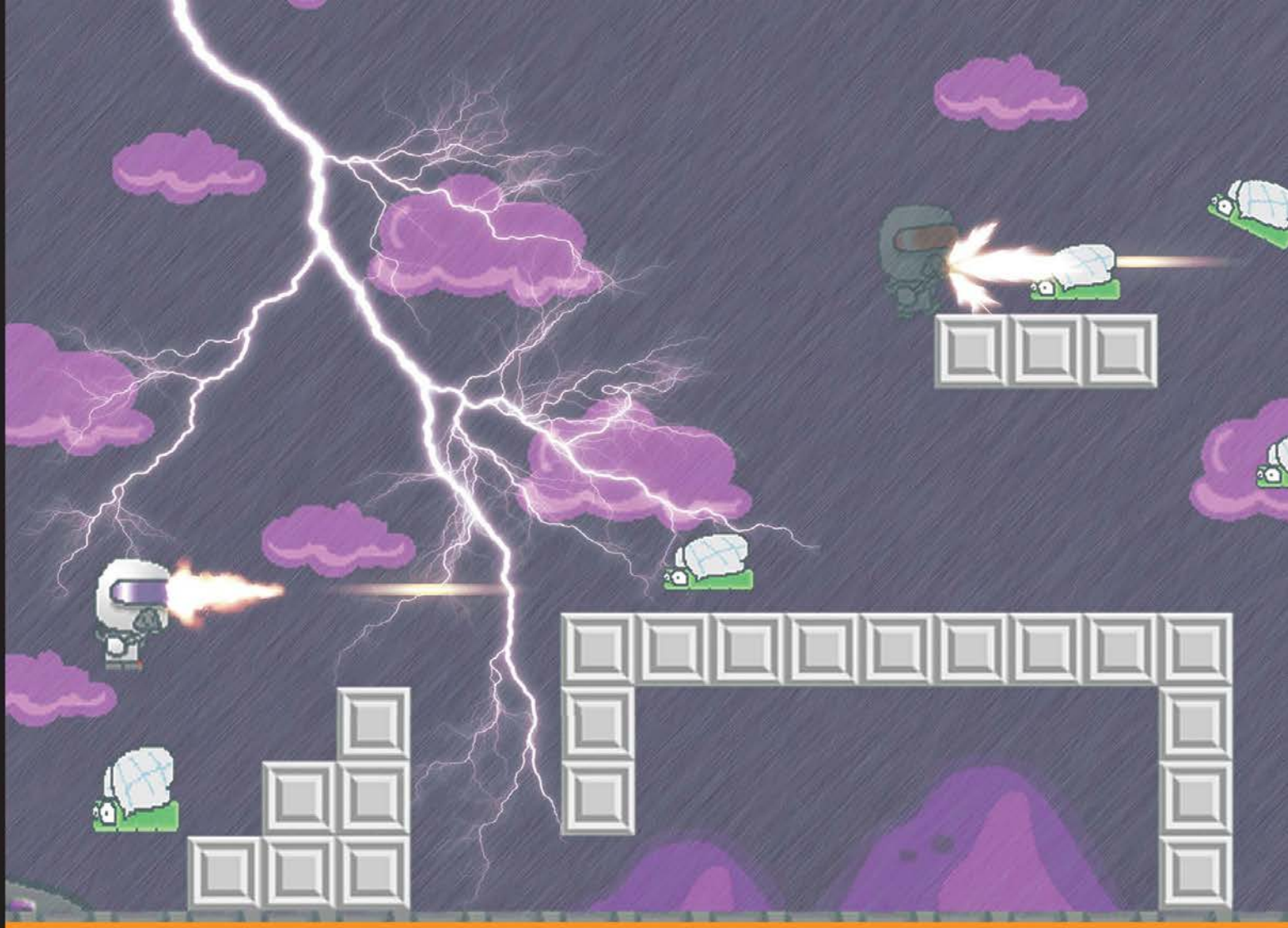


Community Experience Distilled

Mastering iOS Game Development

Master the advanced concepts of game development for iOS to build impressive games

Miguel DeQuadros

[PACKT] PUBLISHING open source* community experience distilled

Mastering iOS Game Development

Master the advanced concepts of game development for iOS to build impressive games

Miguel DeQuadros

[PACKT] PUBLISHING open source community experience distilled

BIRMINGHAM - MUMBAI

Mastering iOS Game Development

First published: December 2015

Production reference: 1181215

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

ISBN 978-1-78355-435-5

www.packtpub.com

Credits

Author
Miguel DeQuadros

Reviewers
Andrew Kenady
Sahil Ramani
Sladjan Trajkovic

Commissioning Editor
Kunal Parikh

Acquisition Editor
Tushar Gupta

Content Development Editor
Athira Laji

Technical Editor
Vivek Pala

Copy Editor
Pranjali Chury

Project Coordinator
Harshal Ved

Proofreader
Safis Editing

Indexer
Monica Ajmera Mehta

Graphics
Disha Haria

Production Coordinator
Nilesh Mohite

Cover Work
Nilesh Mohite

About the Author

Miguel DeQuadros is a game developer and the founder of the independent development studio, Wurd Industries, based in Ontario, Canada. He has been developing iPhone games since the release of the App Store back at the exciting release of iOS 2.0. Since then, he has released 10 games and 1 entertainment app world-wide on the App Store with more to come from Wurd Industries.

He was originally interested in 3D animation and graphical design, which he focused on mainly in 2004. But, he then got the game development bug and has been developing iPhone apps since 2008, which also allows him to use his creativity and knowledge of 3D animation for cut scenes and videos within his apps, and he is loving every minute of it. Starting from his first project, *Toy Tennis*, back in 2008, down to his current project, *SpaceRoads*, for PC, Mac, Wii U, and other platforms, he continues to develop high-quality apps and games alike. Moving away from simple game development tools, he now primarily uses Unity3D, 3D Studio Max, and the Unreal Engine for his current project in an aim to create very high-quality games.

His games can be seen on the App Store on iOS, Steam Greenlight, Amazon, and IndieCity, and of course on his website, `www.wurdindustries.com`. His games have been reviewed on YouTube by Action Soup Studios, and you can also find his interviews there.

I would like to thank my dad, John, even though he called me every two seconds to see how my book was going; his wife, Lucy; my brother and sister in-law, Johnny and Katie; my best friends, Brandon and Kaleb; and everyone else who has been there through everything. A very special thanks to my wonderful wife, Joanne, for encouraging me during the writing of this book (my wife asked me to thank Stella and Mimi, our dog and cat, as well. As if Stella didn't annoy me during the writing stage). Writing a book is very difficult, especially for your spouse and friends, as it does limit your association with them. I would also like to thank Packt Publishing and all the wonderful employees who helped me out both during the initial and final stages of this book, your professionalism and ability to clearly explain things also helped me a lot, thank you for everything. You made the production of the book a very smooth and enjoyable process!

About the Reviewer

Andrew Kenady is a game engineer from Kentucky. He holds a bachelor's degree in computer science from Western Kentucky University and has worked professionally in the game industry since his graduation in 2013. His published titles span multiple genres and platforms and include *Battlepillars* and *Draw a Stickman: EPIC 2*. He is currently working for a Tennessee-based tech company, NC2 Media, on new and promising confidential products for the mobile games sector.

In addition to working on this publication, he has previously reviewed the book *iOS Game Programming Cookbook*, *Bhanu Birani* and *Chhavi Vaishnav*, *Packt Publishing*.

Sladjan Trajkovic is a software engineer with a passion for game development. He has a master's degree in computer science and has been working in the software industry since 2007, where he began his career as a .NET consultant.

Nowadays, he works exclusively with the iOS platform, and he has been involved in several big name applications. He has also released two games on the App Store as an independent developer, *Alien Defense Zone* and *Super Kicks*. Currently, he is working on several new projects, both games and regular applications.

You can follow him on Twitter at `twitter.com/SladanTrajkovic`.

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

Welcome! We all love game development and many of us dream of games we want to play, but alas, we never see them released. You are here to make that game, the one you've been planning, drawing, and dying to create for a long time. With the help of this book and with your awesome ideas, you will succeed in creating your dream game. In this book, you will learn how to use the free software available to you to create, develop, and release your awesome creations.

Welcome and enjoy the ride!

What this book covers

Chapter 1, *What to Expect in Xcode and Game Development*, covers how to become an Apple developer, how to download Xcode and the iOS development kit, and what to expect of the game development industry.

Chapter 2, *Creating the Stuff*, demonstrates how to use the various tools available to you to create sprites and various other images, sound effects, and even videos for your game.

Chapter 3, *Blast Off! Starting with Development*, talks about beginning our project. You will learn how to import various frameworks for our project and create our character and get him moving.

Chapter 4, *Let's Keep Going! Adding More Functionality*, discusses how to add some cool effects, such as particles for flames and rain, to our game now that our project is coming together.

Chapter 5, *Bug Squashing – Testing and Debugging*, covers testing our game and debugging it to ensure that there are no issues in it once we port our game.

Chapter 6, *Making Our Game More Efficient*, shows how we can make our game run better on all devices now that it is running flawlessly. By doing this, we will increase battery life, performance, and the overall user experience.

Chapter 7, *Deploying and Monetizing*, discusses how to show our game to the world and increase our revenue outlets by adding monetizing options.

Chapter 8, *It's Too Dangerous to Go Alone, Take a Friend!*, talks about making our game multiplayer after much planning, as we are now ready to update our game. We will incorporate multiplayer connectivity to find friends to play with!

What you need for this book

For this book, all that you need is Xcode, which is Apple's software development kit for iOS, and any image creation software, such as Gimp or Photoshop.

Who this book is for

This book is for all those who have a little experience of iOS development but really want to hone their skills.

Conventions

In this book, you will find a number of styles of text 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: "We will add another `NSString` class at the top of our `MultiplayerHelper.m` file."

A block of code is set as follows:

```
- (void)match:(GKMatch *)match didFailWithError:(NSError *)error {

    if (_match != match) return;

    NSLog(@"Match failed with error: %@", error.localizedDescription);
    _matchStarted = NO;
    [_delegate matchEnded];
}
```

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

```
- (void)update:(NSTimeInterval)delta {

    CGPoint gravity = CGPointMake(0.0, -450.0);

    CGPoint gravityStep = CGPointMultiplyScalar(gravity, delta);

    CGPoint movingForward = CGPointMake(750.0, 0.0);
    CGPoint movingForwardStep = CGPointMultiplyScalar(walking, delta);

    self.velocity = CGPointAdd(self.velocity, gravityStep);
```

New terms and **important words** are shown in bold. Words that you see on the screen, in menus or dialog boxes for example, appear in the text like this: "For this project, we are—obviously—going to select **iOS** | **Application** | **Single View Application** and then click on **Next**.".

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 may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply send an e-mail to `feedback@packtpub.com`, and mention the book title via 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 on `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 for all Packt books you have purchased from your account at `http://www.packtpub.com`. 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.

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 would 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 on our website, or added to any list of existing errata, under the Errata section of that title. Any existing errata can be viewed by selecting your title from `http://www.packtpub.com/support`.

Piracy

Piracy of copyright 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

You can contact us at `questions@packtpub.com` if you are having a problem with any aspect of the book, and we will do our best to address it.

1
What to Expect in Xcode and Game Development

Welcome to the world of mobile game development! Whether you are a seasoned developer or a fresh new developer, you are in an exciting fast-paced industry. You have purchased this book expecting to master iOS game development, and that's exactly what you are going to do by the end of this book! However, you need to start somewhere, right? Let's see what will discuss in this chapter:

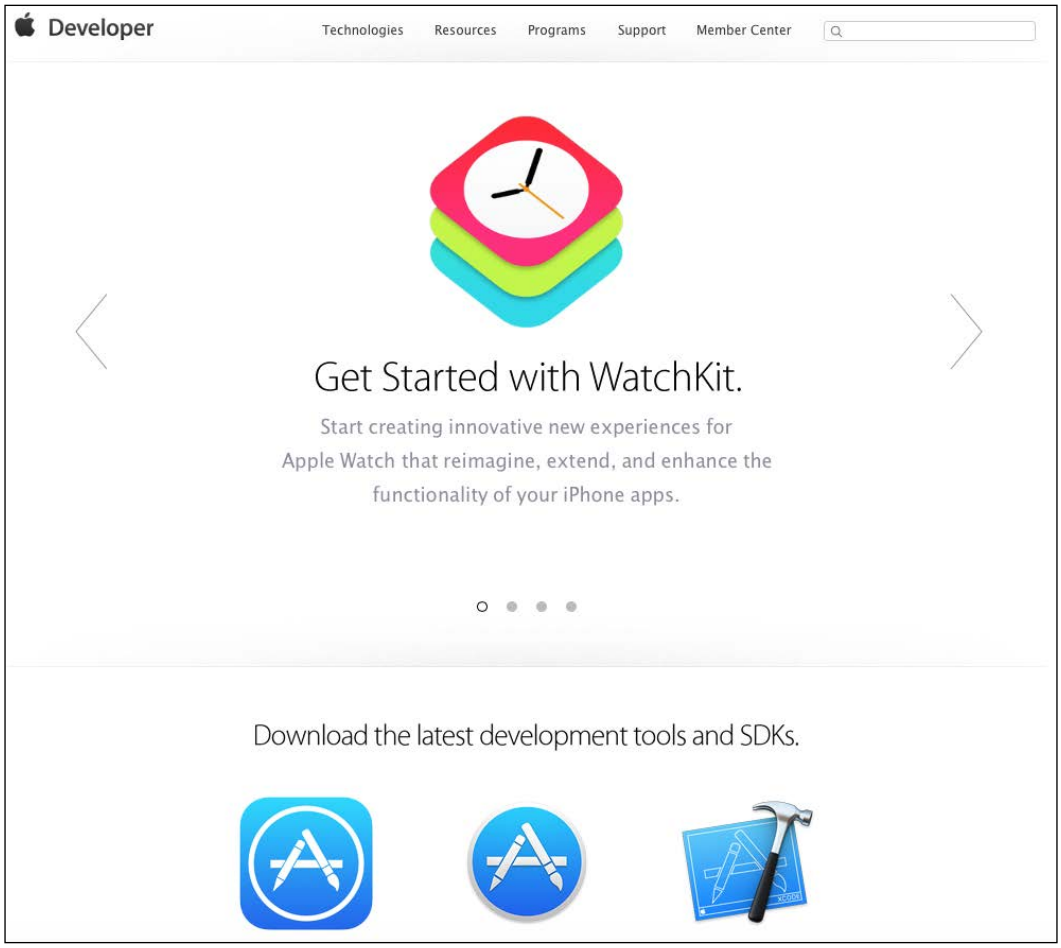
- Signing up to be an iOS developer
- Setting up your Mac for development
- What to expect from the game development market

The mobile game development industry is so exciting! Get ready to take your first steps into game development. We will get you all signed up to be an Apple Developer, set up your Mac with the development environment, and take a tour of some of the features of Xcode and we are going to see what source files (the files that contain our code) look like. Then, we will discuss the game development market and how AMAZING it isn't. I'm kidding, it is a fun market!

Let's dive right into it, shall we?

Registering to be an iOS developer

As with many platforms (Mac, PC, Android, Blackberry, and Windows Mobile), iOS requires you to sign up to be a developer. Don't worry! It's super easy, simply go to `developer.apple.com`, where you will see the home page, as shown in the following screenshot:



Scroll down to find **iOS Apps**. Click on the **iOS Apps** button, and you will be greeted with the iOS Dev Center. Get used to this site as well as iTunes Connect, which we will cover later in this book —you will be using these sites a lot during your game development.

If you don't have a developer account, click on the **register for free** text so that you can get signed up for a new Apple ID. Another great thing about Apple is that their accounts are all linked, so when you register for a developer account, you can use your Apple ID. On the next screen, you will be asked to either sign in with your Apple ID or create a new one. Simply follow the prompts, fill in all your information, and pay the annual fee ($99 USD/ $119 CAD). When you are done, you should have access to the SDK. If you aren't redirected to the iOS Dev Center again, go to `developer.apple.com/ios` and scroll down to the **Downloads** section. As of the time of writing this book, the current version of Xcode is 6.2. Click on the **Download** button. You will be taken to a download screen; however, you will be ported over to the Mac AppStore. Xcode is now hosted on the Mac AppStore, which is a lot better because it automatically updates Xcode when needed. You will see the Xcode installation page, as shown in the following screenshot:

Click on the **Get** button to download Xcode. When it's done downloading (it will probably take a while unless you have fibre optic extreme Internet: It's 2.5 GB. The AppStore will automatically install Xcode for you.

Now, you should have a glittery new icon on your dock. When you open it, it may prompt you to do additional installation to support debugging or what not, go ahead and let it download any extra components needed, as shown in the following screenshot:

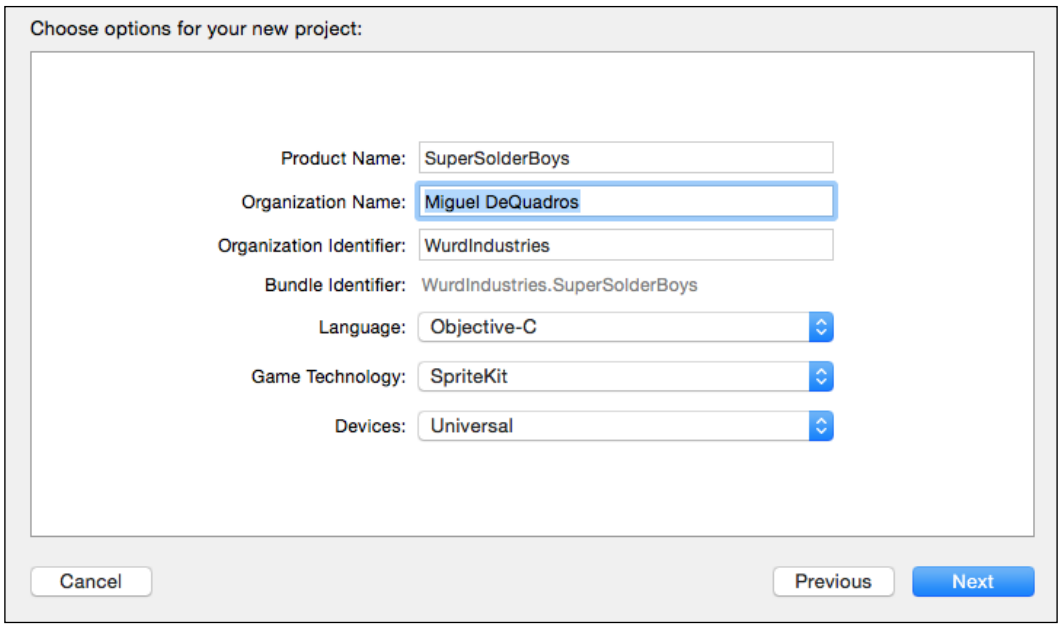
You will now be greeted with the Xcode welcome screen. From here, you can create new projects, open old ones, and even see a list of recent projects.

We will start setting up our game project, so click on **Create a new Xcode project**, and then the new project wizard will pop up. Here, you have a lot of empty project templates to choose from, but for this project, we will select the **Game** template under the iOS selection, under **Application**. Click on **Game**, and then click on **Next**. The next screen will ask for all the game information, which will be broken down like this:

- **Product Name**: This is the name of the project or game.
- **Organization Name**: This is the name of the company or your personal name that you used when signing up with the Apple Developer site.
- **Organization Identifier**: This will show up in your bundle identifier (certificates will be discussed later), usually certificates read `com.yourcompany.yourproduct`, but you can customize it to be `yourcompany.yourproduct`.

- **Bundle Identifier**: This is the identifier for your project. You will see this on your developer certificates, which we will install later, and you will see it when uploading to the AppStore as well.
- **Language**: There are now two languages that we can choose from when developing: Swift and Objective-C. For this book, we will use Objective-C.
- **Game Technology**: This includes he various "kits" that you can use when developing such as SceneKit, SpriteKit, OpenGL ES, and Metal. For this project, we will select SpriteKit.
- **Devices**: The default device is Universal; however, if you want to select your target device, you can do so. Sometimes, an iPad app won't look right on iPhone due to the smaller screen size, things won't fit properly as the resolution is different, and won't display properly.

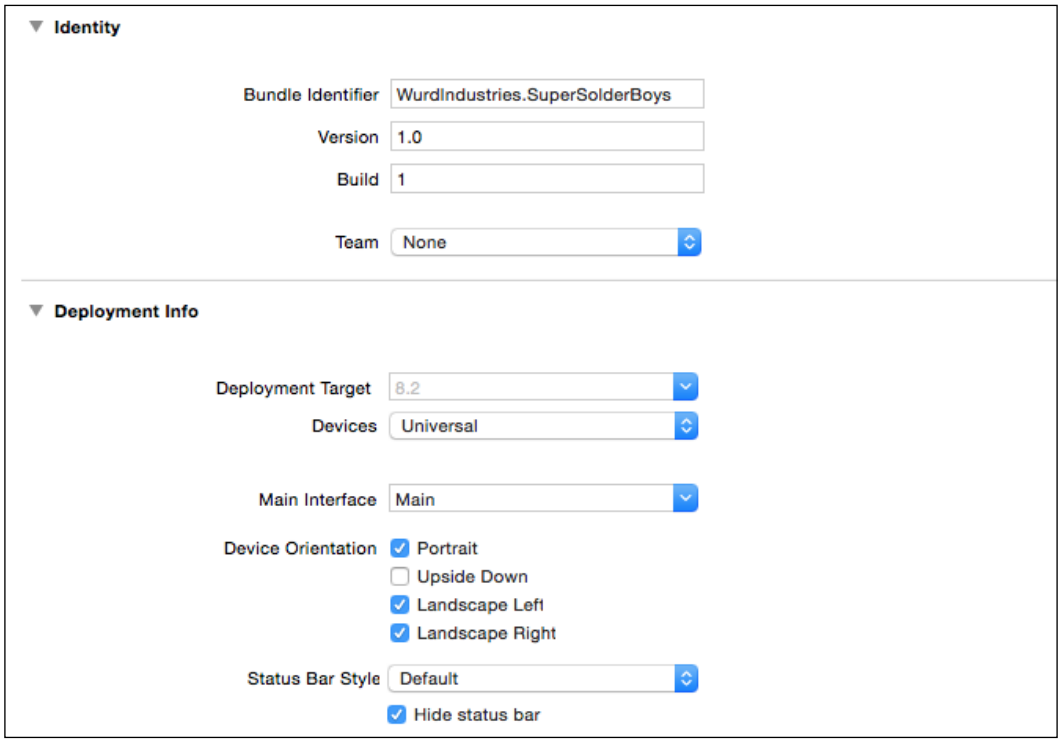
Fill in all the fields of information as shown in the following screenshot:



To recap, for this project, we will select **Objective-C**, **SpriteKit**, and **Universal**. When you have filled in all the information, click on the **Next** button, and save your project in a location that's easy to remember.

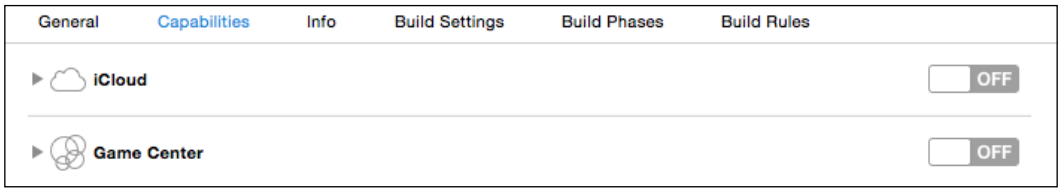
Now you are in Xcode! Confused by what you see? Don't worry, it's actually pretty easy to navigate.

The following screenshot shows the general project settings, which is the first thing you see when you load an Xcode project:



For this project, we only want to support the landscape orientations; so, from the **Deployment Info** section under **Device Orientation**, deselect the **Portrait** option, and leave the two landscape orientations checked.

If you want, you can add in different capabilities for our game. Click on the **Capabilities** button on the topmost bar in the center of the screen, where you will see a ton of different options that you can add to your app. For example, we can add **Game Center** for leader boards and achievements, or we can add iCloud functionality by clicking the button from off to on. We can use iCloud to store game save data remotely, as shown in the following screenshot:



There are a lot of settings and variables that can be changed in the various sections of the topmost bar. We will get into some of them later on in this book.

Let's take a look around our project files to familiarize ourselves with the functions of the files. On the left-hand side bar, you will see `.h` and `.m` files, these are where all your programming goes.

The `.h` files are your header files, we will declare variables (such as integers and Booleans) and outlets (if we were building through the storyboard, we would declare buttons, labels, and a lot more. We would declare them in the header file, and connect them in the storyboard).

Don't worry, we are going to go into a lot more detail later on in this book.

The `.m` file is your main file where the majority of the coding goes. Declarations made in the header files are accessible as long as it's the same set of files, for example, you can access an integer from the `AppDelegate.h` file in the `AppDelegate.m` file. There are ways to access variables from other files, such as frameworks; or as long as you import the header files into the files you are working on. Again these are things we will discuss later. I don't want to confuse you at this point.

Let's look at the `AppDelegate.h` and `AppDelegate.m` files, starting with the `.h` file where you will see the following text:

```
//
//AppDelegate.h
//SuperSolderBoys
//
//Created by Miguel DeQuadros on 2015-03-25.
//Copyright (c) 2015 Miguel DeQuadros. All rights reserved.
//

#import <UIKit/UIKit.h>

@interface AppDelegate : UIResponder <UIApplicationDelegate>

@property (strong, nonatomic) UIWindow *window;


@end
```

There will be variations in the text that is commented (you can comment text out by typing `//` following the comment). You will know text is commented out when it is green in color.

Almost all code files will begin with importing files. The `AppDelegate` class seen in the preceding code shows `#import <UIKit/UIkit.h>`, which will allow you to access commands and functions from the `UIKit` headers. You can import any other header files here, for example, if you want to create an app with an option to make payments via PayPal, you can import the PayPal SDK, then import it into your file by typing `#import <PayPal/PayPal.h>` or whatever the file is that needs to be imported. When you start typing, Xcode will display a list of files that you can import.

We will now look at the next line, that is, `@interface AppDelegate : UIResponder <UIApplicationDelegate>`. This will vary depending on the type of interface you are working with (whether you are working with the app delegate, a view controller for our main menu, or on a PayPal payment view controller); it can be an `ApplicationDelegate` method in this file, or it can be a `UIViewController` or `SKScene` method. Again, it depends on the type of interface you are working with. You will see variations of different interfaces throughout this book.

Finally, we will see the last highlighted line, that is, `@property (strong, nonatomic) UIWindow *window;`. This is where your declarations will go. This line declares the app window. Properties are declarations of various items we use within our classes. For example, a property can be an integer, a Boolean, a window (as mentioned earlier), or even a button.

Finally, we have the `@end` line, which is simply the end of the file.

Now, let's look at the `.m` file:

```
//
//  AppDelegate.m
//  SuperSolderBoys
//
//  Created by Miguel DeQuadros on 2015-03-25.
//  Copyright (c) 2015 Miguel DeQuadros. All rights reserved.
//

#import "AppDelegate.h"

@interface AppDelegate ()

@end

@implementation AppDelegate


- (BOOL)application:(UIApplication *)application didFinishLaunchingWit
hOptions:(NSDictionary *)launchOptions {
```

```
    // Override point for customization after application launch.
    return YES;
}

- (void)applicationWillResignActive:(UIApplication *)application {
    // Sent when the application is about to move from active to
inactive state. This can occur for certain types of temporary
interruptions (such as an incoming phone call or SMS message) or when
the user quits the application and it begins the transition to the
background state.
}


- (void)applicationDidEnterBackground:(UIApplication *)application {
    // Use this method to release shared resources, save user data,
invalidate timers, and store enough application state information to
restore your application to its current state in case it is terminated
later.
    // If your application supports background execution, this method
is called instead of applicationWillTerminate: when the user quits.
}

- (void)applicationWillEnterForeground:(UIApplication *)application {
    // Called as part of the transition from the background to the
inactive state; here you can undo many of the changes made on entering
the background.
}

- (void)applicationDidBecomeActive:(UIApplication *)application {
    // Restart any tasks that were paused (or not yet started) while
the application was inactive. If the application was previously in the
background, optionally refresh the user interface.
}

- (void)applicationWillTerminate:(UIApplication *)application {
    // Called when the application is about to terminate. Save data if
appropriate. See also applicationDidEnterBackground:.
}

@end
```

In this file, you will call various functions when the application reaches certain states. If you read through some of the functions, you will see commented out text (to comment out text type `//` before the line of code) on what each state does, like when the application is terminated, or when it enters the background. With these functions, you can save data, log information, or even pause the game.

Seems fairly simple, right? Once you get the hang of things, it really is!

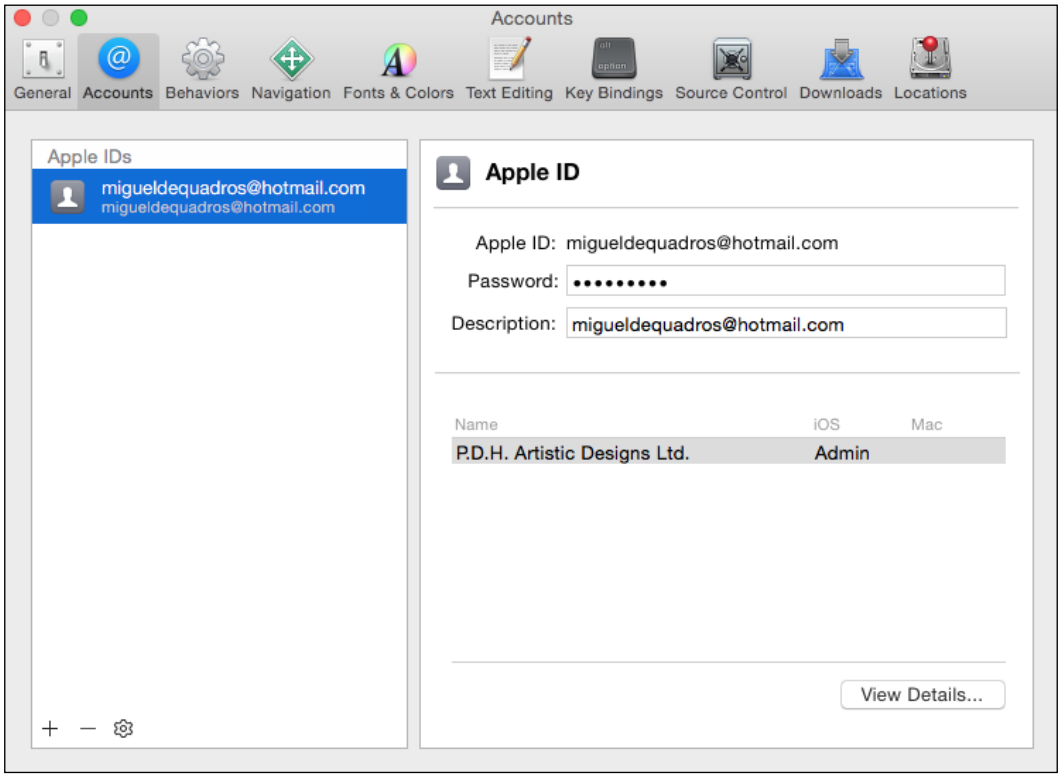
We will now have to add our developer account to the Xcode accounts preferences. This makes signing certificates and provisioning profiles much easier.

To add your account, click on **Xcode** in the topmost bar (where you see file, edit and all that fun stuff), then click on **Preferences**. Alternatively, you can press *CMD* +, for quick access to the menu.

When the preferences window appears, click on the **Accounts** tab, which is the second tab in the window.

In the lower-left corner of the window, click on the **+** button to add an account. When the drop-down menu appears, select **Add Apple ID**.

Again, Xcode uses your normal Apple ID login to sign in to the Developers' Member Center and iTunes Connect, which is super convenient because you don't have to remember multiple logins.



Now that we have added our account to Xcode, we need to add all the developer certificates into our key chain.

Go to `developer.apple.com`, click on **Member Center**, and log in using your Apple ID. Under the **Technical Resources and Tools** section, click on the **Certificates, Identifiers & Profiles** icon.

We will now see a selection for iOS, Mac, and Safari. These will display all the certificates for each platform, organized nice and neatly.

Click on the **Certificates** tab under iOS, and if your membership is new, you will be greeted with a **Getting Started** webpage. Simply follow all the steps, which will guide you into the creation and installation of the certificates. More of this will be discussed later on in this book.

When you have all your certificates installed, you are all done! Now that we have Xcode installed and set up, let's talk about asset creation.

Asset creation for your awesome game

There are literally a plethora of programs that you can use to create the assets for your game. Wait! What's an asset?! Simply put, assets are the various images, sound effects, or videos you use in your game. For example, we have sprites. Wait! What are sprites?! **Sprites** are images you use in your game; this can include characters, platforms, backgrounds, items, and objects. Assets can also be sound effects, music, and videos.

For sprites, I like to use Adobe Photoshop, especially if the game I'm making is a pixel art 2D game. It's excellent because you can make use of the layers for each individual body part that needs to be moved around for the animations. However, if I'm making something that I would want to look a little more realistic, I'll use 3D Studio Max to create my models then render them into a small image file.

These software suites are not cheap. 3DS Max is over $3,000, and the Adobe Suite used to be about $1,600, though they now offer monthly plans.

If you're looking for a free option, check out programs such as **Gimp** for your image creation. I've used it: it's laid out just like Photoshop, and it is just as easy and powerful to use. Alternatively, use **Pixlr**, which is a browser-based image manipulation program exactly like Photoshop.

If you want to do some cool 3D stuff, check out **Blender**, **Wings3D**, and **DAZ Studio**.

For music, many like to use good ol' **Garage Band**. There are lots of instrument options, premade loops, and the ability to use your computer keyboard as a musical keyboard.

For audio manipulation, the main choice between developers and others alike is **Audacity**. You can manipulate audio files by adding fades, clicks, silence, reversing audio, adding , and a lot more. It's pretty awesome.

For videos, I personally like to use **3DS Max**, then import into **iMovie**. The iMovie app is great because it can export your video files for use with an iPhone, so we need not worry about the compatibility of the file, or about screen size. You can use Blender to create some awesome 3D animations in a short time for your introductions, or you can animate them using **Adobe Flash**, which can also export them to specific formats for iPhone and iPad.

We will talk more about the actual creation of assets in the next chapter. In the meantime, let's talk about the game development scene, and what exactly to expect when you get into it.

Entering the game development market

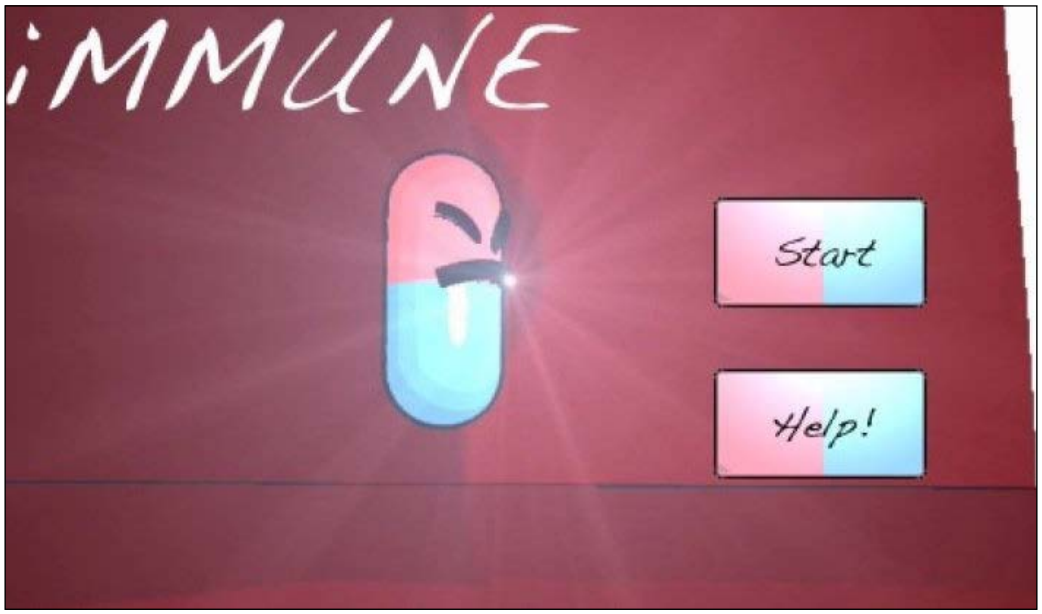
As I mentioned in the introduction of this chapter, you have entered a super exciting market. The game development market is fast paced, always changing, and super fun with a huge potential to make a lot of money as Doge will explain:

However, while it is a fun business with huge potential to make money, it's not all fun and games and instant success.

In the ten and more years that I've been developing games, I have learned that what you think will sell, may not sell. One of the first games I made was called **iMMUNE** and I thought it was quite decent when I was programming it. After much play testing, and getting people's opinions, I felt the game was ready to release, and I hoped for much success, as we all do.

The day of release came and passed, and I saw little or no sales. Naturally, I was pretty bummed, but now I realize I made many mistakes when developing and releasing the game. Let me explain what happened—I got the idea from my grandmother, who (as most grandmothers do) takes a lot of medication for her ailing body. One day, we were discussing the whole swine flu epidemic, and my late teenage brain automatically thought, "Hey! All of us take pills, and this swine flu thing is pretty crazy right now. Why not cash in on that?" I wasn't being uncaring of the state of the world's health, I just thought that, with the current situation, people would be searching for swine flu or pills at the time and my game would be presented to a lot of people.

So, I came up with an idea for iMMUNE. It was simple, you play a pill going throughout the human body and you have to fight the viruses, culminating in having to destroy the swine flu viruses at the end of the game. Here is a screenshot of iMMUNE:



I thought it was a great idea, and so did those around me. So, after many months of rushing to get the game finished, and admittedly, leaving some major features out, such as a main menu with a help button, the first mistake I made was that I uploaded the game to the AppStore. I began to publicize the game, create hype with screenshots and cut scene videos and trailers.

At the time, I felt, "You know what, I've been working on this game for about nine months, I want to get a half decent return from it. I'll set the price to $2.99." This was my second mistake.

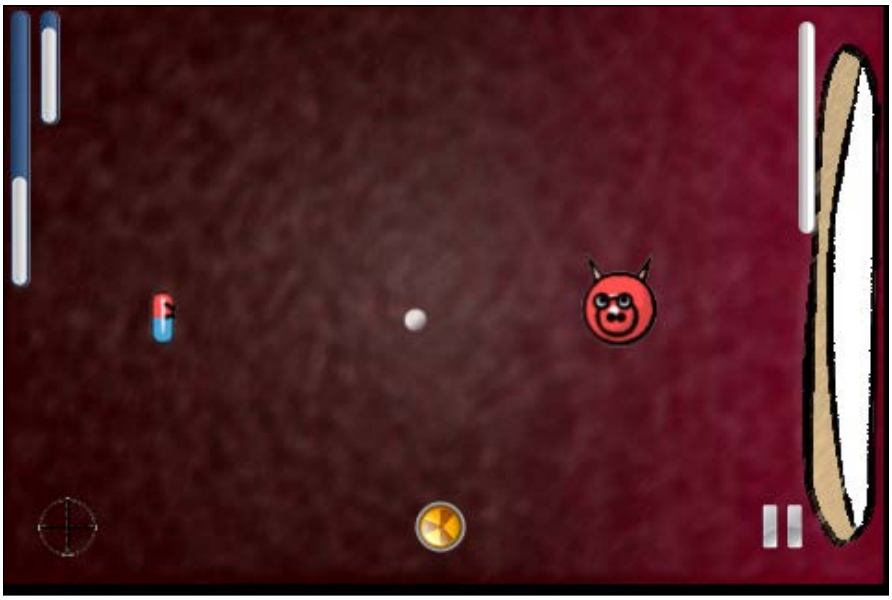
The game released on 5 August, 2009, approximately a year after its development. I started to see one star reviews popping up. Ah! Haters gonna hate. This was my third mistake.

A few days later, my best friend purchased the game and called me up, and this was literally our conversation:

Him: Hey, I just bought your game. How do I play it?

Me: Oh, just tilt the device to move the character and press the crosshair button to shoot the viruses. Then...

I'm pretty sure he fell asleep as I was trying to explain it. (I'll tie this one into my first mistake.) As you can see by the following image, the user interface wasn't well labelled or explained. The interface was just thrown together.



There were so many things on screen the reader had to understand and none of it was explained
There were so many on screen elements the player had to understand and none of it was explained.
It was an unfortunate error.

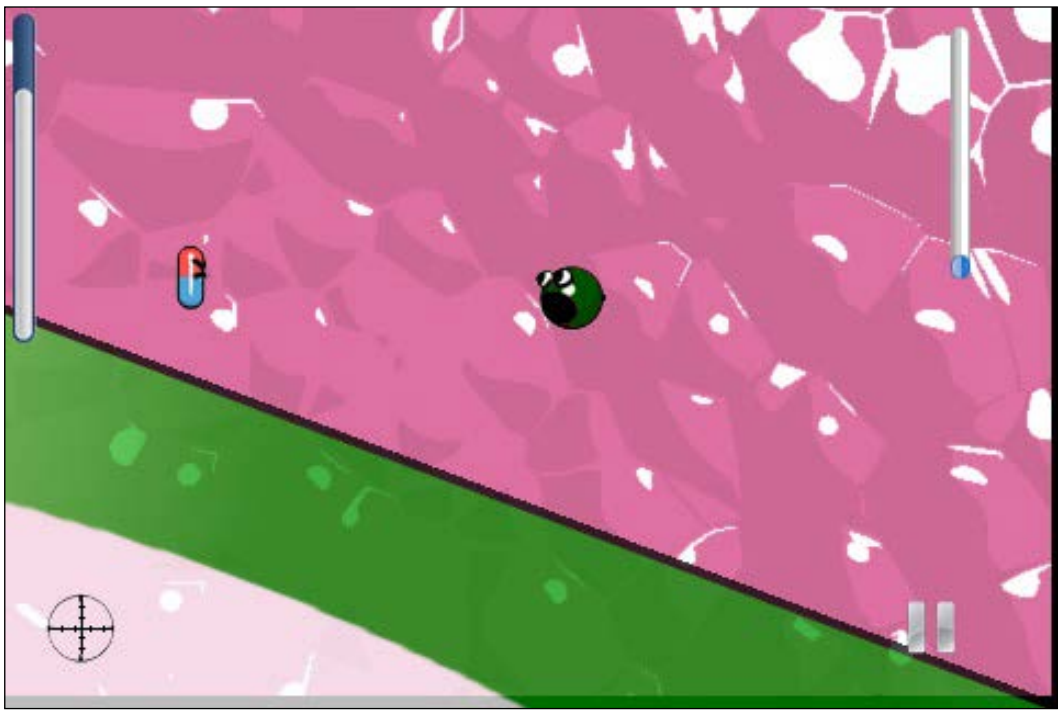
Needless to say, I went on to create a sequel using a program called **GameSalad**, which I've written two books on. I love the program. However, when I was creating iMMUNE 2, GameSalad was in its very early development stages, which meant that it would crash a lot, and there were a lot of features that were missing. Long story short, I miffed that one up too. And I was trying to improve on the first program's mistakes.

So, what went wrong? Let me break it down for you.

- **Mistake 1**: I rushed to get the game out, and I left out crucial features. As noted, I didn't include a main menu, which in itself isn't exactly a terrible thing. What is terrible is the fact I threw the player right into the game without telling them what to do. When the player got frustrated, they went on YouTube to see how to play, only to find out that if they tilted the device too far, the main character would run off the screen. Again, that little bit of code I forgot to write, `if player.position > screen.height = STOP MOVING`, was completely left out. How did I miss that?
- **Mistake 2**: I set the price for what seemed to be a reasonable $2.99. However, what else can you get for $2.99? A cup of coffee, or a ticket for the Toronto Maple Leafs, considering how bad they are lately? There's a lot $2.99 can buy, just go into a dollar store and see. Do people want to pay $2.99 for an app? Nope. Have they ever? Not really. This is why most games have a freemium payment structure. The game is free to download but you can purchase 1,800 carrot seeds for a nickel.
- **Mistake 3**: When I started to see the one star reviews come in, I figured it was just people "hating" because they do! I mean, come on, my game was pretty awesome. Right? *right?* Nope.

What did I learn from these critical errors that caused a waste of year of my time to be a? For one, don't rush projects. High budget projects are rushed to meet deadlines, and we as the players, hate the game because it's so broken. Check every line of code, play test the heck out of it to ensure that everything is working 100 percent. Granted, there are things you will miss that 14 year olds smashing their hand-me-down iPad with their greasy fingers at a rate of 1,000 rpm will catch because that's not how the game is suppose to be played. Ensure that it's working right the first time. Can I suggest focus groups to play test before you release?

The high price tag was a major error. I wanted to get a large return on the time I spent developing the game, which is why I set the price too high—*way too high*. I think we all have that mindset when we first develop a game, and we don't consider the fact our price cannot hold the market. I should have priced it the lowest it could go, which is $0.99 at the time, "freemium" wasn't really a thing. I felt I had put a decent amount of effort into the game, as you can see by the following image, I even put stomach acid that would tilt according to how you were holding the device. I felt I had made a decent, quality game. Oh how wrong I was.



Finally, I should have read the reviews and taken them to heart. I realized what I did to the players, and how it wasn't fair. My game was a little broken, and I admitted it. I worked to fix it and updated it to version 1.1. I added a main menu, a help screen, a help popup at the beginning of the game, and I fixed that stupid bug where the character would float off the edge of the screen instead of just stopping. Also, I reduced the price tag to $0.99. You can probably guess what I'm going to say next. It was too late. By the time iMMUNE 1.1 was released to the general public, the disinterest and general disdain for the previous errors caused the game to ultimately fail.

But, what about new players? Wouldn't they see the new screenshots and gameplay when the game was updated? Yep, they would—for a time frame that is smaller than the thermal exhaust port on the death star. The AppStore is a great place, but apps swarm it like zombies swarm "Rick Grimes", you will see your app go on the new list, and after maybe an hour or so, it's down ten places or more.

This is my personal experience in the game development market. I have had successes, but only after my failures. When I released my latest game *paceRoads* on **Desura** and **IndieRoyale**, I was lucky enough to be told by my developer relations contact to lower the price I had set before the game was even released, which ultimately led to the success of the game.

I know all this sounds like doom and gloom, and that I'm "hating" on the market. But, I'm not, I love game development and the whole community of developers. I just don't want you to make the same mistakes that I did.

I talk about how there's a huge potential to make money in the market, and there really is! If you have time look up the creator of *Trism*, or just look at *Angry Birds*. Enough said, but those are normally huge corporations with billions of dollars to throw at advertising. Normal people like you and I don't have that much money, so our creation, as amazing as it may be, could get lost in the swarm. We will discuss how to overcome this in *Chapter 7*, *Deploying and Monetizing*.

Don't let any of this discourage you because game development and the market is actually a lot of fun. If you have any questions, you can go on a developers' forum and ask a group of developers who are way more knowledgeable than you are, and you'll get an answer—a good one too. I've even had a developer ask me to send him my project so that he could see what was wrong, and he reprogrammed the set of code that was causing me grief.

Trust me when I say you've entered an exciting world! Game development is an expression of, not only your technical knowhow, but also your creativity.

I had recommended in my previous book, *GameSalad Essentials* by Packt Publishing, and I will recommend it again in this book, please take a look at the movie *Indie Game The Movie*. You will see some real world experiences of developers who made it huge in the market, and you'll see the struggles and hardships they endured. I highly recommend watching it.

Enough of my ranting! Let's get to the fun part, let's start creating the assets for our game in the next chapter.

Summary

In this chapter, we signed up to become an iOS developer, set up Xcode on our Mac, and discussed setting up the certificates so that we can begin with the development of our awesome game! We had a quick overview of Xcode and how the coding files work, and we discussed some programs that we can use to create the assets for our game (the ones that I use and the free alternatives). Then, we discussed my personal experience in the game development market, what to expect, and which mistakes to avoid.

Now that we are done with all of that, let's begin with asset creation!

2
Creating the Stuff

Now that you have Xcode all set up, I know you are excited to get started on developing your awesome new game! However, as you know, we have to start slow! All games have assets, such as a 3D model, an image, or super awesome sound effects. If you don't know anything about stuff like this, then you've come to the right place! In this chapter we are going to discuss:

- Creating or obtaining assets for your game
- Creating an optimized asset
- Designing your game

This is where you can start to get really creative! Once you start designing the building blocks of your awesome new project, the fun really begins.

Some assets will be easy to create, some will require a little more effort, some may involve paying people to create them for you, and some may require downloading free assets, as shown in the following image:

Enough intro, let's roll!

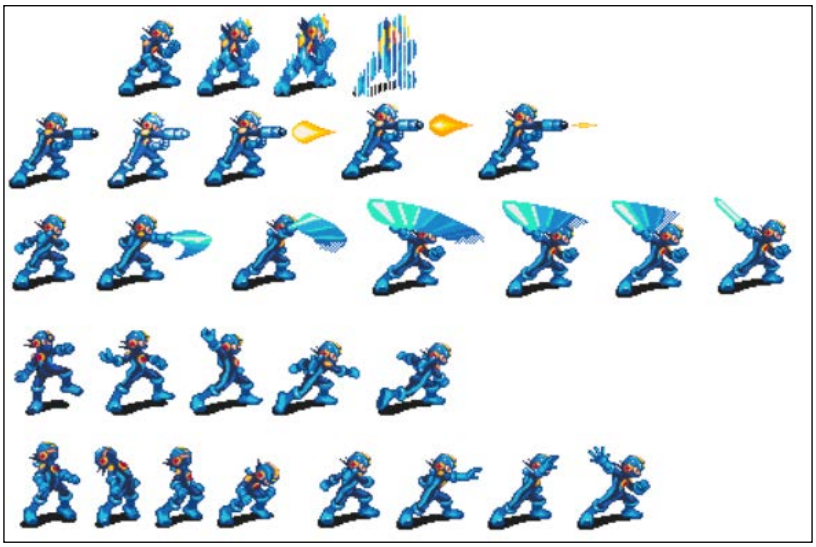
Let's talk assets

Assets are the building blocks of your game. They will essentially make up 100 percent of your game, regardless of whether you are creating a 3D blockbuster, or a 2D platformer as we will in this book.

There are various assets you will either create, purchase, or have created for you for your project. Let's break them down.

Sprites

What?! Don't worry, not a watery sprite as in, say, Homer's The Odyssey. When it comes to video game assets, sprites are essentially images that are used in your game. See that Mario image in the intro of this chapter? That right there is a sprite. Sprites can also be used for special effects, such as 2D lens flares or particles. Sprites are usually created in sprite sheets containing characters and their animations, such as the following image, which is a ripped sprite sheet from one of the many Megaman titles:



Sprites can be super easy to create, or they can be massively complex and take many hours to draw. Yes, draw. Sprites can be drawn in any imaging software, using just the mouse, or a tablet (I highly recommend purchasing one because it makes sprite creation a heck of a lot easier).

On the other hand, sprites can be rendered using a 3D animation program. I have used 3D Studio Max for some of my earlier games, and I was very pleased with the results.

- **Best sprite size**: Varies on device, but where a 32x32 px image will look good on a 3GS, it will be impossible to see on a retina iPad.
- **File type**: PNGs are usually the best file format (in my personal opinion) as they have a great compression (small file sizes = better performance), quality, and they support transparency, which is essential when drawing your characters as you don't want a white box surrounding them!
- **Recommended software**: I am partial to Adobe Photoshop, but as I've mentioned numerous times in my other books, there are a lot of free alternatives to it. A programs such as Gimp is a perfect example.

Sound effects

BOOM! KABLAMO! PAFF! Well these speech bubbles are not required for games! Sound effects are just as important as the sprites you create. Now, sound effects can be tricky to create yourself unless you have professional audio recording equipment. For me, I search for royalty-free sound effects on the Internet. Believe me, there's a plethora of them available for you to use, ranging from cartoon sound effects, to gunshots and explosions. `www.freesound.org` is a great site that I've used many times for my games.

File type

The iPhone audio SDK documentation indicates that iPhone supports audio in several formats, however there are drawbacks to each:

- **MP3**: This is a highly compressed, but also high-quality encoding format that preserves the richness of music and spoken voice. However, only one single MP3 stream can be played at a time on an iPhone because it requires the use of the hardware decoder. This excludes MP3 as a format for sound effects, since it will interrupt background music and/or any user music being played.
- **WAV**: iPhone only supports WAV files if they are PCM-encoded and do not require any compression codes. While WAV supports many different sampling rates and bit-depths, it results in very large files (180k/sec for 16-bit/44.1kHz/stereo audio). This large size becomes prohibitive when you have many effects to play rapidly.
- **AIFF**: This has similar problems to WAV files—Apple Lossless compression is supported.

- **Custom**: You can use iPhone low-level APIs to play your own audio from any source, but this seems like overkill for playing a simple sound effect in a game.
- **Recommended software**: For manipulating sound effects and rewriting/converting them, the best software I've found to date is **Audacity**. It's free, easy to use, and supports the main audio formats. The following image shows the Audacity logo:

Music

Another important asset is music. I love music and wish I could have a song playing at all times in my games. However, I understand in many games music can be used for setting the mood, and can stir up emotions for the following scene. Creation of music isn't difficult, it just takes a good ear and a little bit of creativity. You can either create your own music, or you can download royalty-free audio. You always want to make sure it's royalty free, or else you could make someone very (legally) angry.

Formats

As mentioned earlier, the best format for music is MP3. It allows only one track so it is uninterrupted by other sound effects and provides a high-quality, low file size.

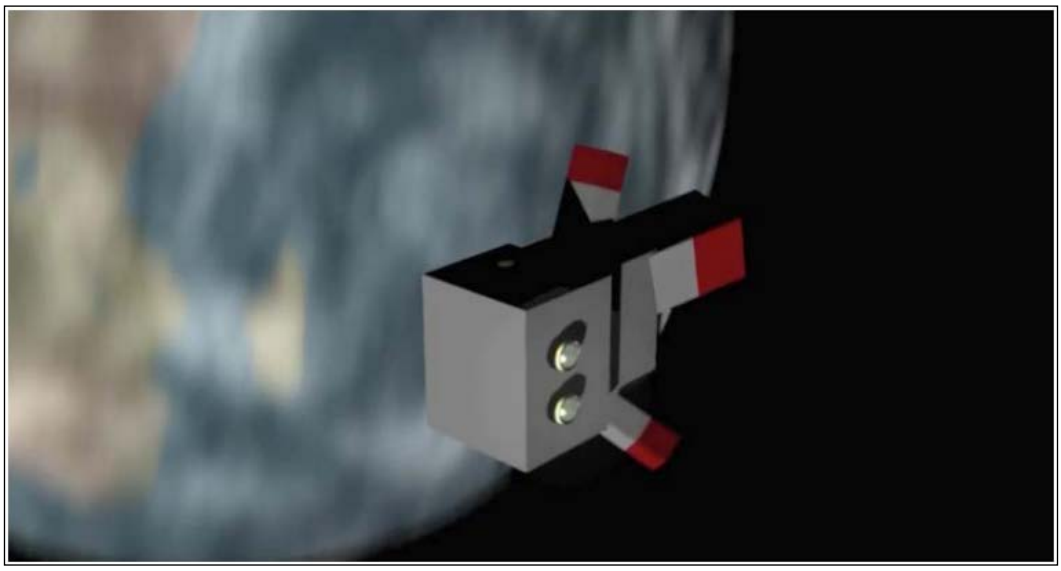
Best software

Garage band for the creation, and for any editing, again use Audacity.

Videos

Personally, I love creating and playing cut scene videos for my games. I feel they provide a good break between game play, but, hey, that's just me. I know many people just skip them as soon as they begin.

The following image is grabbed from the intro video of one of my projects called *LOST*:



File type

M4V (many programs, including QuickTime, can convert any video to an M4V, but if you want a little more control, use **Handbrake**).

Best software

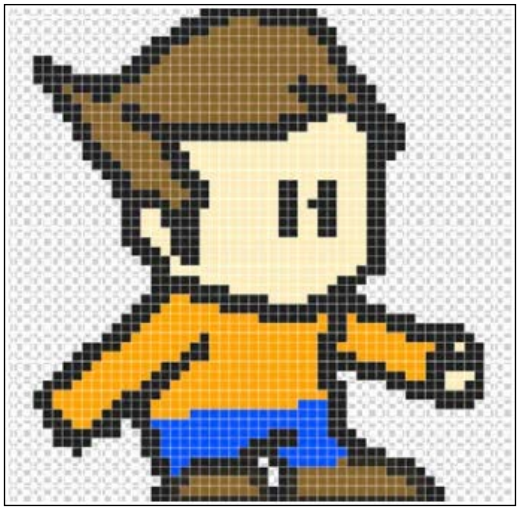
I use 3DS Max a lot for my video creations and then I import into **iMovie** for manipulating. If you are looking for a 2D solution, try Adobe Flash (I believe it used to be called **FlashMX**). For converting the videos use Handbrake, which has a huge array of settings you can change to tweak the video for the desired quality and file size.

Creating optimized assets

Let's start with creating optimized sprites for the best performance. How will this affect performance? Think of a game as a video, and your device as the DVD player, or even your TV. The game plays frame by frame and your device displays each frame. However, it has to render each frame. This happens so fast you will never notice it, but now imagine your device has to render large images (let's think dialup internet loading a huge photo). It can take longer and because the device has to work harder, the battery will drain quicker.

This is only one reason that can slow performance; we will discuss other reasons further on.

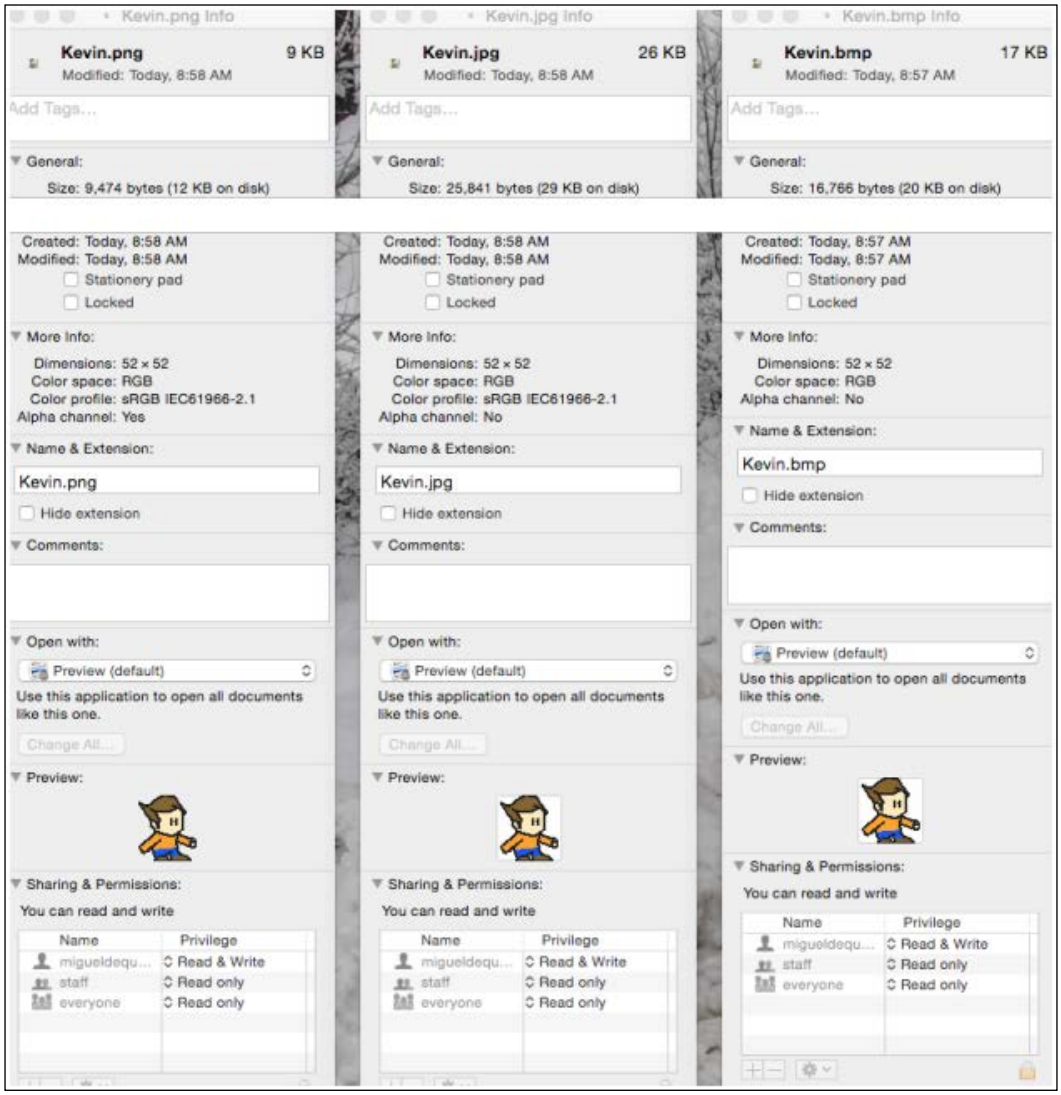
In the following figure, I've drawn a character design from my last book, and I drew him in Photoshop:



For your information, his name is **Kevin**. It doesn't look like much does it? Here are the details of the drawing:

- **Image size**: 52px x 52px
- **Color space**: RGB
- **Compression**: None at the moment
- **File type**: Unsaved

I went ahead and saved the image in three different file types: bitmap, PNG, and JPEG. In the following figure we see the different file types, beginning with PNG on the left, JPEG in the middle, and bitmap on the right:



So what are we looking at here? Let's break it down. We will start with the JPEG, a widely used image format:

- **File size**: 29KB, quite miniscule when you consider we now have multi-terabyte hard drives. However, JPEGs do tout the largest file size.
- **Alpha channel**: No, as mentioned before, jpegs do not allow transparency, which is ok if you are just creating a sprite that fills the whole box and has no background, but for our little character here, a JPEG will do us no good.

Other than that, JPEGs are the same as PNG files:

- **File size**: 9KB, quite a lot smaller then the others, meaning increased performance!
- **Alpha channel**: Yes! This guy is a winner here!

Then on to the bitmap:

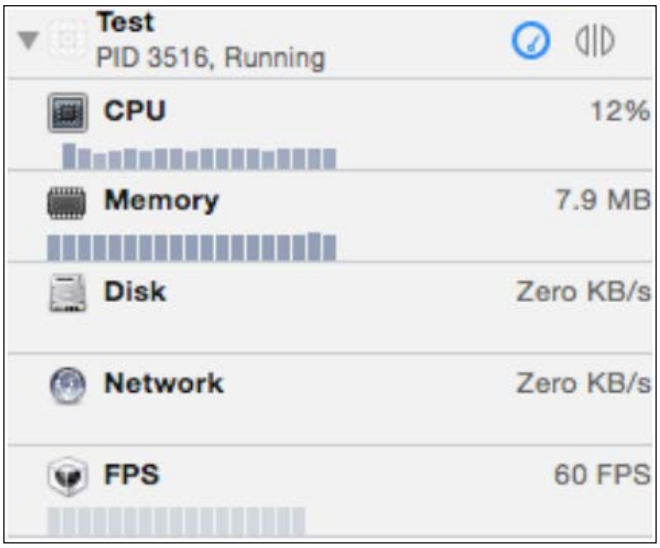
- **File size**: 17KB, again very small and in this case the midrange file size
- **Alpha channel**: Again, no
- **Color profile**: Non-existent

Essentially bitmaps are basic image files with not much information regarding the actual image.

Awesome! So it seems that PNGs are the winner! I went a step further to test my theory: I imported these three images into a blank **SpriteKit** project and wanted to see the difference between rendering each image. Want to know my findings? Of course you do! It's all about the science!:

- **BMP**: Cannot be imported into the `Images.xcassets` section the way I did with the other two images, so that's a bummer... But I worked around it. The CPU had a high 16 percent usage, memory had a range of 7.6 to 7.8MB (very similar to the PNG), and the FPS (or frames per second) dipped when the sprite was created at 56FPS, a completely unnoticeable dip.
- **JPEG**: In Xcode's debugger, simply rendering the sprite file spinning like a top, the CPU seemed to hover around 12 percent usage (I'm using a 16GB iPhone 5S), memory was clocked at 7.9MB, and the FPS remained steady at 60FPS, the best you can get.
- **PNG**: Again, in the debugger, rendering the PNG image spinning like a top, the CPU fluctuated between 11 percent and 12 percent usage, memory ranged between 7.5 and 7.8MB, a little bit lower then the jpeg but not enough to notice just rendering one image, and the FPS was clocked in at a steady 60FPS.

Here is what the findings will look like:



For each file type, I created about 200 images on the screen, all spinning at the same speed, and never saw any hiccups in frame rate, but did see a decent increase in **CPU** and **Memory** (an increase of about 5-10 percent).

All in all, not bad figures. It still seemed as though the PNG was easier to render than the others, which does make a huge difference when rendering multiple objects or images in memory.

Video conversion

When you are converting your videos using a third-party program that doesn't have iOS presets, iOS supports the following formats and settings:

- **H.264 video**: up to 1.5 Mbps, 640 by 480 pixels, 30 frames per second, Low-Complexity version of the H.264 Baseline Profile with AAC-LC audio up to 160 Kbps, 48 kHz, stereo audio in `.m4v`, `.mp4`, and `.mov` file formats
- **H.264 video**: up to 768 Kbps, 320 by 240 pixels, 30 frames per second, Baseline Profile up to Level 1.3 with AAC-LC audio up to 160 Kbps, 48 kHz, stereo audio in `.m4v`, `.mp4`, and `.mov` file formats
- **MPEG-4 video**: up to 2.5 Mbps, 640 by 480 pixels, 30 frames per second, Simple Profile with AAC-LC audio up to 160 Kbps, 48 kHz, stereo audio in `.m4v`, `.mp4`, and `.mov` file formats

For more information, take a look at `https://developer.apple.com/library/ios/documentation/Miscellaneous/Conceptual/iPhoneOSTechOverview/MediaLayer/MediaLayer.html`.

Remember, the developer forums and iOS development documentation will be your best friend!

Audio conversion

There is a fantastic article by Audrey Tam regarding audio encoding for iOS. Here's a quick excerpt from the article found on `raywenderlich.com` (`http://www.raywenderlich.com/69365/audio-tutorial-ios-file-data-formats-2014-edition`):

> *"There are actually just a few (formats) that are the preferred encodings to use. To know which to use, you have to first keep this in mind:*
>
> *You can play linear PCM, IMA4, and a few other formats that are uncompressed or simply compressed quite quickly and simultaneously with no issues.*
>
> *For more advanced compression methods such as AAC, MP3, and ALAC, the iPhone does have hardware support to decompress the data quickly – but the problem is it can only handle one file at a time. Therefore, if you play more than one of these encodings at a time, they will be decompressed in the software, which is slow.*
>
> *So to pick your data format, here are a couple of rules that generally apply:*
>
> **If space is not an issue**, *just encode everything with linear PCM. Not only is this the fastest way for your audio to play, but you can play multiple sounds simultaneously without running into any CPU resource issues.*
>
> **If space is an issue**, *most likely you'll want to use AAC encoding for your background music and IMA4 encoding for your sound effects."*

If you have a lot of audio files being played throughout your game (who doesn't?), you can save a lot of time and guesswork by importing them into iTunes, then right-clicking on them and selecting create AAC version. iTunes will then create a new AAC version that you can locate and import into your project and it will work 100 percent of the time. No guesswork!

I know, it can be confusing at times trying to figure out which format to use and how to convert things, but Apple does make things fairly easy by including audio conversion right in iTunes.

Now let's talk a little bit about design.

How to design your game

Game design can be a lot of fun, especially when you get a really great idea and you just keep rolling with it and adding more bits and pieces to it. No doubt you already have an idea for your game, but it's always a great idea to put things down in writing. I tell people this all the time, write it down, or you'll either forget it completely, or certain critical details. I've done it myself, I've had a great idea for a new game when I awake at 2am, but I don't write it down and by the morning I've completely forgotten my great idea. Not only is it good for remembering things, but it's also good for planning and expediting the creation and development process. The game we are going to create in this book will be in the style of *Contra* in the aspect that it will be a platformer, and we will add the ability to allow playing with your friends! But that will come much later in the book. We are also going to go through the planning process together. The following is be a template game design document that will lay out what we plan to do for the game in this book.

Some of these headings will be left out, which is fine because we can figure out certain details as we go.

Game design document

We will be using the following pattern when creating the game design document:

- Title page
    - **Game name**: Adesa
- Game overview
    - **Game concept**: A game such as Contra, 2D Platform shooter. The player is a cartoon Space Soldier who was launched into space after his ship exploded.
    - **Genre**: Platformer shoot 'em up.
    - **Target audience**: Young children to adults (the game will not contain any graphic violence).
    - **Game flow summary**: The game will be a side scroller and the player will be controlled via touch controls, both in the game and in the menu.
    - **Look and feel**: 8-bit 2D goodness.

- Gameplay
    - **Game progression**: Level by level, no player upgrades
    - **Mission/challenge structure**: Locate lost equipment → find a way off the planet
    - **Puzzle structure**: NA
    - **Objectives**: What are the objectives of the game?

      Find lost equipment

      Hunt for food after finding destroyed rations

      Escape inhabitant captivity and reclaim stolen equipment

      Reach communications relay

      Fight communications officer

      Radio home to find out crash was intentional

      Begin salvaging parts to build a ship

      Build ship

      Fight inhabitants to prevent destruction of your ship

      Return home (space gameplay?)

      Fight nemesis

- **Mechanics**: What are the rules of the game, both implicit and explicit? This is the model of the universe that the game works under. Think of it as a simulation of a real world, how do all the pieces interact? This can actually be a very large section.
    - **Physics**: Normal
    - **Movement in the game**: Run and jump
    - **Objects**: Some moveable by pushing (for example, walking into them)
    - **Actions, including whatever switches and buttons are used, interacting with objects, and what means of communication are used**: Buttons to be pressed (exactly how will be figured out as development continues), communication will be via on-screen text
    - **Combat**: Shooting enemies with telefuser (need to remember this name)
    - **Economy**: None

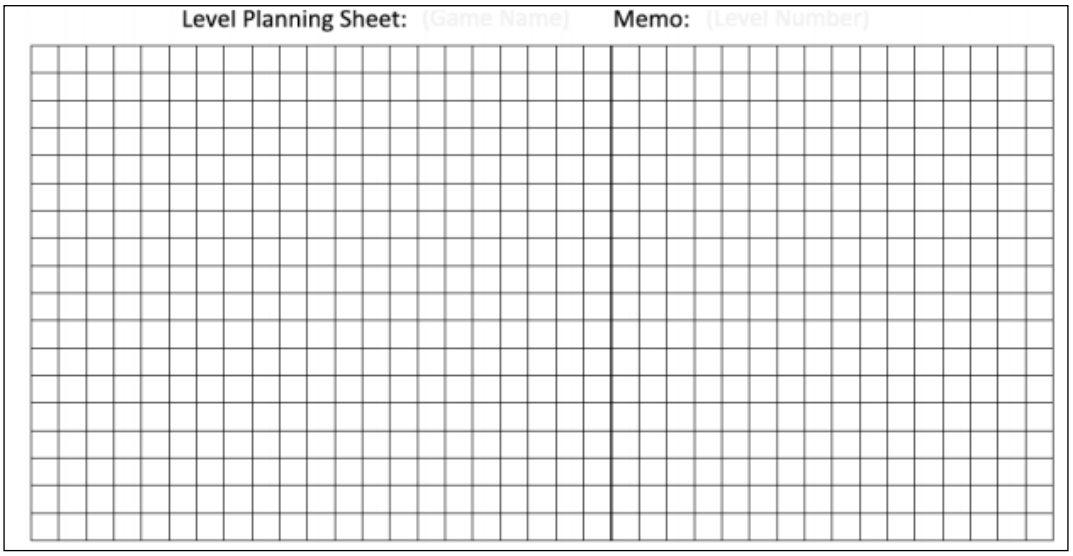
    - **Game options**: Possible difficulty, Facebook integration to post achievements, multiplayer
    - **Cheats and Easter Eggs**: To be figured out as development continues

- Story, setting and character
    - **Story and narrative**: Simple and sweet, your ship explodes on a space mission, vaulting you into deep space. You land in an unexplored world and have to locate your supplies and fight your way back home. You find out the explosion was planned as the main enemy wanted your position in the Space Army.
    - Game world
        - General look and feel of world: Dark and gloomy, almost forest like.
    - **Characters**: Jeff, the main player, is the head of the Space Army. Moly, the second in command, is the second playable character. Nemo is your arch nemesis who planned the destruction of your ship to take over your place in the Space Army.
- Levels
    - **Levels**: Each level should include a synopsis, the required introductory material (and how it is provided), the objectives, and the details of what happens in the level. Depending on the game, this may include the physical description of the map, the critical path that the player needs to take, and what encounters are important or incidental (see section 3.1.4 for level-progression ideas).
    - **Training level**: Locating your equipment will get the player used to the controls and gradually dip their feet into combat.
- Interface
    - **Visual system**: If you have an HUD, what is on it? Health, lives, and gun heat level
    - **Control system**: On-screen touch controls
    - **Audio, music, sound effects**: To be completed at a later time
    - **Help system**: Help screen on main menu and training level
- Artificial Intelligence
    - **Opponent and Enemy AI**: Simple pacing planet inhabitant combatants that shoot at you on sight
    - **Non-combat and friendly characters**: N/A

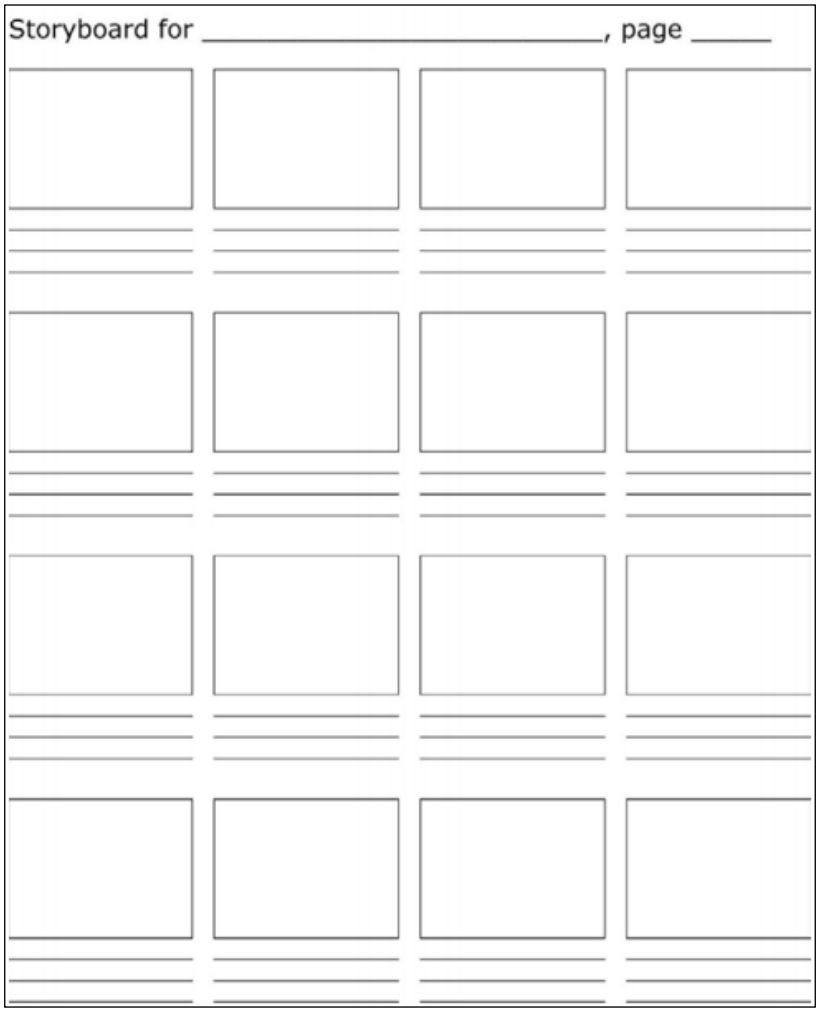
- Technical
    - **Target hardware**: iPhone and iPad
    - **Development hardware and software, including game engine**: SpriteKit
    - **Network requirements**: Bluetooth and possible wireless requirements for online activity
- **Game art**: In development

See? It's not that difficult! It's pretty in-depth but not all of it is required. I actually deleted quite a few lines because they weren't applicable to the game we are going to be creating. Now that the game is essentially planned, there are two other design documents you can make use of. The first is level design sheets, and the second is storyboards.

Level design sheets provide you with a grid to fill in to create your levels on paper:



Blank templates are supplied for you in the resource section of this book. I suggest printing them out and putting them in a binder so you can have a "game design book" as shown in the following image:



I use storyboards for designing cut scenes that will occur throughout the game. This is not required but, hey, it's still a good thing to have things planned out so the development will run quickly and smoothly.

I love the planning process! I think we are about ready to blast off and get into some development don't you? Yeah! That will have to wait until the next chapter though, I've rambled enough in this chapter already. There's a lot to take in when it comes to device performance and optimization, and it can be pretty confusing. I'm all self-taught, so I hope you learned something!

Onwards and upwards, so let's blast off into the development of our awesome game!

Summary

In this chapter we discussed asset creation, what programs to use, as well as the best formats to use. We then had a great comparison between file type rendering to show which file type is more efficient and easier on the system. We then discussed the design aspect of game development. We took a look at a game design document, as well as level design and storyboard documents. In the next chapter, we begin development!

3
Blast Off! Starting with Development

Now the fun really begins! We are going to start the development of our game! Are you as excited as I am? If you aren't, you should be! Just look at what we're going to cover:

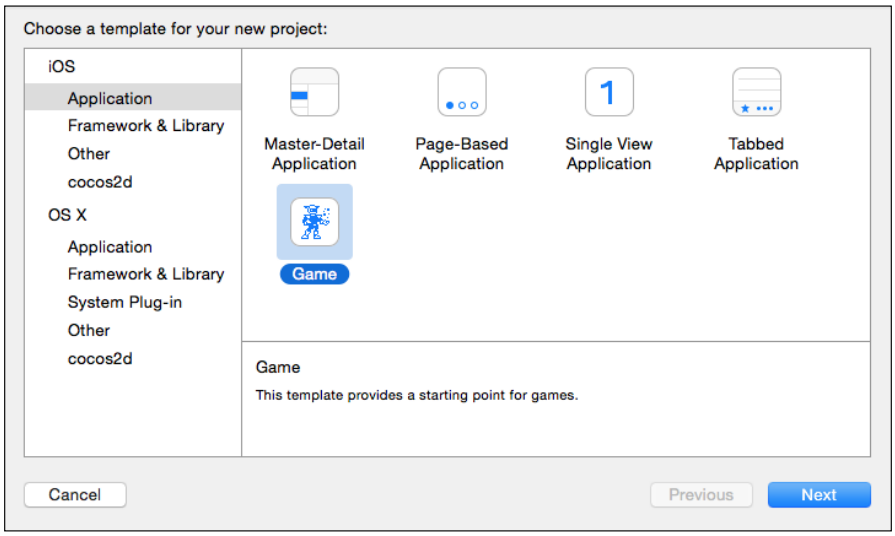
- Creating a SpriteKit project in Xcode
- Level design and implementation
- Gravity – player movement
- Collision detection

We will cover all these and a whole lot more!

Buckle up because we are going to shift into high gear, and we aren't going to look back! Well, maybe we will once or twice, but you get the point. Let's begin! Let's create our project!

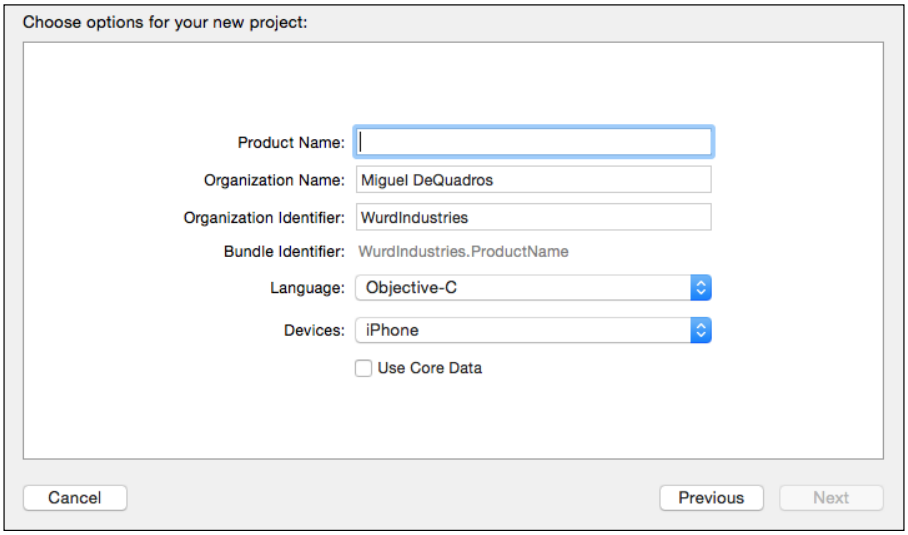
Creating a SpriteKit project in Xcode

Let's begin straight away by opening up Xcode and clicking on **File | New |Project**. You will then be greeted by the New Project wizard, which will look like this:



For this project, we are—obviously—going to select **iOS | Application | Single View Application** and then click on **Next**.

Once you do that, you will be required to fill in some details on the project, such as product name, organization name, and so on. See the following screenshot:

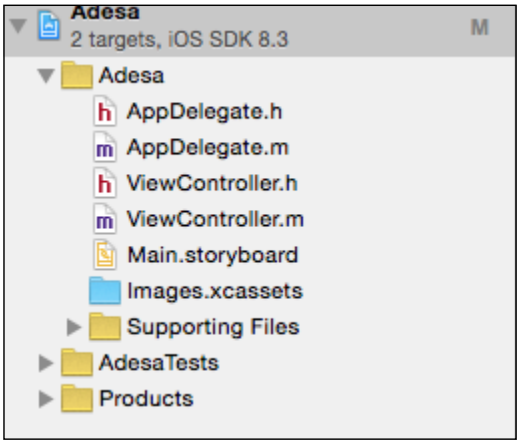


Fill in the required fields (if they aren't already populated) and then click **Next**. To begin, we are going to make some changes to the project. For instance, locate the `LaunchScreen.xib` file in the sidebar on the left and delete it by pressing *Delete* on your keyboard.

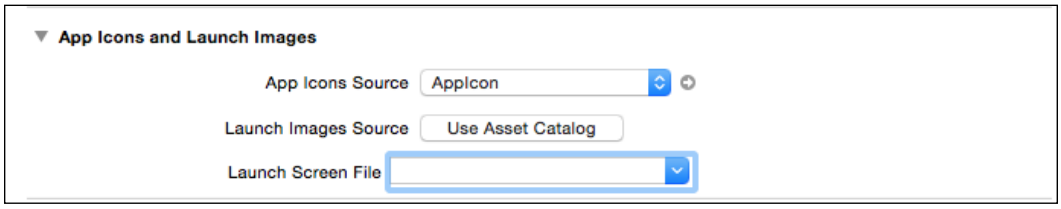
The `.xib` and storyboard files are the interface files that you can use to create the interface of a view. In them you can create buttons, text labels, and other user interface elements.

You will see a popup asking if you want to remove the reference or move it to the trash. We won't need it as we are going to program everything the player will see, so you can safely send it to the trash.

Then click on the main project file in the sidebar on the left; in the middle of the screen, you will see all the project's settings. We are also going to make some changes to this section.



First things first. Under the **App Icons and Launch Images** rollout, locate the **Launch Screen File** dropdown. Where it says `LaunchScreen.xib`, simply select the text and delete it as shown in the following screenshot. Again, because we are going to be programming everything, we won't need it in our case.



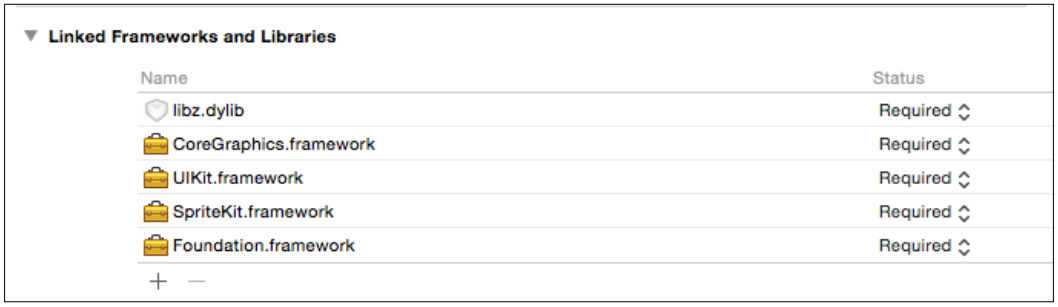
Look down a little further and you will see the **Linked Frameworks and Libraries** roll out.

Frameworks and **Libraries** are extensions you can add to further enhance the functionality of your app. Examples of what you can add are CoreGraphics and SpriteKit; even the Facebook API or Cocos2D can be downloaded and added for additional features.

This is where we add our various frameworks (if you didn't know that already). We are going to add five frameworks and one library. To do this, click on the **+** button at the bottom of the section. The files we are going to add are as follows:

- `libz.dlib`
- `CoreGraphics.framework`
- `UIKit.framework`
- `SpriteKit.framework`
- `GLKit.framework`
- `Foundation.framework`

Here is a screenshot of **Linked Frameworks and Libraries**:



Our game is going to make good use of these frameworks, as we will see throughout this chapter. I like to keep things nice and organized whereas Xcode just throws files at the top of the sidebar on the left-hand side. I selected the new frameworks that were just added, right-clicked on them, selected **New Group from Selection**, and called that folder `Frameworks`. I personally like to keep things good and organized when developing; I even make sure all the paragraphs of code are indented properly.

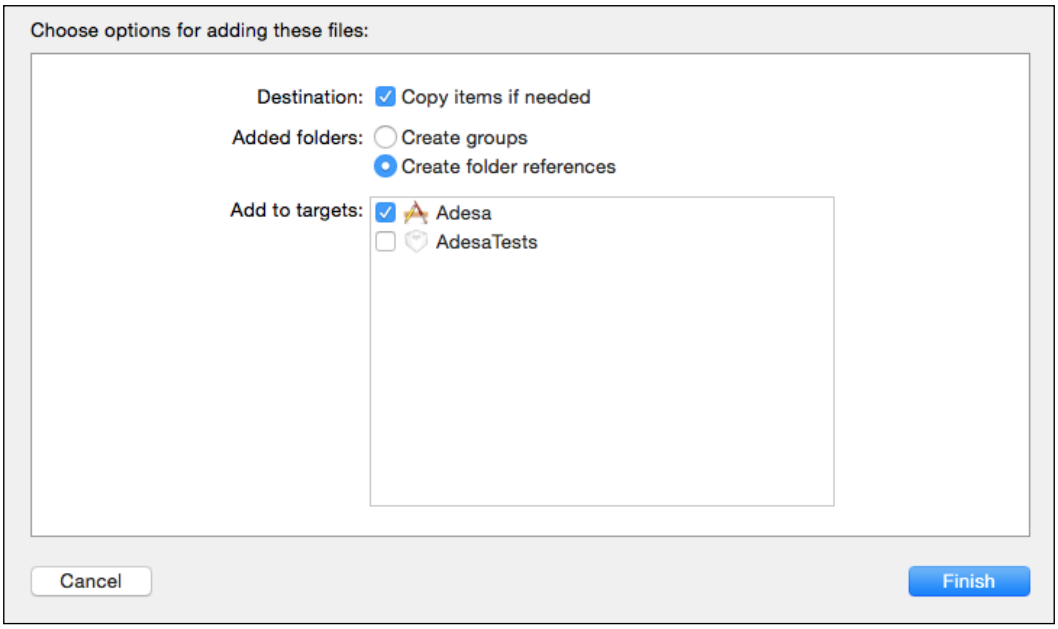
After you've added the new frameworks, you must download the two folders in the resource section of this book. They include additional frameworks that are provided for free online and that will assist us further, especially with our level design.

The Frameworks are `SKTUtils`, which is essentially an extension of `SpriteKit`, and `JSTileMap`, which we will use for our maps.

The `SKUtils` framework is an extension of `SpriteKit` that adds more visual effects as well as further mathematical calculations for additional physics (such as the use of π (pi) in calculations for even more complex physics calculations).

The `JSTileMap` framework allows us to import a tile map file into Xcode. As you will find out later, Xcode doesn't natively support map files.

Once you have them downloaded, simply select the two folders and drag them into the bar on the left-hand side of our project, as shown in the following screenshot:



Make sure you click on the **Copy items if needed** check box; this will copy these folders into your project folder so that any changes do not affect the originals.

Importing the `SKTUtils` framework, I found, threw 20 errors at me, most of which were completely incomprehensible. So after much deliberation and debugging, I realized (halfway through writing this chapter, mind you), that we need to add a **prefix header**. The prefix header file is created to precompile headers a lot faster. So instead of compiling each header one by one, they are compiled once and way ahead of time.

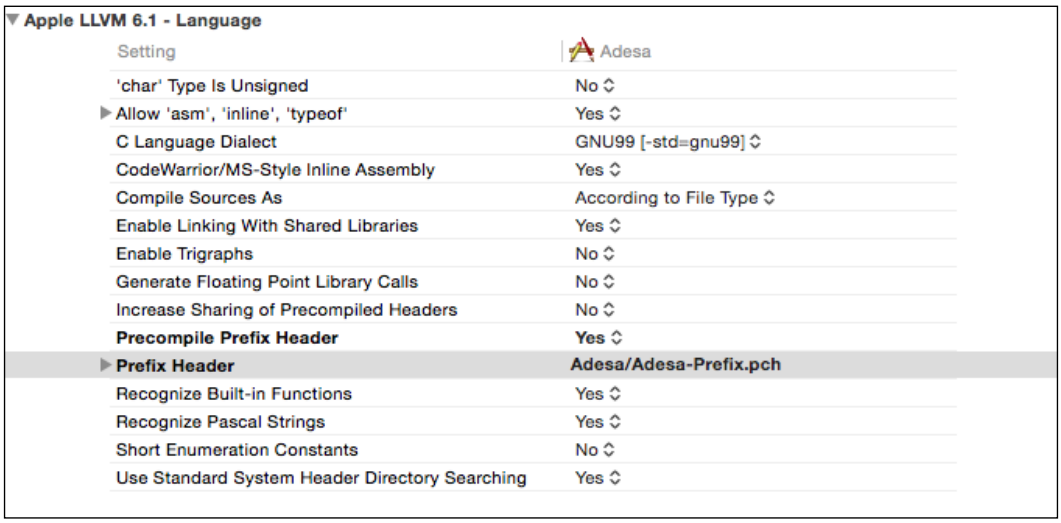
Click on **File** | **New** | **File**. Once the wizard appears, click **Other** under **iOS**, select **PCH File (PreCompiledHeader File)**, and name it something like ***yourProjectName*-Prefix.pch**.

Once that file is created, click on it to edit it; we are going to fill it with the following code:

```
#import <Availability.h>

#ifndef __IPHONE_5_0
#warning "This project uses features only available in iOS SDK 5.0 and
later."
#endif

#ifdef __OBJC__
    #import <UIKit/UIKit.h>
    #import <Foundation/Foundation.h>
#endif
```

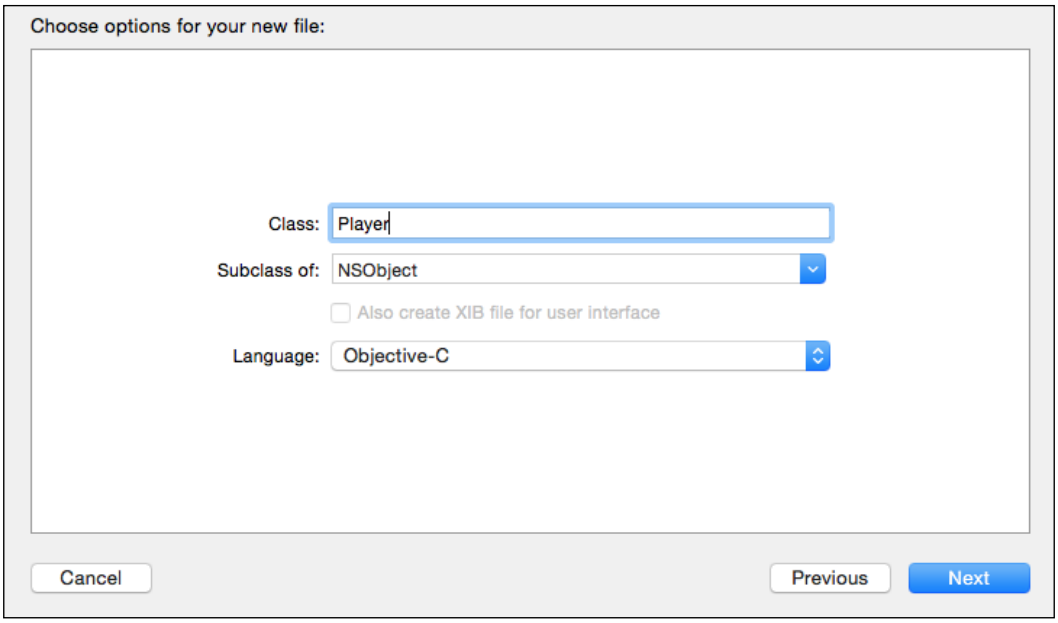
We need to define our PCH file we just created in Xcode or else it will be rendered useless because it won't be doing anything!

Once we are done filling out that file, we are going to click on our project in the bar on the left. You will now see tabs at the top of the navigation bar: **General**, **Capabilities**, **Info**, **Build Settings**, **Build Phases**, and **Build Rules**. Click on **Build Settings**. Scroll down to find **Apple LLVM 6.1 – Language** and change the **Precompile Prefix Header** selection to **Yes**. Below that, double-click on the empty field in the **Prefix Header** section and fill it out according to your project. I named my project Adesa, so I filled in **Adesa/Adesa-Prefix.pch**, as shown in the following screenshot:



Moving on!

Now we have to add in some more main and header files; one set will be called **Player**, and the other will be **GameLevelScene**. Unsure how to add these? No problem! Simply click **File** | **New** | **File** or press *Command* + *N*. For these files, we will select the `Cocoa Class` file and click **Next**, as shown in the following screenshot:



Again fill out the name of the **Class**, and again one will be **Player** and the other **GameLevelScene**. I know setting things up is a little boring, but we have to do it. Now, on to editing our code!

Editing our code files

We need to start making a few changes to our source code files. We will start with the `ViewController` interface set of files, the `.h` and `.m` files, that is—the files that control a defined view. We will start with the `.h` file; at the moment, it should read like this:

```
#import <UIKit/UIKit.h>

@interface ViewController : UIViewController

@end
```

We need to add `#import<SpriteKit/SpriteKit.h>` directly below the line `#import <UIKit/UIkit.h>`.

WHAT?!

This line we just inserted into our code imports the SpriteKit framework into that header file so that we can access the features of SpriteKit in the header we are working on.

That's all for the `.h` file for the moment; we are going to be bouncing back and forth between files as we set our project up.

On to the `ViewController.m` file, which should be edited to look like this:

```
#import "ViewController.h"
#import "GameLevelScene.h"

@implementation ViewController

-(void)viewDidAppear:(BOOL)animated
{
    [super viewDidAppear:animated];

    //Configure the view.
    SKView * skView = (SKView *)self.view;
    skView.showsFPS = YES;
    skView.showsNodeCount = YES;

    //Create and Configure the scene.
    SKScene * scene = [GameLevelScene sceneWithSize: skView.bounds.
size];
    scene.scaleMode = SKSceneScaleModeAspectFill;

    //Present (or show) le scene.
    [skView presentScene:scene];
}

- (BOOL)shouldAutorotate
{
    return YES;
}

- (NSUInteger)supportedInterfaceOrientations
{
```

```
    return UIInterfaceOrientationMaskLandscape;
}


- (void)didReceiveMemoryWarning {
    [super didReceiveMemoryWarning];
    // Dispose of any resources that can be recreated.
}

@end
```

I know what you're thinking: *What does all this mean?* I'll start from the beginning to make it easy on those who don't have coding experience:

```
#import "ViewController.h"
#import "GameLevelScene.h"
```

Again we have the import line; we import the `ViewController` header file where everything will be declared. Next we have the `GameLevelScene` header file; this will host the game scenes, where we will spend probably the majority of time in this file, as shown in the following code:

```
-(void)viewDidAppear:(BOOL)animated
{
    [super viewDidAppear:animated];

    //Configure the view.
    SKView * skView = (SKView *)self.view;
    skView.showsFPS = YES;
    skView.showsNodeCount = YES;

    //Create and Configure the scene.
    SKScene * scene = [GameLevelScene sceneWithSize: skView.bounds.
size];
    scene.scaleMode = SKSceneScaleModeAspectFill;

    //Present le scene.
    [skView presentScene:scene];
}
```

With this paragraph of code, we have set up the SpriteKit view. The first line, `SKView * skView = (SKView *)self.view;`, declares a new instance of a SpriteKit view, named `skView`, and places it in the `ViewController` header file's view, hence `self.view`.

The next two lines are completely optional as they are more for testing purposes, and you will delete them before you release your app. The first line shows the frames-per-second count, and the next shows the number of nodes or objects within the scene. Feel free to remove these lines now if you don't want to see them.

The next line actually creates and configures the scene. As `SKScene * scene = [GameLevelScene scenewithSize: skView.bound.size]` denotes, the data is being pulled from the `GameLevelScene` set of files we still have to set up. For now it will throw an error saying there is no known class method for selector `sceneWithSize`, but don't worry because we haven't done any declarations in the `GameLevelScene` set of files.

After that, we see the scaling for the scene which is set to `AspectFill`. You can select either aspect fit or stretch, but for this example we will use aspect fill. You can change it if you like.

Finally we present the scene!

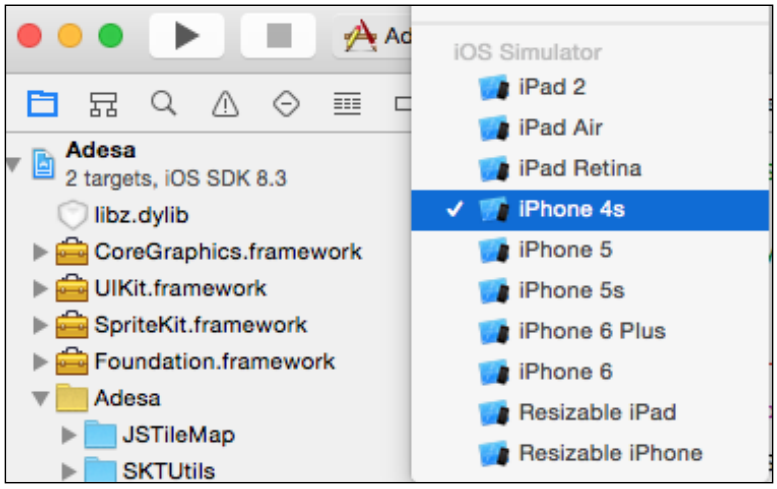
Let's bounce on over to the `GameLevelScene` files. We will again start with the header file, which should read like this:

```
#import <SpriteKit/SpriteKit.h>

@interface GameLevelScene : SKScene

@end
```

Simply, we have changed the `GameLevelScene` class to a `SpriteKit` scene; so now, when we go to build our project, the previously mentioned error will disappear. On to the `.m` file, which will read like this:

```
#import "GameLevelScene.h"

@implementation GameLevelScene

-(id)initWithSize:(CGSize)size {

  if (self = [super initWithSize:size]) {


  }
  return self;
}

@end
```

Again, this is a simple initializing of the scene and setting the size. Now let's build our project! You don't have to have a device plugged in at the moment; we will run on the simulator. If you're running an older computer, I suggest selecting the iPhone 4S for the simulator, as shown in the following screenshot, mainly because it takes less time to load and the screen size is small enough to fit nicely on your screen.



Simply click on the button shown in the previous screenshot (the one beside the play and stop button), and select whatever device you like for the simulator.

Now let's build the project. Click on the **Play** button at the top, or press *cmd* + *B* (or *cmd* + *R* to build and run the project).

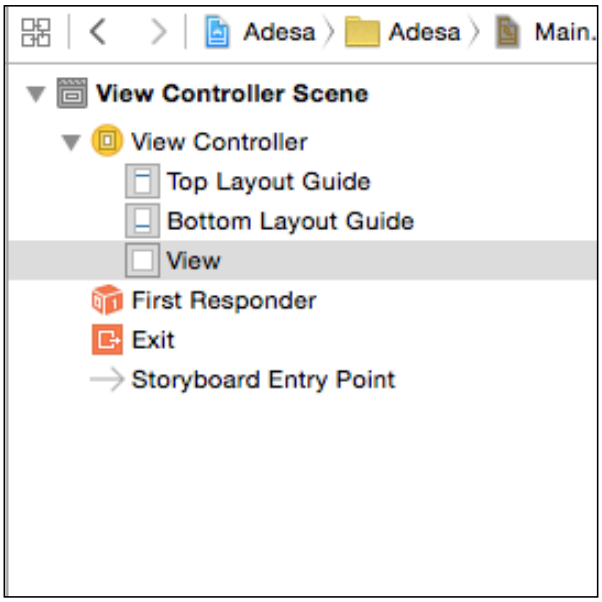
If all goes well, it should build successfully and the iOS simulator should appear. Uh oh! Did you get an error that looks like this?

```
0x22d9a <+138>: movl   %eax, -0x10(%ebp)
2015-04-30 11:33:20.916 Adesa[25543:4462832] -[UIView setShowsFPS:]:
unrecognized selector sent to instance 0x7a660940
2015-04-30 11:33:20.918 Adesa[25543:4462832] *** Terminating app due
to uncaught exception 'NSInvalidArgumentException', reason: '-[UIView
setShowsFPS:]: unrecognized selector sent to instance 0x7a660940'
*** First throw call stack:
(
(LOTS OF WRITING HERE)
)
libc++abi.dylib: terminating with uncaught exception of type
NSException
(lldb)
```

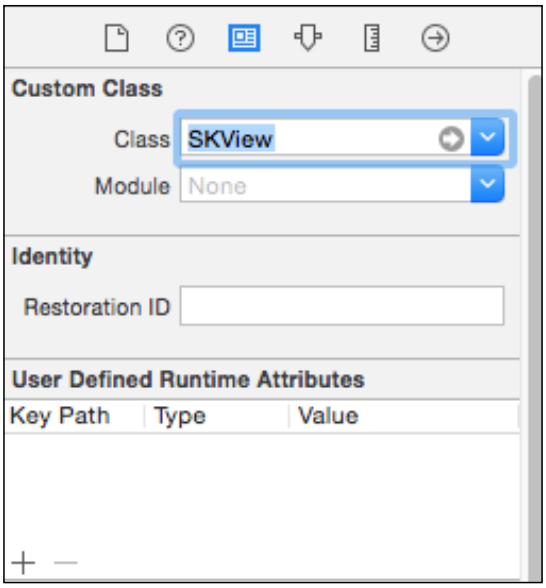
We also need to make changes to the `UIViewController` header file in the Storyboard.

On the left-hand bar of our project, locate the `Main.storyboard` file (remember we discussed the storyboard and `.xib` files earlier?), and click on it. We can arrange the UI of the apps in this file, but we probably won't use it in this project. Now you will see a blank iOS layout, ripe for changing.

All we are going to do here is launch the **ViewController Scene** rollout on the sidebar, just next to the bar where all our project files are, then roll out the **View Controller** rollout, and click on **View**, as shown in the following screenshot:



Now look over to the bar on the far right, and you will see the top bar has six different buttons. Click the third one, in which you see **Show the Identity Inspector**. Now you will see a section right below the buttons called **Custom Class**. In the **Class** text field in this section, we are going to type in **SKView**, as shown in the following screenshot:



Let's try running our project yet again. Does it build successfully? Good! Does it open the iOS simulator? Awesome! Does it show a blank device with the frame rate and zero nodes printed on the bottom of the screen? Amazing! We are right on track!

For our final changes, we are going to quickly edit the `Player` set of files. At this moment, all we have to do is edit the header file. Change it so it reads as follows:

```
#import <SpriteKit/SpriteKit.h>

@interface Player : SKSpriteNode

@end
```

We are now going to work on "pause for effect."

Level design and implementation

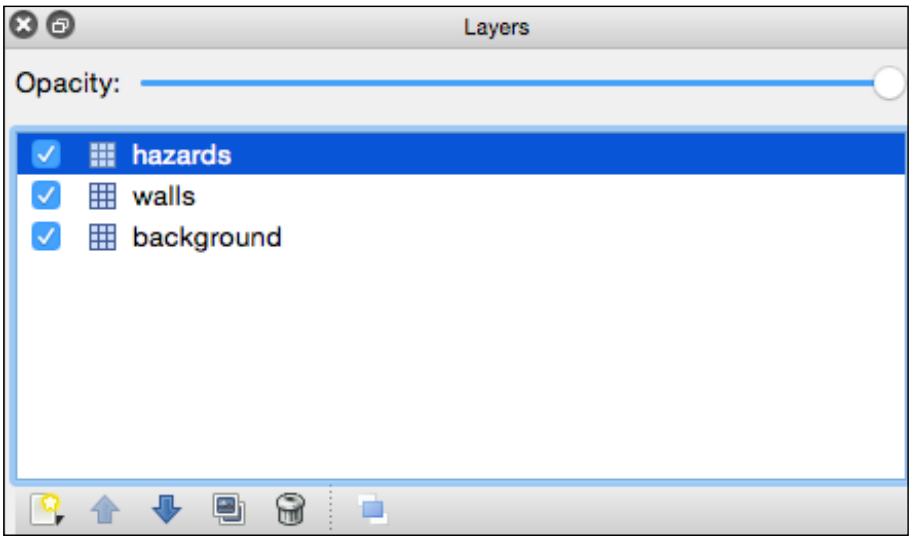
This is where we can have a lot of fun! We are now going to start designing our levels. You can spend hours adding the little details to each level to make them look perfect. Remember, a well-designed level will impress players and will give your game a professional look.

Unfortunately, Xcode just doesn't have the drag-and-drop ease of level design, so we are going to create our maps using a third-party program called *Tiled Map Editor*, which can be downloaded for free at `www.mapeditor.org`.

I have included a sprite sheet for our levels as well as a built level in the *Resources* section of this book. Do you remember the `JSTileMap` library that was included as well in this chapter? It's what is going to display these maps as SpriteKit doesn't support TMXTileMaps. So sad!

Anyway, moving on! Open up the level I have included, entitled `level1.tmx`, and get a feel for how the program works.

The side bar of Tiled shows you the different layers of the levels. In this case, we have **hazards**, which are things like spikes and other objects that could prove detrimental to our player's health. Then we have **walls**, fairly self-explanatory, and then **background**, for scene elements such as clouds and trees.



Try making some edits to the level, and make it your own little masterpiece. The following screenshot shows all the tools you need; the selected tool in the image is the stamp tool, which allows you to place the selected image in the scene on the selected layer. Then you have the Paint Bucket and Eraser tools. Give it a try! When you're done, we are going to program the level into our game.

Let's open up our `GameLevelScene.m` file and add `#import "JSTileMap.h"` at the top of the file, under `#import "GameLevelScene.h"`.

Directly under the import we just inserted, we are going to add in the following lines:

```
@interface GameLevelScene()
@property (nonatomic, strong) JSTileMap *map
@end
```

This is adding a private variable for the map we will be using into our `GameLevelScene` class.

Now we are going to actually load the map. In the `(id)initWithSize:(CGSize) size` block of code, inside the `if` statement, add in the following code to change the color of the sky, as well as load the map:

```
self.backgroundColor = [SKColor colorWithRed:.25 green:.0 blue:.25
alpha:1.0];

self.map = [JSTileMap mapNamed:@"level1.tmx"];
[self addChild:self.map];
```

Run the project to see your awesome level is now in place on the screen. I decided to use a dark purple color for these levels because it's the environment I'm going for. You can go in for whatever you like, but remember to just adjust the colors accordingly.

If for some reason it's not showing up correctly or you are getting an error, make sure your `GameLevelScene.m` file now looks like this:

```
#import "GameLevelScene.h"
#import "JSTileMap.h"

@interface GameLevelScene()
@property (nonatomic, strong) JSTileMap *map;
@end

@implementation GameLevelScene


-(id)initWithSize:(CGSize)size {
    if (self = [super initWithSize:size]) {
        //CUS ITS SKY FALL!...This makes the background purple...
        self.backgroundColor = [SKColor colorWithRed:.25 green:.0
blue:.25 alpha:1.0];

        self.map = [JSTileMap mapNamed:@"level1.tmx"];
        [self addChild:self.map];
    }
```

```
        return self;
    }

    @end
```

Looks pretty cool eh? Tiled is a great program, and you can add a whole lot more detail then I did here.

Something's missing, though, isn't it? Hmm... Oh yes! Our player!

Gravity – player movement

For now, import the `Player` images (in the `sprites.atlas` file) I have provided in the resources section of this book. Unless you have your own! Then by all means use yours.

Let's go back to our `GameLevelScene.m` file and import yet another file. This time, it will be `#import "Player.h"`. Then, after the `@interface` section that we added earlier, we are going to add another property similar to the map property we just added: `@property (nonatomic, strong) Player *player1`. I used player1 because we will be adding in some multiplayer features later!

Then again, inside the `initWithSize` function we will add the following code:

```
self.player = [[Player alloc] initWithImageNamed:@"P1Idle"];
self.player.position = (CGPointMake(100, 50);
self.player.zPosition = 15;
[self.map addChild: self.player];
```

When we build and run our project, we should get similar results as seen in the following image. I think it's starting to look pretty cool!

Hold on cowboy! What did we just do? Well let me explain what happened. The code we just added loaded our little space man as a sprite object, positioned him on the map, and then added him to the map object.

Here's something to keep in mind, as you may be wondering why our little guy is added to the map, instead of adding him directly to the scene. Well, let me tell you! It's all about control. I like to be in full control (and my wife says I can tell people that), and we want to control exactly which layers from our map are in front of and behind our little guy. So for example we can set the background objects, such as the trees and hills, to be way in the back, but if we want to we can position our space dude to be behind them. So then he needs to be a subobject, or a child, of the map class.

Now we need to make this guy move! First off, let's add some gravity into the scene. Back in our `GameLevelScene.m` file, we are again going to add another property in the same location we've been adding them, in the `Interface` section of our code. It will be this:

```
@property (nonatomic, assign) NSTimeInterval previousTime;
```

Refer to the following screenshot to ensure your code is filled out correctly:

```
#import "GameLevelScene.h"
#import "JSTileMap.h"
#import "Player.h"

@interface GameLevelScene()
@property (nonatomic, strong) JSTileMap *map;
@property (nonatomic, strong) Player *player1;
@property (nonatomic, assign) NSTimeInterval previousTime;
@end

@implementation GameLevelScene
```

After this we are going to code in an `update` method, which we will place just before the `-(id)initWithSize:(CGSize)size {` line of code. To create this method, we will add this code:

```
- (void)update:(NSTimeInterval)currentTime
{

    NSTimeInterval delta = currentTime - self.previousTime;

    if (delta > 0.02) {
        delta = 0.02;
    }

    self.previousTime = currentTime;

    [self.player1 update:delta];
}
```

This will throw an error at the moment, saying there is no visible `@interface` for a player that declares the update selector. Confusing, right? Let's take it one step at a time and explain what we just did here.

Firstly, we added in this update method, which is automatically built into a SpriteKit scene, or `SKScene` object. All we have to do is code it in! Then every frame that we call before the scene will be rendered accordingly. The update method provides us with a timer value, or the `NSTimerInterval` value, that is the current time of our program.

Secondly, we get the `delta` value, which is the current time, subtracted by the previous time. What's the `delta` value? It's essentially the interval since the previous time the update was called. With this `delta` time value, we can create movement, gravity, and other forces with a neat and smooth motion, as well as smooth animations.

Then we have an `if` statement; if the `delta` value goes above 0.02, it is kept at 0.02. Sometimes our devices lag, stutter, or slow down, and especially when first booting the game up, the device has a lot to load; thus, this `delta` value could be quite large. We keep it at a consistent value to reduce the chances of the physics acting weird. Why would this happen?

Like I mentioned before, this `delta` value creates smooth movement and gravity by keeping a neat consistent value. If that value goes way over 0.02, the movement or any external forces we program won't work correctly and bad things could occur that we don't want happening. Call it preventative measures so we don't create any game-breaking bugs. After the `if` statement, we then set the current time (which for example could be 0.02), as the previous time so the device can determine the `delta` value. Let's clarify, in order to determine how quickly the time is advancing, we have the current time, and previous time, so before the time advances we set the current time as the previous time, and then the current time advances.

Broken down, the clock begins to count, the current time is 0.05, before the time advances we will set the previous time variable to 0.05, then the current time will increase to 0.06. The device will then calculate the delta value. Make sense? I know it's a lot to take in.

Then we come to the line of code that causes the error. It's calling the error because we haven't implemented the `update` method in our `Player` set of files (or class). Let's do that now!

Go in to our `Player.h` file, and change the code so it looks like this:

```
#import <SpriteKit/SpriteKit.h>

@interface Player : SKSpriteNode //(These lines should be here
already)

@property (nonatomic, assign) CGPoint velocity;
-(void) update:(NSTimeInterval)delta;
@end
```

Whoa, whoa buddy, this doesn't make sense! When programming a game, there's a lot of physics and math that goes into the coding. In the `@property (nonatomic, assign) CGPoint velocity;` line, we are creating a property that will measure how fast the player is moving.

What's a **CGPoint**, you ask? **CG** stands for **CoreGraphics** which is the main graphic rendering framework used by iOS devices, and **point** is a point on the screen, so `CGPoint` holds the positional value of an object; in this case the `velocity` will now have an `x` and `y` value, allowing us to calculate the exact speed the player is moving at and the exact direction they are moving in. Fun!

Let's hop on over to the `Player.m` class file and change the code to the following:

```
#import "Player.h"

#import "SKTUtils.h"

@implementation Player


- (instancetype)initWithImageNamed:(NSString *)name {
    if (self == [super initWithImageNamed:name]) {
        self.velocity = CGPointMake(0.0, 0.0);
    }
    return self;
}

- (void)update:(NSTimeInterval)delta {

    CGPoint gravity = CGPointMake(0.0, -450.0);

    CGPoint gravityStep = CGPointMultiplyScalar(gravity, delta);

    self.velocity = CGPointAdd(self.velocity, gravityStep);
    CGPoint velocityStep = CGPointMultiplyScalar(self.velocity,
delta);

    self.position = CGPointAdd(self.position, velocityStep);
}

@end
```

Let's break down what we just did here. I know things can be confusing, but don't worry! I'm here to help you!

First, we imported the `SKUtils` framework into the `Player` class. After that, we created a new `initWithImageNamed` method and initialized the velocity variable to `0.0`. Then we declared the value of the gravity force. Each time the update method runs, we are increasing the velocity of the player 450 points downward. If the player starts out by standing still, after one second he'll be moving at 450 pixels per second, at two seconds that value will be doubled, and so forth.

Sounds easy.... right?

Next, we used `CGPointMulitplyScalar` to decrease the acceleration down to the size of the current time step. Remember, `CGPointMulitplyScalar` increases the `CGPoint` values by a float value, and returns the `CGPoint` result. This is great because, when the device lags or for some odd reason we see a drop in frame rate, we will still get a consistent acceleration value.

In the `self.velocity =... block`, we calculate the gravity for the current time and then add it to the player's current velocity. With the new velocity calculated, we get the velocity for a single time step.

Finally, with the velocity all calculated, we use the `CGPointAdd` function to change the position of the player. As you can see, `CGPointAdd` is equal to the player's current position, with the gravity added.

OK! Let's run our project!

What? He's falling through the ground? Hmm, I guess that means we need to detect collisions now?

Collision detection

We all know that collision detection is imperative to any game. Regardless of whether it's a hockey game, or Call of Angry eight year olds, the game needs to detect collisions of hockey pucks, bullets, swords, feet on the ground—you name it and there are a ton of collisions that need to be detected. For our game, we are only going to detect simple boxes colliding with each other between the player and enemies, the platforms, and the bullets colliding with enemies.

We are going to make things super easy; first we are going to detect the player's bounding box.

What's a bounding box? Simple! Think of when you're creating your sprites; there's a box surrounding your image? You could think of a collision bounding box like that, the small box that your sprite fits in or within a defined space. You can (and we will) adjust the size of the bounding box to fit the sprite; you can have it larger or smaller depending on your needs.

With that all explained, let's hop into our `Player.h` file and add the following line:

```
-(CGRect)collisionBox;
```

We just created a core-graphics rectangle named `collisionBox`. Easy, right?

Now add the following code into `Player.m`:

```
-(CGRect)collisionBox {
  return CGRectInset(self.frame, 2, 0);
}
```

The value of `CGRectInset` decreases the size of the rectangle, or our collision box, by the last two bracketed numbers, `2` and `0` respectively. We set the player's frame as the base size, and then shrink it by `two` pixels on each side of the player. If you want, you don't have to shrink the bounding box, you can leave the two values at `0`.

Now things will start to get a little more complicated. We need to detect the various images in our level and select which one we want our player to collide with and those that we don't. Let's shoot on over to our `GameLevelScene.m` class and add in the following code:

```
-(CGRect)tileRectFromTileCoords:(CGPoint)tileCoords {
  float levelHeightInPixels = self.map.mapSize.height * self.map.
tileSize.height;
```

This first block locates the pixel origin coordinate; we do this so we know exactly where to place the map in the scene. We need to flip the height coordinate because SpriteKit's origin is in the bottom-left corner of the screen but the tile map's origin is at the top left. In order to detect the origin, we need to add in the following code:

```
  CGPoint origin = CGPointMake(tileCoords.x * self.map.tileSize.
width, levelHeightInPixels - ((tileCoords.y + 1) * self.map.tileSize.
height));


  return CGRectMake(origin.x, origin.y, self.map.tileSize.width, self.
map.tileSize.height);
}
```

Next we add `1` to the tile coordinate. Why do we do this? Actually, the tile coordinate system starts at `0`, so if we have 50 tiles, the 50th tile's actual coordinate will be 49. So we need to add one to get the right value.

```
- (NSInteger)tileGIDAtTileCoord:(CGPoint)coord forLayer:(TMXLayer *)
layer {
  TMXLayerInfo *layerInfo = layer.layerInfo;
  return [layerInfo tileGidAtCoord:coord];
}
```

This next method accesses our saved map's layer info that is saved in *Tiled Map Editor*. Remember we had three layers: Background, walls, and hazards? If you want to have more than three layers, absolutely feel free to do so. I am only doing three in this example because we are going to add in our enemies and some special effects programmatically. This block of code will access those layers.

For our collision system, we are going to detect the surrounding eight tiles of our player. In the following blocks of code, we are going to detect the surrounding tiles, which will inspect the `CGRect` (or Core-Graphics rectangles) for a collision with the player's collision bounding box.

Let's go back to our `GameLevelScene.m` file and add in the following code:

```
- (void)checkForAndResolveCollisionsForPlayer:(Player *)player
forLayer:(TMXLayer *)layer {

    NSInteger indices[8] = {7, 1, 3, 5, 0, 2, 6, 8};
    for (NSUInteger i = 0; i < 8; i++) {
        NSInteger tileIndex = indices[i];
```

```
            CGRect playerRect = [player collisionBox];

            CGPoint playerCoord = [layer coordForPoint:player.position];

            NSInteger tileColumn = tileIndex % 3;
            NSInteger tileRow = tileIndex / 3;
            CGPoint tileCoord = CGPointMake(playerCoord.x + (tileColumn -
1), playerCoord.y + (tileRow - 1));

            NSInteger gid = [self tileGIDAtTileCoord:tileCoord
forLayer:layer];

            if (gid) {

                CGRect tileRect = [self tileRectFromTileCoords:tileCoord];

                NSLog(@"GID %ld, Tile Coord %@, Tile Rect %@,
player rect %@", (long)gid, NSStringFromCGPoint(tileCoord),
NSStringFromCGRect(tileRect), NSStringFromCGRect(playerRect));
                //after this is where we write our collision resolving
            }

        }
    }
```

Wow! So much code! Let's break it down. The first block creates an array that shows the position of the tiles surrounding our cool little player dude. As you can see, we find the eight surrounding tiles and then we store those values in the `tileIndex` variable.

Now, remember what I said about the tile coordinates being flipped? Notice the order of tiles? `7`, `1`, `3`, `5`, `0`, `2`, `6`, `8`. Tile 7 is the tile that is directly below our player, so it needs to be figured out right away. We need to know if he's on the ground or not; if he is, he can jump, but if not, no jumping! If we don't resolve this tile immediately, the player could potentially jump without the character touching the ground—if they pressed the jump button quickly enough.

Then we retrieved the player's collision box that we coded earlier and found the exact tile location of the player. We did this so we could then locate the surrounding tiles of our player. After we located the position of the player, we then divided the `tileIndex` variable we created earlier to find the row and column values around the player.

Let's break this down into an example.

Say the value of `tileIndex` is `3`; the value of `tileColumn` would be `0` (3 % 3 = 0) and the value of `tileRow` would be `1` (3 / 3 = 1).

If our space guy's position was found to be at tile coordinate (`50`, `10`), the surrounding tile at `tileIndex` `3` would be 50 + (0 – 1) and 10 + (1 – 1) or `49` and `10`, respectively. This equals the tile directly to the left of our space man's tile position.

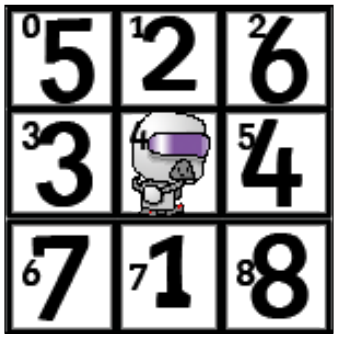
I know that this can be a little confusing, but don't worry; you'll start to get it soon!

In the next step, we look up the GID value for the tile at the coordinate found in the previous instance of `tileIndex`.

Woah, halt! What in the wide, wide world of sports is a GID?

A **GID** is the number that represents the index of an image from a tile set. Each `TMXLayer` class has a tile set that has images arranged in a grid. Simply put, the GID is the position of a particular image.

Next we figure if the GID has a value of `0`. There is no tile. It's just blank space, so we don't resolve or test a collision. However, if there is a value in the GID, we get the `CGRect` position for that tile. Then we simply log the results. This is not a required block of code, but it is very helpful when things aren't working properly—you can look at the debugger to see what's going on. The following figure shows you how tiles are handled:



The big bold numbers represent the order in which tile collisions are handled, bottom first, top second, left-hand side third, and right-hand side fourth, then the corners. The smaller numbers represent the order in which those tiles are stored in the `tileIndex` variable.

Next we are going to go back into our `GameLevelScene.m` file, and add in the following lines:

```
// Add to the @interface section with all our other properties
@property (nonatomic, strong) TMXLayer *walls;

// Add to the init method, after the map is added to the layer
self.walls = [self.map layerNamed:@"walls"];

// Add to the bottom of the update method
[self checkForAndResolveCollisionsForPlayer:self.player1
forLayer:self.walls];
```

If you were to run the project now, it would just crash into the oblivion of **SIGABRT** (**signal abort**). SIGABRT signifies that, though an error wasn't shown within your code, your app failed when it attempted to run a section of code. You will be able to see what's going on and why it happened in the console log. We are going to further discuss debugging later in this book. We need to do a little more work.

When our `Player` class updates its position and the `GameLevelScene` class detects a collision, we will want the player to stop. So we need to create a new variable.

This variable will allow the `Player` class to do all its positioning calculations, and the `GameLevelScene` class will update the position after the collisions have been detected.

Let's go on over to our `Player.h` file and add this new property:

```
@property (nonatomic, assign) CGPoint desiredPosition;
```

We also need to make changes to our `collisionBox` method in the `Player.m` file, which should now read as follows:

```
-(CGRect)collisionBox {
    CGRect boundingBox = CGRectOffset(self.frame, 2, 0);
    CGPoint difference = CGPointSubtract(self.desiredPosition, self.
position);
    return CGRectOffset(boundingBox, difference.x, difference.y);
}
```

This creates a collision bounding box based on the desired position. The layer will now use this for collision detection.

Now let's scroll down to our `update` method and locate this line:

```
self.position = CGPointAdd(self.position, velocityStep);
```

Replace it with the following:

```
self.desiredPosition = CGPointAdd(self.position, velocityStep);
```

Now this will update our `desiredPosition` property instead of the actual position property.

Back in our `GameLevelScene.m` file, look for our `-(void)checkForAndResolveCollisionsForPlayer:(Player *)player forLayer:(TMXLayer *)layer` method. We wrote `CGPoint playerCoord = [layer coordForPoint:player.position];` we have to change this from `player.position` to `player.desiredPosition`.

Back into our `checkForAndResolveCollisionsForPlayer` method, after the commented out text `//after this is where we write our collision resolving`, we need to add in our collision resolution code. So that there's no confusion, the `checkForAndResolveCollisionsForPlayer` method should look like this:

```
- (void)checkForAndResolveCollisionsForPlayer:(Player *)player
forLayer:(TMXLayer *)layer
{
    NSInteger indices[8] = {7, 1, 3, 5, 0, 2, 6, 8};
    player.onGround = NO;  ////Here
    for (NSUInteger i = 0; i < 8; i++) {
        NSInteger tileIndex = indices[i];

        CGRect playerRect = [player collisionBox];
        CGPoint playerCoord = [layer coordForPoint:player.
desiredPosition];

        NSInteger tileColumn = tileIndex % 3;
        NSInteger tileRow = tileIndex / 3;
        CGPoint tileCoord = CGPointMake(playerCoord.x + (tileColumn -
1), playerCoord.y + (tileRow - 1));

        NSInteger gid = [self tileGIDAtTileCoord:tileCoord
forLayer:layer];
        if (gid != 0) {
            CGRect tileRect = [self tileRectFromTileCoords:tileCoord];
            //NSLog(@"GID %ld, Tile Coord %@, Tile Rect %@,
player rect %@", (long)gid, NSStringFromCGPoint(tileCoord),
NSStringFromCGRect(tileRect), NSStringFromCGRect(playerRect));

            if (CGRectIntersectsRect(playerRect, tileRect)) {
```

```
                CGRect intersection = CGRectIntersection(playerRect,
tileRect);

                if (tileIndex == 7) {
                    //tile is directly below Player
                    player.desiredPosition = CGPointMake(player.
desiredPosition.x, player.desiredPosition.y + intersection.size.
height);
                    player.velocity = CGPointMake(player.velocity.x,
0.0);
                    player.onGround = YES;
                } else if (tileIndex == 1) {
                    //tile is directly above Player
                    player.desiredPosition = CGPointMake(player.
desiredPosition.x, player.desiredPosition.y - intersection.size.
height);
                } else if (tileIndex == 3) {
                    //tile is left of Player
                    player.desiredPosition = CGPointMake(player.
desiredPosition.x + intersection.size.width, player.
desiredPosition.y);
                } else if (tileIndex == 5) {
                    //tile is right of Player
                    player.desiredPosition = CGPointMake(player.
desiredPosition.x - intersection.size.width, player.
desiredPosition.y);
                    //3
                } else {
                    if (intersection.size.width > intersection.size.
height) {
                        //tile is diagonal, but resolving collision
vertically

                        player.velocity = CGPointMake(player.
velocity.x, 0.0);
                        float intersectionHeight;
                        if (tileIndex > 4) {
                            intersectionHeight = intersection.size.
height;
                            player.onGround = YES;
                        } else {
                            intersectionHeight = -intersection.size.
height;
                        }
```

```
                        player.desiredPosition = CGPointMake(player.
desiredPosition.x, player.desiredPosition.y + intersection.size.height
);
                    } else {
                        //tile is diagonal, but resolving horizontally
                        float intersectionWidth;
                        if (tileIndex == 6 || tileIndex == 0) {
                            intersectionWidth = intersection.size.
width;
                        } else {
                            intersectionWidth = -intersection.size.
width;
                        }

                        player.desiredPosition = CGPointMake(player.
desiredPosition.x  + intersectionWidth, player.desiredPosition.y);
                    }
                }
            }
        }
    }

    player.position = player.desiredPosition;
}
```

What did we just do?

We used the `CGRectIntersectsRect` method to see if the player and the tile rectangles collide. We then used our `tileIndex` to determine the exact position of that tile and checked to see if it's a vertical or a horizontal collision. We also created a variable to determine the distance required to move our player so he no longer collides with the tile. Then, we checked to see if our player needs to be moved up or down. When that is determined, we either add or subtract the collision height from our player.

We also set up Booleans (true or false statements) that will detect whether the player is colliding with the ground; if he is, make him stop, and set the `onGround` Boolean to `true`.

Lastly, we set the position of our player to finally resolve the collision.

Now, in our `Player.h` file, we need to add the `onGround` Boolean property. With all our other properties, add in the following line of code:

```
@property (nonatomic, assign) BOOL onGround;
```

Now that we have our little man working properly, we are going to get to programming his movement! Let's get him dancing! (I won't actually be programming him to dance; however, if that is something you want to do, absolutely be my guest. I won't judge.)

Making our player dance!

For this game, we are going to make the controls super easy. Touch the right side of the screen, and the player will move forward; touch the left side, and he will jump. You can use the same methods to have him moving forwards and backwards, but this is how we will do it for this example.

In our `Player.h` file, add the following properties:

```
@property (nonatomic, assign) BOOL walking;
@property (nonatomic, assign) BOOL jumping;
```

Popping over to our `GameLevelScene.m` file, we are going to add the following methods:

```
- (void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event
{
    for (UITouch *touch in touches) {
        CGPoint touchLocation = [touch locationInNode:self];
        if (touchLocation.x > self.size.width / 2.0) {
            self.player1.jumping = YES;
        } else {
            self.player1.walking = YES;
        }
    }
}

- (void)touchesMoved:(NSSet *)touches withEvent:(UIEvent *)event {
    for (UITouch *touch in touches) {

        float halfWidth = self.size.width / 2.0;
        CGPoint touchLocation = [touch locationInNode:self];

        //get previous touch and convert it to node space
        CGPoint previousTouchLocation = [touch
previousLocationInNode:self];

        if (touchLocation.x > halfWidth && previousTouchLocation.x <=
halfWidth) {
            self.player1.walking = NO;
            self.player1.jumping = YES;
```

```
        } else if (previousTouchLocation.x > halfWidth &&
touchLocation.x <= halfWidth) {
            self.player1.walking = YES;
            self.player1.jumping = NO;
        }
    }
}

- (void)touchesEnded:(NSSet *)touches withEvent:(UIEvent *)event {

    for (UITouch *touch in touches) {
        CGPoint touchLocation = [touch locationInNode:self];
        if (touchLocation.x < self.size.width / 2.0) {
            self.player1.walking = NO;
        } else {
            self.player1.jumping = NO;
        }
    }
}
```

This is a pretty simple set of methods. We set up the two touch areas on the device's screen, each half of the width of the screen. Once one is touched, it fires the respective Boolean for either walking or jumping, which we will then detect if the Booleans have been fired in our `Player` class. Easy? Yup!

We need to add a small line of code up in our `(id)initWithSize` block of code so that we can enable touch controls in our app. Add this line anywhere in that method:

```
self.userInteractionEnabled = YES;
```

This line just gives us the ability to detect user interaction. If we were to run our app right now and touch the sides of the screen, our player would do absolutely nothing. Because he's a rebel? Not really. We have only set up the touches; we haven't told him what to do when he receives those touches.

Let's jump on over to our `Player.m` class and edit the `update` method so we can get him moving. The update method should look like this, with the highlighted code being the new lines we add:

```
- (void)update:(NSTimeInterval)delta {

    CGPoint gravity = CGPointMake(0.0, -450.0);

    CGPoint gravityStep = CGPointMultiplyScalar(gravity, delta);
```

```
    CGPoint movingForward = CGPointMake(750.0, 0.0);
    CGPoint movingForwardStep = CGPointMultiplyScalar(walking, delta);

     self.velocity = CGPointAdd(self.velocity, gravityStep);

    self.velocity = CGPointMake(self.velocity.x *0.9, self.velocity.y);

   //here he shall fly!


   if (self.walking) {

   self.velocity = CGPointAdd(self.velocity, movingForwardStep);
}

   CGPoint minimumMovement = CGPointMake(0.0, -450);
   CGPoint maximumMovement = CGPointMake(120.0, 250.0);
   self.velocity = CGPointMake(Clamp(self.velocity.x,
minimumMovement.x, maximumMovement.x), Clamp(self.velocity.y,
minimumMovement.y, maximumMovement.y));


    CGPoint velocityStep = CGPointMultiplyScalar(self.velocity,
delta);

    self.desiredPosition = CGPointAdd(self.position, velocityStep);
}
```

Seem easy enough? No? Ok, let's explain things a little more. First we added a moving forward "force" that will be added when the user touches the screen, which is being added at 750 points per second in relation to the delta frame step for smooth motion. Mmmm, smoooooth!

Next we controlled the moving forward force to imitate the friction of the ground so that, when the player stops moving, he glides briefly instead of coming to an immediate halt.

Next we checked if the screen is being touched, and if so we add velocity!

Then came the clamping. Clamping? Think of when you clamp a piece of wood to a work bench, that thing isn't moving. It's the same with the clamp methods; we are "clamping" or limiting the player's maximum and minimum horizontal and vertical speeds. The player will not move outside those limits.

We shall now add in the jumping method. Back in our `update` block of code, just above the `if (self.walking)` statement, we are going to add the following code to make our player jump:

```
CGPoint jumpForce = CGPointMake(0.0, 310.0);
float jumpTime = 150.0;

if (self.jumping && self.onGround) {
  self.velocity = CGPointAdd(self.velocity, jumpForce);
} else if (!self.jumping && self.velocity.y > jumpTime) {
  self.velocity = CGPointMake(self.velocity.x, jumpTime);
}
```

This jumping system is similar to that of *Super Mario* where, if you press and hold the jump button, he will accelerate to a certain point, at which juncture the `jumpTime` function then stops him from further accelerating. However, if the player stops pressing the jump button before the jump reaches the `jumpTime` function cut off, the jump will be reduced.

See that? You're on your way to creating the next *Super Mario*! If you run your project and click on the sides of the screen, he will move around and jump! But you know what, I don't like the fact that our player can only move forward. What if he gets stuck? (Like in my level, he can get stuck almost immediately. Oops!) So let's go back to our `Player.h` class and add another property:

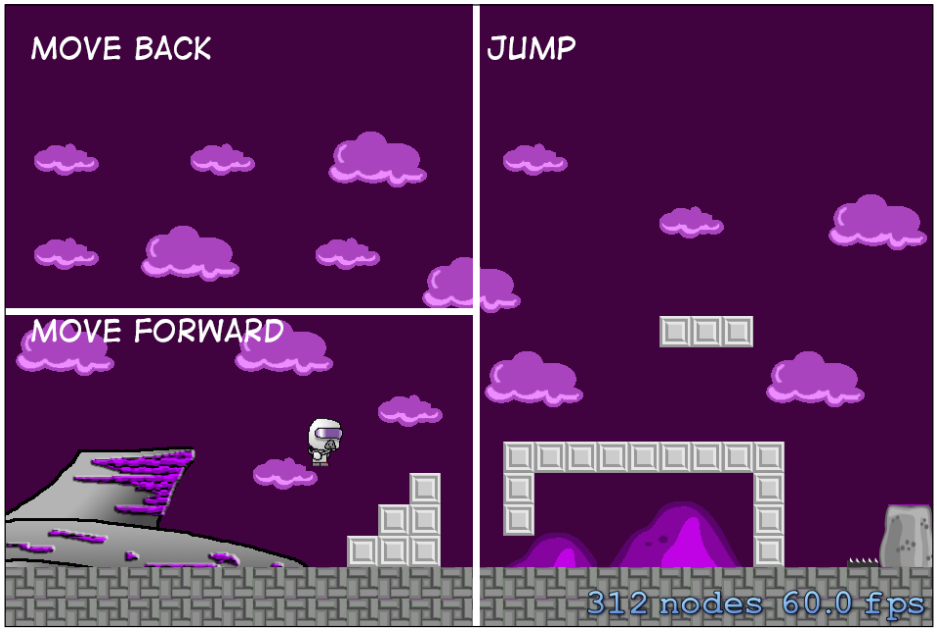
```
@property (nonatomic, assign) BOOL goingBackwards;
```

Then on over to our `GameLevelScene.m` file, where we make a slight change to our `touchesBegan` function and the `TouchesMoved` block of code that we added earlier:

```
- (void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event
{
    for (UITouch *touch in touches) {
        CGPoint touchLocation = [touch locationInNode:self];
        if (touchLocation.x > self.size.width / 2.0) {
            self.player1.jumping = YES;
        }
        else {
            if (touchLocation.x < self.size.width / 2.0) {
                if (touchLocation.y > self.size.height / 2){
                    self.player1.goingBackwards = YES;
        self.player1.xScale = -1.0;
                }
                if (touchLocation.y < self.size.height / 2){
                    self.player1.walking = YES;
            self.player1.xScale = 1.0;
                }
            }

        }
    }
}
- (void)touchesMoved:(NSSet *)touches withEvent:(UIEvent *)event {
    for (UITouch *touch in touches) {

        float halfWidth = self.size.width / 2.0;
        CGPoint touchLocation = [touch locationInNode:self];

        //get previous touch and convert it to node space
        CGPoint previousTouchLocation = [touch
previousLocationInNode:self];

        if (touchLocation.x > halfWidth && previousTouchLocation.x <=
halfWidth) {
```

```
            self.player1.walking = NO;
            self.player1.goingBackwards = NO;
            self.player1.jumping = YES;
        } else if (previousTouchLocation.x > halfWidth &&
touchLocation.x <= halfWidth) {
            //self.player1.walking = YES;
            self.player1.goingBackwards = NO;
            self.player1.jumping = NO;
        }
        else if (previousTouchLocation.x > halfWidth &&
touchLocation.x <= halfWidth) {
            if (touchLocation.y > self.size.height / 2){
            self.player1.walking = NO;
            //self.player1.goingBackwards = YES;
            self.player1.jumping = NO;
            }
        }
    }
}
- (void)touchesEnded:(NSSet *)touches withEvent:(UIEvent *)event {

    for (UITouch *touch in touches) {
        CGPoint touchLocation = [touch locationInNode:self];
        if (touchLocation.x < self.size.width / 2.0) {
            //self.player1.walking = NO;
            if (touchLocation.x < self.size.width / 2.0) {
                if (touchLocation.y > self.size.height / 2){
                    self.player1.goingBackwards = NO;
                }
                if (touchLocation.y < self.size.height / 2){
                    self.player1.walking = NO;
                }
            }

        } else {
            self.player1.jumping = NO;
        }
    }
}
```

So now, instead of having the screen halved with one walking button and one jumping button, we have the screen split into three, with the walking button as one half of the screen, and that half has been split height-wise, as shown in the following screenshot:



Let's pop over to our `Player.m` file and make some very small adjustments to our `update` method. We are going to add in the following lines:

```
    CGPoint movingBackward = CGPointMake(-750.0, 0.0);
    CGPoint movingBackwardStep = CGPointMultiplyScalar(movingBackward,
delta);
```

These lines are a reverse of the walking forward method we created, hence the -750 value. Now under our `if (self.walking)` method, we are going to add the following `if` statement:

```
    if (self.goingBackwards) {
        self.velocity = CGPointAdd(self.velocity, movingBackwardStep);
    }
  //The below value has to be changed to allow a negative x value to
walk backwards.
    CGPoint minimumMovement = CGPointMake(-750.0, -450);
```

This again is just a reverse of the walking forward movement. Test it out to see if it's working; if so, he should be backtracking on his steps like a scared cat!

Now we are going to get the screen scrolling as our player moves towards the edge of the screen. On to our `GameLevelScene.m` class, to which we add the following code:

```
//Add this in the import section
#import "SKTUtils.h"

//Then add this wherever you like, after any of the methods
- (void)setViewpointCenter:(CGPoint)position {
  NSInteger x = MAX(position.x, self.size.width / 2);
  NSInteger y = MAX(position.y, self.size.height / 2);
  x = MIN(x, (self.map.mapSize.width * self.map.tileSize.width) -
self.size.width / 2);
  y = MIN(y, (self.map.mapSize.height * self.map.tileSize.height) -
self.size.height / 2);
  CGPoint actualPosition = CGPointMake(x, y);
  CGPoint centerOfView = CGPointMake(self.size.width/2, self.size.
height/2);
  CGPoint viewPoint = CGPointSubtract(centerOfView, actualPosition);
  self.map.position = viewPoint;
}

//Then add this in the Update method
[self setViewpointCenter:self.player1.position];
```

This block constrains the position of the screen to the player when he reaches the center of the view.

Test it to see if it works!

This is starting to look so good! But if you've noticed, he can't die. Sadly, we do want him to die when he hits spikes or falls down those scary pits, don't we?

Let's integrate the hazards layer in our TMX map. To do this, we have to go to our `GameLevelScene.m` file and add in the following detection method:

```
//Add this at the top of the code
@property (nonatomic, strong) TMXLayer *hazards;

//Add this in the initWithSize method after we set up the Walls
self.hazards = [self.map layerNamed:@"hazards"];

//add this in the checkForAndResolveCollisionsForPlayer
[self handleHazardCollisions:self.player1];

//Add this anywhere!

- (void)handleHazardCollisions:(Player *)player
{
    NSInteger indices[8] = {7, 1, 3, 5, 0, 2, 6, 8};

    for (NSUInteger i = 0; i < 8; i++) {
        NSInteger tileIndex = indices[i];

        CGRect playerRect = [player collisionBox];
        CGPoint playerCoord = [self.hazards coordForPoint:player.
desiredPosition];

        NSInteger tileColumn = tileIndex % 3;
        NSInteger tileRow = tileIndex / 3;
        CGPoint tileCoord = CGPointMake(playerCoord.x + (tileColumn -
1), playerCoord.y + (tileRow - 1));

        NSInteger gid = [self tileGIDAtTileCoord:tileCoord
forLayer:self.hazards];
        if (gid != 0) {
            CGRect tileRect = [self tileRectFromTileCoords:tileCoord];
            if (CGRectIntersectsRect(playerRect, tileRect)) {
                [self gameOver:0];
            }
        }
    }
}
```

This is essentially the same code that we used for our `checkForAndResolveCollisionsForPlayer` function. We only added a `gameOver` method whereby, when it equals `0`, the player dies, and when it's `1`, the player beats the level.

You will have errors showing at the moment. We haven't incorporated our game over feature yet, so let's do that now. Again in our `GameLevelScene.m` file, let's add the following code:

```
//add this at the top with all the other properties
@property (nonatomic, assign) BOOL gameOver;

//Put this in the update method
if (self.gameOver) return;

//Then add this method anywhere in the GameLevelScene
-(void)gameOver:(BOOL)won {

    self.gameOver = YES;

    NSString *gameText;
    if (won) {
        gameText = @"Level Complete!";
    } else {
        gameText = @"You have failed!";
    }


    SKLabelNode *endGameLabel = [SKLabelNode labelNodeWithFontNamed:@"
AvenirNext-Heavy"];
    endGameLabel.text = gameText;
    endGameLabel.fontSize = 40;
    endGameLabel.position = CGPointMake(self.size.width / 2.0, self.
size.height / 1.7);
    [self addChild:endGameLabel];


    UIButton *replay = [UIButton buttonWithType:UIButtonTypeCustom];
    replay.tag = 321;
    UIImage *replayImage = [UIImage imageNamed:@"replay"];
    [replay setImage:replayImage forState:UIControlStateNormal];
    [replay addTarget:self action:@selector(replay:) forControlEvents:
UIControlEventTouchUpInside];
```

```
    replay.frame = CGRectMake(self.size.width / 2.0 - replayImage.
size.width / 2.0, self.size.height / 2.0 - replayImage.size.height /
2.0, replayImage.size.width, replayImage.size.height);
    [self.view addSubview:replay];
}

//Add this into the checkForAndResolveCollisionsForPlayer, after
CGPoint playerCoord = [layer coordForPoint:player1.desiredPosition];
if (playerCoord.y >= self.map.mapSize.height - 1) {
  [self gameOver:0];
  return;
}
```

Let's break it down as we usually do. Firstly, we set up the new game over the Boolean, which we use whenever the player collides with a hazard, or later an enemy.

After that, we set a level win and a level failed string (or text) to pop up whenever the player either beats the level or loses. I used *Avenir Next Heavy* as a font. There are a ton of fonts that you can use for your game. For an awesome site showing all the fonts you can use, check out `iosfonts.com`.

We then created a `UIButton` the user will be able to tap to restart the level. (Don't forget to add in the `replay.png` and `replay@2x.png` image files that I supplied in the resources section of this chapter.)

Finally we added in the method that checks the player's position; if he is below the map—in other words, if he's fallen down a hole or crack in the map—we call game over.

But our poor little guy isn't going to experience just death! He needs to win every once and a while, right? Well we need to add those methods as well! Don't worry, it's super easy!

In our `GameLevelScene.m` file, we need to add a new method:

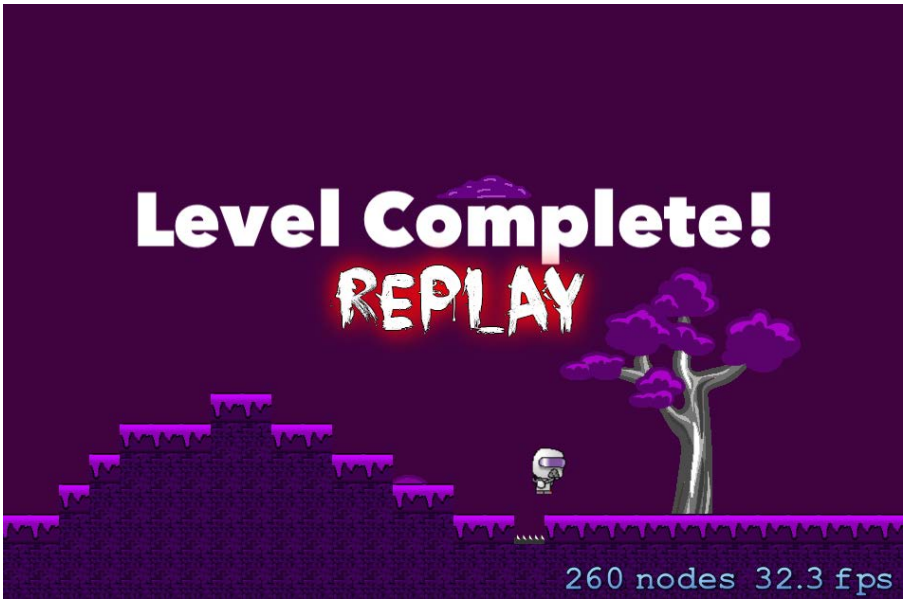
```
-(void)didHeWin {
  if (self.player1.position.x > 3200.0) {
    [self gameOver:1];
  }
}
```

This is a positionally based win, so once our player reaches the `x` value of `3200` on the map we will call a win. We can have a win layer on the TMX map and incorporate that layer, but this just seems easier! Finally, in the `handleHazardCollisions` section, we need to check if our little guy has won:

```
[self didHeWin];
```

We now test this to see if it works and you will see the following screen:

When we fail a level, we land on spikes, ouch! But we reach the end of the level, as shown in the following screenshot:



Summary

Oh my goodness! It's starting to look great, isn't it? We've done a lot of work in this chapter! What with our level design, figuring a way around implementing our maps into our project, creating our little player, and making him move, collide, and jump around, we certainly have done a lot.

For now, let's go take a break! In the next chapter, we are going to do some more creation, such as adding awesome music and sound effects, to polish the game up a bit, and maybe add some menus, particle effects, and even some enemies!

Go get yourself a nice strong <insert beverage of choice here>, and I'll see you in the next chapter!

4
Let's Keep Going! Adding More Functionality

We got a lot accomplished in the last chapter! We figured out level creation, importing our levels into Xcode and getting them and our player all showing up in our game. Are you well rested and ready to tackle even more programming awesomeness? Let's see what we will do in this chapter:

- Adding awesome sound effects
- Character animations
- Playing with particles
- Menus
- Adding some enemies

Our game looks pretty awesome, but there's still tons of work to be completed before we can even remotely consider releasing it!

Let's get into it, shall we?

First things first, we ended the last chapter we had just discussed winning the level, and dying during the level, however if you tested it and you died or beat the level and clicked on the replay button, did the game crash? That's because we need to add one final method:

```
- (void)replay:(id)sender
{
    [[self.view viewWithTag:321] removeFromSuperview];
    [self.view presentScene:[[GameLevelScene alloc] initWithSize:self.
size]];

}
```

This block of code simply removes the button from the screen and resets the game. Let's proceed!

Adding awesome sound effects

Yes! We will give our little guy some sound, specifically, when jumping and dying. That's not all though! No no! We will also get some groovy tunes playing in our levels.

Let's open up our `GameLevelScene.m` file and import the SpriteKit Audio framework to play sounds! At the top of the file where all our import methods are, add the following line:

```
#import "SKTAudio.h"
```

I've included some audio for our use as well, so if you haven't imported them into your project, go ahead and do that now, or you can use your own music if you like. Once your file has been imported into the project, back in our `GameLevelScene.m` file, within our `-(id)initWithSize` method, we will add the following line of code to get our music playing:

```
[[SKTAudio sharedInstance] playBackgroundMusic:@"BackgroundAudio.
mp3"]; //change the file name to whatever file you imported
```

Test the project and now you should have some rocking tunes playing in the background! So cool!

I think we should now create a sound effect for the jump movement, shouldn't we?

Hopping (no pun intended) on to our `Player.m` file, we will locate the block of code where we make our player dance. That method is found in the `update` method, and in that method, we will locate the '`if (self.jumping && self.onGround)` statement. In the `if` statement, we will add the following code just after the open braces of the `if` statement (That is, {):

```
[self runAction:[SKAction playSoundFileNamed:@"jump.wav"
waitForCompletion:NO]];
```

Let's break this function down a little bit. We tell **self** to run an action, which self in this case is the player class inherited from SKNode, which is where that `runAction` method comes from. Then, we declare that action as a SpriteKit action, which is used to play a sound file. We declare the sound file, then we tell SpriteKit not to wait for completion.

Awesome! Now the character will emit a little blip noise every time the player taps the jump button. You can apply these methods to anywhere you want to play a sound, be it dying, shooting, walking—you name it!

Notice how we used a different method to play the background music then we did to play the jumping sound effect? Do you know why we did this?

When you play the game, there will be a bunch of sound effects playing at one a time. If we play the sound effects using the same method that we use to play our music, it would potentially stop playing the music in order to play a new sound effect that has been called, as we can only play one background music file at a time. So when we play a sound effect, we will not interfere with our background music as it plays on a different channel.

This way, the music will be played without any interruption, unless you get a call while playing it, as there's nothing we can do about that.

Our game is now coming together, but I don't like how our player just stays still when he's walking or jumping. Let's add in some animations!

Character animations

Almost all games contain character or object animations. They add life to the objects or characters, and they make the game look way more appealing. Take the game we are making as an example. Say we pushed the game for sale, and the player just remained in his idle pose all the time, dying, walking, or jumping—there he was just standing as he is.

Doesn't look proper, does it? We need to change that!

We've already imported the Sprite atlas file that contains all the images required to create our character animations, so that saves a lot of the work. The Sprite atlas is where all the sprites will go for SpriteKit to access. Instead of having images randomly imported all over the place in our project, the Sprite atlas organizes them all nicely.

We will go into our `Player.m` class, and just below our `@implementation Player` line, add the following line of code so that our implementation looks like this:

```
@implementation Player
{
  NSArray *walkingAnimation
}
```

We need to add an array of images that will make up our walking animations or any other animations that you would like to create.

Now, in our `initWithImageNamed` method, we need to create the array, locate the walking images within the atlas, and then add them to our array. To do this, add the following code directly under the opening bracket for our `init` method:

```
    NSMutableArray *walkingFrames = [NSMutableArray array];
    SKTextureAtlas *playerAtlas = [SKTextureAtlas
atlasNamed:@"sprites"];

    int numberOfImages = 8;
    for (int i=1; i <= numberOfImages/2; i++) {
        NSString *imageName = [NSString stringWithFormat:@"P1Walking
%d", i];
        SKTexture *temporaryTexture = [playerAtlas textureNamed:
imageName];
        [walkingFrames addObject:temporaryTexture];
    }
    walkingAnimation = walkingFrames;
```

Let's discuss what just happened here because it's a lot of confusing jargon!

In the first line, we added sets to an array to hold all the walking images within the atlas. Next, we load the texture atlas that contains all our images. What's great is that the SpriteKit automatically loads the correct resolution for the device we are using, hence the @2x images to make up for the higher resolution images used for retina displays.

Next, we tell Xcode to search through the atlas for images named `P1Walking`; the `%d` automatically searches through all the images named `P1Walking`. So, this will start at image `0`, then `1`, `2`, `3`, and so forth. Finally, we add the images into our walking animation array.

We now need to add an action that will fire the animation and stop it when called. We need to add the following methods into our `Player.m` file:

```
-(void)playWalkingAnim {
[self runAction:[SKAction repeatActionForever:[SKAction animateWith
Textures:walkingAnimation
  timePerFrame:0.1f
  resize:NO
  restore:YES]]
  withKey:@"PlayerWalking"];

return;
}

-(void)PlayerStoppedMoving {
 [self removeAllActions];
}
```

We add an action key, `PlayerWalking` to be able to stop the animation when needed.

We call this method once when the player is touching the walking zones we created in the previous chapter. Let's go over to our `Player.h` file and create another property. With all our other properties, add the following:

```
@property (nonatomic, assign) BOOL animateWalking;
```

Now, to get our little guy actually animated, we need to jump on to `GameLevelScene.m` file. In our `touchesBegan` method, where we call the `walking` and `goingBackwards` Booleans, we need to add the following code:

```
self.player1.animateWalking = YES;
```

So, now our `touchesBegan` method will look like this (with the new code highlighted):

```
- (void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event
{
    for (UITouch *touch in touches) {
        CGPoint touchLocation = [touch locationInNode:self];
        if (touchLocation.x > self.size.width / 2.0) {
            self.player1.jumping = YES;
        }
        else {
            if (touchLocation.x < self.size.width / 2.0) {
//This will check our touch position on the left hand side of the
screen
                if (touchLocation.y > self.size.height / 2){
//This checks if we are in the top left hand corner of the screen, if
so trigger walking backward
                    self.player1.goingBackwards = YES;
                    self.player1.animateWalking = YES;
                    self.player1.xScale = -1.0;
                }
                if (touchLocation.y < self.size.height / 2){
//This checks if we are touching the bottom left corner. If we are
trigger walking forward.
                    self.player1.walking = YES;
                    self.player1.animateWalking = YES;
                    self.player1.xScale = 1.0;
                }
            }

        }
    }
}
```

This will now set our Boolean `true` whenever the player touches these walking zones. We also need to set them `false` when the touches end, so we will scroll down to our `touchesEnded` method and add the following where we turn our walking and `goingBackwards` method to `no`:

```
self.player1.animateWalking = NO;
```

Now let's go back to our `Player.m` file, and we are going to add the following lines of code to our `update` section:

```
if (self.animateWalking) {
        [self actionForKey:@"PlayerWalking"];
    }
    else {
        [self PlayerStoppedMoving];

    }
    if (![self actionForKey:@"PlayerWalking"]) {

        [self playWalkingAnim];
    }
```

Now when the player touches a walking zone, it will set the `animateWalking` Boolean to `true`, which will call the `PlayerWalking` action. When that action is called, we will then animate the character with a walking animation, and when the player removes their fingers from the walking zone, it will stop animating.

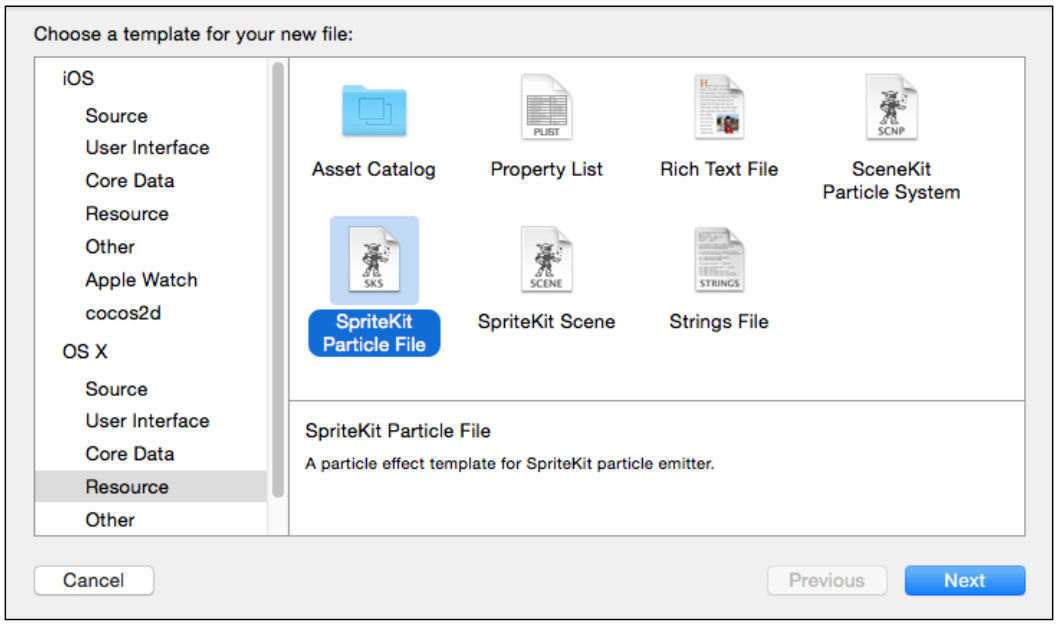
Whew! That's a lot of code for something so simple, but the results look great!

Bit by bit, our little game is coming together! Let's make this level look a little dark and dreary. Let's play with some particles to add depth!

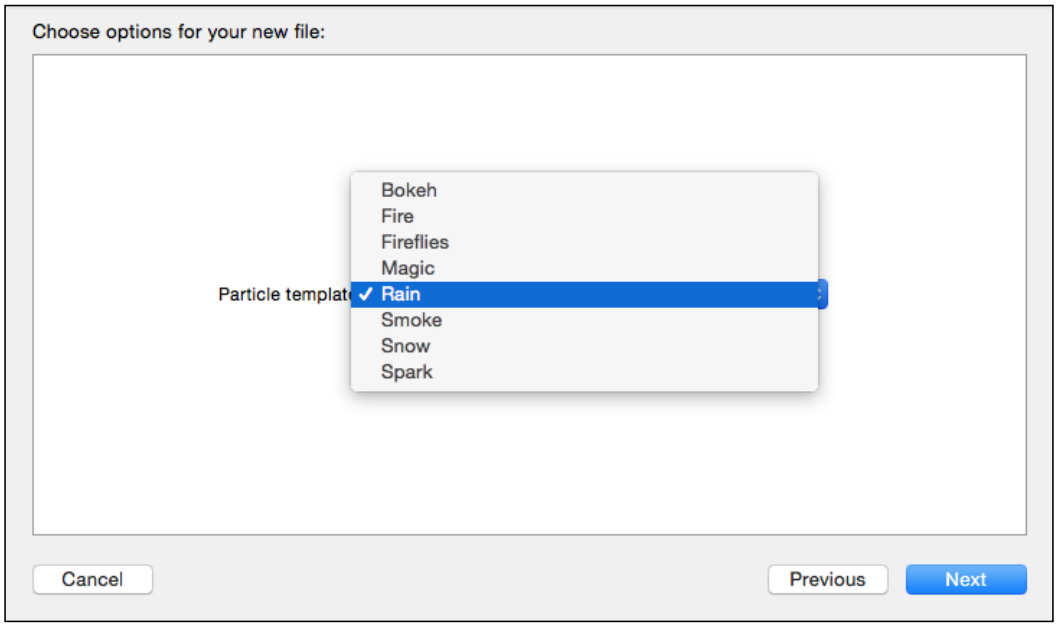
Playing with particles

SpriteKit makes particles super easy, so easy that, in fact, a lot of common particle effects, such as flames, smoke, and rain are premade templates available when creating our particle file.

In Xcode, to create a new particle emitter, simply navigate to **File** | **New** | **File**. Under the template creator in the **iOS** section (ignore the fact that I have OS X selected in the previous screenshot) and under **Resource**, select **SpriteKit Particle File**, as shown in the following screenshot:

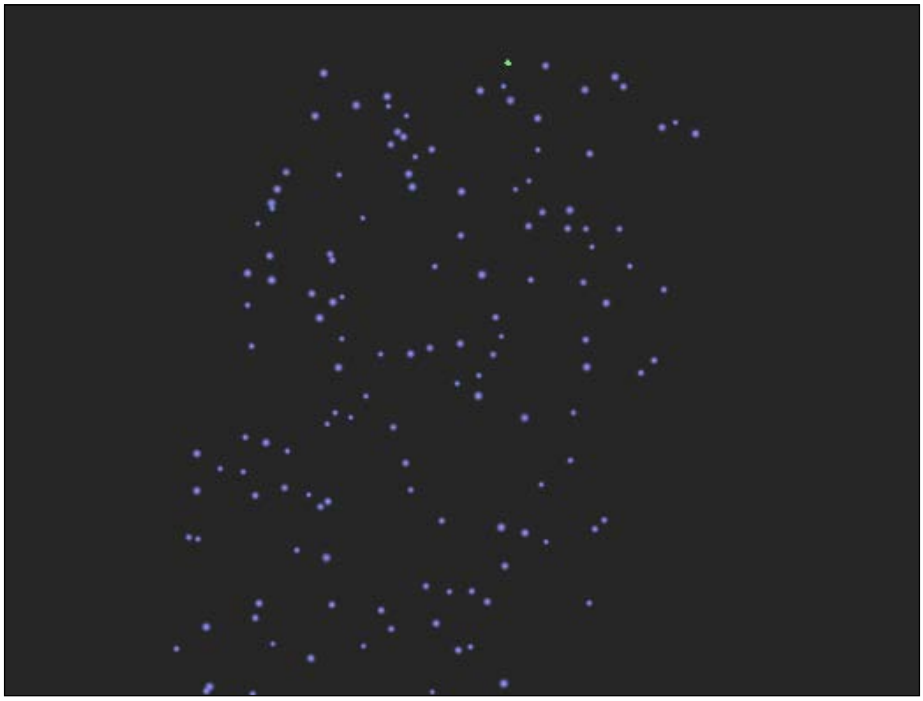


This is where creating the effects are super easy, simply select **Rain** from the particle template.



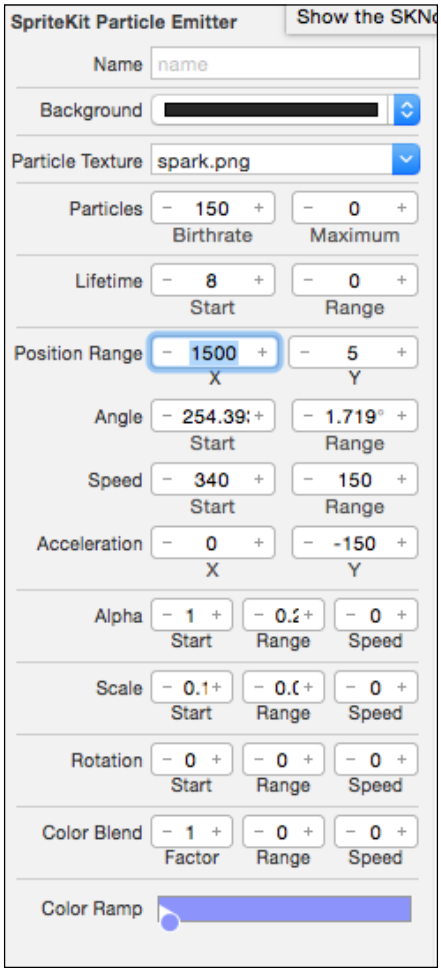
Yes, it's that easy to create particles!

Now, all that we have to do is save it, which I just did in the `ADESA` project folder. Now, you will see our new particle emitter file in the project explorer in Xcode. Click on it and you should see something similar to the following screenshot:



Don't worry if you don't exactly like the look of the particle effect, we can do a whole bunch of adjustments to it.

At the top of the right-hand side bar, click on the **Show SK Node inspector** button to see all the adjustments we can make. The only thing I changed here is the position range, I changed the value of **X** to **1500** so that it will stretch across the whole screen.



If you want to make other adjustments, let me tell you what all these options do.

Background

The background option is provided so you can test the visibility of the particles against various colored backgrounds. Changing this option will have no effect on the particles.

Particle texture

The particle texture is the image file that the emitter will use for the particles. The standard provided image is simply a soft white sphere, so you get a nice smooth and soft particle. This image is great because it can be used for pretty much all types of particles.

Particle birthrate

The particle birthrate is the rate at which new particles are emitted by the emitter. The higher the value, the faster the new particles are spat out. (Be warned, less particles give better performance). The total number of particles to be emitted may also be specified. If you set the value to **0**, this will cause particles to be emitted indefinitely. If you specify a maximum value, the emitter will stop when particles in the scene reach that value.

Particle life cycle

The particle life cycle controls the length of time in seconds for which a particle remains alive. The range property can be used to vary the duration of the life of particles, for example, if you create an explosion, you can use a larger range so you have some particles visible for longer than the others.

Particle position range

The particle position range option defines the location from which particles are created (self-explanatory, right?). The **X** and **Y** values can be used to declare an area around the center of the node location from which particles will be created randomly.

Angle

The angle option relates to the angle at which a newly emitted particle will travel away from the creation point in counterclockwise degrees, where a value of **0** degrees equates to rightward movement. When we set a range value, it will vary the direction in which the particles are emitted.

Particle speed

Again, the particle speed option is pretty self-explanatory. It deals with the speed at which particles move when they are created. When we set a range value, it will vary the direction in which the particles are emitted.

Particle acceleration

The acceleration properties control the speed at which a particle accelerates or decelerates after emission. I'll use explosions for an example again, with this option, you can have them fly out fast but have the shrapnels slow down.

Particle scale

The particle scale option obviously refers to the size of the particles, which again can be varied with the range setting.

Particle rotation

The particle rotation controls the speed at which a particle rotates. Again, you can have a shrapnel rotating as it flies from the explosion.

Particle color

The particles created by an emitter can change colors during their life. To add a new color in the life cycle timeline, click on the color ramp at the location where the color is to change and select a new color (think of creating a gradient in Photoshop or Illustrator). You can also change an existing color by double-clicking on the marker to display the color selection.

To remove a color from the gradient, click and drag it downward.

The color blend option controls how many colors in the particles texture image, and how they blend with the main color in the color gradient.

The greater the **Factor** option, the more the colors blend, where **0** causes no blending.

Particle blend mode

The blend mode option controls the way in which the particle image blends with the scene. The available options are as follows:

- **Alpha**: This blends transparent backgrounds in the particle image.
- **Add**: This adds the particle pixels to the corresponding background image pixels.
- **Subtract**: This subtracts the particle pixels from the corresponding background image pixels.
- **Multiply**: This multiplies the particle pixels by the corresponding background image pixels. This results in a darker particle effect.

- **MultiplyX2**: This creates a darker particle effect than the standard multiply mode.
- **Screen**: This inverts pixels, then multiplies and inverts them a second time. This results in lighter particle effects (great for flame effects or sparks).
- **Replace**: This results in no blending with the background. Only the particle's colors are used.

To implement our particles into the scene, let's go on over to our `GameLevelScene.m` file, and in the `initWithSize` method inside the `if (self = [super initWithSize:size])` parentheses, add the following block of code:

```
NSString *rainParticles =
        [[NSBundle mainBundle] pathForResource:@"Rain" ofType:@"sks"];

        SKEmitterNode *rainEmitter =
        [NSKeyedUnarchiver unarchiveObjectWithFile:rainParticles];

        rainEmitter.position = CGPointMake(0, self.scene.size.height);

        [self addChild:rainEmitter];
```

Build and run to see the great results!

It looks great, but now that we have it in the scene, I don't quite like the look of the rain; I want to tweak it so that it looks a little more realistic. Let's go to the particle editor (again, by clicking on the particle file in our project explorer) and increase the birthrate of the particles.

Currently, the birthrate is at 150, but I want it to be pouring, so I will increase it to 2500. I will also change the scale down from 0.1 to 0.02 because I think the raindrops look way too big. I'll also change the speed to 500 so that it looks a little more torrential.

That does look a lot better! It's tough to tell on paper but the effect looks great on screen.

Now that we have the rain looking awesome, let's add in some flames and smoke effects to our wrecked ship. Create a new particle file, and instead of selecting rain in the template creator, select fire, then save it.

Our fire effect will be implemented differently as we want it to scroll with the map. With the rain, we added it to the current screen so that it doesn't move with the scene but with the fire. If we were to add it to the scene the same way we did the rain, the particles would stay in the same position according to the screen, not the whole map.

Back in our `GameLevelScene.m` file, in the same location where we added our rain emitter code, add in the following method:

```
        NSString *fireParticles =
        [[NSBundle mainBundle] pathForResource:@"Flames"
ofType:@"sks"];

        SKEmitterNode *fireEmitter =
        [NSKeyedUnarchiver unarchiveObjectWithFile:fireParticles];
        fireEmitter.position = CGPointMake(25, 50);

        [_map addChild:fireEmitter];
```

See the difference in code? Instead of adding the `fireEmitter` as a child to `self` or `GameLevelScene` class, we add it directly to the map at `25x` and `50y`, so now when the player begins to scroll through the scene, the fire stays put, burning away the poor crashed ship.

Ignore the extreme drop in frame rate in the screenshot here; whenever I take a screenshot, the frame rate plummets, as I am running a slightly older iMac. The fire looks good though!

I did, however, notice that the frame rate stuck at an almost constant 30fps with the flames on the screen. This is where the performance and battery drainage comes into play.

I'm running the iPhone 4S simulator in Xcode because it fits properly on my screen, which means people with older devices will have difficulty running this game, especially when graphic-intensive elements come onto screen like the particle effects.

However, running it on my iPhone 5S, the game hit an average of 60fps. These are all the things we have to take into consideration before we publish our game. We will discuss this later on in this book.

Now, let's create a menu system!

Creating menus and multiple levels

Our game is looking really good, but instead of simply throwing the player into the gameplay, let's talk about menus.

Menus are pretty important (obviously). It's the stepping stone into the gameplay, and its absence can leave the players dazed and confused. We don't want that now, do we?

As we've been going crazy with our level development, we haven't focused on the structure of the game; namely, we've done everything without taking menus or any other functionality into consideration.

First off, I've included some more images for your use that need to be imported into the project for the following code to work. If you have your own images, simply change the name of the button image accordingly.

Now, we have to make some substantial modifications to our `GameLevelScene.m` file. First things first, we need to create a new integer variable. This will count our levels, with level `0` being our main menu. So, in the `@interface GameLevelScene ()` method where we begin our declarations, let's add in one more declaration:

```
@property (nonatomic, assign) NSInteger level;
```

Now, let's do some editing!

Let's start off with the `initWithSize` method. We will edit it so that it looks as follows:

```
-(id)initWithSize:(CGSize)size {
    if (self = [super initWithSize:size]) {
```

```
        self.userInteractionEnabled = YES;
        self.backgroundColor = [SKColor colorWithRed:.0 green:.0
blue:.0 alpha:1.0];


        if (_level == 0){
        SKLabelNode *playLabel = [SKLabelNode labelNodeWithFontNamed:@
"AvenirNext-Heavy"];
        playLabel.text = @"Adesa";
        playLabel.fontSize = 40;
        playLabel.position = CGPointMake(self.size.width / 2.0, self.
size.height / 1.7);
        [self addChild:playLabel];

        SKSpriteNode *playButton = [SKSpriteNode spriteNodeWithImageN
amed:@"play"];
        playButton.position = CGPointMake(self.size.width /2.0 , self.
size.height / 2.5);
        playButton.name = @"playButton";
        [self addChild:playButton];
        }

    }
    return self;
}
```

All the init code we had in there originally is now gone. We have placed a nice little label and a button. We also changed the background to black but that part isn't important. Notice how we name the `playButton`? This comes in handy when we detect player touches, which you'll see in a moment.

Now, let's scroll on down to our `touchesBegan` method and add the following code inside the `for` method:

```
SKNode *node = [self nodeAtPoint:touchLocation];

        if ([node.name isEqualToString:@"playButton"]) {
            _level = 1;
            [self removeAllChildren];

            [[SKTAudio sharedInstance] playBackgroundMusic:@"Backgrou
ndAudio.mp3"];
            self.map = [JSTileMap mapNamed:@"level1.tmx"];
            [self addChild:self.map];
            NSString *rainParticles =
```

```
            [[NSBundle mainBundle] pathForResource:@"Rain"
ofType:@"sks"];

            SKEmitterNode *rainEmitter =
            [NSKeyedUnarchiver unarchiveObjectWithFile:rainParticles];

            rainEmitter.position = CGPointMake(0, self.scene.size.
height);

            [self addChild:rainEmitter];

            NSString *fireParticles =
            [[NSBundle mainBundle] pathForResource:@"Flames"
ofType:@"sks"];

            SKEmitterNode *fireEmitter =
            [NSKeyedUnarchiver unarchiveObjectWithFile:fireParticles];
            fireEmitter.position = CGPointMake(25, 50);

            [_map addChild:fireEmitter];

            self.player1 = [[Player alloc]
initWithImageNamed:@"P1idle"];
            self.player1.position = CGPointMake(100, 50);
            self.player1.zPosition = 15;
            [self.map addChild:self.player1];
            self.walls = [self.map layerNamed:@"walls"];
            self.hazards = [self.map layerNamed:@"hazards"];
        }
```

Here, we add a new SpriteKit node that is created at the location of your touch. We then detect whether that node is touching our play button, and then we set up all our level 1 stuff.

I will also change how the game over is handled. As of this moment, when the player dies, the screen pops up saying **You have died**, followed by the big `replay` button. If we were to leave it as it is, that `replay` button method resets the entire game, and I don't think people want to lose their progress when they die in a level.

Make the following changes to the code. We will delete the highlighted code and replace it with this text:

```
-(void)gameOver:(BOOL)won {
    if (_level > 0){
    self.gameOver = YES;
```

```
    NSString *gameText;
    if (won) {
        gameText = @"Level Complete!";

    } else {
        gameText = @"You have died!";
        //add the following lines of code here:
         self.player1.position = CGPointMake(100, 50);
    self.player1.zPosition = 15;
    [self setViewpointCenter:self.player1.position];
    self.gameOver = NO;
    }
}
}
    UIButton *replay = [UIButton buttonWithType:UIButtonTypeCustom];
    replay.tag = 321;
    UIImage *replayImage = [UIImage imageNamed:@"replay"];
    [replay setImage:replayImage forState:UIControlStateNormal];
    [replay addTarget:self action:@selector(replay:) forControlEvents:
UIControlEventTouchUpInside];
    replay.frame = CGRectMake(self.size.width / 2.0 - replayImage.
size.width / 2.0, self.size.height / 2.0 - replayImage.size.height /
2.0, replayImage.size.width, replayImage.size.height);
    [self.view addSubview:replay];

}
}

- (void)replay:(id)sender
{
    [[self.view viewWithTag:321] removeFromSuperview];
  //  [self.view presentScene:[[GameLevelScene alloc]
initWithSize:self.size]];

}
```

Now, when you test it and the player dies, he simply teleports to the beginning of the level instead of having the `replay` button pop up, which could prove to be intrusive after a while.

With our menu built, when you build and run the project, the main menu should look like this:

We will now implement multiple levels, as I'm pretty sure players would get bored of playing the same level over and over again. I created a new level for you in the resources section of this book. For this level, our player will locate his lost equipment, but in order to find it, the player needs to work around a little trap in the level.

Once you've imported the `level2.tmx` file into your project, hop on to `GameLevelScene.m` file and scroll down to our `didHeWin` method and edit it to look as follows:

```
-(void)didHeWin {
    if (self.player1.position.x > 3200.0) {

        if (_level == 1) {
        [self.map removeFromParent];
        self.map = [JSTileMap mapNamed:@"level2.tmx"];
        [self addChild:self.map];

        }
        self.player1 = [[Player alloc] initWithImageNamed:@"P1idle"];
        self.player1.position = CGPointMake(100, 50);
        self.player1.zPosition = 15;
        [self.map addChild:self.player1];
        self.walls = [self.map layerNamed:@"walls"];
        self.hazards = [self.map layerNamed:@"hazards"];
    }
}
```

This new method still detects the player's position, but now we set up a secondary `if` statement that detects the level the player is on. In this case, if the level is equal to 1, we then remove the current map from the view, and we add the `level2.tmx` file to the view as a child. After this, we relocate the player to the beginning position in the level and check again for the map layers. Fairly easy, right?

Don't forget to reimport the `tileSet.png` and `tileSet@2x.png` files with the ones provided for this chapter, as they contain the new tiles. If you don't reimport these, Xcode will either throw an error or our new tiles just won't appear!

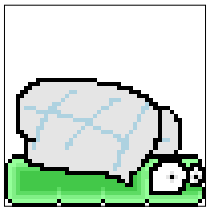
You can now build and run our project; now, when the player reaches the end of the level, it should switch over to the next level, which as you can see by the following image, looks pretty awesome!

Yes! Our game is coming together more and more! However, as awesome as our game looks, it's pretty darn boring without any enemies! Let's give our little spaceman some competition.

Creating enemies

This game isn't simply an exploration game! We need to make this game intense and exciting!

Go ahead and import the Squiggy set of images that I've included, or again you can use your own.



In our `GameLevelScene.m` file, we will add a new method to begin spawning random enemies. These ones will start off easy, but we will get into some baddies that will actually try to kill you!

Anyways, anywhere within the `GameLevelScene.m` file, add the following method:

```
- (void)addSquiggy {

    SKSpriteNode * squiggy = [SKSpriteNode spriteNodeWithImageNamed:@
"Squiggy"];

    int minY = squiggy.size.height / 2;
    int maxY = self.frame.size.height - squiggy.size.height / 2;
    int rangeY = maxY - minY;
    int actualY = (arc4random() % rangeY) + minY;
    squiggy.xScale = -1.0;
    squiggy.position = CGPointMake(self.player1.position.x + 1000 +
squiggy.size.width/2, actualY);

    squiggy.name = @"squiggy";

    [self.map addChild:squiggy];

    // Setting the Speed of SQUIGGY!
    int minDuration = 10.0;
    int maxDuration = 20.0;
    int rangeDuration = maxDuration - minDuration;
    int actualDuration = (arc4random() % rangeDuration) + minDuration;
```

```
    SKAction * actionMove = [SKAction moveTo:CGPointMake(-squiggy.
size.width/2, actualY) duration:actualDuration];
    SKAction * actionMoveDone = [SKAction removeFromParent];
    [squiggy runAction:[SKAction sequence:@[actionMove,
actionMoveDone]]];
}
```

This method creates the Squiggy sprite node. Then, we create a random *y* axis coordinate to spawn our Squiggy. Why a random *y* axis? *y* you ask? (ah, pun intended). Our Squiggy is a flying creature, of course! We also spawn him a little past the visible edge of the screen so that it looks like he's flying into the scene, hence we will add the Squiggy 1000px beyond the position of the player.

We will add him as a child of the map because we want him to scroll with the map, as well as give him a name so that we can do our collision detection.

Then, we set a random speed range for our Squiggies to be flying at. Then we create the actions for SpriteKit to get them flying.

We now need to implement two new properties in our `GameLevelScene.m` class. These new classes will control the spawn rate of our Squiggies. So, at the top of the class file, add the following implementations:

```
@property (nonatomic) NSTimeInterval lastSpawnTimeInterval;
@property (nonatomic) NSTimeInterval lastUpdateTimeInterval;
```

We will use the `lastSpawnTimeInterval` method to keep track of the elapsed time since we spawned a Squiggy and the `lastUpdateTimeInterval` method to detect the time elapsed since the last update.

We are now going to create a new update method, but don't worry, it won't interfere with our main `update` method. Go ahead and add this method anywhere in the `GameLevelScene.m` file:

```
- (void)updateWithTimeSinceLastUpdate:(CFTimeInterval)timeSinceLast {

    self.lastSpawnTimeInterval += timeSinceLast;
    if (self.lastSpawnTimeInterval > 2) {
        self.lastSpawnTimeInterval = 0;
        [self addSquiggy];
    }
}

//Then add this method into our regular update method to call our new
updateWithTimeSinceLastUpdate method
```

```
        CFTimeInterval timeSinceLast = currentTime - self.
lastUpdateTimeInterval;
    self.lastUpdateTimeInterval = currentTime;
    if (timeSinceLast > 1) {
        timeSinceLast = 1.0 / 60.0;

    }

    [self updateWithTimeSinceLastUpdate:timeSinceLast];
```

Build, run, and forecast calls for scattered showers and Squiggies! The result looks pretty cool! We need to make our player get hurt when he touches a Squiggy. Hey, they look cute, but they're dangerous!

We need to do some work with physics here, so bare with me.

At the top of our `GameLevelScene.m` class, between the `@interface` and the `#import` section, add the following two lines of code:

```
static const uint32_t playerCategory = 0x1 << 0;
static const uint32_t enemyCategory = 0x1 << 1;
```

These lines set up two categories, one for the enemies and one for the players. We will later add another category for player and enemy projectiles when we incorporate shooting.

We now need to set up the physics of the world, so inside of our `initWithSize` method, we need to add the following lines of code:

```
self.physicsWorld.gravity = CGVectorMake(0,0);
self.physicsWorld.contactDelegate = self;
```

The preceding code sets up the world gravity and sets the scene as the delegate that will be notified when two objects collide (or physics bodies).

Physic body is anything within the level, such as the character, objects, or enemies that are controlled by physics.

Inside of our `addSquiggy` method, we need to add the following lines of code, just after we create the actual Squiggy sprite:

```
squiggy.physicsBody = [SKPhysicsBody bodyWithRectangleOfSize:squiggy.
size];
    squiggy.physicsBody.dynamic = YES;
    squiggy.physicsBody.categoryBitMask = enemyCategory;
    squiggy.physicsBody.contactTestBitMask = playerCategory;
    squiggy.physicsBody.collisionBitMask = 0;
```

What does this code do, you ask? Well, let me explain!

The first line creates a physics body for our Squiggy sprite. The body is the rectangle surrounding the sprite. The next line sets the sprite to be dynamic, meaning that the SpriteKit physics engine will have no bearing over the movement of our Squiggy. The only thing that controls it is our code. The third line puts our Squiggy into the `enemyCategory` method.

We then tell the engine which category method, if a collision occurs between the two objects, should notify us. Obviously, we selected the `playerCategory` method.

The next and final line we added is a bit of a doozy to understand, we set the `collisionBitMask` value to `0`. The collision bit mask defines the response of each object when a collision occurs. In our case, we set it to `0`, meaning that they won't react to each other as far as a ricochet or bounce is concerned.

We need to add similar code for our player when we create it, so let's create a new method that we will call each time our player is created (that is, just after he dies, or spawns a new level, and so on):

```
-(void)setUpPlayerPhysics{
```

```
    self.player1.physicsBody = [SKPhysicsBody
bodyWithRectangleOfSize:self.player1.size];
    self.player1.physicsBody.dynamic = YES;
    self.player1.physicsBody.categoryBitMask = playerCategory;
    self.player1.physicsBody.contactTestBitMask = enemyCategory;
    self.player1.physicsBody.collisionBitMask = 0;
}
```

This is the same method that we used to set up our enemy's physics; we just switched around the categories.

Now, at the bottom of the `didHeWin` method, as well as in the `touchesBegan` method where we set up our player after we touch the button, add this line of code:

```
[self setUpPlayerPhysics];
```

Super duper easy peasy! Now we need to actually detect the collisions between the two objects.

We need to add the following method anywhere in our `GameLevelScene.m` class file:

```
- (void)didBeginContact:(SKPhysicsContact *)contact
{
    SKPhysicsBody *firstBody, *secondBody;

    uint32_t collision = (contact.bodyA.categoryBitMask | contact.
bodyB.categoryBitMask);

    if (collision == (playerCategory | enemyCategory)) {
        [contact.bodyA.node removeFromParent];
        [contact.bodyB.node removeFromParent];
        [self gameOver:0];
    }

    else if (collision == (bulletCategory | enemyCategory)) {
        for (SKSpriteNode *playerBullet in _playerBullets) {
            if (playerBullet.hidden == NO) {
                [contact.bodyA.node removeFromParent];
                [contact.bodyB.node removeFromParent];
            }
        }
    }

    if (contact.bodyA.categoryBitMask < contact.bodyB.categoryBitMask)
    {
        firstBody = contact.bodyA;
        secondBody = contact.bodyB;

    }
```

```
        else
        {
            firstBody = contact.bodyB;
            secondBody = contact.bodyA;

        }
    }
```

We still have to add the `bulletCategory` function, so if you were to build the project right now, it would show you an error. Don't worry, we will add it in just a few moments. Finally, we will make some slight adjustments to our `GameOver` method:

```
-(void)gameOver:(BOOL)won {
    if (_level > 0){
    self.gameOver = YES;
        [_player1 removeFromParent];
    NSString *gameText;
    if (won) {
        gameText = @"Level Complete!";

    } else {
        gameText = @"You have died!";
        self.player1 = [[Player alloc] initWithImageNamed:@"P1idle"];
        self.player1.position = CGPointMake(100, 50);
        self.player1.zPosition = 15;
        [self setUpPlayerPhysics];
        [self.map addChild:self.player1];
        [self setViewpointCenter:self.player1.position];
        self.gameOver = NO;
    }
}
}
```

Here, we changed our code so every time our player is killed, he is removed from the scene, and a new player is created at the beginning of the level. Also, the physics is set up yet again.

After testing a few methods, the above method is the one that worked the best. I simply teleported our player about 100px ahead of the normal beginning position. No idea why. This works though!

Now let's get shooting!

To begin with, we will add another category like we did earlier; this will be as follows:

```
static const uint32_t bulletCategory = 0x1 << 2;
```

We will now use this for our player's bullets. Also, we want to add a new definition, so just under the `#import` section, add the following line of code:

```
#define kNumBullets 20
```

This definition will assist in creating an array for all our bullets, which we will do a little further in just a bit.

Just above the `@implementation` line and just before the `@end` line of our property declarations, add the following two lines of code:

```
NSMutableArray *_playerBullets;
int _nextPlayerBullet;
```

We created the player bullet array that will allow us to create multiple bullets at once. We then defined the number of the next bullet within the array.

Inside our `setUpPlayerPhysics` method, add the following block of code (don't forget to import all the images located in the resource section of this book):

```
  #pragma mark - Setup the bullets
    _playerBullets = [[NSMutableArray alloc]
initWithCapacity:kNumBullets];
    for (int i = 0; i < kNumBullets; ++i) {
        SKSpriteNode *playerBullet = [SKSpriteNode spriteNodeWithImage
Named:@"Bullet"];
        playerBullet.hidden = YES;
        [_playerBullets addObject:playerBullet];
        [self.map addChild:playerBullet];
    }
```

Welcome to arrays! What does all this mean? Easy! We take the `_playerBullets` array, initialize it with a predefined number of bullets in the array; the `kNumBullets` variable which was `20`. So now the array has a value of `20`. Then, for every entry in the array, from `0-20`, add a bullet. It really is that easy.

We will create a new method, which will be as follows:

```
-(void) startTheGame {
    for (SKSpriteNode *playerBullet in _playerBullets) {
        playerBullet.hidden = YES;
    }
}
```

We will call the preceding method when the player presses the play button, so in our `touchesBegan` method, at the bottom of the `if ([node.name isEqualToString:@"playButton"]) {` statement, add the following:

```
[self startTheGame];
```

Now, we need to create a new touch zone. So, in our `touchesBegan` method, we need to edit our `if (touchLocation.x > self.size.width / 2.0) {` statement to the following:

```
 if (touchLocation.x > self.size.width / 2.0) {
            if (touchLocation.y < self.size.height / 2.0) {
            self.player1.jumping = YES;
            }
            else if (touchLocation.y > self.size.height / 2.0) {
                NSLog(@"PEW");
                if (touchLocation.x > self.size.width / 2.0) {
                    if (touchLocation.y < self.size.height / 2.0) {
                        self.player1.jumping = YES;
                    }
                    else if (touchLocation.y > self.size.height / 2.0)
{
                        if (self.player1.xScale == - 1.0) {
                            SKSpriteNode *bullet = [_playerBullets
objectAtIndex:_nextPlayerBullet];
                            _nextPlayerBullet++;
                            if (_nextPlayerBullet >= _playerBullets.
count) {
                                _nextPlayerBullet = 0;
                            }


                            bullet.position = CGPointMake(_player1.
position.x-bullet.size.width/2,_player1.position.y+0);
                            bullet.hidden = NO;
                            [bullet removeAllActions];


                            CGPoint location = CGPointMake(_player1.
position.x - 1000, _player1.position.y);
                            SKAction *bulletMoveAction = [SKAction
moveTo:location duration:2.5];

                            SKAction *bulletDoneAction = [SKAction
runBlock:(dispatch_block_t)^() {
                                bullet.hidden = YES;
```

```
                                }];


                                SKAction *moveBulletActionWithDone =
    [SKAction sequence:@[bulletMoveAction,bulletDoneAction]];
                                [bullet runAction:moveBulletActionWithDone
    withKey:@"bulletFired"];
                            }


                           else {
                                SKSpriteNode *bullet = [_playerBullets
    objectAtIndex:_nextPlayerBullet];
                                _nextPlayerBullet++;
                                if (_nextPlayerBullet >= _playerBullets.
    count) {
                                    _nextPlayerBullet = 0;
                                }


                                bullet.position = CGPointMake(_player1.
    position.x+bullet.size.width/2,_player1.position.y+0);
                                bullet.hidden = NO;
                                [bullet removeAllActions];


                                CGPoint location = CGPointMake(_player1.
    position.x + 1000, _player1.position.y);
                                SKAction *bulletMoveAction = [SKAction
    moveTo:location duration:2.5];

                                SKAction *bulletDoneAction = [SKAction
    runBlock:(dispatch_block_t)^() {
                                    bullet.hidden = YES;
                                }];


                                SKAction *moveBulletActionWithDone =
    [SKAction sequence:@[bulletMoveAction,bulletDoneAction]];
                                [bullet runAction:moveBulletActionWithDone
    withKey:@"bulletFired"];
                            }
                        }
                    }
                }
            }
```

Here, we selected one of the bullets within the array. We then set the position of that bullet at the player's position. Next, we set the bullet's final position off the screen, which is why we added two methods, one if the player presses the button when walking backward, and the other going forward.

Now, when you test it you will see the following output:

Lots of pew pew! And it shoots according to the direction you're facing too, so that's pretty awesome.

Now, back in our `setUpPlayerPhysics` method, where we set up the bullets, we need to set the `physicsBody` value for each bullet. In the `for` loop, just before we add the bullet to the map as a child, add the following code:

```
playerBullet.hidden = YES;
        playerBullet.physicsBody = [SKPhysicsBody bodyWithRectangleOfS
ize:playerBullet.size];
        playerBullet.physicsBody.dynamic = YES;
        playerBullet.physicsBody.categoryBitMask = bulletCategory;
        playerBullet.physicsBody.contactTestBitMask = enemyCategory;
        playerBullet.physicsBody.collisionBitMask = 0;
```

This is the same as our previous `physicsBody` setup, except we changed the bullet's category to the `bulletCategory`.

Now, when you test your project, you should be shooting, dying, and killing without a care in the world! Isn't it great?

If it is working for you, great! I'm happy for you! We've done a lot of work in this chapter and our game really isn't even close to finishing. Why not try to take what you've learned in this chapter and create your own levels? Spawn some new enemies, and maybe try having some shooting baddies? If not, I'll cover some crazier baddies later on in this book. For now let's go take a break!

Our game is running efficiently, running at an almost constant 60 fps! Awesome! In case you don't have things working just right, the following is the complete source code you should have up until now:

```
//  GameLevelScene.m


#import "GameLevelScene.h"
#import "JSTileMap.h"
#import "Player.h"
```

```
#import "SKTUtils.h"
#import "SKTAudio.h"
#define kNumBullets 20

static const uint32_t playerCategory = 0x1 << 0;
static const uint32_t enemyCategory = 0x1 << 1;
static const uint32_t bulletCategory = 0x1 << 2;

@interface GameLevelScene() <SKPhysicsContactDelegate>
@property (nonatomic, strong) JSTileMap *map;
@property (nonatomic, strong) Player *player1;
@property (nonatomic, assign) NSTimeInterval previousTime;
@property (nonatomic, strong) TMXLayer *walls;
@property (nonatomic, strong) TMXLayer *hazards;
@property (nonatomic, assign) BOOL gameOver;

@property (nonatomic) NSTimeInterval lastSpawnTimeInterval;
@property (nonatomic) NSTimeInterval lastUpdateTimeInterval;
@property (nonatomic, assign) NSInteger level;
@end

NSMutableArray *_playerBullets;
int _nextPlayerBullet;

@implementation GameLevelScene


-(id)initWithSize:(CGSize)size {
    if (self = [super initWithSize:size]) {


        self.userInteractionEnabled = YES;
        self.backgroundColor = [SKColor colorWithRed:.0 green:.0
blue:.0 alpha:1.0];
        self.physicsWorld.gravity = CGVectorMake(0, 0);
        self.physicsWorld.contactDelegate = self;



        if (_level == 0){
        SKLabelNode *playLabel = [SKLabelNode labelNodeWithFontNamed:@
"AvenirNext-Heavy"];
        playLabel.text = @"Adesa";
        playLabel.fontSize = 40;
        playLabel.position = CGPointMake(self.size.width / 2.0, self.
size.height / 1.7);
```

```
            [self addChild:playLabel];

            SKSpriteNode *playButton = [SKSpriteNode spriteNodeWithImageN
    amed:@"play"];
            playButton.position = CGPointMake(self.size.width /2.0 , self.
    size.height / 2.5);
            playButton.name = @"playButton";
            [self addChild:playButton];


            }

        }
        return self;
    }

    -(void) startTheGame {
        for (SKSpriteNode *playerBullet in _playerBullets) {
            playerBullet.hidden = YES;
        }
    }

    - (void)addSquiggy {

        SKSpriteNode * squiggy = [SKSpriteNode spriteNodeWithImageNamed:@
    "Squiggy"];
        squiggy.physicsBody = [SKPhysicsBody
    bodyWithRectangleOfSize:squiggy.size];
        squiggy.physicsBody.dynamic = YES;
        squiggy.physicsBody.categoryBitMask = enemyCategory;
        squiggy.physicsBody.contactTestBitMask = playerCategory;
        squiggy.physicsBody.collisionBitMask = 0;


        int minY = squiggy.size.height / 2;
        int maxY = self.frame.size.height - squiggy.size.height / 2;
        int rangeY = maxY - minY;
        int actualY = (arc4random() % rangeY) + minY;
        squiggy.xScale = -1.0;
        squiggy.position = CGPointMake(self.player1.position.x + 1000 +
    squiggy.size.width/2, actualY);
        [self.map addChild:squiggy];

        // Setting the Speed of SQUIGGY!
        int minDuration = 10.0;
        int maxDuration = 20.0;
        int rangeDuration = maxDuration - minDuration;
```

```
    int actualDuration = (arc4random() % rangeDuration) + minDuration;

    SKAction * actionMove = [SKAction moveTo:CGPointMake(-squiggy.
size.width/2, actualY) duration:actualDuration];
    SKAction * actionMoveDone = [SKAction removeFromParent];
    [squiggy runAction:[SKAction sequence:@[actionMove,
actionMoveDone]]];
}

- (void)updateWithTimeSinceLastUpdate:(CFTimeInterval)timeSinceLast {

    self.lastSpawnTimeInterval += timeSinceLast;
    if (self.lastSpawnTimeInterval > 2) {
        self.lastSpawnTimeInterval = 0;
        [self addSquiggy];

    }
}


-(CGRect)tileRectFromTileCoords:(CGPoint)tileCoords {
    float levelHeightInPixels = self.map.mapSize.height * self.map.
tileSize.height;
    CGPoint origin = CGPointMake(tileCoords.x * self.map.tileSize.
width, levelHeightInPixels - ((tileCoords.y + 1) * self.map.tileSize.
height));
    return CGRectMake(origin.x, origin.y, self.map.tileSize.width,
self.map.tileSize.height);
}

- (NSInteger)tileGIDAtTileCoord:(CGPoint)coord forLayer:(TMXLayer *)
layer {
    TMXLayerInfo *layerInfo = layer.layerInfo;
    return [layerInfo tileGidAtCoord:coord];
}

- (void)handleHazardCollisions:(Player *)player
{
    [self didHeWin];

    NSInteger indices[8] = {7, 1, 3, 5, 0, 2, 6, 8};

    for (NSUInteger i = 0; i < 8; i++) {
        NSInteger tileIndex = indices[i];
```

```
            CGRect playerRect = [player collisionBox];
            CGPoint playerCoord = [self.hazards coordForPoint:player.
    desiredPosition];

            NSInteger tileColumn = tileIndex % 3;
            NSInteger tileRow = tileIndex / 3;
            CGPoint tileCoord = CGPointMake(playerCoord.x + (tileColumn -
    1), playerCoord.y + (tileRow - 1));

            NSInteger gid = [self tileGIDAtTileCoord:tileCoord
    forLayer:self.hazards];
            if (gid != 0) {
                CGRect tileRect = [self tileRectFromTileCoords:tileCoord];
                if (CGRectIntersectsRect(playerRect, tileRect)) {
                    [self gameOver:0];
                }
            }
        }
    }

    -(void)gameOver:(BOOL)won {
        if (_level > 0){
        self.gameOver = YES;
            [_player1 removeFromParent];
        NSString *gameText;
        if (won) {
            gameText = @"Level Complete!";

        } else {
            gameText = @"You have died!";
            self.player1 = [[Player alloc] initWithImageNamed:@"P1idle"];
            self.player1.position = CGPointMake(100, 50);
            self.player1.zPosition = 15;
            [self setUpPlayerPhysics];
            [self.map addChild:self.player1];
            [self setViewpointCenter:self.player1.position];
            self.gameOver = NO;
        }
    }
    }

    -(void)setUpPlayerPhysics{
        self.player1.physicsBody = [SKPhysicsBody
    bodyWithRectangleOfSize:self.player1.size];
        self.player1.physicsBody.dynamic = YES;
        self.player1.physicsBody.categoryBitMask = playerCategory;
```

```
    self.player1.physicsBody.contactTestBitMask = enemyCategory;
    self.player1.physicsBody.collisionBitMask = 0;

#pragma mark - Setup the bullets
    _playerBullets = [[NSMutableArray alloc]
initWithCapacity:kNumBullets];
    for (int i = 0; i < kNumBullets; ++i) {
        SKSpriteNode *playerBullet = [SKSpriteNode spriteNodeWithImage
Named:@"Bullet"];
        playerBullet.hidden = YES;
        playerBullet.physicsBody = [SKPhysicsBody bodyWithRectangleOfS
ize:playerBullet.size];
        playerBullet.physicsBody.dynamic = YES;
        playerBullet.physicsBody.categoryBitMask = bulletCategory;
        playerBullet.physicsBody.contactTestBitMask = enemyCategory;
        playerBullet.physicsBody.collisionBitMask = 0;

        [_playerBullets addObject:playerBullet];
        [self.map addChild:playerBullet];
    }
}

-(void)didHeWin {
    if (self.player1.position.x > 3200.0) {

        if (_level == 1) {
        [self.map removeFromParent];
        self.map = [JSTileMap mapNamed:@"level2.tmx"];
        [self addChild:self.map];
        }
        self.player1 = [[Player alloc] initWithImageNamed:@"P1idle"];
        self.player1.position = CGPointMake(100, 50);
        self.player1.zPosition = 15;
        [self.map addChild:self.player1];
        self.walls = [self.map layerNamed:@"walls"];
        self.hazards = [self.map layerNamed:@"hazards"];
        [self setUpPlayerPhysics];
    }
}


- (void)checkForAndResolveCollisionsForPlayer:(Player *)player
forLayer:(TMXLayer *)layer
{
```

```
    NSInteger indices[8] = {7, 1, 3, 5, 0, 2, 6, 8};
    player.onGround = NO;
    for (NSUInteger i = 0; i < 8; i++) {
        NSInteger tileIndex = indices[i];

        CGRect playerRect = [player collisionBox];
        CGPoint playerCoord = [layer coordForPoint:player.
desiredPosition];
        if (playerCoord.y >= self.map.mapSize.height - 1) {
            [self gameOver:0];
            return;
        }

        NSInteger tileColumn = tileIndex % 3;
        NSInteger tileRow = tileIndex / 3;
        CGPoint tileCoord = CGPointMake(playerCoord.x + (tileColumn -
1), playerCoord.y + (tileRow - 1));

        NSInteger gid = [self tileGIDAtTileCoord:tileCoord
forLayer:layer];
        if (gid != 0) {
            CGRect tileRect = [self tileRectFromTileCoords:tileCoord];

            if (CGRectIntersectsRect(playerRect, tileRect)) {
                CGRect intersection = CGRectIntersection(playerRect,
tileRect);

                if (tileIndex == 7) {

                    player.desiredPosition = CGPointMake(player.
desiredPosition.x, player.desiredPosition.y + intersection.size.
height);
                    player.velocity = CGPointMake(player.velocity.x,
0.0);
                    player.onGround = YES;
                } else if (tileIndex == 1) {

                    player.desiredPosition = CGPointMake(player.
desiredPosition.x, player.desiredPosition.y - intersection.size.
height);
                } else if (tileIndex == 3) {

                    player.desiredPosition = CGPointMake(player.
desiredPosition.x + intersection.size.width, player.
desiredPosition.y);
                } else if (tileIndex == 5) {
```

```
                    player.desiredPosition = CGPointMake(player.
desiredPosition.x - intersection.size.width, player.
desiredPosition.y);

                } else {
                    if (intersection.size.width > intersection.size.
height) {

                        player.velocity = CGPointMake(player.
velocity.x, 0.0);
                        float intersectionHeight;
                        if (tileIndex > 4) {
                            intersectionHeight = intersection.size.
height;
                            player.onGround = YES;
                        } else {
                            intersectionHeight = -intersection.size.
height;
                        }
                        player.desiredPosition = CGPointMake(player.
desiredPosition.x, player.desiredPosition.y + intersection.size.height
);
                    } else {

                        float intersectionWidth;
                        if (tileIndex == 6 || tileIndex == 0) {
                            intersectionWidth = intersection.size.
width;
                        } else {
                            intersectionWidth = -intersection.size.
width;
                        }

                        player.desiredPosition = CGPointMake(player.
desiredPosition.x  + intersectionWidth, player.desiredPosition.y);
                    }
                }
            }
        }
    }

    player.position = player.desiredPosition;
    [self handleHazardCollisions:self.player1];
}


- (void)didBeginContact:(SKPhysicsContact *)contact
{
```

```
    SKPhysicsBody *firstBody, *secondBody;

    uint32_t collision = (contact.bodyA.categoryBitMask | contact.
bodyB.categoryBitMask);

    if (collision == (playerCategory | enemyCategory)) {
        [contact.bodyA.node removeFromParent];
        [contact.bodyB.node removeFromParent];
        [self gameOver:0];
    }

    else if (collision == (bulletCategory | enemyCategory)) {
        for (SKSpriteNode *playerBullet in _playerBullets) {
            if (playerBullet.hidden == NO) {
                [contact.bodyA.node removeFromParent];
                [contact.bodyB.node removeFromParent];
            }
        }
    }

    if (contact.bodyA.categoryBitMask < contact.bodyB.categoryBitMask)
    {
        firstBody = contact.bodyA;
        secondBody = contact.bodyB;

    }
    else
    {
        firstBody = contact.bodyB;
        secondBody = contact.bodyA;

    }
}



- (void)setViewpointCenter:(CGPoint)position {
    NSInteger x = MAX(position.x, self.size.width / 2);
    NSInteger y = MAX(position.y, self.size.height / 2);
    x = MIN(x, (self.map.mapSize.width * self.map.tileSize.width) -
self.size.width / 2);
    y = MIN(y, (self.map.mapSize.height * self.map.tileSize.height) -
self.size.height / 2);
    CGPoint actualPosition = CGPointMake(x, y);
    CGPoint centerOfView = CGPointMake(self.size.width/2, self.size.
height/2);
```

```
    CGPoint viewPoint = CGPointSubtract(centerOfView, actualPosition);
    self.map.position = viewPoint;
}

- (void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event
{

    for (UITouch *touch in touches) {
        CGPoint touchLocation = [touch locationInNode:self];

        SKNode *node = [self nodeAtPoint:touchLocation];

        if ([node.name isEqualToString:@"playButton"]) {
            _level = 1;
            [self removeAllChildren];
            self.backgroundColor = [SKColor colorWithRed:.25 green:.0
blue:.25 alpha:1.0];
            [[SKTAudio sharedInstance] playBackgroundMusic:@"Backgrou
ndAudio.mp3"];
            self.map = [JSTileMap mapNamed:@"level1.tmx"];
            [self addChild:self.map];
            NSString *rainParticles =
            [[NSBundle mainBundle] pathForResource:@"Rain"
ofType:@"sks"];

            SKEmitterNode *rainEmitter =
            [NSKeyedUnarchiver unarchiveObjectWithFile:rainParticles];

            rainEmitter.position = CGPointMake(0, self.scene.size.
height);

            [self addChild:rainEmitter];

            NSString *fireParticles =
            [[NSBundle mainBundle] pathForResource:@"Flames"
ofType:@"sks"];

            SKEmitterNode *fireEmitter =
            [NSKeyedUnarchiver unarchiveObjectWithFile:fireParticles];
            fireEmitter.position = CGPointMake(25, 50);

            [_map addChild:fireEmitter];

            self.player1 = [[Player alloc]
initWithImageNamed:@"P1idle"];
            self.player1.position = CGPointMake(100, 50);
            self.player1.zPosition = 15;
```

```
            [self.map addChild:self.player1];
            self.walls = [self.map layerNamed:@"walls"];
            self.hazards = [self.map layerNamed:@"hazards"];
            [self setUpPlayerPhysics];
            [self startTheGame];
        }

        if (touchLocation.x > self.size.width / 2.0) {
            if (touchLocation.y < self.size.height / 2.0) {
            self.player1.jumping = YES;
            }
            else if (touchLocation.y > self.size.height / 2.0) {
                NSLog(@"PEW");
                if (touchLocation.x > self.size.width / 2.0) {
                    if (touchLocation.y < self.size.height / 2.0) {
                        self.player1.jumping = YES;
                    }
                    else if (touchLocation.y > self.size.height / 2.0)
{
                        if (self.player1.xScale == - 1.0) {
                            SKSpriteNode *bullet = [_playerBullets
objectAtIndex:_nextPlayerBullet];
                            _nextPlayerBullet++;
                            if (_nextPlayerBullet >= _playerBullets.
count) {
                                _nextPlayerBullet = 0;
                            }


                            bullet.position = CGPointMake(_player1.
position.x-bullet.size.width/2,_player1.position.y+0);
                            bullet.hidden = NO;
                            [bullet removeAllActions];


                            CGPoint location = CGPointMake(_player1.
position.x - 1000, _player1.position.y);
                            SKAction *bulletMoveAction = [SKAction
moveTo:location duration:2.5];

                            SKAction *bulletDoneAction = [SKAction
runBlock:(dispatch_block_t)^() {
                                bullet.hidden = YES;
                            }];
```

```
                            SKAction *moveBulletActionWithDone =
[SKAction sequence:@[bulletMoveAction,bulletDoneAction]];
                            [bullet runAction:moveBulletActionWithDone
withKey:@"bulletFired"];
                        }


                        else {
                            SKSpriteNode *bullet = [_playerBullets
objectAtIndex:_nextPlayerBullet];
                            _nextPlayerBullet++;
                            if (_nextPlayerBullet >= _playerBullets.
count) {
                                _nextPlayerBullet = 0;
                            }


                            bullet.position = CGPointMake(_player1.
position.x+bullet.size.width/2,_player1.position.y+0);
                            bullet.hidden = NO;
                            [bullet removeAllActions];


                            CGPoint location = CGPointMake(_player1.
position.x + 1000, _player1.position.y);
                            SKAction *bulletMoveAction = [SKAction
moveTo:location duration:2.5];

                            SKAction *bulletDoneAction = [SKAction
runBlock:(dispatch_block_t)^() {
                                bullet.hidden = YES;
                            }];


                            SKAction *moveBulletActionWithDone =
[SKAction sequence:@[bulletMoveAction,bulletDoneAction]];
                            [bullet runAction:moveBulletActionWithDone
withKey:@"bulletFired"];
                        }
                    }
                }
            }
        }

        else {
            if (touchLocation.x < self.size.width / 2.0) {
                if (touchLocation.y > self.size.height / 2){
```

```
                    self.player1.goingBackwards = YES;
                    self.player1.animateWalking = YES;
                    self.player1.xScale = -1.0;
                }
                if (touchLocation.y < self.size.height / 2){
                    self.player1.walking = YES;
                    self.player1.animateWalking = YES;
                    self.player1.xScale = 1.0;
                }
            }

        }
    }
}

- (void)touchesMoved:(NSSet *)touches withEvent:(UIEvent *)event {
    for (UITouch *touch in touches) {

        float halfWidth = self.size.width / 2.0;
        CGPoint touchLocation = [touch locationInNode:self];


        CGPoint previousTouchLocation = [touch
previousLocationInNode:self];

        if (touchLocation.x > halfWidth && previousTouchLocation.x <=
halfWidth) {
            self.player1.walking = NO;
            self.player1.goingBackwards = NO;
            self.player1.jumping = YES;
        } else if (previousTouchLocation.x > halfWidth &&
touchLocation.x <= halfWidth) {

            self.player1.goingBackwards = NO;
            self.player1.jumping = NO;
        }
        else if (previousTouchLocation.x > halfWidth &&
touchLocation.x <= halfWidth) {
            if (touchLocation.y > self.size.height / 2){
            self.player1.walking = NO;
            self.player1.jumping = NO;
            }
        }
    }
}
```

```
- (void)touchesEnded:(NSSet *)touches withEvent:(UIEvent *)event {

    for (UITouch *touch in touches) {
        CGPoint touchLocation = [touch locationInNode:self];
        if (touchLocation.x < self.size.width / 2.0) {
            if (touchLocation.x < self.size.width / 2.0) {
                if (touchLocation.y > self.size.height / 2){
                    self.player1.goingBackwards = NO;
                    self.player1.animateWalking = NO;
                }
                if (touchLocation.y < self.size.height / 2){
                    self.player1.walking = NO;
                    self.player1.animateWalking = NO;
                }
            }

        } else {
            self.player1.jumping = NO;
        }
    }
}



- (void)update:(NSTimeInterval)currentTime
{


    CFTimeInterval timeSinceLast = currentTime - self.
lastUpdateTimeInterval;
    self.lastUpdateTimeInterval = currentTime;
    if (timeSinceLast > 1) {
        timeSinceLast = 1.0 / 60.0;
        self.lastUpdateTimeInterval = currentTime;
    }

    [self updateWithTimeSinceLastUpdate:timeSinceLast];

    if (self.gameOver) return;

    [self setViewpointCenter:self.player1.position];
    NSTimeInterval delta = currentTime - self.previousTime;

    if (delta > 0.02) {
        delta = 0.02;
    }
```

```
        self.previousTime = currentTime;

        [self.player1 update:delta];

        [self checkForAndResolveCollisionsForPlayer:self.player1
    forLayer:self.walls];
    }

    @end //(which is fitting because this is the end of the chapter... and
    end of the page :D)
```

Summary

Let's recall all that we covered in this chapter. We added some music and cool sound effects to our game. Then, we gave our player a proper walking animation so that he doesn't look like he's floating in the air! However, if that's not an effect you're looking for, you can save yourself a bunch of typing. Next, we made some really awesome looking particles! We made it rain, and we made the wrecked ship burn in flames!

We briefly discussed menus, and how we can create a simple main menu. We will polish this up later when preparing for publishing.

Then, the enemies! We made our Squiggies, those cute yet dangerous enemies in our game. Things are going to get scary, so buckle up and enjoy the ride.

Note: It will only get scary for our player, and not for us because we are the ones creating the madness.

5
Bug Squashing – Testing and Debugging

I'm still trying to catch my breath from the last chapter! Boy, did we ever do a lot of work! Now it's time to slow it down a little bit. After all that coding, we are going to discuss testing and debugging. Debugging is a critical step in the entire development process, as the appearance of bugs can ruin the experience of your app. You spend countless hours on it, and, if there are bugs, people will think it's unprofessional. That's certainly not ideal. Let's see what we will discuss in this chapter:

- Testing our code on the simulator
- Testing our project on various devices
- Setting up TestFlight for beta testers
- Squishing those bugs

We will slow things down for this chapter and will resume the coding soon! So, let's sit back and treat this chapter as a breather. But don't think it will be a cake walk! No no! While this isn't exactly as action-packed as our last chapter, as I mentioned, this topic is super-critical in all projects, so it's super-important we pay attention to it.

Boom!

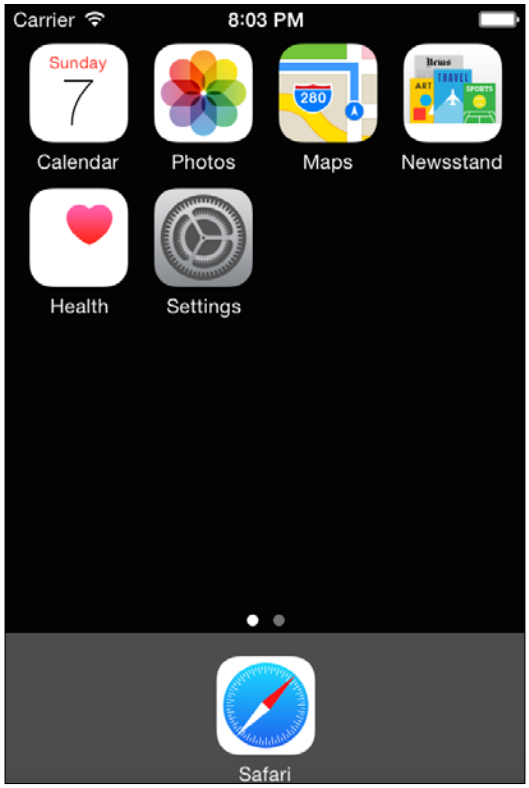
On to it!

Testing our project

No doubt you've been messing around with the iOS simulator (a feature that, to be honest, is very helpful, and that I found easier to access than the Android SDK emulator), which is awesome, but there is actually quite a lot that you can do with it without having to actually install anything on a physical device.

Let's open up our device simulator. To do this, inside Xcode, click on **Xcode** on the bar at the top, then click on **Open Developer Tool**, and finally click on **iOS Simulator**.

You will now be greeted with a pixel-for-pixel simulation of any iOS device of your choice:



As I mentioned earlier, I like to use the iPhone 4S simulator because it's small enough to fit on my screen, but you do have your pick of the litter. Is your simulator set to a device that you don't want? Want to change it to something else such as the glittery iPhone 6+? No problem!

Simply click on **Hardware** on the top bar. Then, under **Device**, you can select any awesome Apple mobile device you like! When you select a new device, you will need to allow some time for the new device to boot.

Let's explore!

If you click on **File**, you can snap a screenshot or at least see what key combination you can press to create a screenshot of the device screen (this is great for uploading to Facebook or iTunes).

When you click on **Edit**, you can copy the text, copy the screen, paste, start dictation, and see the emojis.

Back to **Hardware**, again you can select your device, you can rotate the device left or right, trigger a shake device gesture, trigger the home and lock buttons, or force a reboot.

You can also simulate a memory warning, trigger an in-call status bar, and select your keyboard settings as well as an external device.

Under the **Debug** setting, you can choose to slow down animations, such as UI animations for opening and closing and so on.

You can also do some funky color stuff with the screen!

What?

Yes! You can color blended layers, copied images, misaligned images, and offscreen rendered items. In other words, you can highlight them to make them look as seen in the following screenshot. Kinda funky, right?

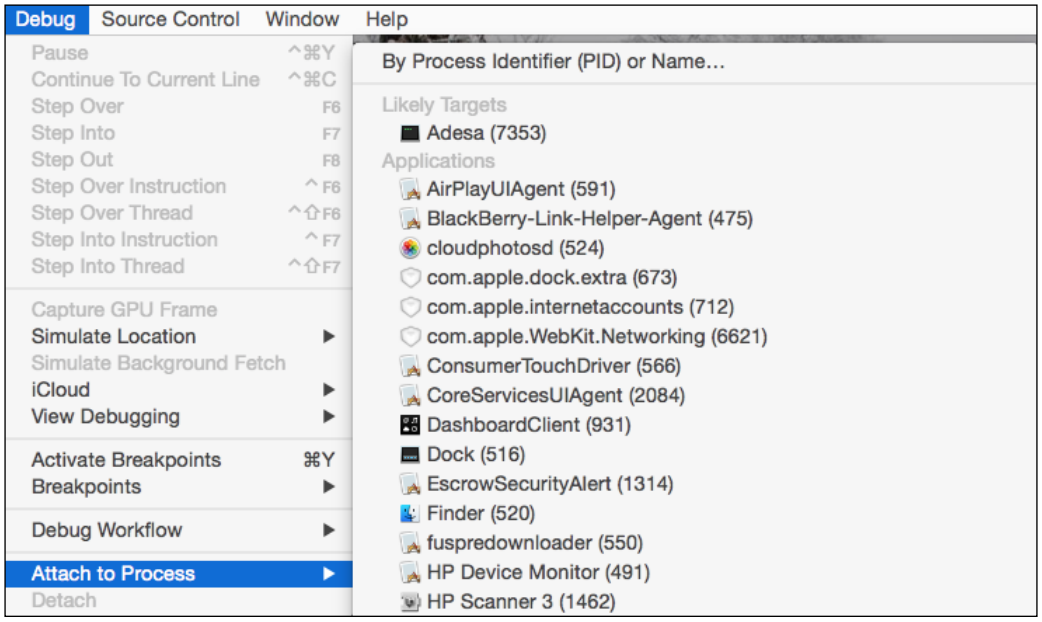
Now, why would one use this feature? Simple; sometimes sprites can get lost when debugging and you would like to keep track of them. Using this feature, you can easily track their movement. If you have a lot of sprites flying around your screen, this is a great feature to utilize. As seen in the following image, all images and sprites are tracked and clearly outlined:

Moving on, you can also open the system log, which is fantastic for debugging. You can also trigger an iCloud sync, and set the device mock location.

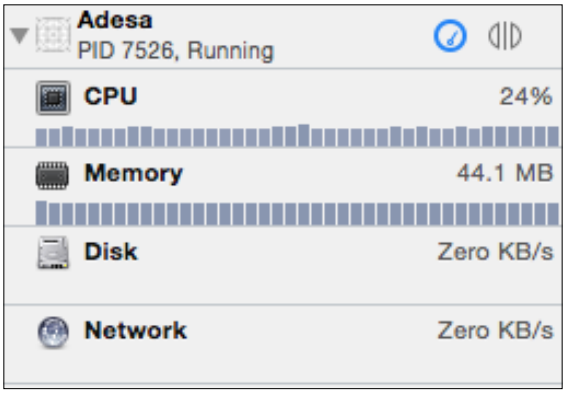
You can explore the device a little, but you will notice some settings and apps are missing. For example, you won't be able to select a background for the device, make calls, or send messages. You have a phone for that kind of stuff!

Moving on, we will build and run our project (again, by clicking on the **Play** button in **Xcode**). We should still have the SpriteKit frame rate label, but that's not exactly as in-depth as we would like.

To get some more technical statistics, click on back to go to **Xcode** and then on **Debug** on the bar at the top. Hover over **Attach to Process** to find the name of the project that we are working on. It will show up under **Likely Targets**. When you click on your project, Xcode will display some technical information, as shown in the following screenshot:



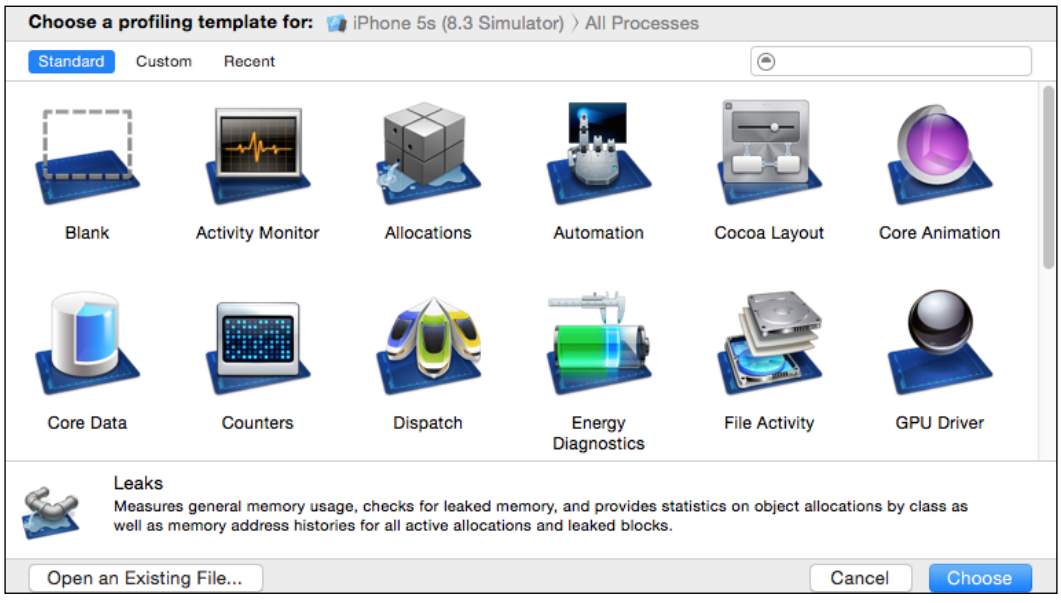
Among other details, Xcode will tell you what the **Process Identifier** (**PID**) of our app is as well as the **CPU**, **Memory**, **Disk**, and **Network** usage, as shown in the following screenshot:



Yes, that stuff is pretty great, but again, still not quite as in-depth as a developer would like to see. Want to get some really cool in-depth instrumentation?

Back in Xcode, on the bar at the top click on **Xcode**, scroll down to **Open Developer Tool**, and select **Instruments**.

Now this is where we can get crazy and do some awesome in-depth testing:



Just look at all those awesome choices! Let's go through all the options you have here to help you debug:

- **Activity Monitor**: Similar to the performance figures we have seen before, we do have more insight into what's going on within out app, as well as being able to record what's going on, and go back through our log to see at what points our performance drops.
- **Allocations**: This section tracks the app's virtual memory and breaks it down into class names as well.
- **Automation**: This executes a simple script that will simulate interaction with the user interface and, of course, record it all.
- **Cocoa Layout**: This section monitors the `NSLayoutConstraints` that is created when laying out your UI via the Storyboard editor in Xcode to see where things get out of place.
- **Core Animation**: This measures the app's graphical performance and CPU usage.
- **Core Data**: This measures the Core Data system activity. In other words, the disk or memory usage.

Moving past on to some more important ones:

- **Energy Diagnostics**: This is an excellent tool to monitor battery usage
- **Leaks**: This measures memory usage and checks for any leaking memory
- **OpenGL ES Analysis**: This measures and analyzes the rendering of our project and detects **OGLES** (as I like to call it) correctness as well as performance issues

Pretty cool, eh? In the next chapter, we will explore more of this when we discuss making our game more efficient.

Now that you have the simulator all figured out and we've looked at some of the awesome tools that Apple provides, let's move on to installing our app onto an actual device, which is actually incredibly easy.

All you have to do is connect your device to your development computer using the lightning cable or the 30-pin connector (if you have an older device) via USB.

In Xcode, in the same place where you would select the iOS simulator that you want to use, click on the Scheme toolbar, and select your device from the menu.

Xcode will automatically register the device for you to use as a development device. As opposed to how things used to be when you would have to manually add the device's UUID number via `developer.apple.com`, Apple has really streamlined this process.

Your device could possibly be disabled in the scheme editor, which would mean it isn't an eligible device. For example, say you have the beta version of iOS 9 that was just released at the time of writing this book. If you had that installed, then an older version of Xcode installed, your device would not be eligible. You would have to install the latest version of Xcode.

Now, simply clicking on the **Run** button will let Xcode install your app on the device. It's that simple!

You may have Xcode asking whether or not you want to allow codesign in the key in your keychain. Click on **always allow**. You don't want that to pop up every time you click on the **Run** button. By clicking on **always allow**, Xcode will have the permission to automatically run the project on your device without having to ask.

In the same way that we connected the debuggers and logs to our iOS simulator, you can link them to the project running on your device.

In fact, you may very well get different read-outs due to the fact that you are running on a simulator. The device could potentially run it much better or worse depending on the device and the number of flappy clones you have installed, the number of messages you have stored, the amount of storage you have remaining, and so on. It's usually a good idea (if possible) to have a device dedicated to development with nothing but your projects installed on it. That way, you know your app will perform 100% each time.

Now that we have things figured out on how to link debugging to our project and we know how to install our apps on our device, it's time to talk about getting beta testers to help you out.

Setting up TestFlight users

Why should I set up TestFlight users, you ask? Simply put, you can test your code for hours on end, and still not be able to find certain bugs. Trust me, I've done it before, only to find out after thousands of people have installed the app that there is a game destroying bug that could have been found if I let someone else play the game in a back-busting style.

Beta testers, or TestFlight users can be your friends who want to help you test your game, or even people in a focus group who will test your game and give you honest feedback. Let's discuss how to set up Test Flight users.

What's a TestFlight user?

You can define a TestFlight user as a beta tester. When you are ready for users to begin testing, you can set up TestFlight users that will receive a notification on their device when you have pushed a new version to test. From there, they can log any issues they see, so you will know if anything needs fixing.

We will start off by logging in to `itunesconnect.apple.com` with your developer Apple ID, and then you will see the following page:

From the main screen, simply click on the **Users and Rolls** button. You will see a **TestFlight Beta Testers** button on the top bar on the next page. Click on it to begin adding testers!

You will now be at the TestFlight Beta tester page. From here, you can see your internal testers (which will include you, as well as anyone else in your team) as well as your external testers.

To add people to test for you, click on the **External** button. On the next page, you will see the text, **External Tester**, next to which there should be a blue plus button, as you can see in the following screenshot:

Internal testers

Get feedback quickly by sharing your beta builds with up to 25 members of your team who have been assigned the Technical or Admin role in iTunes Connect. Each member can test on up to 10 devices.

External testers

Once you're ready, you can invite up to 1,000 users who are not part of your development organization to beta-test an app that you intend for public release on the App Store. Apps made available to external testers require a **Beta App Review** and must comply with the full **App Store Review Guidelines** before testing can begin. A review is required for new versions of your app that contain significant changes. Up to 10 apps can be tested at a time, internally or externally.

You will then be able to add as many beta testers as required. Besides entering their e-mail address, and optionally their first and last name, you can add these people to a group, which can help keep multiple testers organized, as shown in the following screenshot:

The user will then get an e-mail requesting them to set up an account. From there, they can set up TestFlight on their device. We are now going to discuss how a TestFlight user can get started testing your awesome creation.

Using TestFlight as a beta tester

Installation

You can use TestFlight on up to 10 devices and test multiple apps for multiple developers; there is no limit to the number of apps that you can test.

TestFlight (as seen by the following icon image), can only be used to test iOS apps on iPhone, iPad, and iPod touch running iOS 8 or later—not Mac apps.

Testing

Once a tester accepts the invitation, they will be able to download a test version of the app.

If they already have the live app installed on their device, the beta version of the app will replace the live version.

When they've downloaded the beta app, they will see an orange dot next to its name that identifies it as a beta.

The TestFlight app will notify the tester each time a new build is available and provide instructions on what parts of the app they should keep an eye on.

Testers can easily offer feedback by tapping the **Provide Feedback** button in the **App Details** view in TestFlight. An e-mail automatically opens with app and device details, to which the tester can add additional details and screenshots.

Apps in beta do expire. The beta period lasts for 30 days, starting from the day it's released to testers. In TestFlight, the number of days remaining appears under the **Open** button for each app.

If a beta app has **In-App Purchases**, you do not have to purchase them, as In-App Purchases made with beta builds are free.

Opting out of testing

If the tester doesn't accept your e-mail invitation, the beta app will not be installed and they will not be listed as a tester.

Also, a tester can unsubscribe from testing by using the link at the bottom of the invitation e-mail to notify you (the developer) that they would like to be removed from TestFlight.

If a tester accepts the invitation but they don't want to test the app, they can delete themselves as a tester in the **App Details** page in TestFlight. Because, you know, people are selfish.

So really, Apple has created an awesome system to allow easy, quick testing of your app.

Keep in mind that, each time you want to test a new version of your app, you need to upload the new version as well as submit the new version for review (as weird as that is, I know). It does seem redundant, but hey, it's Apple, and they know what they're doing.

Now, let's talk about crushing bugs!

Squashing those bugs!

This is a tedious thing to do. To be honest, debugging will take you many hours of figuring things out, especially when you're just starting out and don't know the code all that well. Trust me, it will take some time, but don't worry; it's worth the effort.

Let's take the app we've created, I sent it to a friend to test, and here is a snippet of the device log he lovingly sent to me:

```
Size (256.000000, 256.000000)
2015-06-11 21:24:04.357 Adesa[11289:1870205] Layer background has
zPosition -60.000000
2015-06-11 21:24:04.363 Adesa[11289:1870205] Layer walls has zPosition
-40.000000
2015-06-11 21:24:04.364 Adesa[11289:1870205] Layer hazards has
zPosition -20.000000
2015-06-11 21:24:06.725 Adesa[11289:1870205] PEW
2015-06-11 21:24:21.971 Adesa[11289:1870205] PEW
2015-06-11 21:24:22.158 Adesa[11289:1870205] PEW
2015-06-11 21:24:22.391 Adesa[11289:1870205] PEW
2015-06-11 21:24:22.511 Adesa[11289:1870205] PEW
2015-06-11 21:24:22.911 Adesa[11289:1870205] PEW
2015-06-11 21:24:23.110 Adesa[11289:1870205] PEW
2015-06-11 21:24:23.976 Adesa[11289:1870205] PEW
2015-06-11 21:24:24.145 Adesa[11289:1870205] PEW
2015-06-11 21:24:24.344 Adesa[11289:1870205] PEW
2015-06-11 21:24:37.587 Adesa[11289:1870205] PEW
2015-06-11 21:24:46.171 Adesa[11289:1870205] PEW
```

```
2015-06-11 21:24:46.304 Adesa[11289:1870205] PEW
2015-06-11 21:24:46.470 Adesa[11289:1870205] PEW
2015-06-11 21:24:46.671 Adesa[11289:1870205] PEW
2015-06-11 21:24:46.821 Adesa[11289:1870205] PEW
2015-06-11 21:25:04.079 Adesa[11289:1870205] PEW
2015-06-11 21:25:04.310 Adesa[11289:1870205] PEW
2015-06-11 21:25:05.054 Adesa[11289:1870205] PEW
2015-06-11 21:25:05.222 Adesa[11289:1870205] PEW
2015-06-11 21:25:05.422 Adesa[11289:1870205] PEW
2015-06-11 21:25:05.974 Adesa[11289:1870205] unexpected nil window in
_UIApplicationHandleEventFromQueueEvent, _windowServerHitTestWindow:
<UIClassicWindow: 0x7fa9ebb01370; frame = (0 0; 320 568);
userInteractionEnabled = NO; gestureRecognizers = <NSArray:
0x7fa9e9708080>; layer = <UIWindowLayer: 0x7fa9e9505e70>>
2015-06-11 21:25:22.856 Adesa[11289:1870205] PEW
2015-06-11 21:25:23.073 Adesa[11289:1870205] PEW
2015-06-11 21:25:23.223 Adesa[11289:1870205] PEW
Message from debugger: Terminated due to signal 15
```

He noted no issues when testing. All he noticed was the device (an iPhone 5S) seemed to slow down when he first started the game, and his phone showed the rain and the flames all at once. This is something we will consider in the next chapter.

Here are some other things to remember when debugging. As we discussed earlier, after you click on the **Run** button in the workspace toolbar and your app builds successfully, Xcode runs your app and starts a debugging session. You can debug your app directly within the source editor with graphical tools, such as data tips and Quick Look, for the value of variables.

The debug area and the debug navigator let you inspect the current state of your running application and control its execution.

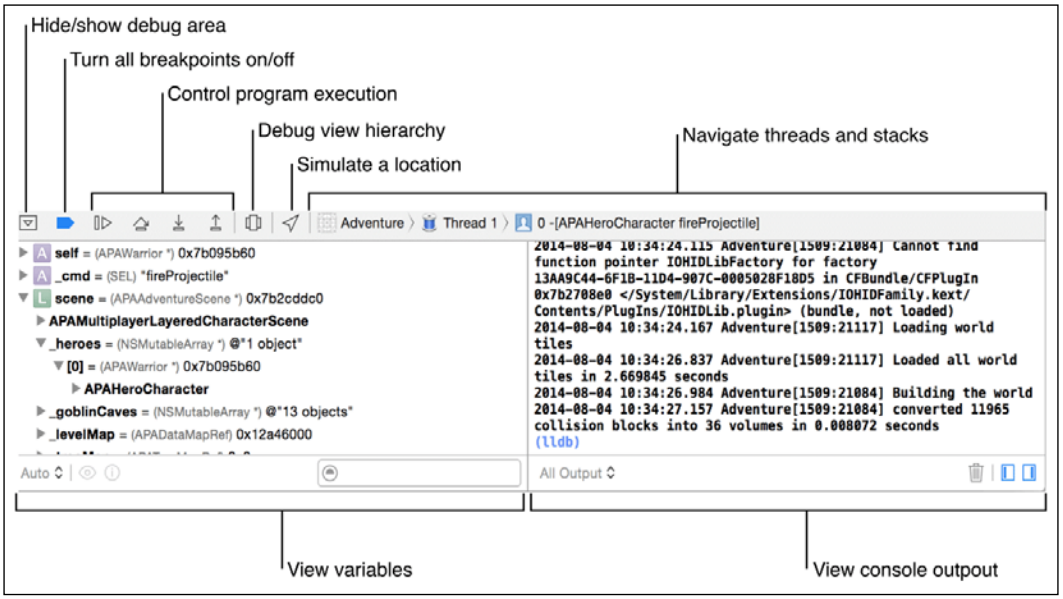
Creating a quality app (like, come on, you already have and will create many more awesome quality apps... but moving on...) requires that you minimize the app's impact on the user's device. Use the debug gauges (that we talked about earlier) in the debug navigator to gain insight into your app's resource consumption, and when you spot a problem, use instruments to measure and analyze your app's performance. Again, we are going to discuss this a little further in the next chapter.

If you are developing an iOS app, use iOS Simulator to find major problems during design and early testing.

You can configure Xcode to help you focus on debugging tasks. For example, when your code hits a breakpoint, you can make Xcode automatically play an alert sound and create a window tab named **Debug**, where Xcode displays the debug area, the debug navigator, and your code at the breakpoint.

Xcode lets you step through your code line by line to view your program's state at a particular stage of execution, which is a fantastically awesome feature that can really help you pick out problem code.

You can also use the debug area to control the execution of your code, view program variables and registers, view its console output, and interact with the debugger. In addition to this, you can also use the debug area to navigate to the OpenGL calls that render a frame and to view the rendering-state information at a particular call. The following image breaks down the user interface, and what everything does. Don't worry, I'll explain everything following the image:



You can suspend the execution of your app by clicking on the pause button (which toggles between "to pause" and "to continue") in the debug area toolbar.

To set a breakpoint, open a source code file and click on the **gutter** next to the line where you want the execution to pause. A blue arrow in the **gutter** (again) indicates the breakpoint.

When your app is paused, the line of code that is currently being executed will be highlighted in green.

You can step through the execution of your code using the **Step Over**, **Step Into**, and **Step Out** buttons located in the bar at the top of the **Debug** area. Clicking on **Step Over** will execute the current line of code, including any methods.

If the current line of code calls a method, step into starts execution at the current line, and then stops when it reaches the first line of the called method. The **Step Out** button executes the rest of the current method or function.

When the execution pauses, the debug navigator opens to display a stack trace (a report of the active stack frames at a certain point in time during the app's execution). Select an item in the debug navigator to view information about the item in the editor area and the debug area. As you continue to debug, expand or collapse threads to show or hide stack frames.

Hover over any variable in the source code editor to see a data tip displaying the value for the variable. Click on the inspector icon next to the variable to print the Objective-C description of the object to the debug area console and to display that description in an additional popup.

Click on the **Quick Look** icon to see a graphical display of the variable's contents. You can implement a custom Quick Look display for your own objects.

When you build and run an OpenGL ES application on a connected device, the debug area toolbar includes a **Frame Capture** button. Click on this button to capture a frame. You can use OpenGL ES frame capture to do the following:

- Inspect the OpenGL ES state information
- Introspect OpenGL ES objects such as view textures and shaders
- Step through the state calls (current debug state information) that precede each draw call and watch the changes with each call
- Step through draw calls (current state of rendering of the app) to see exactly how the image is constructed
- See which objects are used by each draw call by checking in the assistant editor
- Edit shaders (how an image is rendered, such as color information, bloom, lens flares, and the like of special effects) to see the effect upon your application

The debug navigator on the left shows parts of the rendering tree, and the main debug view shows the color and depth sources for the rendered frame as well as other image sources.

Click on the **Debug View Hierarchy** button in the bar at the top of the debug area to inspect a 3D rendering of the view hierarchy of your paused app. You can do the following:

- Rotate the rendering by clicking and dragging in the canvas. What's the canvas? That is where you see the app rendered in Xcode.
- Increase or decrease the spacing between the view layers using the slider on the lower left.
- Change the range of visible views using the double ended slider on the lower right. Move the left handle to change the bottom-most visible view. Move the right handle to change the top-most visible view to get the desired view. After all, there's a lot more going on in the scene that certain views won't show!
- Reveal any clipped content of the selected view by clicking on the **Show clipped content** button.
- You can also reveal any clipped content of the selected view by clicking on the **Show Clipped Content** button. You can do this if your view is clipped due to smaller screen sizes.
- Increase and decrease the magnification using the **Zoom In (+)** and **Zoom Out (-)** buttons.

The iOS Simulator helps you find major problems in your app during design and early testing as we've discussed throughout this book, as it is exactly that, a simulator. Not everyone can afford every new Apple device!

As I sit in front of my 27" iMac, 13" MacBook Pro, texting from my 5S, charging my iPad, currently thinking about buying the Apple watch and wearing my Apple Store shirt.... hmm... it's no wonder I have no money.

Moving on!

In every simulated environment in the iOS Simulator, the home screen provides access to apps—such as Safari, Contacts, Maps, and Passbook—that are included with iOS on the device.

You can perform an initial testing of your app's interaction with these apps in iOS Simulator. For example, just like we did, if you are testing a game, use iOS Simulator to test that the game uses Game Center correctly.

The **Accessibility Inspector** in iOS Simulator helps you test the usability of your app regardless of a person's limitations or disabilities.

The Accessibility Inspector displays information about each accessible element in your app and enables you to simulate VoiceOver interaction with those elements. To start the Accessibility Inspector, click on the **Home** button on iOS Simulator. Click on **Settings** and go to **General** | **Accessibility**. Slide the **AccessibilityInspector** switch to **On**.

You can test your app's localizations in iOS Simulator by changing the language. In **Settings**, navigate to **General** | **International** | **Language**.

Now, these are obviously some pretty over-and-beyond steps you can take. I don't think every game will need accessibility (or that every developer will include it in their app), but these are good things to keep in mind if you are developing an app that will be seen worldwide and you want everyone to be able to use it.

Although you can test your app's basic behavior in iOS Simulator, the simulator is limited as a test platform for multiple reasons. Why's this? Take the following into consideration:

- The iOS Simulator is an app running on a Mac and it has access to the computer's memory, which is much greater than the memory found on a device.
- The iOS Simulator runs on the Mac CPU rather than the processor of an iOS device.
- iOS Simulator doesn't run all threads that run on devices.
- iOS Simulator can't simulate hardware features, such as the accelerometer, gyroscope, camera, or proximity sensor. *So don't throw your computer to simulate accelerometer controls!* Please don't... don't ask... I promise I didn't do that.

While developing your app, ensure that you run and test it on all of the iOS devices and iOS versions that you intend to support. It's only common sense to do so, as you can adjust performance for each device.

I know it's a lot to keep in mind and probably a lot you won't consider every time you are testing your app or even building a new app. It is good information to keep in the back of your mind in case something ever comes up in your own development.

We did discuss some performance issues, such as the fact that rendering a lot of particles on an iPhone 4S cripples the performance and almost causes the device to completely crash out. That's what we will discuss in the next chapter—how to manage performance and keep battery drainage to a minimum!

Summary

Let's see what we talked about in this chapter. Well, what didn't we talk about?! We discussed all things debugging. We started off talking about installing and testing our app on the iOS simulator. Being as limited as it is, we then moved on to testing on a physical device because hey, it's pretty tough to click on two areas at once as you are trying to test a platformer like the one we've created.

We then discussed setting up TestFlight so that we can get beta testers to help us out in finding some ravenous bugs hidden deep within the confides of our code. Again, a seven-year-old child bashing the screen is bound to find a bug a helluva lot quicker than a 20+ device-respecting developer.

We then discussed the fun of debugging and all the great tips and tricks for debugging. We also covered some things to keep in mind and various ways to test our code line by line as well as slowing animations down so we can see things as they happen.

It's been a lot of reading for this chapter, and we will get further into it in the next one.

Don't worry; we're only two chapters away from the best part: Monetization. Also, we are going to return to creating the game. I didn't forget that!

I'll see you in the next chapter!

6
Making Our Game More Efficient

We have discussed quite a few debugging techniques, and there are a lot of them. Not all of them, however, help us when it comes to efficiency. What do we mean by efficiency? Your app can run great, but that doesn't mean it's going to cause the device to run great. Overheating, extreme battery drain, and other such things can ruin the gameplay experience.

With that in mind, let's get an idea of what we will cover in this chapter:

- Optimizing our game
- Preventing extreme battery drain
- Preventing lag (or at least minimizing it)

This is a critical step in your game development, as you don't want people to complain and delete your app because it's too laggy (or causes the app to stutter because the device is having issues rendering the current scene) or it kills the battery.

I know you may have never experienced such an issue, but it happens more often than not.

What would happen if you had downloaded a game that was extremely laggy and killed your battery after an hour of gameplay? You'd, no doubt, delete it off your phone or request a refund.

So, let's dive into how we can optimize things!

Managing effects

As we've been programming our game so far, we have been haphazardly creating without thinking about how our effects will run on older devices.

Have no fear! Because we can detect which device the player is using and adjust our particle effects and the number of enemies that will spawn on the screen at a time.

To begin, in our `GameLevelScene.m` file, we will add the following line of code at the top:

```
#import <sys/utsname.h>
```

This framework will allow us access to detecting the exact device we are working on.

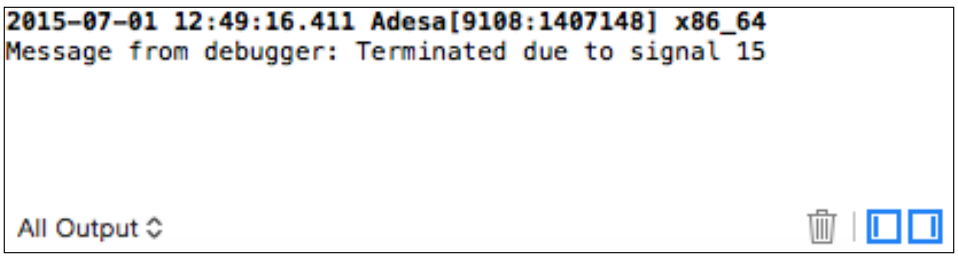
Just under the `@implementation` line, add the following function:

```
NSString* deviceName()
{
    struct utsname systemInfo;
    uname(&systemInfo);

    return [NSString stringWithCString:systemInfo.machine
                              encoding:NSUTF8StringEncoding];
}
```

Now, if you want to see which device appears, you can add the `deviceName()` function to the `NSLog` function in our init section, which would look like this:

```
NSLog(deviceName());
```

So, your log will now display the device on which you are running, as shown in the following figure:



Not sure what **x86_64** means? In this case, because I am running off the simulator, **x86_64** will be displayed on a 64-bit iMac, or if you are running a 32-bit machine (for older Macs), you will see **i386**.

Here's a breakdown of what will appear for each device:

- **i386** on 32-bit Machine
- **x86_64** on 64-bit Machine
- **iPod1,1** on iPod Touch
- **iPod2,1** on iPod Touch Second Generation
- **iPod3,1** on iPod Touch Third Generation
- **iPod4,1** on iPod Touch Fourth Generation
- **iPhone1,1** on iPhone
- **iPhone1,2** on iPhone 3G
- **iPhone2,1** on iPhone 3GS
- **iPad1,1** on iPad
- **iPad2,1** on iPad 2
- **iPad3,1** on 3rd Generation iPad
- **iPhone3,1** on iPhone 4 (GSM)
- **iPhone3,3** on iPhone 4 (CDMA/Verizon/Sprint)
- **iPhone4,1** on iPhone 4s
- **iPhone5,1** on iPhone 5 (model A1428, AT&T/Canada)
- **iPhone5,2** on iPhone 5 (model A1429, everything else)
- **iPad3,4** on 4th Generation iPad
- **iPad2,5** on iPad Mini
- **iPhone5,3** on iPhone 5c (model A1456, A1532 | GSM)
- **iPhone5,4** on iPhone 5c (model A1507, A1516, A1526 (China), A1529 | Global)
- **iPhone6,1** on iPhone 5s (model A1433, A1533 | GSM)
- **iPhone6,2** on iPhone 5s (model A1457, A1518, A1528 (China), A1530 | Global)
- **iPad4,1** on 5th Generation iPad (iPad Air) - Wifi
- **iPad4,2** on 5th Generation iPad (iPad Air) - Cellular
- **iPad4,4** on 2nd Generation iPad Mini - Wifi
- **iPad4,5** on 2nd Generation iPad Mini - Cellular
- **iPhone7,1** on iPhone 6 Plus
- **iPhone7,2** on iPhone 6

Pretty extensive, isn't it? However, it will help when it comes to developing for all devices.

Now, we can begin setting up the functions to change our particle emission according to the device we are running. For example, as we saw in the previous screenshot, we are running **x86_64**, or in other words, a Mac computer. So, to test this function (while running our game via the simulator), in our `GameLevelScene.m` file, scroll down to the `- (void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event` function and add this block of code just before the `[self addChild:rainEmitter];` function where we create our rain emitter:

```
 if ([deviceName()  isEqual: @"x86_64"])
{
  rainEmitter.particleBirthRate = 150;
}
```

You can now play with the birthrate number to see what looks good and performs well. If you adjust the birthrate and test your project, you will see the particles adjust accordingly, as shown in the following screenshot:

For example, in the preceding image, the birth rate is set to `15,000`. While it does look like torrential downpours, the frame rate suffers considerably. Keep in mind that when I take a screenshot, the framerate does plummet quite a bit, but you get the idea that it was averaging about 25fps, because there was such an immense amount for the device to render.

Now if we drop it down to `1500`, we see quite a different result:

It doesn't look as torrential as it did previously, but we see a major increase in the framerate.

Don't want to crank down the rain? Let's try adjusting the fire effects on the screen in the same way that we reduced the rain by adding the following code:

```
if ([deviceName()  isEqual: @"x86_64"])
{
  fireEmitter.particleBirthRate = 100;
}
```

This drops the original birthrate that was set to `200` down to `100`. While a difference of 100 may seem like a huge number, and our particles won't look as good; it's just enough to reduce the lag and not affect the visual appeal of the flames themselves.

Nice! A solid 60 fps *while* taking the screenshot! It looks pretty good too!

Now, let's keep in mind that this is set for the iOS simulator on our Mac, so you can even expect the app to run a little differently, but it does give you a great idea of how it will perform.

Hoping that we have already decided which devices we will support on our game, we will now add the functions to test the supported devices. For this example, I will only support the iPhone 4S and up, including the iPad 4th generation and up.

The code that would be added to test these devices is as follows:

```
if ([deviceName()  isEqual: @"x86_64"]) {//ios sim
                rainEmitter.particleBirthRate = 1500;
            }
            else if ([deviceName()  isEqual: @"iPhone4,1"]) {//4s
                rainEmitter.particleBirthRate = 1000;
            }
            else if ([deviceName()  isEqual: @"iPad3,4"]) {//4th gen
ipad
                rainEmitter.particleBirthRate = 1000;
            }
```

```
            else if ([deviceName()  isEqual: @"iPad2,5"]) {//ipad mini
                rainEmitter.particleBirthRate = 1000;
            }
            else if ([deviceName()  isEqual: @"iPad4,1"]) {//ipad air
                rainEmitter.particleBirthRate = 1200;
            }
            else if ([deviceName()  isEqual: @"iPad4,2"]) {//ipad air
cellular
                rainEmitter.particleBirthRate = 1200;
            }
            else if ([deviceName()  isEqual: @"iPad4,4"]) {//ipad mini
w. retina
                rainEmitter.particleBirthRate = 1300;
            }
            else if ([deviceName()  isEqual: @"iPad4,5"]) {//ipad mini
w. retina cellular
                rainEmitter.particleBirthRate = 1300;
            }
            else if ([deviceName()  isEqual: @"iPhone5,1"]) {//iphone
5 at&t/canada
                rainEmitter.particleBirthRate = 1200;
            }
            else if ([deviceName()  isEqual: @"iPhone5,2"]) {//iphone
5 world-wide
                rainEmitter.particleBirthRate = 1200;
            }
            else if ([deviceName()  isEqual: @"iPhone5,3"]) {//iphone
5c gsm
                rainEmitter.particleBirthRate = 1300;
            }
            else if ([deviceName()  isEqual: @"iPhone5,4"]) {//iphone
5c china/global
                rainEmitter.particleBirthRate = 1300;
            }

            else if ([deviceName()  isEqual: @"iPhone6,1"]) {//iphone
5s gsm
                rainEmitter.particleBirthRate = 1500;
            }
            else if ([deviceName()  isEqual: @"iPhone6,2"]) {//iphone
5s china/global
                rainEmitter.particleBirthRate = 1500;
            }
            else if ([deviceName()  isEqual: @"iPhone7,1"]) {//iphone
6 plus
```

```
                rainEmitter.particleBirthRate = 2500;
            }
             else if ([deviceName()  isEqual: @"iPhone7,1"]) {//iphone
6
                rainEmitter.particleBirthRate = 2500;
            }
```

I know it's a lot to have to write and test as well. These numbers were all guesstimated, as I don't own all these devices. However, I did run test them on each simulator and all of them seemed to run at a solid framerate.

We begin with an `if` statement to check which device we are running. If we do not detect the device checked, we use an `else if` statement. This will cause the code to check the next line for the device. When it finds the running device, it will adjust the birth rate accordingly.

You can do the same when it comes to the fire effects or any other effect that you have running in your game.

You can also limit the amount of enemies being spawned simply by changing your update timer function, as shown in the following code:

```
- (void)updateWithTimeSinceLastUpdate:(CFTimeInterval)timeSinceLast {

    self.lastSpawnTimeInterval += timeSinceLast;
    if (self.lastSpawnTimeInterval > 2) {
        self.lastSpawnTimeInterval = 0;
        [self addSquiggy];

    }
}
```

Simply adjusting the `if (self.lastSpawnTimeInterval > 2)` statement to a higher number will reduce the amount of enemies being spawned at a time, as we are increasing the time we will spawn a new enemy. So, the higher the interval, the longer it takes to spawn enemies.

By reducing these simple things, we will greatly improve the performance of the device, especially older ones that are getting tired and are unable to perform as they used to.

Remember, better performance (the device not running so hard) equals extended battery life, extended play time, and happier customers.

There are a plethora of other ways in which we can optimize things; for example, as we mentioned earlier in this book, we can look at the sprites we created, and if they are large files, we can try saving other formats to save space and make it easier for the device to render them.

You've come here to master iOS development, and you want to learn everything! So, let me explain everything.

The first thing we will discuss is how to prevent, or at least how to limit the amount our app runs in the background.

Battery management – doing less in the background

When the user isn't actively using your game/app, the device will place it into a background state. The device may eventually suspend the app if it's not performing important tasks, such as finishing a task the user initiated (for example, sending photos or updating a newsfeed) or running in a specially declared background execution mode.

To save further battery life, your app shouldn't wait to be suspended by the device. It should begin pausing activity immediately once the app has notified that the state has changed (refer to the following image).

When your app completes any queued tasks, it should notify the device that the background activity is complete. Failing to do so causes the app to remain active and draw energy unnecessarily, unnecessarily, as can be summed up perfectly in the following image:.

Why don't the batteries last longer? It plagues us all...Well, yes, they are improving the battery usage with each new iPhone released, but they are adding more things in the OS that drains the battery, so you're not really getting anywhere.

The *Common Causes of Energy Wasted by Background Apps* section of *The Energy Efficiency Guide for iOS App* states:

> *Apps performing unnecessary background activity (like music, Facebook or news apps constantly working in the background) waste energy. The following are the common causes of wasted energy in background refreshing apps:*

- Not notifying the system when background activity is complete
- Playing silent audio
- Performing location updates
- Interacting with Bluetooth accessories

These are all things, regardless of whether you are creating a game or an app, that you should keep in mind when developing:

- **Notifying the system when background activity is complete**: For apps, if you are pulling information from a website, you will need to either pause it until the app resumes or set a refresh interval.
- **Playing silent audio**: For our games, again, when the app enters the background, the sounds and music need to be paused.
- **Performing location updates**: Have you noticed that annoying arrow icon on the status bar? That equals battery drainage. When the app enters the background, turn location services off.
- **Interacting with Bluetooth accessories**: This applies for apps and games, especially for something as intensive as Bluetooth speakers or even a Bluetooth keyboard or gamepad. Ensure that Bluetooth disconnects and goes into a rest mode when the app goes into the background. Don't forget to resume once the app reenters the foreground!

Firstly, you can to scan for peripherals, such as bluetooth controllers, or other devices only when needed. To do this, use the following block of code:

```
-(void)scanForDevice {

myCentralManager = [[CBCentralManager alloc] initWithDelegate:self
queue:nil options:nil]; //this creates the Bluetooth Manager

[myCentralManager scanForPeripheralsWithServices:nil options:nil];
//this will scan for a broadcasting device
```

```
}

- (void)centralManager:(CBCentralManager *)central
     didDiscoverPeripheral:(CBPeripheral *)peripheral
     advertisementData:(NSDictionary *)advertisementData
     RSSI:(NSNumber *)RSSI {
// Connect to the newly discovered device

      // Stop scanning for devices
      [myCentralManager stopScan];
    }
```

Pretty easy stuff, eh? I know, this is a lot to take in. Trust me, though, this is all super important stuff.

Then, when you are finished with the Bluetooth device, simply use the following two methods to disconnect:

```
//This will Unsubscribe you from a characteristic value
[peripheral setNotifyValue:NO forCharacteristic:interestingCharacteri
stic];

// Disconnect from the device
[myCentralManager cancelPeripheralConnection:peripheral];
```

How do you suspend activity when your app becomes inactive or moves to the background? Implementing the `UIApplicationDelegate` methods in your app delegate (if it hasn't been already, since our app already has this method in the `AppDelegate.m` file) will allow the device to receive notifications and suspend activity when your app becomes inactive or transitions from the foreground to the background.

The `applicationWillResignActive` method is called when the app enters an inactive state, such as when a telemarketer calls you to clean your ducts, a text comes in, or the player switches to another app and your app begins the transition to a background state.

This is where you want to pause any activity, save data, and prepare for any suspension:

```
- (void)applicationWillResignActive:(UIApplication *)application {

}
```

Now, one big thing to keep in mind, is you don't want to rely strictly on saving data when the application enters the background. You always want to save data at proper points during the gameplay, such as at the end of the level. If you rely strictly on saving data during a state change, such as a level change, or the player pausing or exiting the game, the game won't save as reliably and the poor player could lose some information. For example, let's say you're playing your favourite game, but the game only saves when you finish 3 levels, you could lose important high score, or collected items. Plus the player wouldn't want to have to replay those completed levels.

Another state, `applicationDidEnterBackground:` is called as soon as your app enters the background. With this method, you can stop operations, animations, and update methods immediately, using the following code:

```
- (void)applicationDidEnterBackground:(UIApplication *)application {

}
```

This method is only called for a few seconds, so if the app requires more time to complete called requests, you would have to request more execution time using the `beginBackgroundTaskWithExpirationHandler:` method, which is to be called if extra time is required.

When the background tasks are completed, you will be required to call the `endBackgroundTask:` method to let the device know the tasks are finished. If you don't call the method, iOS will allow the app a little more time to complete any additional functions. If any saving or unloading methods don't complete in the additional time iOS provides, the app will be suspended.

To expand, if your app is still wrapping up any methods by the time your app enters the background and the app becomes suspended, all the important performance tweaks, save game, will not occur. Potentially ruining the user's experience.

Now that we have talked about what to do when the app enters the background, it's time to talk about what to do (or how to do it) when the app resumes.

We have two methods that will allow us to call any reconnection methods that we may have disconnected, such as Bluetooth devices, location services, and many more:

```
- (void)applicationWillEnterForeground:(UIApplication *)application {
    // Called as part of the transition from the background to the
inactive state; here you can undo many of the changes made on entering
the background.
}
```

This method is called immediately before the app transitions from the background to active. Start resuming operations, loading data, reinitializing the User Interface, and getting your app ready for the user:

```
- (void)applicationDidBecomeActive:(UIApplication *)application {
    // Restart any tasks that were paused (or not yet started) while
the application was inactive. If the application was previously in the
background, optionally refresh the user interface.
}
```

The `ApplicationDidBecomeActive` function is called immediately after the app becomes an active app after being launched by the device or transitioning from a background or inactive state. Essentially, we are fully resuming any operations that were halted.

These functions are a few great things to keep in mind when developing timers, Bluetooth connections, and so on.

Another thing to keep in mind, is avoiding extreme graphics and animations. As you have seen in our game, we haven't used any intensive animations. Instead, a majority of our characters have two to four images per animation.

If your app uses only the standard windows and controls, such as an app like notes, you don't really need to worry about updates to your content, as the system APIs are created with maximum efficiency in mind.

If you have custom windows and controls, let's say **Shazam** for example, you really have to ensure that the code in which you draw everything is efficient.

For example, you don't want to be refreshing everything on the screen all the time, especially items that are hidden, or even by going overkill on your animations. I know they look good, but they do waste battery life.

Keep in mind that every time an app draws images and text to the screen, it requires CPU usage, GPU usage, and the screen needs to be active. Ergo, the more you are displaying and refreshing, the more the battery will drain, hence why certain apps murder battery life.

So you should be mindful of over updating or even inefficient content drawing, as it will draw significant power from the battery.

Here are some more things to keep in mind. You should always reduce the number of views that are in the app (all of which will be drawing power from the battery).

If you can help it at all, which I know isn't always possible, you should tune down the number of onscreen effects you use, such as opacity, transparency, and so on. If effects are needed, you should avoid using them on items that refresh frequently, as this will again draw battery power because both the opacity objects and the content underneath need to be updated.

In regards to animations, use a maximum of 60fps per animation, anything faster will be troublesome when rendering, as it will require more CPU/GPU to boost the framerate. Also, try to keep all your animation speeds at the same frame rate so the engine isn't struggling to render your character at 60fps and an enemy at 30fps, for example. While having extra frames in the animation may look great, those extra frames are inefficient and will require extra computing, which you guessed it, equals more battery drainage.

It's even recommended to only use SpriteKit, SceneKit, and Metal when developing games, as these frameworks are made to give you the best performance and efficiency.

This doesn't mean that you can't use a framework, such as Cocos2d. Cocos2D is actually built on top of the SpriteKit framework and is just as easy to use and just as powerful.

When testing your app, keep a keen eye set on the debugging tools we discussed in the previous chapter.

Watch out for these signs:

- Battery drain
- Activity when you expect your app to be idle
- An unresponsive or slow user interface

- Large amounts of work on the main thread
- High use of animations
- High use of view opacity
- Swapping
- Memory stalls and cache misses
- Memory warnings
- Lock contention
- Excessive context switches
- Excessive use of timers
- Excessive drawing to screen
- Excessive or repeated small disk I/O
- High-overhead communication, such as network activity with small packets and buffers
- Preventing device sleep

In the previous chapter, we didn't look at measuring the energy usage with Instruments in Xcode.

Instruments provide us with a graphical timeline of the usage on our device. You will be able to gather info about the app's CPU usage, network usage, disk usage, and the graphic usage.

By viewing this timeline, you will be able to analyze the performance of your app, and glean certain points in your app which may be causing lag, slow downs, or potential drops in frame rate so you can make adjustments at those exact moments.

To access these energy diagnostics, launch Instruments, connect it to your app, and then click on the **Energy Diagnostics** template.

The **Energy Diagnostics** template monitors factors that affect energy usage on an iOS device, as mentioned earlier, including the CPU and network activity, screen brightness, and much more. Then, you can see the areas in which the usage is highest, and check whether you can reduce the impact in those areas. For example, you might find an opportunity to defer certain Bluetooth tasks until more energy-efficient times, such as when the device is plugged in or connected to Wi-Fi.

It's all discretionary; as you start to play with the debugging Instruments, you will see what works for you and when is the best time to use certain methods. As you can see by the following image, there are so many excellent tools at your disposal to make your app the most efficient app possible:

With our fantastic game running, click on **Choose a profiling template for** and select the device you are running on (which is the only way you can test battery usage), then select your app.

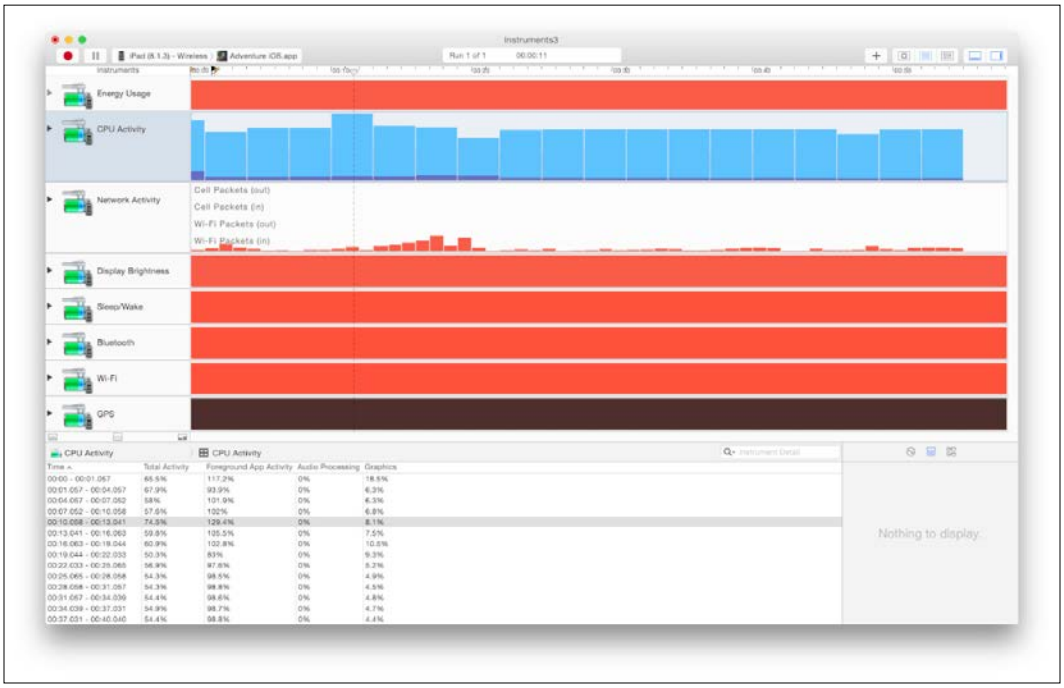
Double-click on the **Energy Diagnostics** profile template. Then, at the top of the new window that pops up, click on the red record button to begin recording our energy usage. Play your game as you normally would. Don't worry, **Instruments** will be recording all your data as you play, so you can focus on looking for bugs while recording your energy usage.

Awesome!

When you have completed testing, simply click on the **Stop** button or press *Command + R* to stop recording.

Now, you will have a complete log of all the data that was recorded. Scroll through and check whether there are any spikes in the log. If so, go back and check the code in those areas where there was a spike to check whether there is an issue causing the spike.

As a side note, the energy usage template shows readings from 0-20 which will indicate how much your app is using the energy at that time:



If, let's say you have some awesome particles flying around, and you are connected to a Bluetooth device during gameplay, you will no doubt see that your energy usage is high all the time. This doesn't mean that your app has an issue. It just means your app requires more power to run everything you've implemented.

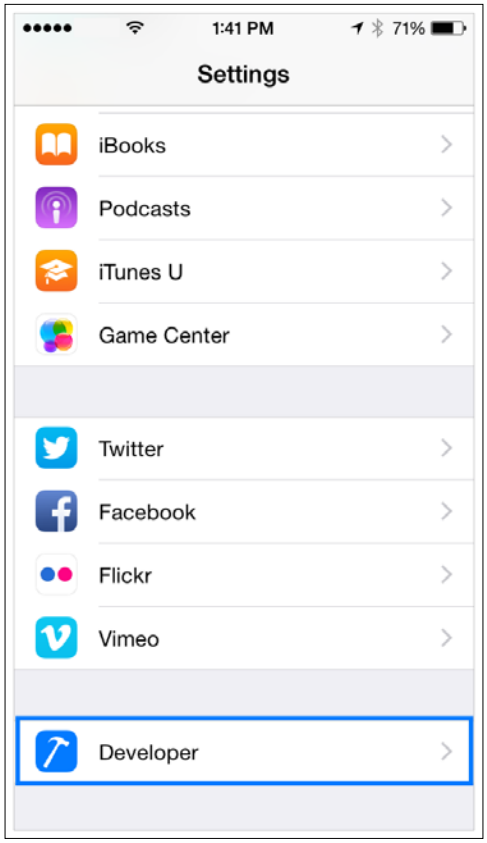
I'll take a certain app I use on a regular basis as an example. The app is from a website loaded with some cool and funny photos. Anyways, when I use this app, my phone heats up within minutes and the battery level starts to plummet. Does this mean something is wrong with the app? No! Especially not when you start to figure what the app is doing as you are using it.

Not only does the app need to load photos from the server, but it uses your Wi-Fi and location services and shows ads—all this while keeping you logged in to their website so that you can make comments on the photos.

Now, this does not equate to the ideal user experience, but if you want access photos, you need to expect this type of performance. So don't worry if your device gets warm or the battery drains a little bit—this is a normal byproduct of device usage.

Don't have access to your computer and need to log energy? Log the usage right on your device!

If you have already connected your device to Xcode, you will now have access to developer options on your device, as shown in the following screenshot:



From this option, you can select **Logging** under the **Instruments** section, then click on **Energy** to enable logging, and then click on the **Start Recording** button to begin logging your energy usage, as shown in the following screenshot:

Again, use your app the same way you normally would, and **Instruments** will log the data for you as your play.

To finish, go back to this setting. You will see that the **Start Recording** button has now changed to **Stop Recording**. Simply click on **Stop Recording**.

Now, back in Xcode Instruments, in the **Energy Diagnostics** option, navigate to **File | Import Logged Data from Device**.

Now, you will be greeted with the same data log as you would if you ran the test directly in Xcode.

Whew! That's a lot to consider!

I know I've been talking a lot, it's all extremely important information.

As I've been testing the app, everything ran just perfectly. In fact, the only issued that seemed to bog the device down at all were the particles! And we managed to fix that programmatically by checking what device we are running and then adjusting our crazy particles accordingly.

Summary

We've discussed a lot in this chapter, a lot of which was best practices on how to manage your device's performance and battery drainage.

You are now well prepared to tackle the testing process and ensure that your device is running as efficiently as possible so that no customers return your app or give you low ratings because of how terribly your app affects their device.

Get excited for the next chapter boys and girls! We will talk about the awesomeness that is deploying your app and monetizing it so that you can make bajillions of dollars! Well, maybe not, but at least enough to pay for your developer cost would be OK.

Get ready because now is the time when it gets fun, and you get to see all your hard work come together.

I'll see you in the next chapter my friends!

7
Deploying and Monetizing

Wow! You have done a lot of work so far! And we're not even done yet! It's about time for your hard work to pay off. In this chapter, we will discuss the following things to keep in mind during this final part of development:

- Preparing your app for deployment
- Monetizing (Making MOAR money!)

All your hard work has come down to this, and now it's time to shine by making the money you deserve for working super hard! Of course there are way more options then simply charging for your app and, you know what, let's just dive right in to it shall we?

Let's begin making money...

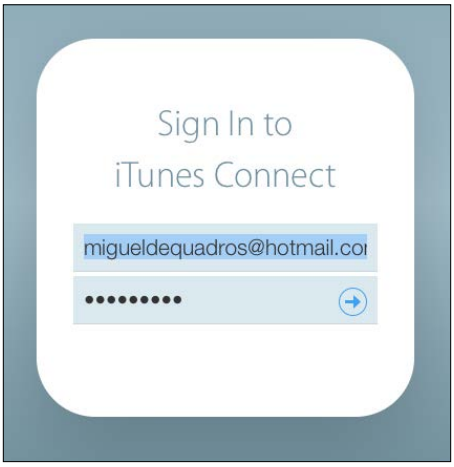
Preparing to deploy

It's finally time! It's finally that time; all your work has come down to this deployment.

For me, this has always been the most exciting time of development, especially when you get the e-mail from Apple saying that your app is in review.

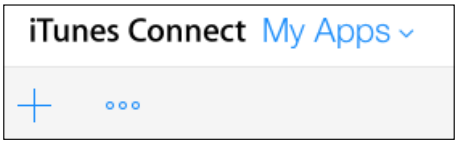
Then, when it gets released, just seeing your app in the AppStore is so exciting! I'll never forget the first app I released, the moment I saw my first app I went around to show it to my whole family. I had finally done it. Now it's your turn.

To begin, we are going to create our app within iTunes Connect. Simply go to `itunesconnect.apple.com` and log in with your Apple ID that is linked to your developer account, as shown in the following screenshot:



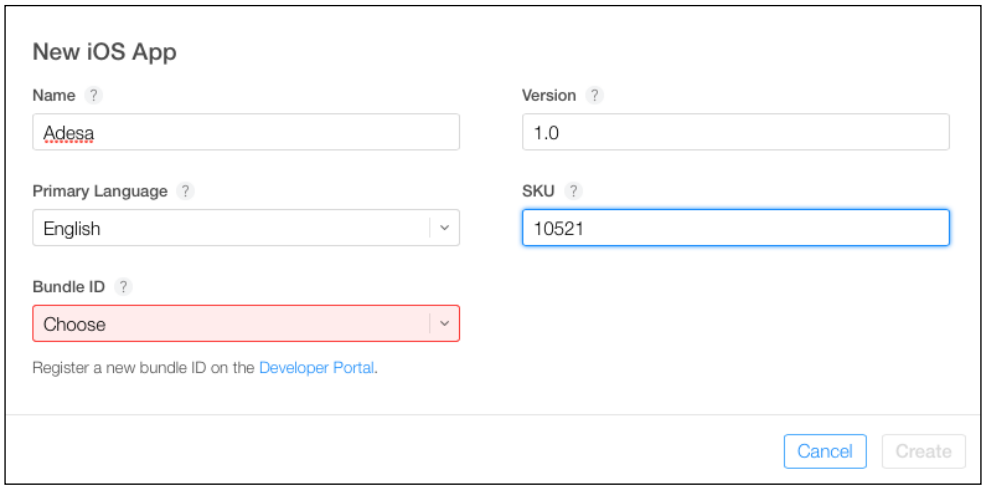
Once you've logged in, click on the **My Apps** icon:

From here, you will now be able to see all the apps you have created (if any):



Just below the **iTunes Connect** logo, simply click on the **+** button to begin creating a new app. You will now see a roll-out menu appear, asking what type of app you would like to create. We will create a new iOS App.

You will then be greeted by a new popup window asking for some basic information for the app, such as the app name, app language, bundle ID (which I will discuss in just a second), version number, and SKU for your app (for the SKU, I literally just use the time I am filling it in, but you can use whatever you like):



Don't have a bundle ID? Simply click on the **Register a new bundle ID on the Developer Portal** button below the **Bundle ID** selection.

From there, it's simply a matter of filling in all the information, such as the App ID description, prefix, suffix, and services you want to include in the app.

Once you have it all filled out, return to the app creation screen and select the new bundle id (if it hasn't appeared yet, simply cancel the creation and try again).

Your app is now created! Well, somewhat...

The next page requires you to fill in the complete information regarding the app—that is, the description, keywords, screenshots, copyright information, and that kind of stuff.

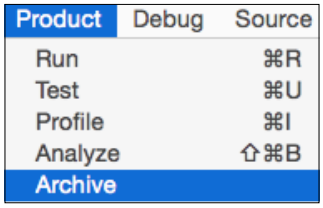
Now, it's time to submit your app to the AppStore for review!

There are two ways of doing this. One is directly through Xcode and the second is through the Application Loader (both app icons can be seen in the following image); we will discuss how to do this.

No matter which method you choose to upload your app, though, both ways will require you to archive it. Before you archive the app, I highly recommend doing another test just to ensure that everything is working as it should.

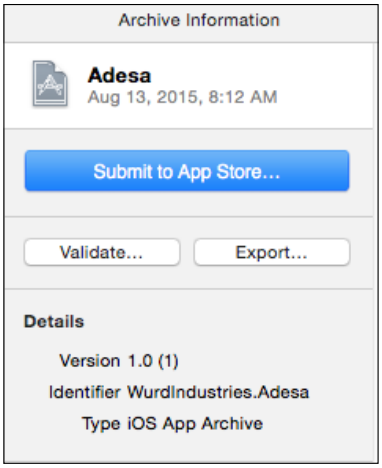
To create the archive in Xcode, ensure that you have **iOS Device** selected under the scheme selection. To recap, schemes are presets on the simulator or iOS device you want to run your project on. The following screenshot shows the **iOS Device** option:

Once you have that selected, simply click on **Product** on the toolbar, then select **Archive**. By archiving, you are essentially putting your whole project into a single, compressed file so that you can upload it to the app store.



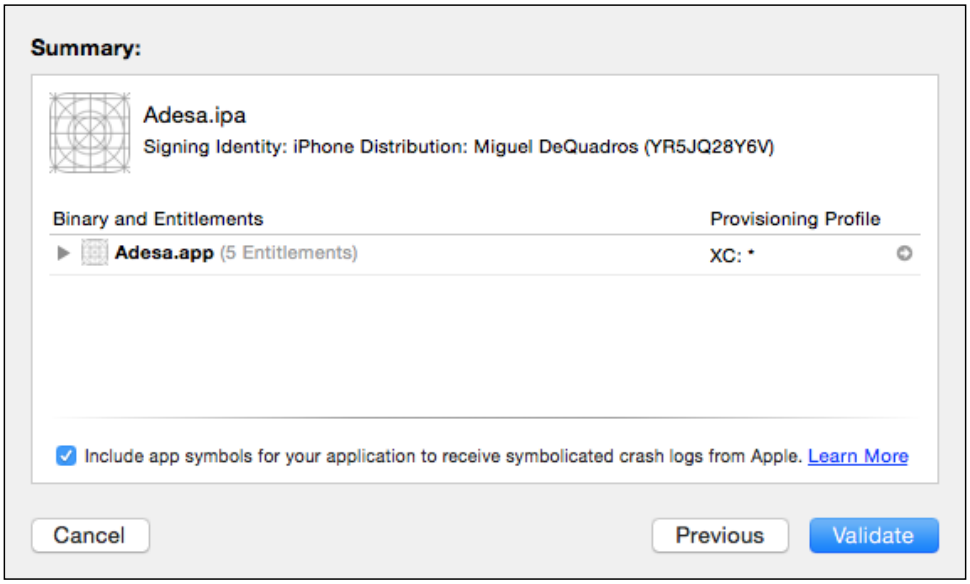
Now, all you have to do is wait... Depending on how large your app is, you may be waiting for a while.

Once it's done, you will encounter the Archive window, which will show you a list of all the archives you have created for your awesome apps:

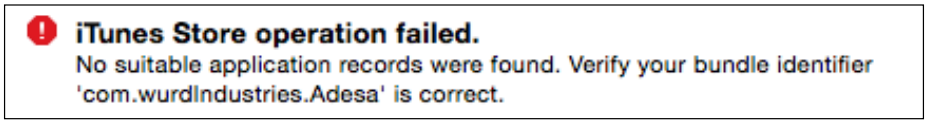


Now, it's a good idea to validate your app through the iTunes validation process.

How do you do this? Simple! Click on the **Validate** button you see in the **Archive** window:



If all goes well, you will get a window saying it's been validated. However, if you're like me, you will get an error, as shown in the following screenshot:



Sure enough, I had made a blooper and wasn't using the correct provisioning profile.

Once that's all done, we can upload it!

Simply click on the big **Submit to App Store** button!

In the dialog that appears, choose a team from the pop-up menu and click on **Choose**.

If need be, Xcode will create a distribution certificate and distribution provisioning profile for you. The name of the distribution provisioning profile begins with the text **XC**.

In the dialog that appears, review the app, its entitlements, and provisioning profile, and click on **Submit**.

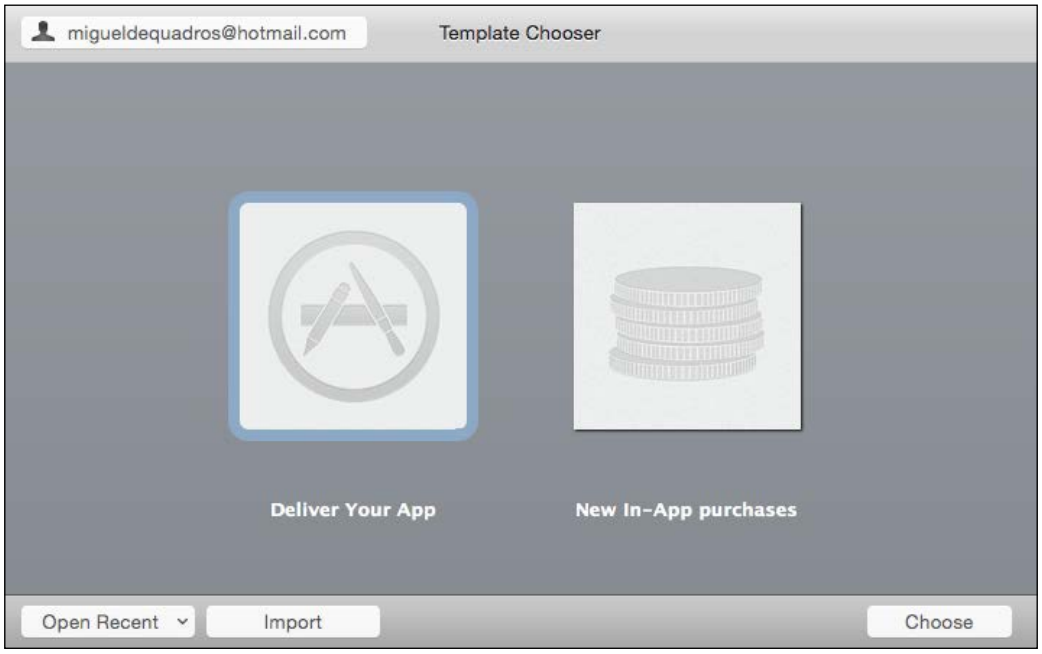
Xcode will then upload the archive to iTunes Connect. If a dialog appears stating that no application record can be found, click on **Done**, create an app record in iTunes Connect, and repeat these steps.

If issues are found, click on **Done** and fix them before continuing.

If no issues are found, click on **Submit** to upload your app.

You have now uploaded your app to the App Store! Await an e-mail saying your app is in review!

This is the easiest way to upload your app, but alternatively you can upload your app via the **ApplicationLoader** option. Simply open the **ApplicationLoader** tab; you will be greeted by the **Template Chooser**, which will allow you to either upload your app or upload an in-app-purchase, as shown in the following screenshot:

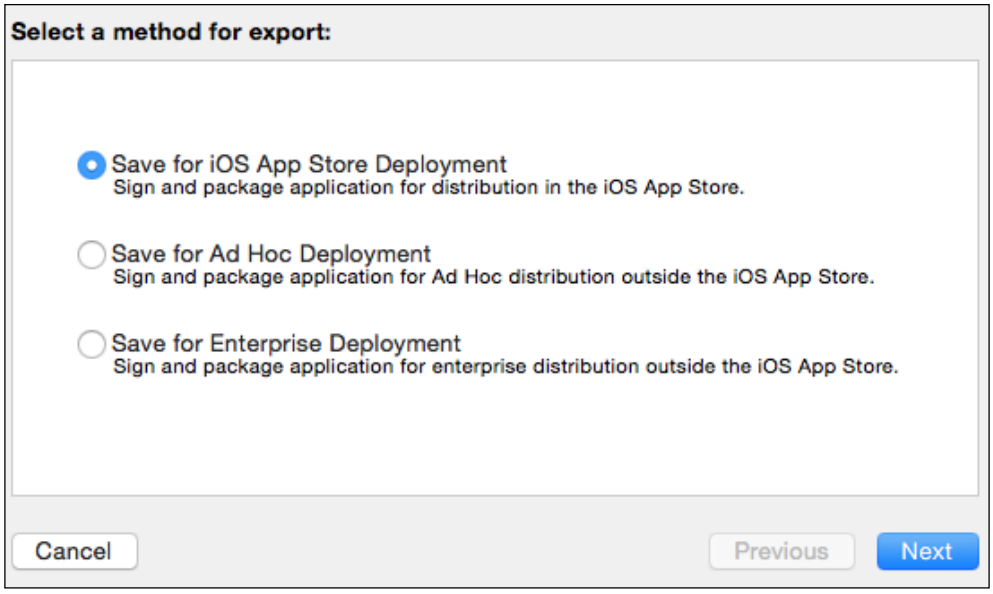


Before we click on **Deliver Your App**, we need to export the project as an archive back in Xcode.

In Xcode, click on **Window** in **Organizer** and we will be back to the Archive of our app. Now, instead of clicking on **Submit to AppStore**, we will click on **Export**.

Next, it will ask how you want to save the app as— **Save for iOS App Store Deployment**, **Save for Ad Hoc Deployment**, or for **Save for Enterprise Deployment**.

Select **Save for iOS AppStore Deployment**, then click on **Next**, as shown in the following screenshot:



Again, it will ask you which development team you would like to use; select the team you are using.

It will again run validation tests, and then ask you where you want to save the archive.

Simply select an easy-to-remember location, then pop back to **Application Loader** and click on **Deliver your App**.

Select the app you just created. Application loader will then search for the application and display a window with all the application details, such as the name, version number, SKU number, primary language, copyright, type, and Apple ID.

Click on **Next**. **Application Loader** begins uploading your application binary file to the App Store.

Once it's done, all you have to do is wait!

You can always log in to iTunes Connect to stay up-to-date on the status of your app.

Now, it's time to make some dough!

Tips for monetizing

Simply charging $0.99 (or a similar conversion for your country) for your app isn't the only way to make money on it. Actually, in my experience, it's one of the worst ways to make money from your app.

In fact, I have made more money with in-app advertising then I ever have with the funds from app purchases.

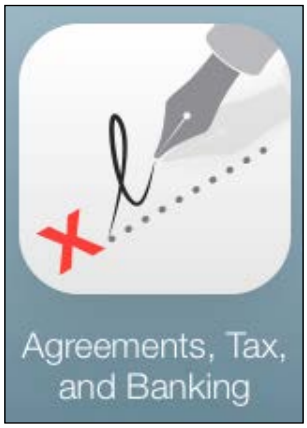
The best thing that you can do is make your app free, then incorporate some kind of advertising outlet such as **iAds**, **Admob**, **Chartboost**, or any other iOS-enabled advertising API.

Let's go through a few of the advertising APIs and how you can sign up for them and integrate them into your game.

iAds

Since you are already signed up as an iOS developer, you don't need to go through any sign-up process. You do, however, need to (if you haven't already) set up the appropriate contracts for iAds.

To do this, log in to `itunesconnect.apple.com` and click on the **Agreements, Tax, and Banking** icon:



On the next screen, you will see **Request Contracts** if you haven't yet set them up. Simply request the **iAd App Network** contract. I recommend that you request all the agreements, but for this section of the chapter we only need iAds.

Once you have clicked on **Request**, you will be asked for legal entity information—in other words, information about the person who has the authority to agree to the contracts. Once done, simply agree and go back to the contracts screen.

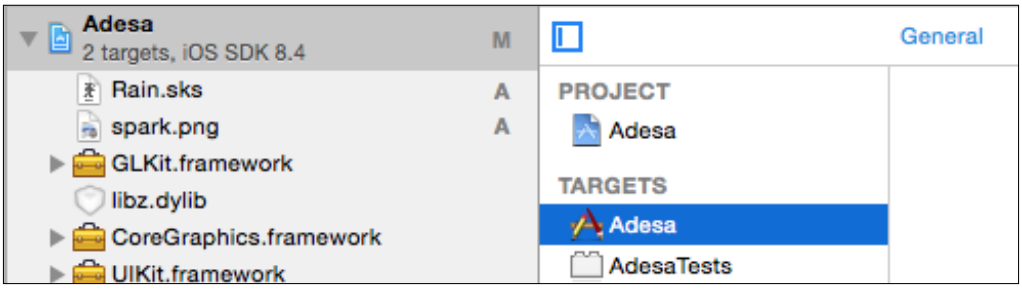
When you're back at the contracts screen, you will be required to fill in your banking information, tax information, and so on, as shown in the following screenshot:

Are you getting excited? I know I am! Here's some exciting information at a glance about iAds:

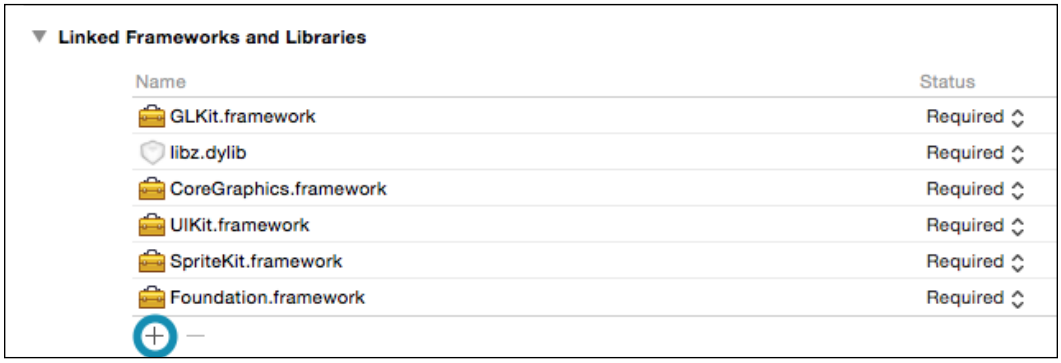
- **Banner views use just a portion of the screen to display a banner ad**: We've all seen them, those tiny little ads at the bottom of a game we are playing or an app we are using.
- **Full screen ads provide larger advertisements to iPad apps**: We can even show a full screen ad! This can come in handy when, let's say, the player beats a level. You can show a fullscreen ad, so it's not exactly disturbing a player's gameplay.
- **You can pause nonessential activities while users interact with advertisements**: Once the user taps on the ad, it will launch into a fullscreen interactive experience for the user. This obscures everything happening underneath, but fortunately we can pause everything going on in the game, so our player's progress isn't being hampered by an ad displaying, and our poor player runs into an enemy. That wouldn't be fun...
- **Canceling advertising negatively impacts your app**: You can programmatically cancel the interactive ad experience and force the user back to your app if your app needs the attention of the user. However, Apple recommends you only do this if absolutely necessary because it may affect your fill rates.

Now we can start integrating!

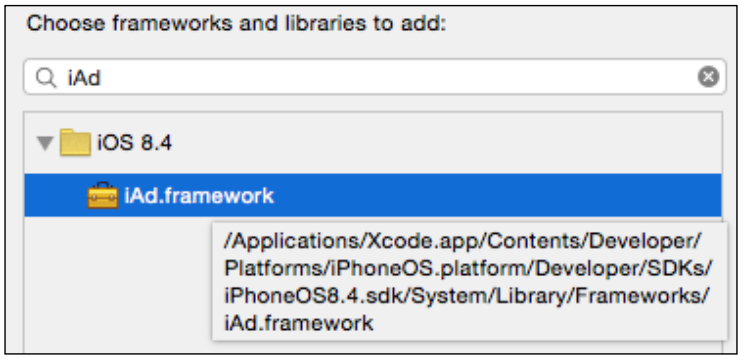
It's really simple, so let's start by adding the iAds framework to our project. Do you remember how to add frameworks? The following screenshot shows how to do this:



Don't worry if you don't remember! In your Xcode, select your project in the sidebar. Then, in the center window, with your **Target** selected (see the previous screenshot), under general, scroll down to the bottom of the window where you see **Linked Frameworks and Libraries**:



Simply click on the **+** button to add a new framework. In the search bar, type the screenshot shows 'iAd':



Next, click on the **Ok** button!

It's that easy! The iAd framework is officially integrated into our project! Now, all that we have to do is insert the code to display the ads.

We will hop in to our **ViewController.h** file and we will adjust the code so that it looks like the following:

```
#import <UIKit/UIKit.h>
#import <SpriteKit/SpriteKit.h>
#import <iAd/iAd.h>

@interface ViewController : UIViewController <ADBannerViewDelegate> {
    ADBannerView *adView;
}

@end
```

To explain, we imported the iAd framework into the `ViewController` class, then we declared the `ViewController` class and `AdBannerViewDelegate` to control iAds.

We then declared the `ADBannerView` variable as `adView`.

Let's go in to our `ViewController.m` file now and change the following methods to display our ads. The bold text is what's new:

```
- (void)viewDidAppear:(BOOL)animated
{
    [super viewDidAppear:animated];
    [[NSNotificationCenter defaultCenter] addObserver:self selector:@
selector(handleNotification:) name:@"hideAd" object:nil];
    [[NSNotificationCenter defaultCenter] addObserver:self selector:@
selector(handleNotification:) name:@"showAd" object:nil];
    // Configure the view.
    SKView * skView = (SKView *)self.view;
    skView.showsFPS = NO;
    skView.showsNodeCount = NO;

    // Create and configure the scene.
    SKScene * scene = [GameLevelScene sceneWithSize:skView.bounds.
size];
    scene.scaleMode = SKSceneScaleModeAspectFill;

    // Present the scene.
    [skView presentScene:scene];
```

```
        self.canDisplayBannerAds = YES;

        adView = [[ADBannerView alloc] initWithFrame:CGRectZero];
        adView.frame = CGRectOffset(adView.frame, 0, 0.0f);
        adView.delegate=self;
        [self.view addSubview:adView];

    }

    - (void)handleNotification:(NSNotification *)notification
    {
        if ([notification.name isEqualToString:@"hideAd"]) {
            adView.hidden = YES;
        }else if ([notification.name isEqualToString:@"showAd"]) {
            adView.hidden = NO;
        }
    }
```

Now, we will go on over to our `GameLevelScene.m` file. Let's scroll down to our `initWithSize` method and add the following code (again the bold text is added):

```
    -(id)initWithSize:(CGSize)size {
        if (self = [super initWithSize:size]) {
            /* Setup your scene here */
            NSLog(deviceName());

            self.userInteractionEnabled = YES;
            self.backgroundColor = [SKColor colorWithRed:.0 green:.0
    blue:.0 alpha:1.0];
            self.physicsWorld.gravity = CGVectorMake(0, 0);
            self.physicsWorld.contactDelegate = self;



            if (_level == 0){
            [[NSNotificationCenter defaultCenter]
    postNotificationName:@"showAd" object:nil];
            SKLabelNode *playLabel = [SKLabelNode labelNodeWithFontNamed:@
    "AvenirNext-Heavy"];
            playLabel.text = @"Adesa";
            playLabel.fontSize = 40;
            playLabel.position = CGPointMake(self.size.width / 2.0, self.
    size.height / 1.7);
```

```
        [self addChild:playLabel];

        SKSpriteNode *playButton = [SKSpriteNode spriteNodeWithImageN
amed:@"play"];
        playButton.position = CGPointMake(self.size.width /2.0 , self.
size.height / 2.5);
        playButton.name = @"playButton";
        [self addChild:playButton];
        }

    }
    return self;
}
```

Then, we will scroll down even further to our `touchesBegan:` method and, within the `if (node.name isEqualToString:@"playButton"])` method, add the following code:

```
[[NSNotificationCenter defaultCenter] postNotificationName:@"hideAd"
object:nil];
```

Confused? No problem!

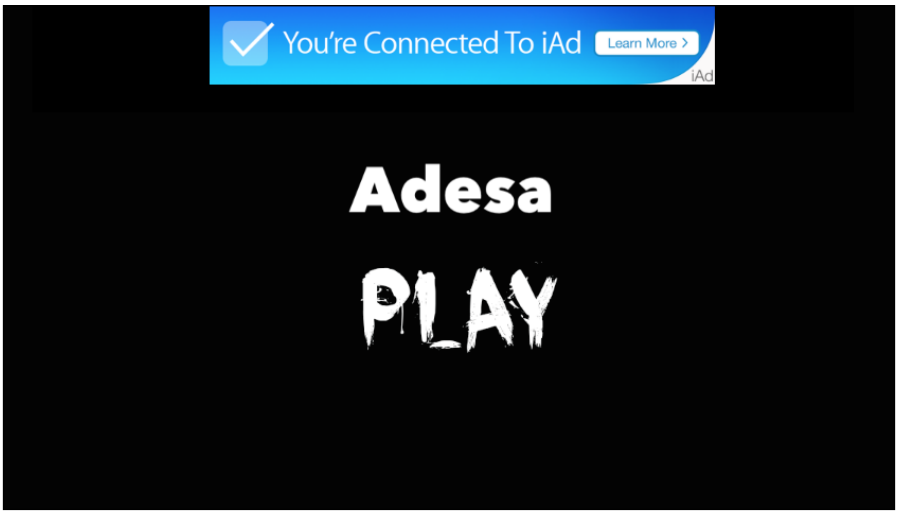
What we did is set up a notification center to accept certain, well, notifications. This is an easy way to call methods between different class files. All we did is set up the notification center in our `ViewController` class to receive a notification to show and hide the banners.

Why do we do it this way?

Banner views are only able to be displayed on a `ViewController` class. In the case of our game, we only have one view controller, whereas our `GameLevelScene` is not a `ViewController` class; it's a SpriteKit scene that is displayed on our `ViewController` class. Ergo, if you were to set up and display the Banner View in our scene, it wouldn't show up because it's a SpriteKit scene, not a `ViewController` class.

Makes sense?

Once you have everything coded, test to see what happens!



Then, when you click on the play button, you will see the following:

All gone!

Keep in mind that no actual ads will appear until the game has been released; until then, it will display placeholder ads.

That's it! You are now well on your way to making ad money!

Now, let's talk about the other advertising outlets.

AdMob

After downloading the framework (which you can find easily on Google), extract it to a place that's easy to locate again.

All we have to do is right-click on our project on the left-hand side bar, and choose **Add Files To (Your Project Name)**:

Locate the `GoogleMobileAds.framework` you just downloaded, and add it.

The AdMob SDK depends on the following iOS development frameworks, which may not already be part of your project:

- AdSupport
- AudioToolbox
- AVFoundation
- CoreGraphics
- CoreMedia
- CoreTelephony
- EventKit
- EventKitUI
- MessageUI
- StoreKit
- SystemConfiguration

Once you have imported these frameworks and referenced the AdMob SDK, Xcode will automatically link to the required frameworks.

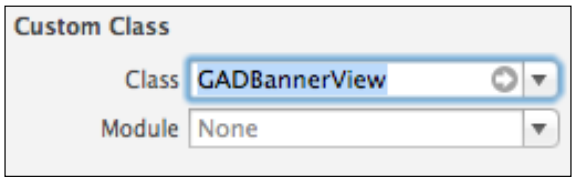
Following Google's own documentation, add the following code in your `ViewController` class to set up ads:

```
@import GoogleMobileAds;

#import "ViewController.h"
```

```
@implementation ViewController

- (void)viewDidLoad {
 [super viewDidLoad];

 NSLog(@"Google Mobile Ads SDK version: %@", [GADRequest sdkVersion]);
}

- (void)didReceiveMemoryWarning {
 [super didReceiveMemoryWarning];
}

@end
```

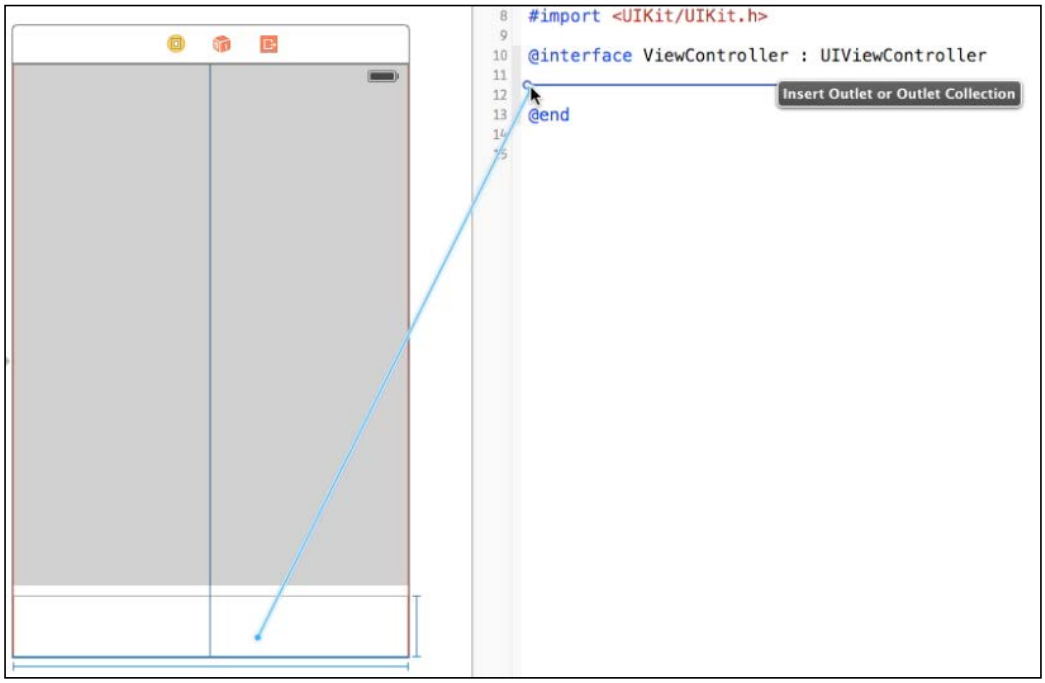
Now that our project has the SDK referenced, let's put banner ads into it.

A **GADBannerView** (**GoogleAd Banner View**) can be created from a storyboard or from code. For regular apps, things are put together in the Storyboard editor but the same methods work for a game such as ours, just remember to keep the code in your `ViewController` class:



Open the main storyboard and, in the `Object` library in the bottom-right corner, search for `UIView` and drag a `UIView` element into your view controller. Then, in the **Identity inspector** button in the top-right corner, we will change the class of this view to a `GADBannerView`, as seen in the previous image:

The `GADBannerView` class needs a reference in code to load ads into it. Open the **Assistant Editor** by navigating to **View** | **Assistant Editor** | **Show Assistant Editor**:



In the assistant editor, ensure that the `ViewController.h` file is displayed. Next, holding the Control key (or just right-click and drag), click on the `GADBannerView` element and drag your cursor over to the `ViewController.h` file.

This is the easiest way to link outlets in your storyboard to code. This has been extremely streamlined and takes a bunch of work out of programming; previously, we would have had to manually type the outlet, then link it manually in the storyboard editor, which wasn't fun. We are going to add the following code, by doing the following, we are going to import the Google ads framework:

```
@import GoogleMobileAds;

@interface ViewController : UIViewController

@property (weak, nonatomic) IBOutlet GADBannerView *bannerView;
```

You then add the following code and make the required adjustments to the methods that are already in your `ViewController.m` file.

Now, to display an ad, just insert the following method in either the `ViewDidLoad` section or wherever you would like it to appear:

```
NSLog(@"Google Mobile Ads SDK version: %@", [GADRequest sdkVersion]);
self.bannerView.adUnitID = @"ca-app-pub-3940256099942544/2934735716";
self.bannerView.rootViewController = self;
[self.bannerView loadRequest:[GADRequest request]];
```

It's (almost) that easy!

There are a few changes that need to be made, though. First, you need to create a new `AppID` variable through the website. All you gotta do is copy and paste the ID in place of the one that is after `self.bannerView.adUnitID =`.

Implementing Chartboost!

Importing the framework will be the same no matter what framework you import. Here are the required frameworks for Chartboost:

- StoreKit
- CoreGraphics
- UIKit
- Foundation

You will have to add only one of these. The other three should be included with a SpriteKit project. For Chartboost, we need to start in our `AppDelegate` class to initialize the app ID and app signature, both of which are set up under Chartboost's website. At the top of the `AppDelegate.m` file, we need to import the following:

```
#import <Chartboost/Chartboost.h>
#import "AppDelegate.h"
#import <CommonCrypto/CommonDigest.h>
#import <AdSupport/AdSupport.h>
```

Then, inside our `Application didFinishLaunchingWithOptions` method, we need to add the following block of code:

```
[Chartboost startWithAppId:@"YOUR_CHARTBOOST_APP_ID"
                appSignature:@"YOUR_CHARTBOOST_APP_SIGNATURE"
                    delegate:self];
```

There is a note, Chartboost must be initialized this way, as they record app bootups to track analytics for you. If you don't do it this way, you won't get any ads. Then you cry. Want to know what's next? The following code will show us the ad:

```
[Chartboost showInterstitial:CBLocationHomeScreen];
```

It's really that easy! Simply add that line of code when you want to display your ad and boom! Done! I can smell the revenue now!

You know, you guys are a great audience, I really hope you've enjoyed and learned a lot from this book.

I really feel you will be able to create your own full-fledged games and monetize them with extreme ease.

But you know what... we aren't done yet. No no! We are actually going to talk about updating your app, and adding multiplayer functionality!

Buckle up, this last chapter is going to be a blast!

It is too dangerous to go alone... Does the following image seem familiar?

Summary

In this chapter, you learned how to deploy your app on the App store and how to make money on your published app by incorporating advertising outlets, such as iAds, Admob, and Chartboost. Now, you can develop and deploy as many app as you want. Enjoy developing, deploying, and earning money on your apps.

8
It's Too Dangerous to Go Alone, Take a Friend!

You've done it! You've officially released your game and it's doing great! But now you want to add more features and push an update, right? Because the same thing over and over is kind of repetitive and we don't want to lose people's interest. This is what we will discuss in this chapter:

- Multiplayer integration
- Game Center integration (we mix multiplayer and Game Center)
- Pushing an update to the AppStore

Now, we can make things super fun for our players because, really, who doesn't like playing with friends? It's going to be a fair amount of work, but hey, it will be totally worth it!

Let's do this...

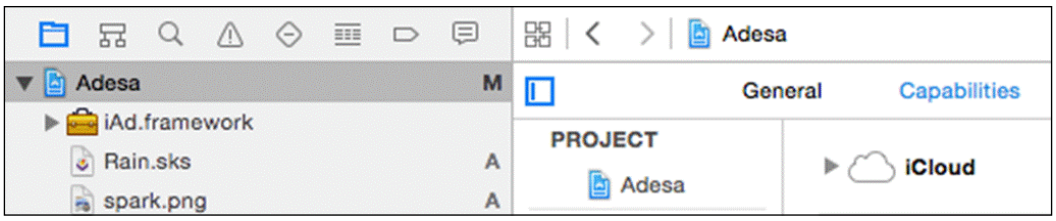
Multiplayer integration

I'm sure all of us remember sitting down in front of our old CRT TV with a friend and loading up *Super Mario 3* (blowing on the cartridge before inserting it to ensure that it works) and playing together for hours on end.

Then, the split screen multiplayer was introduced and that blew everyone's minds. "We can play together at the same time? We don't have to take turns? Awesome sauce!"

Now, we are in the age of Bluetooth/online multiplayer games. No more are we sitting down with friends in the same room to play our games; no, we are antisocial now. That's not an issue though; it does leave a smaller mess to clean up when you're done playing. This is what will integrate—multiplayer with matchmaking—and it will be so cool!

In order to do this, however, we do need to enable and integrate **Game Center** into our game. To start doing this, let's open up our project in Xcode. With our project selected on the left-hand side, click on the **Capabilities** tab in the center of the screen, as shown in the following screenshot:



Then, scroll through the list of capabilities we can add and find **Game Center** (it's usually the third one in this list). Click on the arrow to roll out the options. You will see a button that says **OFF**, click that button to turn on **Game Center**.

We can't run our game and expect **Game Center** to automatically work, no no! For that, we actually have to authenticate the user by logging them in when our app opens up.

In order to ensure that everything is organized within our app, we will create a new group (or folder) within our project and name it `Multiplayer`.

We will then have to create a new class (by navigating to **File | New | File**) and create a new `.h` file as well as a new `.m` file and name them both `MultiplayerHelper`. Then, drag the two files into the `Multiplayer` folder we just created.

We will now replace the code within `MultiplayerHelper.h` with the following code:

```
@import GameKit;

@interface MultiplayerHelper : NSObject

@property (nonatomic, readonly) UIViewController
*authenticationViewController;
@property (nonatomic, readonly) NSError *lastError;

+ (instancetype)sharedGameKitHelper;

@end
```

This imports the `GameKit` API (we will use this to connect the two players together) and then defines two properties—one is a view controller that we will use to display the Game Center authentication and the other is used to keep track of the last error (if any) that occurred while interacting with Game Center.

Now, let's pop on over to our `.m` file and change the code to the following:

```
#import "MultiplayerHelper.h"

@import GameKit;

@implementation MultiplayerHelper {
    BOOL _enableGameCenter;
}

+ (instancetype)sharedMultiplayerHelper
{
    static MultiplayerHelper *sharedMultiplayerHelper;
    static dispatch_once_t onceToken;
    dispatch_once(&onceToken, ^{
        sharedMultiplayerHelper = [[MultiplayerHelper alloc] init];
    });
    return sharedMultiplayerHelper;
}

- (id)init
{
    self = [super init];
    if (self) {
        _enableGameCenter = YES;
    }
    return self;
}

- (void)authenticateLocalPlayer
{

    GKLocalPlayer *localPlayer = [GKLocalPlayer localPlayer];


    localPlayer.authenticateHandler  =
    ^(UIViewController *viewController, NSError *error) {

        [self setLastError:error];
```

```
            if(viewController != nil) {

                [self setAuthenticationViewController:viewController];
            } else if([GKLocalPlayer localPlayer].isAuthenticated) {

                _enableGameCenter = YES;
            } else {

                _enableGameCenter = NO;
            }
        };
    }

    - (void)setAuthenticationViewController:(UIViewController *)
    authenticationViewController
    {
    }

    - (void)setLastError:(NSError *)error
    {
    }

    @end
```

I know this does seem a bit daunting at first, but let's break it all down. At once, we get an instance of the `GKLocalPlayer` class, which represents the currently authenticated player. We then give `GKLocalPlayer` an authentication handler, which the `GameKit` API will call.

We set up a method to store any errors that may appear for easy debugging with the `setLastError:` method.

Next, we check whether the player is logged in to Game Center in either the GC app or anywhere else within your device. If not, the `GameKit` API will attempt to authenticate the user. This is where we display the authentication window (we all know the Game Center login popup window, don't we?). It's ideal for authenticating the user as soon as possible. No one wants to be interrupted with the authentication window midway through gameplay or while typing game settings.

Alternatively, if the player is logged in, the `authenticated` property of the `Game Kit` local player will be set to `true`.

Finally, the last method is just turning Game Center off. Who knows, maybe the player doesn't want to be bothered by it, so all the features get shut off.

As Game Center authentication occurs in the background of the app, the game can authenticate at any time when the app is open, irrespective of whether the player is navigating screens or fighting a boss battle. That's not what we want. In order to counteract this from killing our game, we will use a bit of trickery. Basically, we will get Game Center to create a notification, and the current view that we are on will be responsible for displaying it. So, for example, if the notification is called in the main menu, no problem, we will call it right away. However, if we are mid-level and the notification is called, we will need to display the authentication window at a convenient time, such as when the player pauses or dies.

To do this, we need to define the notification. So, in our `MultiplayerHelper.m` file, we will add the following line directly at the top of the file, just under the `@import GameKit` line:

```
NSString *const PresentAuthenticationViewController = @"present_
authentication_view_controller";
```

Further down in our `setAuthenticationViewController:` method, add the following block of code:

```
if (authenticationViewController != nil) {
        _authenticationViewController = authenticationViewController;
        [[NSNotificationCenter defaultCenter]
         postNotificationName:PresentAuthenticationViewController
         object:self];
    }
```

All that this method does is store and send the notification to the current view controller. Are you following me? I know it's a lot to take in, isn't it?

Let's now scroll down to our `-(void)setLastError:` method and add the following code within the parentheses:

```
_lastError = [error copy];
  if (_lastError) {
    NSLog(@"MultiplayerHelper ERROR: %@",
          [[_lastError userInfo] description]);
  }
```

This will just send a log into the console if anything awry occurs. We set `MultiplayerHelper` first before the error so in the log we will actually see `MultiplayerHelper Error: "Connection Failed"` This way it is much easier to see what error is being thrown. There can be a lot of times a connection fails, by doing this we will be able to tell, not only what the error is, but what is causing the error. This is a good practice to follow; it makes things a heck of a lot easier when debugging issues!

We then need to hop on over to our `MultiplayerHelper.h` file, and we need to add an external linkage method just above the `interface` method:

```
extern NSString *const PresentAuthenticationViewController;
```

This will allow us to access this function through other parts of our project, so we can show the authentication view whenever needed.

Finally, we need to add in a declaration to authenticate the local player just above `@end`:

```
(void)authenticateLocalPlayer;
```

That's all we need to authenticate the player!

For the moment, the content of your `MultiplayerHelper.h` file should look like the following:

```
@import GameKit;
extern NSString *const PresentAuthenticationViewController;
@interface MultiplayerHelper : NSObject

@property (nonatomic, readonly) UIViewController
*authenticationViewController;
@property (nonatomic, readonly) NSError *lastError;


- (void)authenticateLocalPlayer;
+ (instancetype)sharedMultiplayerHelper;

@end
```

Also, the content of your `MultiplayerHelper.m` file should look like the following:

```
#import "MultiplayerHelper.h"

@import GameKit;

NSString *const PresentAuthenticationViewController = @"present_
authentication_view_controller";

@implementation MultiplayerHelper {
    BOOL _enableGameCenter;
}

+ (instancetype)sharedMultiplayerHelper
{
```

```
    static MultiplayerHelper *sharedMultiplayerHelper;
    static dispatch_once_t onceToken;
    dispatch_once(&onceToken, ^{
        sharedMultiplayerHelper = [[MultiplayerHelper alloc] init];
    });
    return sharedMultiplayerHelper;
}

- (id)init
{
    self = [super init];
    if (self) {
        _enableGameCenter = YES;
    }
    return self;
}

- (void)authenticateLocalPlayer
{

    GKLocalPlayer *localPlayer = [GKLocalPlayer localPlayer];


    localPlayer.authenticateHandler  =
    ^(UIViewController *viewController, NSError *error) {

        [self setLastError:error];

        if(viewController != nil) {

            [self setAuthenticationViewController:viewController];
        } else if([GKLocalPlayer localPlayer].isAuthenticated) {

            _enableGameCenter = YES;
        } else {

            _enableGameCenter = NO;
        }
    };
}

- (void)setAuthenticationViewController:(UIViewController *)
authenticationViewController
{
```

```
        if (authenticationViewController != nil) {
            _authenticationViewController = authenticationViewController;
            [[NSNotificationCenter defaultCenter]
             postNotificationName:PresentAuthenticationViewController
             object:self];
        }
    }

    - (void)setLastError:(NSError *)error
    {
        _lastError = [error copy];
        if (_lastError) {
            NSLog(@"MultiplayerHelper ERROR: %@",
                  [[_lastError userInfo] description]);
        }
    }

    @end
```

With that all sorted out (hopefully everything is working perfectly for you), we will hop into our `ViewController.m` file, and we will add the following functions inside the `(void)viewDidAppear:` method:

```
    [[NSNotificationCenter defaultCenter]
       addObserver:self
       selector:@selector(showAuthenticationViewController)
       name:PresentAuthenticationViewController
       object:nil];

    [[MultiplayerHelper sharedMultiplayerHelper]
     authenticateLocalPlayer];
```

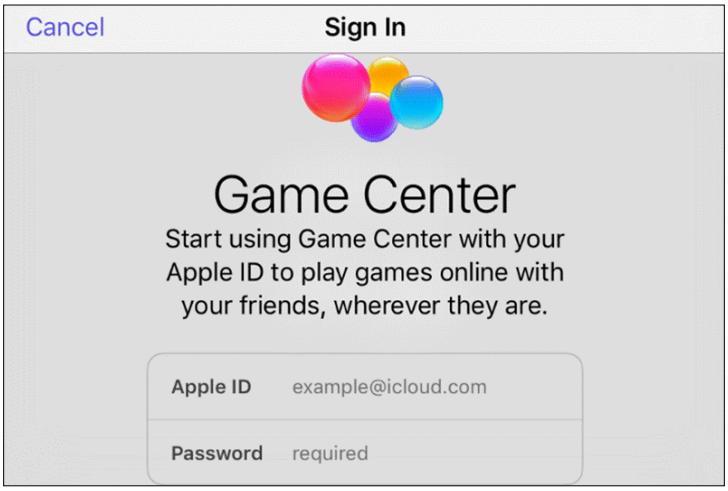
Again, we will use the `NSNotificationCenter` method, which will allow each `ViewController` class to send a notification, in this case to display the authentication view controller, so it can be handled differently depending on where it's being called in the `ViewController` class.

Now, we need to add the method to actually show the authentication view controller. Further in the `ViewController.m` file, add the following function:

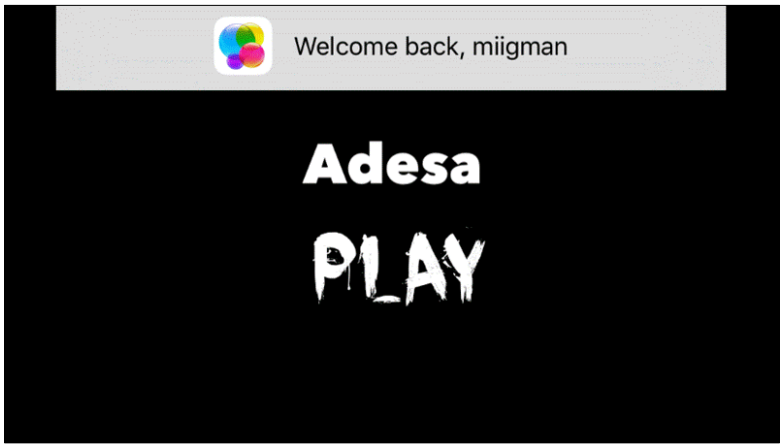
```
- (void)showAuthenticationViewController
{
    MultiplayerHelper *multiplayerHelper =
    [MultiplayerHelper sharedMultiplayerHelper];

    [self presentViewController:
```

```
        multiplayerHelper.authenticationViewController
                                                animated:YES
                                              completion:nil];
}
- (void)dealloc
{
    [[NSNotificationCenter defaultCenter] removeObserver:self];
}
```

When you run and test the game after a few seconds, you will be greeted with the following page:



When you type your credentials, you will be logged in, and then you will see the following page:



We aren't done yet! We still need to search for other players to play with. The great thing about Game Center is that a matchmaking system is built right into the API, so we don't have to do any funky complicated programming or GUI creation.

Let's hop back over to our `MultiPlayerHelper.h` file so that we can make the following changes by adding this block of code after our `@import` line:

```
@protocol MultiPlayerHelperDelegate
-(void)matchStarted;
-(void)matchEnded;
-(void)match:(GKMatch *)match didReceiveData:(NSData *)data
fromPlayer:(NSString *)playerID;
@end
```

We then have to modify our `@interface` line, so we can support the matchmaking protocol we just created:

```
@interface MultiplayerHelper : NSObject<GKMatchmakerViewControllerDele
gate, GKMatchDelegate>
```

We will then add these functions after our `@interface` line:

```
@property (nonatomic, strong) GKMatch *match;
@property (nonatomic, assign) id <MultiPlayerHelperDelegate> delegate;

- (void)findMatchWithMinPlayers:(int)minPlayers maxPlayers:(int)
maxPlayers
                 viewController:(UIViewController *)viewController
                 delegate:(id<MultiPlayerHelperDelegate>)delegate;
```

Whoa, whoa! Settle down there, cowboy! That's some trickery I've never seen before! Let's break it down!

First, we added a new protocol entitled `MultiPlayerHelperDelegate`. This is so we can notify other objects and functions when certain events occur, such as a new game starting or ending.

Next, the `MultiplayerHelper` class defines two new protocols. The first protocol is the `GKMatchmakerViewControllerDelegate` function, which enables the `MultiplayerHelper` class to notify the player when it's found a new match. The second protocol is the `GKMatchDelegate` function, which is for Game Center to notify `MultiplayerHelper` if new data is coming in or if we lose the connection.

Finally, the next action allows the `MultiplayerHelper` class to search for someone to play with.

Easy, right? YES!

We will now hop over to our `GameKitHelper.m` file so that we can add more functions!

The first function has to be added within our `@implementation` line:

```
BOOL _matchStarted;
```

This is a simple `true` or `false` statement that we will call if there is a new match beginning. Now, we will add the following function (I added it just after the `(void) authenticateLocalPlayer` method):

```
- (void)findMatchWithMinPlayers:(int)minPlayers maxPlayers:(int)
maxPlayers
                 viewController:(UIViewController *)viewController
                       delegate:(id<MultiPlayerHelperDelegate>)
delegate {

    if (!_enableGameCenter) return;

    _matchStarted = NO;
    self.match = nil;
    _delegate = delegate;
    [viewController dismissViewControllerAnimated:NO completion:nil];

    GKMatchRequest *request = [[GKMatchRequest alloc] init];
    request.minPlayers = minPlayers;
    request.maxPlayers = maxPlayers;

    GKMatchmakerViewController *mmvc =
    [[GKMatchmakerViewController alloc] initWithMatchRequest:request];
    mmvc.matchmakerDelegate = self;

    [viewController presentViewController:mmvc animated:YES
completion:nil];
}
```

This is the function that will allow a match to be found. We set it up so that, if the player doesn't sign in to Game Center, it will invalidate and do nothing.

After this, we begin the search for a new match. This method allows us to customize the type of match we want, for example, the minimum or maximum players desired in the match. You can create a GUI so that the matchmaker can customize the match himself like in your typical FPS game.

Then, we create a new instance of the Game Kit `MatchMakerViewController` function by setting the delegate to our `MultiplayerHelper` object, and it then pops it into the screen.

Finally, the `MatchMakerViewController` function begins to start searching. It will send out some call-backs, which we will now add. Insert the following directly after that method we just added; the first will be called when the user cancels searching for a friend:

```
- (void)matchmakerViewControllerWasCancelled:(GKMatchmakerViewControll
er *)viewController {
    [viewController dismissViewControllerAnimated:YES completion:nil];
}
```

The next is when the matchmaking process has failed; we then show in the log what happened:

```
- (void)matchmakerViewController:(GKMatchmakerViewController *)
viewController didFailWithError:(NSError *)error {
    [viewController dismissViewControllerAnimated:YES completion:nil];
    NSLog(@"Well that didn't work. Here's why: %@", error.
localizedDescription);
}
```

Now, we've located a match!

```
- (void)matchmakerViewController:(GKMatchmakerViewController *)
viewController didFindMatch:(GKMatch *)match {
    [viewController dismissViewControllerAnimated:YES completion:nil];
    self.match = match;
    match.delegate = self;
    if (!_matchStarted && match.expectedPlayerCount == 0) {
        NSLog(@"Ready to start playing!");
    }
}

#pragma mark GKMatchDelegate
```

The following block of code tells us whether the match has received any data sent from the player:

```
- (void)match:(GKMatch *)match didReceiveData:(NSData *)data
fromPlayer:(NSString *)playerID {
    if (_match != match) return;

    [_delegate match:match didReceiveData:data fromPlayer:playerID];
}
```

This tells us if there's a change in the connection:

```
- (void)match:(GKMatch *)match player:(NSString *)playerID didChangeSt
ate:(GKPlayerConnectionState)state {
    if (_match != match) return;

    switch (state) {
        case GKPlayerStateConnected:

            NSLog(@"Player connected!");

            if (!_matchStarted && match.expectedPlayerCount == 0) {
                NSLog(@"Ready to start match!");
            }

            break;
        case GKPlayerStateDisconnected:

            NSLog(@"Player disconnected!");
            _matchStarted = NO;
            [_delegate matchEnded];
            break;
    }
}
```

This will tell us if the game can't start for some odd reason:

```
- (void)match:(GKMatch *)match connectionWithPlayerFailed:(NSString *)
playerID withError:(NSError *)error {

    if (_match != match) return;

    NSLog(@"Failed to connect to player with error: %@", error.
localizedDescription);
    _matchStarted = NO;
    [_delegate matchEnded];
}
```

Finally, this tells us if there was an error in connecting to any player:

```
- (void)match:(GKMatch *)match didFailWithError:(NSError *)error {

    if (_match != match) return;

    NSLog(@"Match failed with error: %@", error.localizedDescription);
    _matchStarted = NO;
    [_delegate matchEnded];
}
```

We will add another `NSString` class at the top of our `MultiplayerHelper.m` file. Add the following declaration just under the definition of our `present_authentication_view_controller` line:

```
NSString *const LocalPlayerIsAuthenticated = @"local_player_
authenticated";
```

We will then scroll down to our `authenticatedLocalPlayer` method, and we will edit it so that it looks like the following:

```
- (void)authenticateLocalPlayer
{

    GKLocalPlayer *localPlayer = [GKLocalPlayer localPlayer];

    if (localPlayer.isAuthenticated) {
        [[NSNotificationCenter defaultCenter] postNotificationName:Loc
alPlayerIsAuthenticated object:nil];
        return;
    }


    localPlayer.authenticateHandler  =
    ^(UIViewController *viewController, NSError *error) {

        [self setLastError:error];

        if(viewController != nil) {

            [self setAuthenticationViewController:viewController];
        } else if([GKLocalPlayer localPlayer].isAuthenticated) {

            _enableGameCenter = YES;
            [[NSNotificationCenter defaultCenter] postNotificationName
:LocalPlayerIsAuthenticated object:nil];
        } else {

            _enableGameCenter = NO;
        }
    };
}
```

Again, what we did here is create a new notification (the `NSString` class) to be called later on when the player is authenticated, as we will handle the calling of that notification momentarily.

Let's pop back over to our `MultiPlayerHelper.h` file, where we will add the following line of code just below the `@import` line:

```
extern NSString *const LocalPlayerIsAuthenticated;
```

Again, this is an external linkage that we will access later. For the moment, we will change our `ViewController.h` file:

```
#import <UIKit/UIKit.h>
#import <SpriteKit/SpriteKit.h>
#import <iAd/iAd.h>
#import "MultiplayerHelper.h"

@interface ViewController : UIViewController <ADBannerViewDelegate,
MultiPlayerHelperDelegate> {
    ADBannerView *adView;
}

@end
```

We needed to adjust the `ViewController` interface to implement the `MultiPlayerHelperDelegate` method.

We now need to adjust our `ViewController.m` file with the following additions and changes. First, we will add this block to our `(void)viewDidAppear` function:

```
    [[NSNotificationCenter defaultCenter] addObserver:self selector:@
selector(playerAuthenticated)
name:LocalPlayerIsAuthenticated object:nil];
```

The following code shows us how we accept the notification we created earlier to authenticate the player:

```
- (void)playerAuthenticated {
    [[MultiplayerHelper sharedMultiplayerHelper]
findMatchWithMinPlayers:2 maxPlayers:2 viewController:self
delegate:self];
}
```

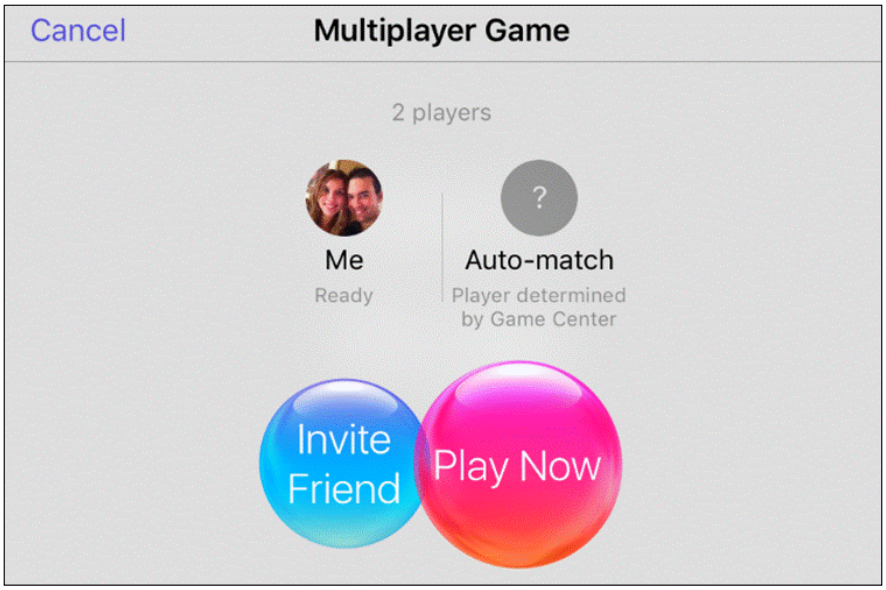
Now, add the following definitions under the `dealloc` function:

```
#pragma mark MultiPlayerHelperDelegate

- (void)matchStarted {
    NSLog(@"Game started");
}

- (void)matchEnded {
    NSLog(@"Game ended");
```

```
}

- (void)match:(GKMatch *)match didReceiveData:(NSData *)data
fromPlayer:(NSString *)playerID {
    NSLog(@"Received data");
}
```

These will simply log each notification that comes through the `MultiPlayerHelperDelegate` method, so when a game begins or ends or when data is received, you will see a log in the console.

Now run it, and you will see the following output:



Awesome!! That part is looking pretty cool, but we still have quite a way to go.

Game Center integration

Matchmaker, matchmaker find me a match! Now's the time to start finding our friends!

Let's pop over to our `MultiplayerHelper.h` file so that we can add in a new dictionary to store the players we find. Just after the `@interface` line, add in the following declaration:

```
@property(nonatomic, strong) NSMutableDictionary *playersDict;
```

This dictionary allows Game Kit to easily look up player data.

Now, we will go over to our `MultiplayerHelper.m` file and make a few new changes. First, we will add a new method just after we authenticate our local player:

```
- (void)lookupPlayers {

    NSLog(@"Looking up %lu players...", (unsigned long)_match.
playerIDs.count);

    [GKPlayer loadPlayersForIdentifiers:_match.playerIDs
withCompletionHandler:^(NSArray *players, NSError *error) {

        if (error != nil) {
            NSLog(@"Error retrieving player info: %@", error.
localizedDescription);
            _matchStarted = NO;
            [_delegate matchEnded];
        } else {

            // fill up that there dictionary
            _playersDict = [NSMutableDictionary
dictionaryWithCapacity:players.count];
            for (GKPlayer *player in players) {
                NSLog(@"Found this person to play with: %@", player.
alias);
                [_playersDict setObject:player forKey:player.
playerID];
            }
            [_playersDict setObject:[GKLocalPlayer localPlayer]
forKey:[GKLocalPlayer localPlayer].playerID];

            // Let me know if the match can start k?
            _matchStarted = YES;
            [_delegate matchStarted];
        }
    }];
}
```

Next, we need to actually call this method, and we will call it in two separate areas. In the first method, we will adjust the following code:

```
- (void)matchmakerViewController:(GKMatchmakerViewController *)
viewController didFindMatch:(GKMatch *)match {
    [viewController dismissViewControllerAnimated:YES completion:nil];
    self.match = match;
```

```
        match.delegate = self;
        if (!_matchStarted && match.expectedPlayerCount == 0) {
            NSLog(@"Ready to play!");
            [self lookupPlayers];
        }
    }
```

We need to adjust yet another method:

```
- (void)match:(GKMatch *)match player:(NSString *)playerID didChangeSt
ate:(GKPlayerConnectionState)state {
    if (_match != match) return;

    switch (state) {
        case GKPlayerStateConnected:
            // handle a new player connection.
            NSLog(@"Player connected!");

            if (!_matchStarted && match.expectedPlayerCount == 0) {
                NSLog(@"LET'S PLAY YA");
                [self lookupPlayers];
            }

            break;
        case GKPlayerStateDisconnected:
            // a player just disconnected.
            NSLog(@"Player disconnected!");
            _matchStarted = NO;
            [_delegate matchEnded];
            break;
    }
}
```

Now, if you were to test our code on two devices, you should get the following in the console:

```
2015-10-26 18:52:13.867 ADESA[787:60b] Ready to start match!
2015-10-26 18:52:13.874 ADESA[787:60b] Looking up 1 players...
2015-10-26 18:52:13.894 ADESA[787:60b] Found player: miigman
2015-10-26 18:52:13.895 ADESA[787:60b] Match has started successfully
```

That's it! Now all that is required to finish our multiplayer integration is to handle the controls between the two connected devices.

Now that we have the multiplayer features all integrated, it's up to you to make both players appear! (You didn't think I would make it that easy for you, did you?)

Most of the work has been completed for you; just keep in mind how to work between multiple classes. Want a hint? Send messages! For example, when you press the jump button, send the `MultiPlayerHelper` method a message to make `player2` move within the `GameScene` class. Oh, and don't forget to set up `player2`.

I know you can do it!

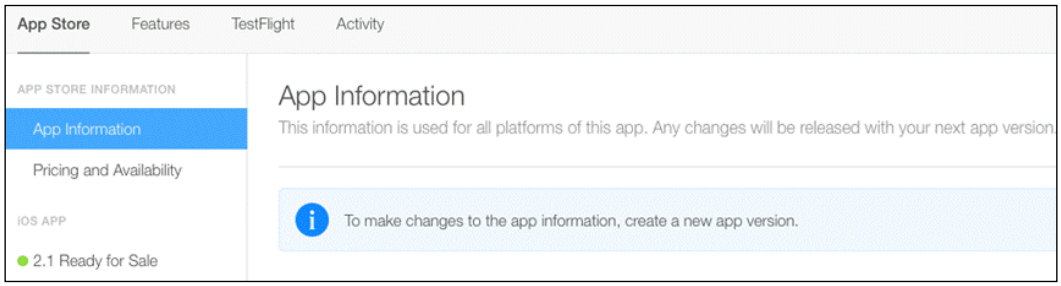
This will be the final challenge for you.

When you're done...

Pushing updates to the AppStore!

Now that you have completed your awesome new multiplayer game, it's time to push the latest version of your game to the AppStore! Do you remember how we pushed our game in the first place? The steps are very similar!

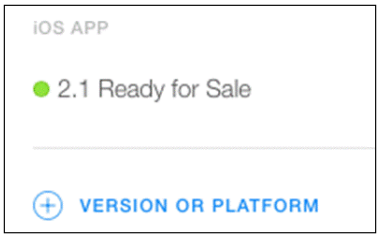
To begin, simply log in to `itunesconnect.apple.com` and, in the **My Apps** section, click on the app you want to update.



Once you have selected the app, you will be greeted with a big blue circle with an **i** in the center asking you to create a new app version if you want to change the app information. Well how the heck do I do that?!

It's easy, try to stay calm.

On the sidebar, simply click on the **+** button where it says **VERSION OR PLATFORM**, as shown in the following screenshot:

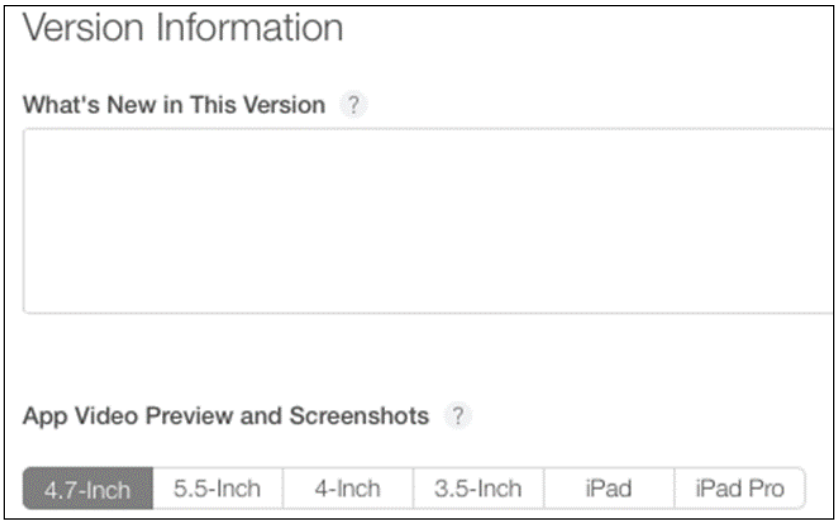


At the time of writing this book, it will ask you whether you want to create a new iOS version or a tvOS version.

For this book (because we didn't cover tvOS).

ITunes connect will then ask you to type in the new version number. When you click on **Done**, it will appear above the current version that says **Ready for Sale**.

Once you click on the new version, you will be asked to fill in new version information and upload new screenshots, as shown in the following screenshot:

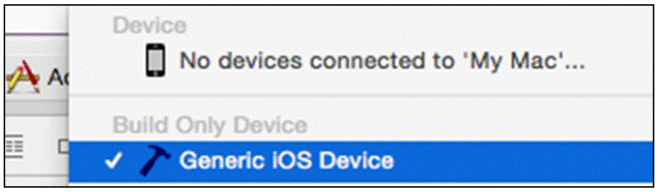


You can then change the description of the app and contact information and select when you want the app to be released. When you have it all filled out and all the new screenshots uploaded, you can exit your browser and open up Xcode again.

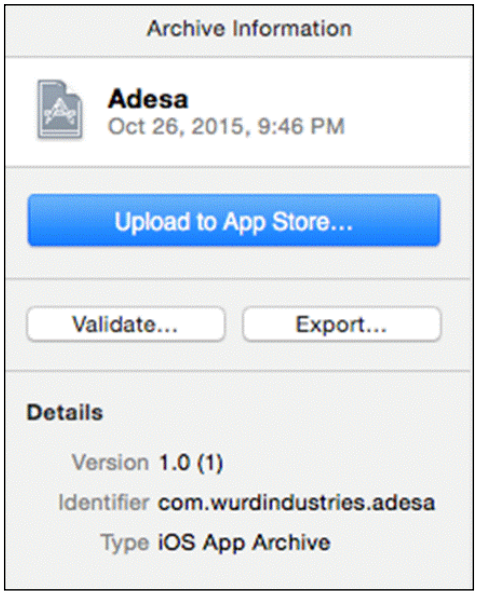
I know, earlier we used Application Loader, but Xcode seems to be a much easier way to do it because you can build and submit directly through it.

Do you remember how we do this? No problem!

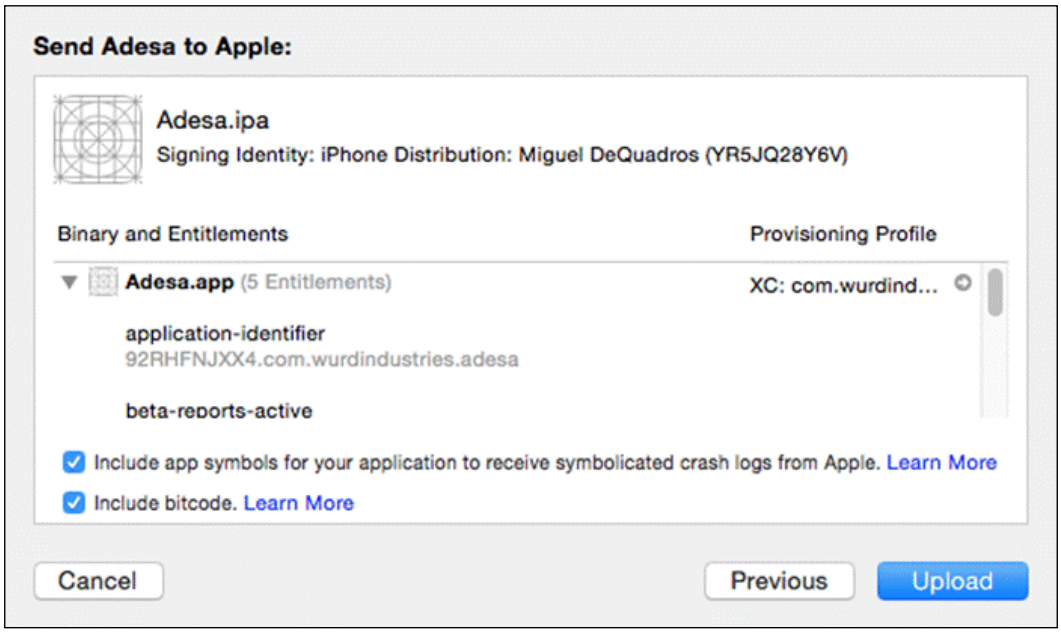
Click on the scheme at the top of our project and ensure that we have a device build selected (otherwise, the archive option will not be available), as shown in the following screenshot:



Once you have a device build selected, navigate to **Product** | **Archive**. Xcode will now build our app and archive it for AppStore submission. Once it's done, it will show you the organizer with all the builds of our app and the other apps we have built in the past. The most recent build will automatically be selected. On the sidebar, simply click on **Upload to App Store**.



Xcode will validate our package, sign it, and then archive it.



Once that's done, simply click on the **Upload** button! Xcode will automatically upload your project file to the AppStore. Don't forget to change the version number in the list before you upload it, or else it will come up with an error after the upload.

Now all you have to do is wait for your awesome new version to be released!

I'm so proud of you guys! You've come such a long way!

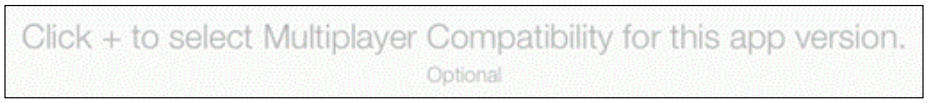
I didn't forget... I just missed it

Did you have any issues when implementing the multiplayer functionality of our game? Did it show you it wasn't recognized by Game Center? Silly me! I forgot to mention a somewhat critical step to follow... Oops! Hey, it happens!

So, if you did get that error when running your app on a device or even in the simulator, it's because your game hasn't been registered in Game Center.

How do you do this?

Log in to `itunesconnect.apple.com` again, go to **My Apps**, and then open up your app. Scroll down and, if you see a gray box with the text **Click + to select Multiplayer Compatibility for this app version** under the **Multiplayer Compatibility** section, then you know where ~~I've, you've, I've~~, you've gone wrong.



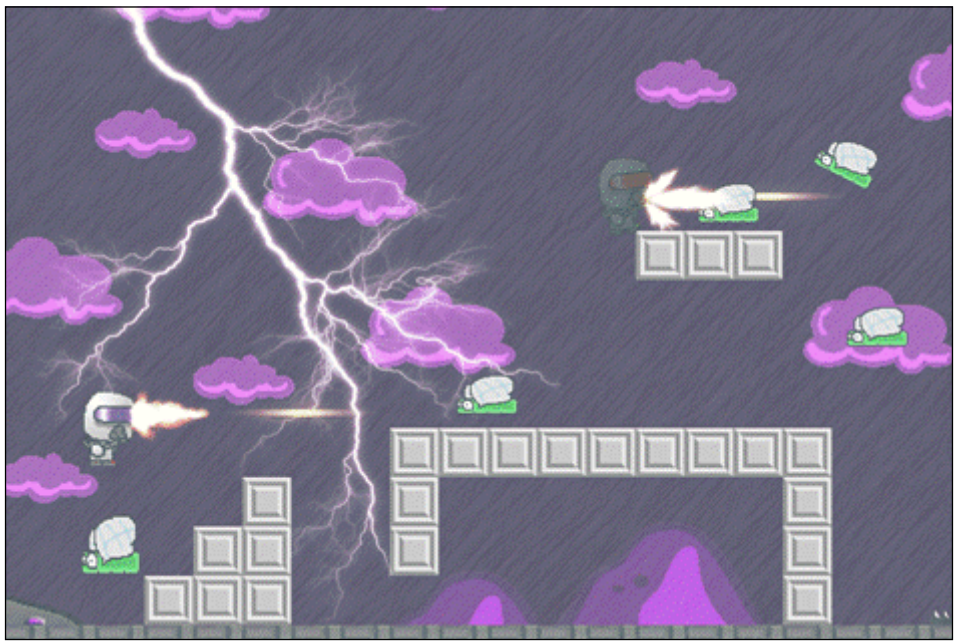
Simply click the **+** button and select your app to add multiplayer compatibility.

You're done!

Guys, I'm so happy you've come all this way with me! I know, at times it may have been confusing or a lot to take in, but you've made it!

I hope you will be able to Master iOS Development with all the knowledge and tips and tricks I have taught you!

Have an awesome game development career, and never give up on your dreams and passions. The following image shows us the final result when the game is all completed.



Summary

We discussed implementing multiplayer integration! We set up GameCenter for our game, and we even got player authentication and match searching all ready to go. It's all up to you to finish the awesome project. We also discussed how to upload an update of our game, so the millions around the world can enjoy your newest update.

Index

T

V

X

Thank you for buying
Mastering iOS Game Development

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.

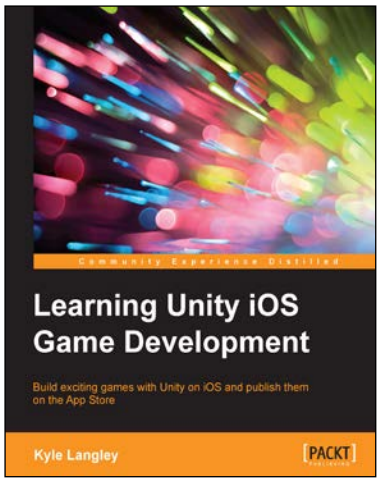
We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.

iOS Game Development By Example

ISBN: 978-1-78528-469-4 Paperback: 220 pages

Learn how to develop an ace game for your iOS device using Sprite Kit

1. Learn about the Sprite Kit engine and create games on the iOS platform from the ground up.
2. Acquaint your Sprite Kit knowledge with Swift programming and turn your 2D game conceptualization into reality in no time.
3. An abridged and focused guide to develop an exhaustive mobile game.



Learning Unity iOS Game Development

ISBN: 978-1-78439-980-1 Paperback: 230 pages

Build exciting games with Unity on iOS and publish them on the App Store

1. Take advantage of Unity 5's new tools to create a fully interactive mobile game.
2. Learn how to connect your iTunes developer account and use Unity 5 to communicate with it.
3. Use your Macintosh computer to publish your game to the App Store.

Learning Unreal® Engine iOS Game Development

ISBN: 978-1-78439-771-5 Paperback: 212 pages

Create exciting iOS games with the power of the new Unreal® Engine 4 subsystems

1. Learn each step in the iOS game development process, from start to finish.
2. Develop exciting iOS games with the Unreal Engine 4.x toolset.
3. Step-by-step tutorials to build optimized iOS games.

UDK iOS Game Development Beginner's Guide

ISBN: 978-1-84969-190-1 Paperback: 280 pages

Create your own third-person shooter game using the Unreal Development Kit to create your own game on Apple's iOS devices, such as the iPhone, iPad and iPod Touch

1. Learn the fundamentals of the Unreal Editor to create gameplay environments and interactive elements.
2. Create a third person shooter intended for the iOS and optimize any game with special considerations for the target platform.
3. Take your completed game to Apple's App Store with a detailed walkthrough on how to do it.