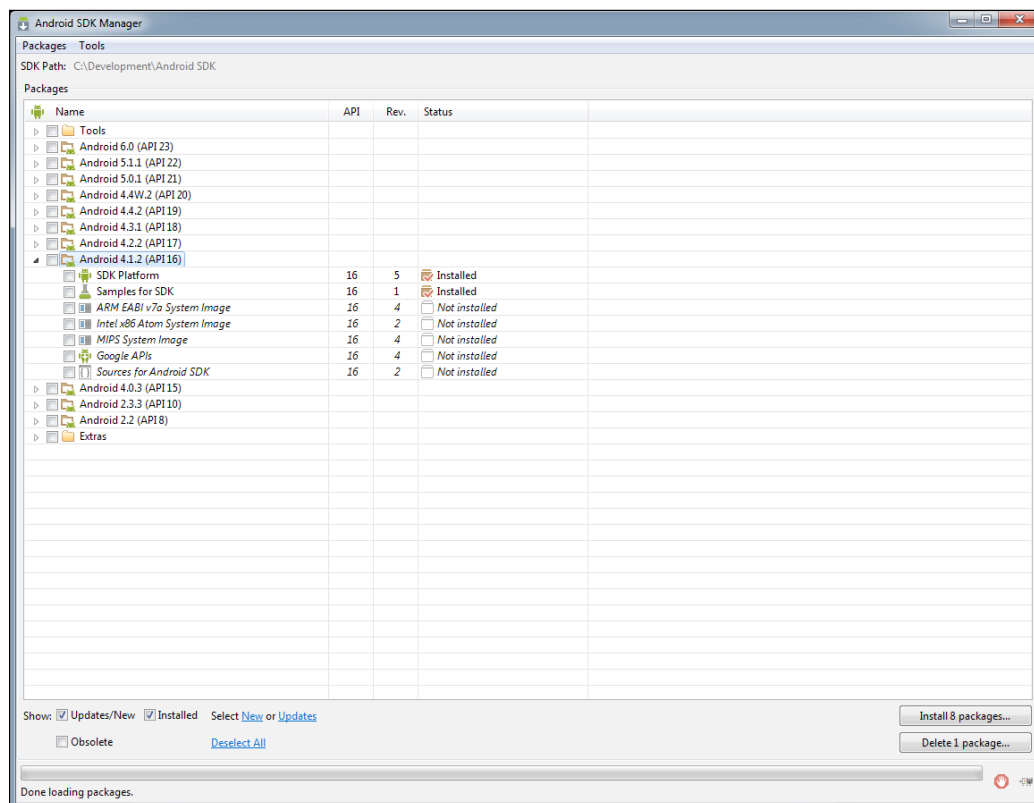
This will build and run an OS X version of the game. The application is located in the `bin` folder under `export/mac64/cpp` and will be named `StarWalrus.app`. It's self-contained so that one file alone can be distributed.

Building for Android

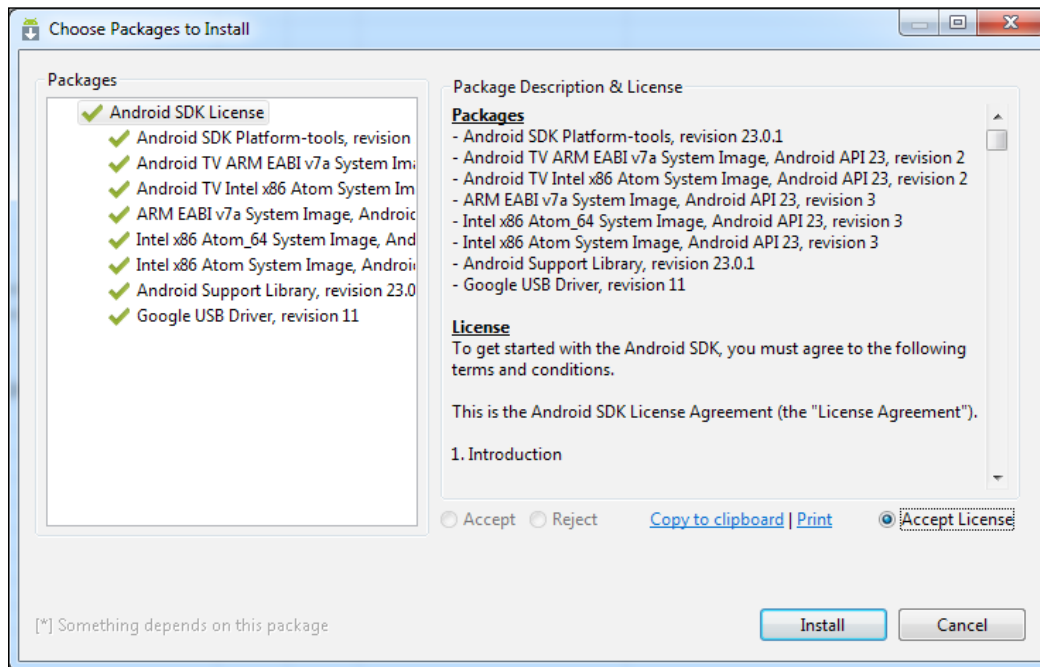
Building for Android is fairly simple, and the only extra step needed is to download an Android SDK for a specific API version. HaxeFlixel defaults to using Android API version 16, specifically, so we're going to get it now.

In the folder where you installed the Android SDK, locate an executable named `SDK Manager` and run it.

Navigate down to the section labeled **Android 4.1.2 (API16)** and check the box next to **SDK Platform**. Then, hit the **Install packages...** button.



This will bring up a window to accept license terms. Hit the **Accept License** radio button and then hit **Install**. Wait for the files to download and install, then you can build for Android.



If you're using FlashDevelop on Windows, you can just run the game by selecting the release configuration and then choosing the Android target build.

If you're not using FlashDevelop, you can just run this command:

```
lime test android -release
```

This will create Android project files and an APK that you can install on a device. It will also try to run the game on a device if you have one connected to your computer and it's setup for development.

To install manually, go to the `bin` folder under `export\android\bin` and copy the `StarWalrus-debug.apk` file to your Android device. Then, open the `.apk` file on the device to start the install process. You may need to enable installation from unknown sources.

The project files that are generated can be opened in your Android IDE of choice. You can use this to get the build ready for deployment to the various Android application stores.

Building for iOS

We're now going to cover building for iOS. This is definitely the most complicated platform to build for but once you have everything set up, it's rather easy to work with.

Most of the work here actually lies with the setup on the Apple developer site and Xcode. The work on the Haxe side of things is pretty simple. If you've built apps or games for iOS in the past and already have a wildcard-provisioning profile, you can skip to the section titled *Running the game on iOS*. Building for iOS requires the following things:

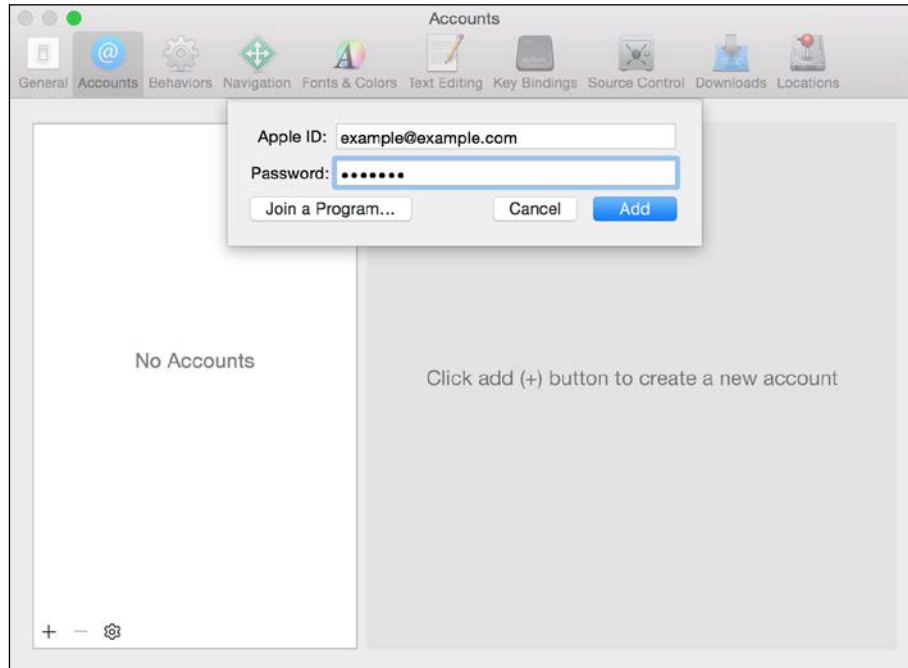
- An Apple computer running a version of OS X, typically to support the newest versions of iOS. This also means running the latest version of OS X, which is Yosemite at the time of writing.
- An Apple developer account, and you can enroll at <https://developer.apple.com/programs/>.
- An iOS device, preferably something that's running one of the newer versions of iOS. This is because iOS users tend to upgrade fairly fast, and Apple requires you to support the newest versions when submitting to the App Store.

Xcode setup

In the first chapter, you downloaded and installed Xcode. If you haven't run it yet, be sure to do so because it will need to initialize and may need to pull down SDK files that let you build.

Once Xcode is set up and is running, you'll need to login to your developer account. To do this, go to the title bar menu, then navigate to **Xcode | Preferences**. Then, click on the **Accounts** tab and hit the **+** button and then select **Add Apple ID...**

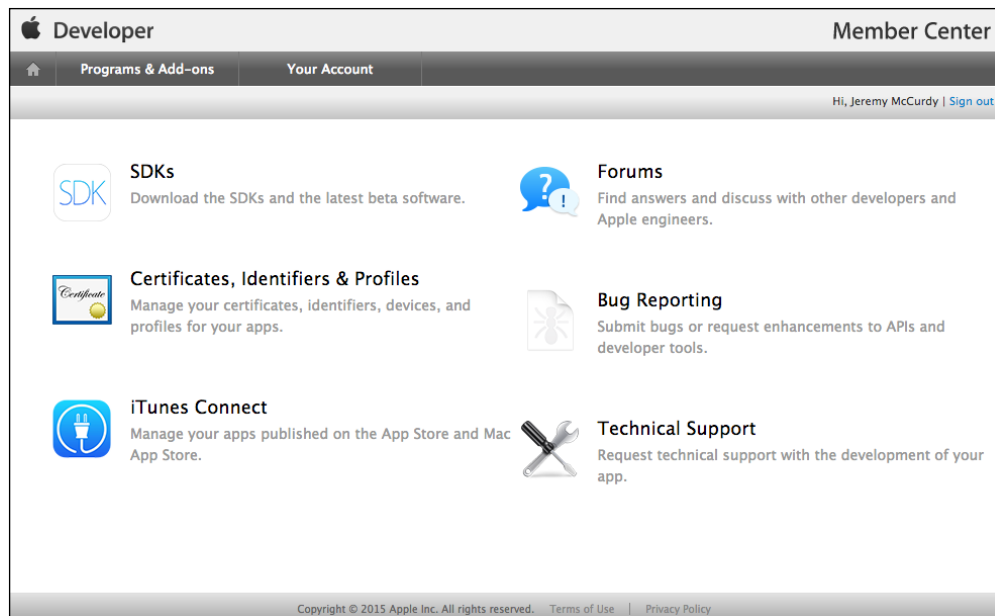
This will prompt you for your credentials; enter them and you are now logged in to your developer account. This will enable Xcode to search and find the things it needs from the developer portal to build your game.



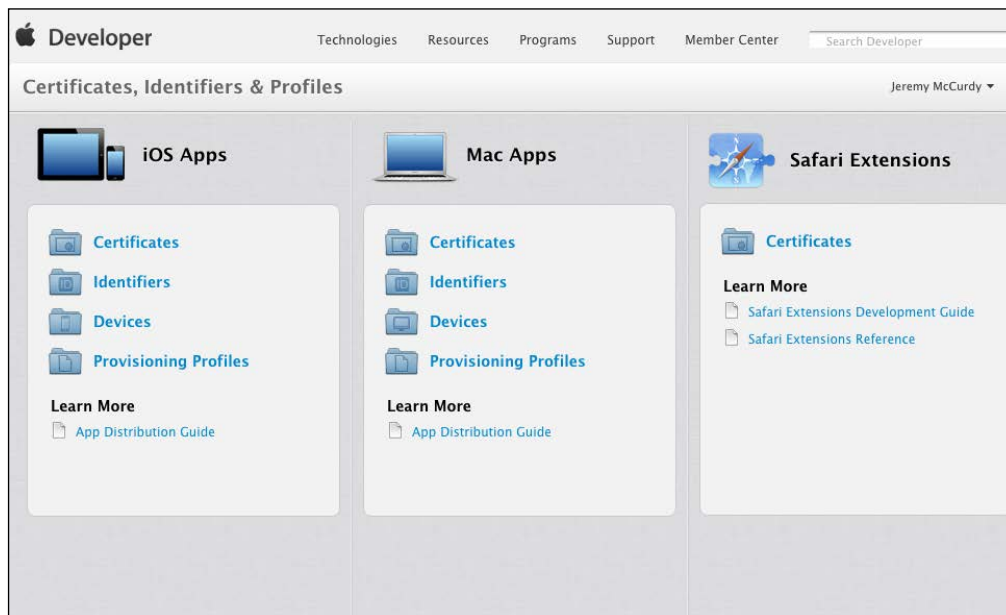
With this, Xcode is set up and is ready to use.

Setting up a certificate

Now we need to set up our things we need to have in order to build for iOS. In a web browser, log in to the Apple developer site at <http://developer.apple.com>. Click on **Certificates, Identifiers & Profiles**.



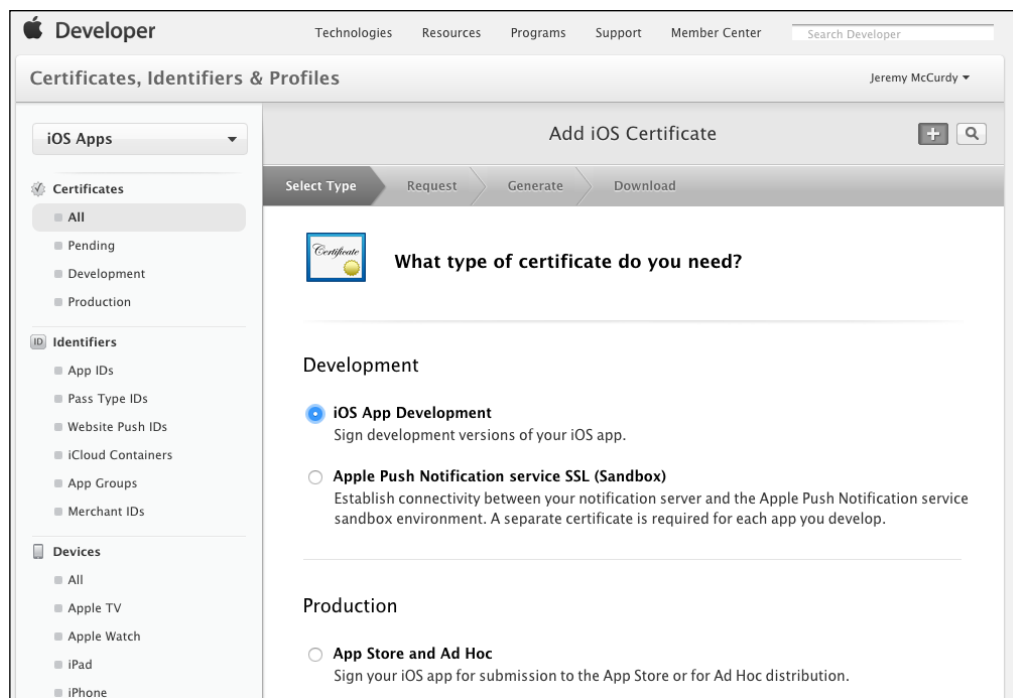
You'll then be taken to that section of the site, under the **iOS Apps** section. Click on **Certificates**, as shown here:



Now you'll need to create a certificate that will be used to sign apps you create. This essentially ensures that Apple knows that you are the person who created the application. To start, click on the link that says **Certificate Signing Request**.

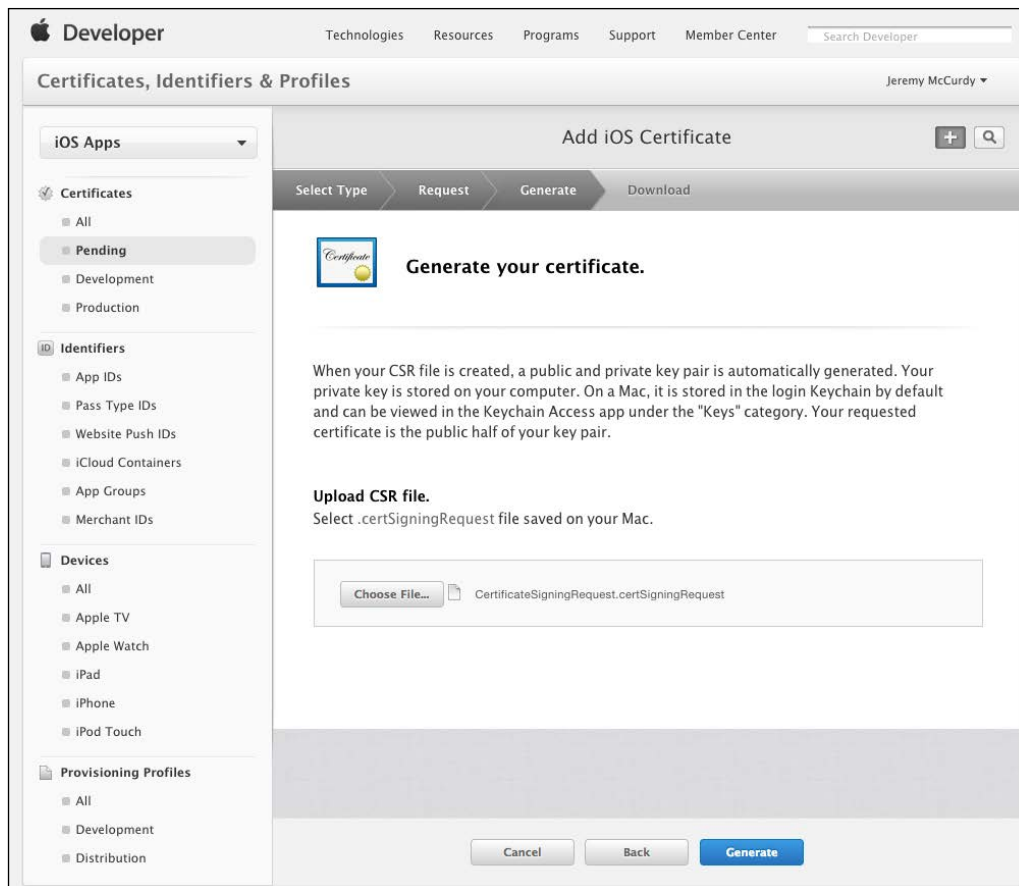
On the next page, select **iOS App Development** under the development heading. This indicates that you are going to use the certificate while you develop the game.

If you wanted to make a build of the game that could be submitted to the App Store, you would need to select **App Store and Ad Hoc** under the production heading. We won't be covering App Store submission, but you now know where to look if you decide to submit a game.



After this, click on the **Continue** button at the bottom of the page. This will take you to the **About Creating a Certificate Signing Request** section; follow the instructions on this page to create a certificate signing request and then click on **Continue**.

On the next page, click on the **Choose File...** button to select the `.certSigningRequest` file you created. Hit the **Generate** button to continue.



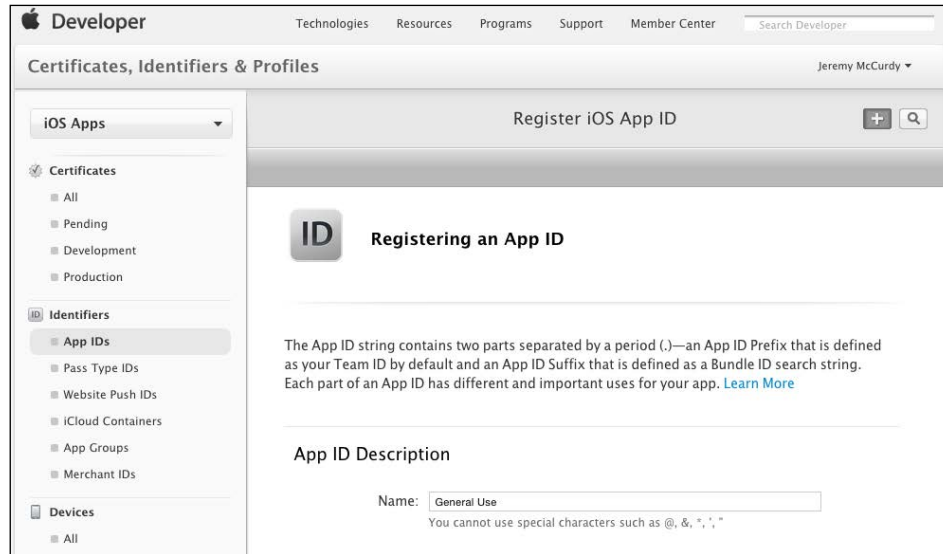
Finally, you'll be taken to a page where you can download your certificate. Click on the **Download** button and save the certificate somewhere on your file system. Navigate to the location where you downloaded the file and double-click on it. This will add the file to your computer's keychain.

Setting up an App ID

Now we're going to create an App ID. This is used to uniquely identify your app although we're going to create a general-purpose wildcard ID. This will be an ID that you can use on anything you work on until you need to submit to the App Store.

To start, click on the **App IDs** section under **Identifiers** on the sidebar. Then, click on the link that says **Register your App ID**.

Next, enter a name that will describe the App ID. Since this will be a general use App ID, name it `General Use` or something else that will make sense to you later.



After this, go to the **App ID Suffix** section and select **Wildcard App ID**. Then, just type in an asterisk (*) for the Bundle ID. This will allow any app with any bundle identifier to work with this App ID.

Note that in order to submit your app to the App Store, you will need to create an explicit App ID because Apple uses this to identify specific apps. This also means it must be a totally unique ID.

App ID Suffix

☐ **Explicit App ID**

If you plan to incorporate app services such as Game Center, In-App Purchase, Data Protection, and iCloud, or want a provisioning profile unique to a single app, you must register an explicit App ID for your app.

To create an explicit App ID, enter a unique string in the Bundle ID field. This string should match the Bundle ID of your app.

Bundle ID:

We recommend using a reverse-domain name style string (i.e., com.domainname.appname). It cannot contain an asterisk (*).

☒ **Wildcard App ID**

This allows you to use a single App ID to match multiple apps. To create a wildcard App ID, enter an asterisk (*) as the last digit in the Bundle ID field.

Bundle ID:

Example: com.domainname.*

After this, scroll down to the bottom and hit the **Continue** button. On the following page, just hit the **Submit** button.

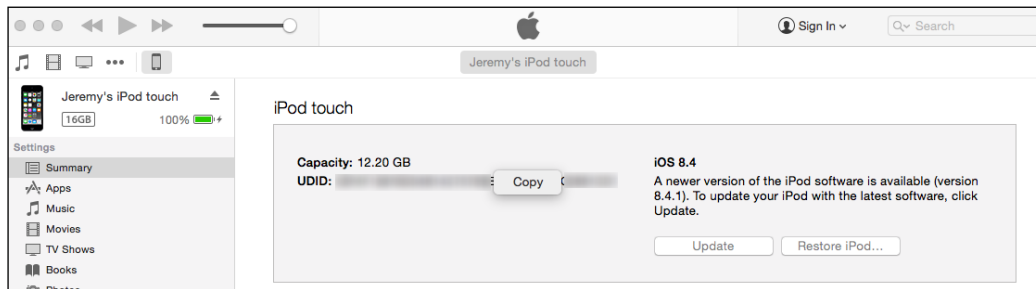
Registering a device

When building your app to iOS, you have to have at least one device to complete the process. Xcode has an iOS simulator that will let you test out different devices to check things such as screen layout, but in order to do a meaningful development, you have to have at least one iOS device.

So, we've now going to go through the process of registering an iOS device. To start, we're going to need a **UDID**, which stands for **Unique Device Identifier**. To get the UDID, plug your iOS device into your computer and open iTunes.

Once started, click on the device icon on the bar below the playback controls. This will take you to the summary screen for the device.

Once you're on this screen, click on the serial number. It will change to say UDID and show you the UDID value. Right-click on it and click on the **Copy** button, and this will copy it to the clipboard.



Now go back to the developer site and click on **All** under **Devices**. Click on the **Add** button to go to the device registration screen.

On the device registration screen, enter a descriptive name and paste it in your UDID. Click on **Continue** to complete the registration.

Register Device
Name your device and enter its Unique Device Identifier (UDID).

Name:

UDID:

Please enter a valid Hardware UDID

There is one thing to keep in mind: once you register a device, it will remain registered for at least one year. You will get an e-mail from Apple after a year with the option to remove devices if you choose so. This is the only time you can remove a device from your account for the year.

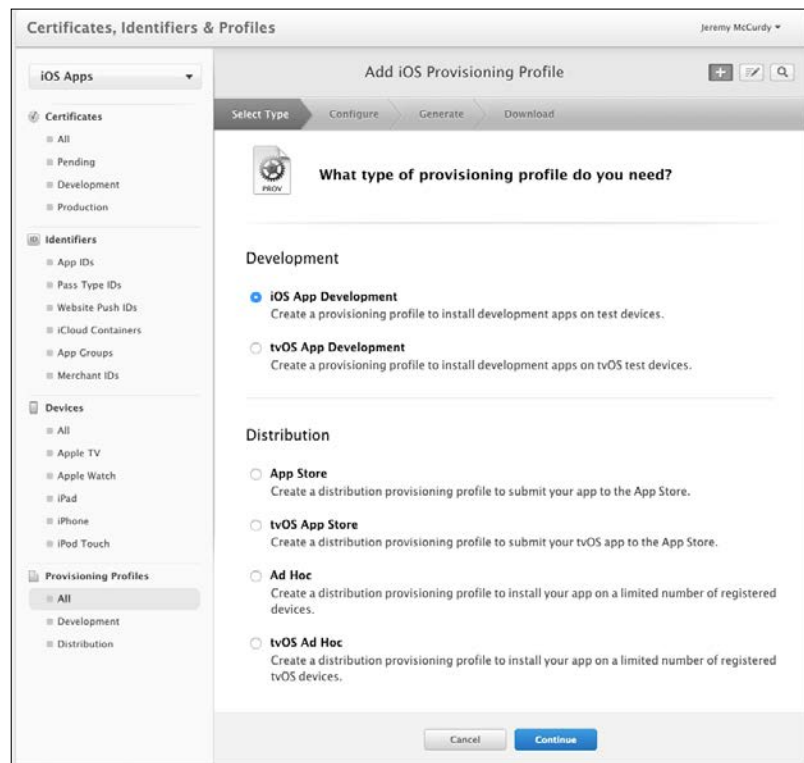
It's also worth noting that registering your device on the developer site doesn't lock it to your developer profile. The device can be registered with multiple developers at a time. You're just registering it so that you can use it for your games and apps.

Creating a provisioning profile

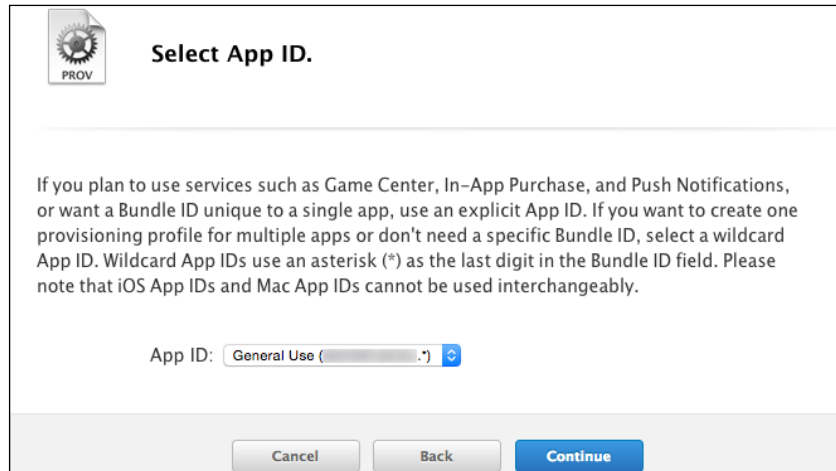
Now we're reaching the last step – creating a provisioning profile. A provisioning profile is what Xcode will use to create builds. It will contain all of the information we have entered so far.

To start, click on **All** under **Provisioning Profiles** and then click on **Manually generate profiles**.

Next, click on **iOS App Development** under the **Development** heading and click on the **Continue** button.

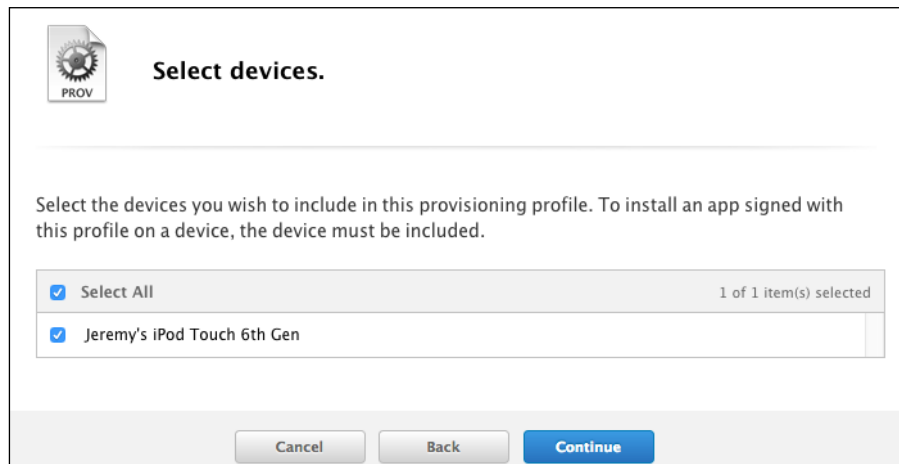


On the next screen, if the App ID you just created isn't selected, pick it from the drop-down box and then click on **Continue**.

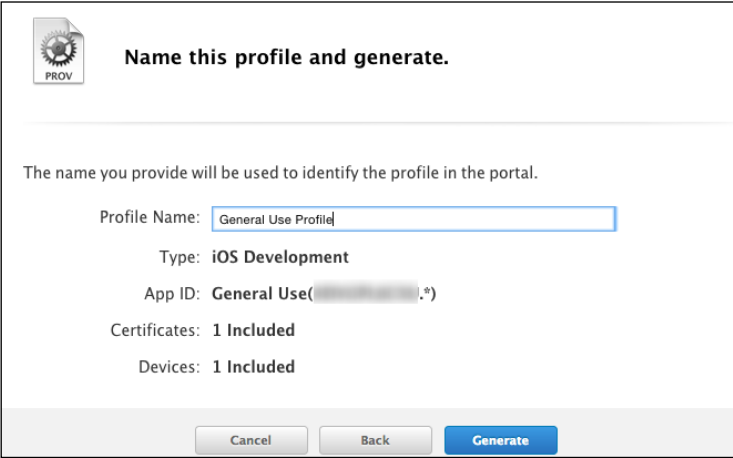


On the following screen, select the certificate you created earlier and click on **Continue**.

On the next screen, choose the device you registered. Here, you're picking the devices your game will run on for development purposes. So, if you have several devices you want to run the game on, you should register and select them all. Once your device is chosen, click on the **Continue** button.



Now you just have to enter a name for your profile. Since this profile won't be specific to one game or app, be sure to give it a general-purpose name. After entering a name, click on **Generate**.



Name this profile and generate.

The name you provide will be used to identify the profile in the portal.

Profile Name:

Type: iOS Development

App ID: General Use()

Certificates: 1 Included

Devices: 1 Included

Finally, you just need to download your profile. Click on the **Download** button and choose a location to save it. Then, navigate to the file's location and double-click on it; this will load it into Xcode, and it is now ready to use.

Running the game on iOS

Once you have everything set up in Xcode and the developer site, building and running the game is actually very easy. Ensure that your device is plugged in and open a terminal window and navigate to the game's root folder.

Once you're there, run this command:

```
lime test ios - release
```

The game should build and then immediately run on your device. It's also installed on the device; if you close the game, you can run it again later.

If you navigate to `export/ios` under your project's root directory, you'll see that the compiler has created an Xcode project for you. You can open this project inside Xcode and tweak it as needed; you can also test the game at various screen sizes using the simulator. Just bear in mind that the simulator will run the game more slowly than on an actual device. So, don't rely on it in order to check performance.

Summary

In this chapter, you learned how to build the various platforms that HaxeFlixel supports. You experienced how easy it is to build games that run on these platforms with no code changes and minimal amounts of work. You also learned the essentials of setting up an iOS application, which have some very broad uses.

To continue learning, I suggest looking into the project files created for both Android and iOS, looking at what you can change, and playing around with running them through their respective IDEs.

In the next chapter, we'll go over what we accomplished with our game so far and talk about features that could be added to enhance the experience and build up what's been learned. After this, you will be directed to additional features of Haxe, OpenFL, and HaxeFlixel to check out and grow more as a game developer.

To finish things off, you will be directed to useful sites and resources to keep learning.

8

What's Next?

You've come a long way. Over the course of this book, you learned about the Haxe language, OpenFL, and HaxeFlixel. You learned how to build gameplay and user interfaces and implemented audio, configuration files, and animations. You topped it off by learning how to build your game for five different platforms!

You've taken the steps needed to start diving deeper into Haxe, OpenFL, and HaxeFlixel. We're now going to have a look at ways the game could be expanded and changed, talk about some advanced features you can take advantage of, and look at some useful resources to continue learning. Here's a rundown of the sections in this chapter:

- Expanding the game
- Exploring advanced features
- Continuing with learning resources

Expanding the game

Star Walrus was a game built to help you learn Haxe; it's funny and silly, but it can be more. At the end of each chapter, I suggested ways to tweak the game or expand on what you've learned, but the sky is the limit. To help with this, I've included the original art files used to make the elements in the game. They require Adobe Illustrator and Photoshop.

Let's go over a few possibilities.

Level-based gameplay

Loading in predefined enemy patterns works for the quick pick up and play experience, but you could add more depth to the experience by expanding the game to take place over several predefined levels.

You've learned to work with both JSON and XML; either would be excellent to build out levels. To make your life a little easier, you could have levels composed of enemy patterns instead of making every single thing in every level unique. All you would need to do is add an `id` property to the patterns and reference those in an integer array.

Reusing prebuilt elements like that is a common practice that saves you time and makes levels easier to balance.

Power-ups and collectibles

Most side-scrolling shooters have power-ups and things you collect. They help give the players extra motivation beyond just getting a high score.

Having a power-up that doubles your shoot speed would be very easy to do. If you wanted to go crazy, you could add different types of projectiles that fire in different patterns. How cool would it be to have a spread shot like in *Contra*?

For collectibles, collecting in-game currency would make sense. It could be used just to increase score, or could be spent to increase the walrus' move or shoot speed.

You could also have health packs that heal you or screen-clearing bombs that kill all enemies on screen when you collide with them. Putting collectibles in spots that are dangerous to approach would also introduce an element of risk/reward gameplay.

More enemies and bosses

For the sake of simplicity, we kept the game to a single enemy type, but there's nothing stopping you from adding more.

You could add enemies that fire back at you or follow you around screen for a short period of time. Having things that require the player to change up their tactics keep them engaged and having fun.

Boss fights would also add a lot to the game. It's often quite a challenge to script out a compelling boss fight, but a good boss fight will stay in a player's memory for a lifetime.

I would suggest having a health UI for a boss or alternatively have a boss character composed of multiple pieces that can become visibly damaged over time.

Communicating boss fight progress to the player is very important. If you don't, the boss will feel like a bullet sponge and become tedious really fast.

Exploring advanced features

HaxeFlixel has a ton of features we didn't touch on in this book. The engine is frequently updated and has a large community of contributors. This gives you a lot of power and a lot of things that are just fun to play with. Let's go over a few.

Gamepads

HaxeFlixel has built-in support for gamepads such as the Xbox 360 controller. This means that you can create more hardcore console style experiences if you choose.

There is explicit support for Xbox, PS3, PS4, Ouya, and Logitech controllers. To learn more about HaxeFlixel's controller support, check their docs page at <http://haxeflixel.com/documentation/gamepads/>.

Physics

The add-ons library for HaxeFlixel adds support for the Nape Physics Engine. This enables you to create 2D rigid body simulations. Games such as Angry Birds use similar physics engines to create their chaotic and silly physics-based experiences.

To learn more about Nape and what it can do, check it out at <http://napephys.com/>.

Tilemaps

HaxeFlixel supports tile maps to build out levels. This enables you to use visual tools to build out levels; this can also enable nonprogrammers to contribute to your game because they can build levels for your game while you add functionality.

You can see an example implementation of a tilemap at <http://haxeflixel.com/demos/TileMap/>.

If you want to take it a step further and use a visual tool, you can implement support for the Tiled Map Editor, which is a completely free tile map editor. You can get tiled at <http://www.mapeditor.org/>.

You can see an example implementation of a Tiled Editor map at <http://haxeflixel.com/demos/TiledEditor/>.

Continuing on with learning resources

We're now going to look at some resources to continue learning. We'll cover Haxe, OpenFL, and HaxeFlixel, and a couple of good community resources.

Haxe

The Haxe website has an excellent manual that goes into great detail about how the language's syntax works. If you're ever unsure of the correct way to write something in Haxe, I suggest you to check out the manual at <http://haxe.org/manual/>.

There's also a well-documented API for Haxe. It's the perfect place to understand what each core Haxe class does. It also lists the platforms that each class is available in. Since Haxe supports such a vast number of platforms, you don't get every class on every platform, so having documentation about it is crucial. You can access the API at <http://api.haxe.org/>.

If you want to start using additional Haxe libraries, you can refer to the documentation on Haxelib at <http://haxe.org/manual/haxelib.html>.

You can find all of the libraries that are publicly available through Haxelib at <http://lib.haxe.org/>.

OpenFL

The OpenFL site is equally useful when learning about OpenFL-specific functionality. It's worth noting that even though we used a game engine built on top of OpenFL, it's entirely possible to write entire games and applications using just OpenFL with relative ease.

To learn more about OpenFL, check out their learning section at <http://www.openfl.org/learn/>.

The OpenFL site also has an API section. Part of its API is copied from the Haxe API, so it might be a little tricky to find OpenFL-specific classes, but all you have to do is stick to the `openfl` and `lime` packages in the API.

You can find the OpenFL API at <http://docs.openfl.org/>.

HaxeFlixel

We already looked at some HaxeFlixel resources in the *Exploring advanced features* section, but let's look a little more deeply.

To start, the HaxeFlixel website has very solid documentation. They explain many of the engine's features, and some of the *gotchas* you can encounter in different use cases. It's worthwhile to read their docs top to bottom. You can find their documentation at <http://haxeflixel.com/documentation/>.

HaxeFlixel also has an API section, and it builds on top of the OpenFL API, so it contains entries for it and also for Haxe in general. If you're looking for HaxeFlixel-specific classes, stick to the `flixel` package. You can access the API documentation at <http://api.haxeflixel.com/>.

There are some things that aren't clearly detailed in the manual; since HaxeFlixel does so much, it's very difficult to document every single thing. This being said, they do have a demo section on their site that shows off various cool features and how to implement them.

Each example links to a GitHub page where you can read the code and see what makes it tick. You can access the demos at <http://haxeflixel.com/demos/>.

Community resources

The Haxe community is growing at a steady pace, and there are a lot of helpful folks out there helping to spread information about Haxe and its various frameworks.

One of the biggest voices in the community is the Haxe Roundup, a site dedicated to giving weekly updates on what's new and cool in the world of Haxe. There's a lot to learn from there, and it's great to keep a finger on the pulse of the community. You can find the Haxe Roundup at <http://haxe.io/>.

There are also forums for Haxe, OpenFL, and HaxeFlixel. These forums have lots of people posting with questions and support. They're excellent places to post and ask questions if you're stuck on something.

You can find the Haxe Google Group at <https://groups.google.com/forum/?hl=en#!forum/haxelang>.

The OpenFL forum is at <http://community.openfl.org/>.

The HaxeFlixel Google Group is at <https://groups.google.com/forum/?hl=en#!forum/haxeflixel>.

Summary

In this chapter, we went over steps to add more life to Star Walrus, some cool features to explore with HaxeFlixel, and looked at a bunch of resources to continue down the path of the Haxe Developer. It's been a bit of a journey to get here, but you now have the knowledge you need to become a top-notch Haxe developer.

I greatly appreciate you reading this book, and I hope that you had as much fun following along as I did writing it.

Index

A

access modifiers

- inline 16
- private 16
- public 16
- static 16

advanced features

- exploring 153
- gamepads 153
- Physics 153
- Tilemaps 153

Android 6, 7, 137

Apple developer account

- URL 139

Apple developer site

- URL 140

assets

- adding 38, 62
- default state, changing 38, 39

audio folder

- about 100
- music_fanfare 100
- music_ingame 100
- sound_click 100
- sound_damage 100
- sound_explosion1 100
- sound_explosion2 100
- sound_shoot 100
- sound_whoosh 100

B

background

- adding 22, 23

basic UI

- adding 22, 23

C

community resources 155

D

data folder

- movementPatterns.json 117
- strings.xml 117

data types

- Array 15
- Bool 15
- Dynamic 15
- Float 15
- Int 15
- Map 15
- null 15
- String 15
- Void 15

E

end screen, creating

- about 29
- background, adding 30, 31
- end screen class, creating 29, 30
- score text, adding 31
- variables, adding 30

enemies

- animation, adding 77
- death functionality, adding 79
- imports, adding 76

- onSpawn function, updating 79
- spawning 122
- spawn position, resetting 78
- updating 75
- variables, adding 76

enemies, creating

- about 24
- display, setting up 25
- enemy class, creating 24
- spawn timer, creating 26
- variables, adding 25

enemy spawner

- building 119
- enemy spawner class, creating 119, 120
- imports, adding 120
- JSON data, parsing 121
- pool, populating 121
- variables, adding 120

explosion effects

- animation, completing 70, 71
- class, creating 63
- creating 63
- everything, adding to group 68
- explosion bulb 63
- explosion bulb, displaying 69
- explosion line 63
- explosion sequence, starting 68
- imports, adding 64
- object pooling, preparing for 72
- particle emitter, creating 67
- smoke, displaying 70, 71
- smoke particle effect 63
- sprites, adding 66
- Texture Packer data, setting up 65, 66
- variables, creating 64, 65

F

Flash

- about 5, 135
- debugging 6
- reference 5

FlashDevelop 135

G

game

- bosses 152
- collectibles 152
- enemies 152
- expanding 151
- level-based gameplay 151, 152
- power-ups 152
- running, on iOS 148

game, building for Android 137, 138

game, building for Flash

- about 135
- deploying, to Web 136

game, building for iOS

- about 139
- App ID, setting up 143-145
- certificate, setting up 140-143
- provisioning profile, creating 146-148
- Xcode setup 139, 140

game, building for OS X 136

game, building for Windows 136

gameplay sounds

- adding 110, 111
- functions, creating 112
- playback functions, calling 112, 113
- variables, creating 111

game timer, implementing

- about 33
- end screen, importing 33
- level timer, creating 33
- timer callback function, creating 34, 35
- variables, adding 33

H

Haxe

- about 1, 135, 154
- HaxeFlixel 2
- Hello World project 9
- IDE, setting up 3, 4
- installation 3
- OpenFL 2
- target platforms, setting up 5

HaxeFlixel

- about 2, 154, 155
- features 2
- installing 4, 5

HaxeFlixel project

- anatomy 20, 21
- assets, adding 22
- creating 20
- settings, changing 21
- setting up 20

Haxe syntax

- about 13
- access modifiers 14
- classes 18, 19
- data types 14
- for loops 18
- functions 17
- operators 14
- structures 14
- variables 14

heads-up display (HUD) 37

health display

- hearts, changing 89
- updating 89

Hello World project

- about 9
- assets, copying 10
- creating 9
- modifying 10
- running 9, 10

I

instances, of enemies

- creating 27
- mouse events, adding 27, 28
- new imports, adding 27
- variables, adding 27

iOS

- about 7, 8, 139
- game, running on 148

J

JSON

- versus XML 116

JSON configuration file

- about 117
- adding, to projects 116
- data folder, creating 117
- JSON format 118
- simple example 117, 118
- using 115

L

level

- restarting 32

level end screen

- animation sequence, creating 52
- best score sprite, displaying 51
- group, adding 49
- housekeeping 48
- imports, adding 47
- play again button, adding 51
- scores, adding 50
- updating 47
- variables, adding 47
- window, setting up 49

M

menu sounds

- adding 108
- playback functions, calling 109, 110
- playback functions, creating 108, 109

movement patterns, adding to enemies

- about 123
- imports, adding 123
- old functionality, removing 124
- onSpawn, updating 125
- resetSpawn, updating 125
- startPattern function, creating 124
- variables, adding 123

movementPoints array 123

movementSpeed variable 123

music

- adding 107
- playback functions, calling 107
- playback functions, creating 107

muzzle flashes

- class, creating 72
- creating 72
- flash 72
- flash, playing 74
- imports, adding 73
- rings 72
- rings, displaying 75
- sprites, adding 74
- variables, creating 73

N

new game HUD, adding

- about 56
- housekeeping 57
- HUD, adding 58
- imports, adding 56
- variables, adding 57

new game HUD, creating

- about 53
- GameHUD class, creating 54
- imports, adding 54
- laying out 55, 56
- score, changing 56
- variables, adding 54

O

object pooling

- defining 61, 62

OpenFL

- about 2, 154
- installing 3

operators

- addition operator 15
- assignment operator 16
- decrement operator 16
- equivalence operator 16
- greater than operator 16
- greater than or equal to operator 16

increment operator 16

less than operator 16

less than or equal to operator 16

multiplication operator 16

subtraction operator 15

P

player character

- animation, adding 83, 84
- class, creating 81
- creating 81
- imports, adding 82
- movement functions, building 88
- muzzle flash, creating 85
- player death, handling 89
- player input, handling 86, 87
- projectile pool, creating 84
- projectiles, shooting 85, 86
- shoot timer, starting 85
- variables, creating 82, 83

PlayState

- collisions, detecting 94
- collisions, handling between player and enemies 95
- collisions, handling between projectiles and enemies 95
- defining 90
- enemies, killing 93
- EnemySpawner, using 127
- explosion object pool, creating 92
- imports, adding 90
- old functionality, removing 126
- player, adding 92
- player, creating 92
- score, updating 94
- scrolling backdrop, adding 92
- things, removing 91
- updating 126
- variables, adding 90

projectiles

- class, creating 80
- creating 80
- imports, adding 80
- sprite and setting velocity, displaying 81
- variables, creating 80

project settings
 updating 63

R

resources, for continue learning
 about 154
 community resources 155
 Haxe 154
 HaxeFlixel 154, 155
 OpenFL 154

S

singleton pattern 105
sound constants
 class, creating 103
 creating 103
 sound ID constants, creating 103
 volume constants, creating 104
sound effects
 adding 99
 audio folder, adding 100
 project settings, updating 100-102
sound manager class
 building 105
 class, creating 105
 imports, adding 105
 singleton pattern, implementing 105, 106
splash screen
 animation sequence, starting 43
 building 39
 imports, adding 40
 instructions, adding 42
 navigating back to game 45, 46
 next stage of sequence, creating 44
 sequence, finishing 45
 sprites, adding 41
 variables, creating 40
string data, pulling from XML
 about 129
 StringIDs class, building 131
 Strings class, building 129
 XML data, handling 129, 130

strings, replacing
 about 132
 GameHUD, updating 132
 LevelEndScreen, updating 133
 MenuState, updating 132

T

target platforms
 Android 6, 7
 desktop 8
 Flash 5, 6
 iOS 7, 8
 setting up 5
Texture Packer
 about 62
 URL 62

U

Unique Device Identifier (UDID) 145

V

variables, enemy spawner
 currentPattern 121
 numEnemies 121
 patternData 121
 spawnTimer 121

W

Windows 136

X

XML file
 about 128
 contents 128
 string data, pulling from 129



Thank you for buying Haxe Game Development Essentials

About Packt Publishing

Packt, pronounced 'packed', published its first book, *Mastering phpMyAdmin for Effective MySQL Management*, in April 2004, and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution-based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern yet unique publishing company that focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website at www.packtpub.com.

About Packt Open Source

In 2010, Packt launched two new brands, Packt Open Source and Packt Enterprise, in order to continue its focus on specialization. This book is part of the Packt Open Source brand, home to books published on software built around open source licenses, and offering information to anybody from advanced developers to budding web designers. The Open Source brand also runs Packt's Open Source Royalty Scheme, by which Packt gives a royalty to each open source project about whose software a book is sold.

Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to author@packtpub.com. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, then please contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.



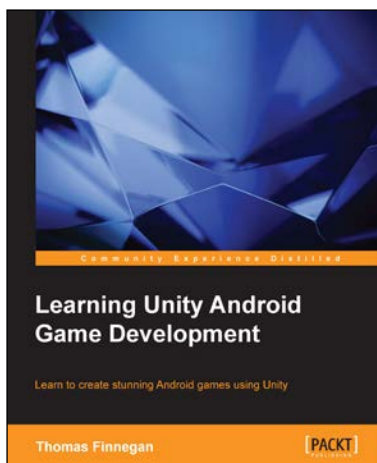
Learning iOS 8 Game Development Using Swift

ISBN: 978-1-78439-355-7

Paperback: 366 pages

Create robust and spectacular 2D and 3D games from scratch using Swift – Apple's latest and easy-to-learn programming language

1. Create engaging games from the ground up using SpriteKit and SceneKit.
2. Boost your game's visual performance using Metal - Apple's new graphics library.
3. A step-by-step approach to exploring the world of game development using Swift.



Learning Unity Android Game Development

ISBN: 978-1-78439-469-1

Paperback: 338 pages

Learn to create stunning Android games using Unity

1. Leverage the new features of Unity 5 for the Android mobile market with hands-on projects and real-world examples.
2. Create comprehensive and robust games using various customizations and additions available in Unity such as camera, lighting, and sound effects.
3. Precise instructions to use Unity to create an Android-based mobile game.

Please check www.PacktPub.com for information on our titles



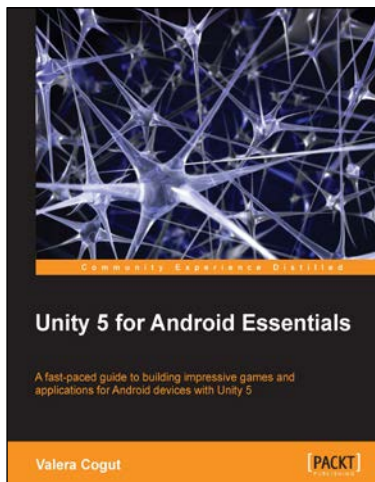
HTML5 Game Development by Example Beginner's Guide *Second Edition*

ISBN: 978-1-78528-777-0

Paperback: 354 pages

Make the most of HTML5 techniques to create exciting games from scratch

1. Get to grips with the latest HTML5, Canvas, CSS, and JavaScript techniques.
2. Learn to create exciting puzzle games, action games, multiplayer, and Box2D physics.
3. A task-based approach to achieve practical skills required for game development.



Unity 5 for Android Essentials

ISBN: 978-1-78439-919-1

Paperback: 200 pages

A fast-paced guide to building impressive games and applications for Android devices with Unity 5

1. Design beautiful effects, animations, physical behaviors, and other different real-world features for your Android games and applications.
2. Optimize your project and any other real-world projects for Android devices.
3. Follows a tutorial-based approach to learning the best practices for accessing Android functionality, rendering high-end graphics, and expanding your project using Asset Bundles.

Please check www.PacktPub.com for information on our titles