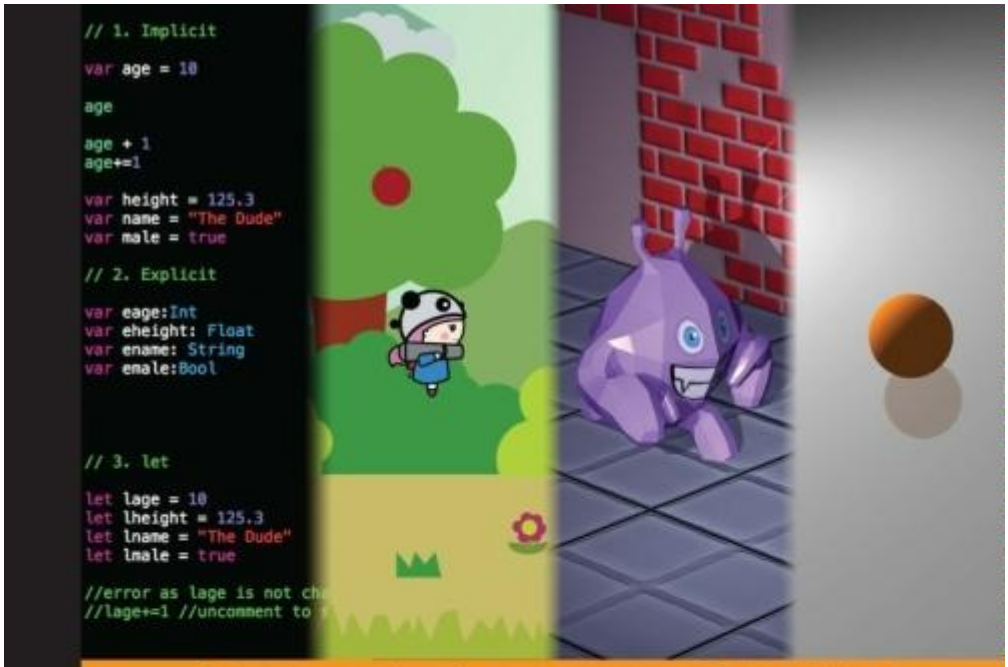


Community Experience Distilled

Learning iOS 8 Game Development Using Swift

Create robust and spectacular 2D and 3D games from scratch using Swift – Apple's latest and easy-to-learn programming language

Siddharth Shekar

[PACKT] PUBLISHING

Table of Contents

Learning iOS 8 Game Development Using Swift
Credits
About the Author
Acknowledgments
About the Reviewers
www.PacktPub.com
    Support files, eBooks, discount offers, and more
        Why subscribe?
        Free access for Packt account holders
Preface
    What this book covers
    What you need for this book
    Who this book is for
    Conventions
    Reader feedback
    Customer support
        Downloading the example code
        Errata
        Piracy
        Questions
1. Getting Started
    Downloading and installing Xcode
    Creating an iOS developer account
    Introducing Swift
    Introducing Playground
    Exploring SpriteKit
        New features in SpriteKit
        Looking at the default SpriteKit project
    Exploring SceneKit
        Looking at the default SceneKit project
    Understanding 3D objects
    2D and 3D coordinate systems
    The basics of SceneKit
    Introducing Metal
    The graphics pipeline
    Summary
2. Swift Basics
    Variables
    Operators
        Arithmetic operators
        Comparison operators
        Logical operators
        Arithmetic increment/decrement
        Composite operations

- Statements
  - Decision-making statements
    - The if statement
    - The if else statement
    - The else if statement
    - Conditional expressions
    - The switch statement
  - Looping statements
    - The while loop
    - The do while loop
    - The for loop
    - The for in loop
- Arrays
  - Looping through arrays
  - Adding, removing, and inserting objects into arrays
  - Important array functions
- Dictionary
  - Adding and removing objects from the dictionary
  - Looping through items in the dictionary
  - Dictionary functions
- Functions
  - Simple functions
  - Passing a parameter
  - Passing more than one parameter
  - Returning a value
  - Default and named parameters
  - Returning more than one value
- Classes
  - Properties and initializers
  - Custom methods
  - Inheritance
  - Access specifiers
- Optionals
- Summary

3. An Introduction to Xcode
- Xcode application types
  - Master-Detail Application
  - Page-Based Application
  - Tabbed Application
  - Single View Application
- The Xcode interface
  - The toolbar
  - The Navigation panel
  - The Project Navigation tab
  - The Symbol Navigator tab
  - The Find Navigator tab
  - The Issue Navigator tab
  - The Test Navigator tab

- The Debug Navigator tab
- The Breakpoint Navigator tab
- The Report Navigator tab
- The Utility panel
- The Single View Project
- The project root
- General
- The Capabilities tab
- The Info tab
- The Build Settings tab
- The Build Phases tab
- The Build Rules tab
- The project folder
- The Utility panel (Redux)
- The inspector tabs
- The File Inspector tab
- The Quick Help Inspector tab
- The Identity Inspector tab
- The Attributes Inspector tab
- The Size Inspector tab
- The Connections Inspector tab
- The library
- The File Template tab
- The Code Snippet Library tab
- The Object Library tab
- The Media Library tab
- The Debug panel
- Running the app on the device
- Summary

4. SpriteKit Basics
- Introduction to SpriteKit and SKScene
- Adding a main menu scene
- Adding a gameplay scene
- Adding a background and a hero
- Updating the position of the hero
- Adding player controls
- Adding enemies
- Adding enemy bullets
- Collision detection
- Keeping score
- Displaying the score
- Displaying the game over screen
- Adding the main menu button
- Saving the high score
- Resetting the high score count
- Summary

5. Animation and Particles
- Sprite sheet animation

- Basic SpriteKit animation
- Exploring Texture Packer
  - Data
  - Texture
  - Layout
  - Sprites
- Creating the hero spritesheet
- Animating the hero
- Particle systems
  - Designing particles
    - Name
    - Background
    - Particle Texture
    - Particles
    - Lifetime
    - Position Range
    - Angle
    - Speed
    - Acceleration
    - Alpha
    - Scale
    - Rotation
    - Color Blend
    - Color Ramp
- Creating particle effects
- Adding a particle system to the game
- Summary

6. Audio and Parallax Effect
- Audio file formats
- Downloading and installing Audacity
- Converting the audio format
- Adding sound effects
  - Adding the fireRocket sound effect
  - Adding the enemy-kill sound effect
  - Adding a sound effect at GameOver
- Adding background music
  - Adding audio loops
- Parallax background theory
- Implementing the parallax effect
- Summary

7. Advanced SpriteKit
- Lighting and shadows
- Sprite Illuminator
  - The Sprites panel
  - The Preview panel
  - The Tools panel
- Physics
- Objective-C in Swift

- Glyph Designer
  - Implementing a Bitmap font
- Skeletal animation
- Summary

8. SceneKit
- Creating a scene with SCNScene
- Adding objects to the scene
  - Adding a sphere to the scene
  - Adding light sources
  - Adding a camera to the scene
  - Adding a floor
- Importing scenes from external 3D applications
- Adding objects and physics to the scene
  - Accessing the hero object and adding a physics body
  - Adding the ground
- Adding an enemy node
  - Updating objects in the scene
  - Checking for contact between objects
- Adding a SpriteKit overlay
  - Adding labels and buttons
- Adding touch interactivity
- Finishing the game loop
  - Making the hero jump
  - Setting a game over condition
  - Fixing the jump
- Adding wall and floor parallax
- Adding particles
- Summary

9. Metal
- Overview
- The graphics pipeline and shaders
  - The preparation/initialization stage
    - Get device
    - Command queue
    - Resources
    - Render pipeline
    - View
  - The draw stage
    - Start render pass
    - Get command buffer
    - Draw
    - Commit the command buffer
- The basic Metal project
- The colored quad project
- The texture quad project
- Summary

10. Publishing and Distribution
- Getting Ms. tinyBazooka ready

Generating the distribution certificate
The iTunesConnect portal
Creating the app
Creating an ad hoc app
References
Alternative frameworks/engines
Final remarks
Summary
Index

Learning iOS 8 Game Development Using Swift

Learning iOS 8 Game Development Using Swift

Copyright © 2015 Packt Publishing

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

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

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

First published: May 2015

Production reference: 1250515

Published by Packt Publishing Ltd.

Livery Place

35 Livery Street

Birmingham B3 2PB, UK.

ISBN 978-1-78439-355-7

www.packtpub.com

Credits

Author

Siddharth Shekar

Reviewers

M. Adil

Peter Ahlgren

Marco Amador

Julien Lange

Commissioning Editor

Sarah Crofton

Acquisition Editor

Shaon Basu

Content Development Editor

Samantha Gonsalves

Technical Editor

Faisal Siddiqui

Copy Editors

Hiral Bhat

Vikrant Phadke

Sameen Siddiqui

Project Coordinator

Sanchita Mandal

Proofreaders

Stephen Copestake

Safis Editing

Indexer

Tejal Soni

Graphics

Abhinash Sahu

Production Coordinator

Melwyn Dsa

Cover Work

Melwyn Dsa

About the Author

Siddharth Shekar is a game developer with over 4 years of experience. He has been programming for the last 10 years and is adept in languages such as C++, C#, Objective-C, Java, JavaScript, Lua, and Swift. He has experience in developing games for the Web, mobile phones, and desktops using Flash, Cocos2d, Cocos2d-x, Unity3D, and Unreal Engine. Siddharth has also worked with graphics libraries such as DirectX, OpenGL, and OpenGLES.

He is the founder and CEO of Growl Games Studio, and has developed several games and published them on iOS, Android, and Windows Phone stores. Apart from developing games, he has conducted game development workshops in major engineering colleges, and is a visiting faculty in game development institutes in India. He is also the author of the book *Learning Cocos2d-x Game Development*, *Packt Publishing*.

In his spare time, Siddharth likes to experiment with the latest game development frameworks and tools. Apart from playing games, he has an avid interest in animation and computer graphics. He also likes to listen to all kinds of music and plays the guitar.

You can find more information about Growl Game Studios and Siddharth at www.growlgamesstudio.com. You can tweet him at `@sidshekar` or also follow his blog at http://growlgamesstudio.tumblr.com/, and his Facebook page at www.facebook.com/GrowlGamesStudio to get the latest updates.

Acknowledgments

First and foremost, I would like to thank my mom, Shanti Shekar, and dad, R. Shekar, for their continuing, unconditional love and support.

Also, three people I can't thank enough are Andreas from Code'n'Web, Tom from 71Squared, and Soren of EsotericSoftware for their awesome tools. Otherwise, I would have spent most of my time developing tools rather than creating any games. Additionally, thanks to Michael for writing the Spine Runtime for Swift. Thanks to the team at Apple for developing an awesome language and framework, and for making the lives of game developers a little less like hell.

A special thanks to Packt Publishing for putting this book together. I would like to thank Shaon Basu and Samantha Gonsalves for helping and guiding me through every step in the process of writing this book. Thanks to the technical reviewers for the technical feedback and tips. I have really learned a lot of new things in this process.

Finally, I would like to thank my friends, followers, and well-wishers for being a part of my life and tolerating me for all these years.

About the Reviewers

M. Adil has an engineering degree in computer science. He likes well-crafted iOS applications. He has a very good understanding of the Objective-C and Swift programming languages and the iOS SDK. He has been working on iOS since 2009. Currently, Adil is working as a principal software engineer at Appifytech (www.appifytech.com). You can also contact him at www.itsaboutcode.com, his personal website.

I would like to thank my wife, Bushra, for being next to me and encouraging me in my adventures.

Peter Ahlgren is a professional freelancing web and app developer from Sweden. He has been freelancing since 2000, and has worked with clients such as Disney, Warner Bros, the Swedish Cancer and Allergy Fund, and many more.

Peter has a firm of his own, BrainLab, and mostly works on frontend web development and iOS app development. He has also worked on a few Swedish books on web development. You can visit his website at http://peterahlgren.com.

Marco Amador is a husband and a father of two beautiful daughters, and has more than 20 years of experience in the software industry. He is an experienced software engineer who has worked in several mission-critical systems, developing highly available and scalable software using mainly Java/JEE technologies.

Lately, Marco has been involved in IoT, Big Data, and Cloud technologies. He is also the co-founder and currently the chief technology officer of Maisis – Information Systems.

Julien Lange is a 34-year-old expert in software engineering. He started developing on an Amstrad CPC464 with the BASIC language when he was 7. He later learned Visual Basic 3/4, then VB.NET, and then C#. For several years until the end of his education, he developed and maintained several e-business websites based on PHP and ASP.NET. After his graduation, Julien continued to learn more and more about software, which included software architecture and project management, and always tried to acquire new skills. Since 2011, he has been working as an IT project manager on the lead management middleware of Axa France (a French insurance company). This middleware is based on SOA architecture. As it is used by the frontend exposed on the Internet, performance is the top

priority each time he delivers a new release of the system. Scalability and robustness are really important in his everyday work.

Julien first developed an interest in mobile development in 2009. After discovering the massive potential of iPhone games and software, he decided to find an improved game engine, which would allow him to concentrate only on the main purpose of a game: developing the game itself and not a game engine. His choice was Unity 3D, thanks to its compatibility with C# and its high frame rate on the iPhone. In addition to his main work as an IT consultant, he created www.iXGaminG.com in December 2010.

Julien currently has several projects in mind, including a game based on a French board game, and a website that delivers new services to developers. He is searching for a few partners to work with. In addition to this, and after reviewing three books by Packt Publishing, he has now decided to write a new book for them. It will be about applying design patterns with Swift to build robust and scalable applications.

I would like to thank my family, and most of all my wife, for allowing me to take some time to review books on my computer. I would also like to thank a few people with whom I work everyday: Alain, Adrien, Juliette, Raphael, Christelle, Stephane, and Helmi.

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.

Preface

With Xcode 6, Apple has made game development more accessible to anyone wanting to learn it. No additional installations are required. All the features needed for making a game are right at your fingertips. You can create a simple game for iOS devices with just a single touch control, or you can create a full-fledged big-budget game for OS X. Your imagination is the limit.

Apple's new programming language Swift is at the heart of all this magic. Programming doesn't get easier than this. You don't have struggle between creating an interface and implementation for each class. Say goodbye to square brackets. And yet, it looks and feels like any other programming language so that you can hit the ground running. If you are coming from a JavaScript or ActionScript background, you will feel right at home. Even if you are proficient only in Objective-C, there is no need to worry as all the game development technologies work with both Objective-C and Swift.

With improved lighting and physics in SpriteKit, you don't have to write extra code to make your games come to life. Scenes in games will have proper lighting; all you have to do is provide the location of the light source. You can provide a normal map, which will caste shadows depending on the direction of the light source. With tools such as Glyph Designer, Texture Packer, and Spine, you can really make your game shine and stand out, technically and visually.

SceneKit is finally here on iOS devices. So now you can create 3D games with ease. You can also create texture geometries, camera, and lighting right in code. You can even import COLLADA files of scenes made in your favorite 3D package into SceneKit and see them running without adding any extra code. With SceneKit's inbuilt physics and collision system, you can make your game dynamic. The coolest thing is that you can also add a SpriteKit layer for the 2D GUI and the button interface.

If all that wasn't enough, Apple has also introduced Metal, a new graphics library developed only for Apple devices. Developed from the ground up, it greatly reduces the size of the overhead and brings the code much closer to the GPU so that you can add more stuff onto the screen and still maintain a smooth 60 Hz frame rate. At the same time, Apple gives you the freedom to develop games using OpenGLES as well.

Seeing that we have our work cut down for us, let's dive right into learning Swift, SpriteKit, SceneKit, and Metal.

What this book covers

Chapter 1, *Getting Started*, shows you how to download Xcode and create an iOS Developer account. You also get an overview of Swift, Playground, SpriteKit, SceneKit, and Metal, and a basic understanding of a graphics pipeline.

Chapter 2, *Swift Basics*, covers the basics of Swift, including a look at different data types, statements, arrays, dictionaries, functions, classes, and optionals.

Chapter 3, *An Introduction to Xcode*, demonstrates the different types of applications you can create with Xcode. You also get a general understanding of the interface of Xcode and get to learn how to run an app on the device.

Chapter 4, *SpriteKit Basics*, explains the creation of a basic game in SpriteKit, while you understand the basic elements of SpriteKit, such as sprites, scenes, and labels. We also create a basic physics engine and add collision detection and scoring.

Chapter 5, *Animation and Particles*, illustrates the addition of animation and particle effects in the game using SpriteKit's inbuilt animation and particle system.

Chapter 6, *Audio and Parallax Effect*, covers the addition of background music and sound effects to the game. We also add a parallax effect to the game to make it more dynamic.

Chapter 7, *Advanced SpriteKit*, teaches you about the advanced features in SpriteKit, such as adding lighting and physics to your game. Additionally, it teaches you how to integrate Objective-C classes with Swift to implement tools such as Glyph Designer and Spine.

Chapter 8, *SceneKit*, shows the creation of a basic 3D game in SceneKit. This chapter teaches you how to add objects such as geometries, lighting, and cameras to the game scene. You also get to import COLLADA files into the scene and integrate the SpriteKit interface to add the GUI and the button interface.

Chapter 9, *Metal*, contains an in-depth look at Metal's graphics pipeline and shaders. It also teaches you how to get access the device, create vertex and texture buffers, and draw an image on the screen.

Chapter 10, *Publishing and Distribution*, prepares your app for distribution. You also get to create an app on iTunesConnect, and this chapter shows you how to publish your app on the iOS App Store. Additionally, it shows you how to create an ad hoc build and run it on any registered device.

What you need for this book

You will need the latest version of Xcode 6 and A Mac running OS X Yosemite, or at least Maverick version 10.9.4. Although a simulator is included with Xcode, it is better to get an iOS device running iOS 8. For running Metal, a device with the A7 chip or higher is mandatory, as it won't run on a simulator.

Who this book is for

This book is for hobbyists and game developers who want to develop 2D and 3D games and release them on the App Store. This book is also for graphics programmers and veteran developers who want to get a general idea of Metal's graphics pipeline and shader language.

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 can include other contexts through the use of the `include` directive."

A block of code is set as follows:

```
var height = 125.3
var name = "The Dude"
var male = true
```

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

```
var height = 125.3
var name = "The Dude"
var male = true
```

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

```
#
cp /usr/src/asterisk-
addons/configs/cdr_mysql.conf.sample
     /etc/asterisk/cdr_mysql.conf
```

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: "Clicking on the **Next** button moves you to the next screen."

> ## Note
>
> Warnings or important notes appear in a box like this.

> ## Tip
>
> 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.

Chapter 1. Getting Started

So you want to create games for iOS? With the introduction of **SpriteKit** in **Xcode 5**, it has become a breeze to create two-dimensional games for iOS. Previously, even before thinking about creating games, you had to think about what framework to use to create games. There were so many frameworks available, and each had its own pros and cons. Also, what if you wanted to create your own framework? In that case, you had to write it from the ground up using **OpenGLES**, which required writing a whole lot of code just to display a triangle. And let's not even talk about creating three-dimensional games using the frameworks, as most of the frameworks don't even support it.

Apple solved all of these questions and issues by giving all the required tools with Xcode 6. In Xcode 6, you are really limited by your imagination. You can create 2D or 3D games using **SpriteKit** and **SceneKit**. If you want to create your own 2D or 3D engine, there is Metal available, which makes it easy to communicate with the **GPU** (short for **Graphics Processing Unit**). But if you are a veteran and have used OpenGL ES to create games, don't worry! That option is still there, so you are not tied down to using only Metal. So, let's get started with iOS 8 game development with Xcode 6.

The following topics will be covered in this chapter:

- Downloading and installing Xcode
- Creating an iOS developer account
- Introducing Swift
- Introducing Playground
- Introducing SpriteKit
- Looking at the default SpriteKit project
- New features in SpriteKit
- Looking at the default SceneKit project
- Understanding 3D objects
- The 2D/3D coordinate system
- Exploring SceneKit
- Introducing Metal
- The graphics pipeline

Downloading and installing Xcode

If you have an Apple account, you can open the desktop version of the App Store and search for Xcode. Click on **Install** so that you can start downloading it. For this book, I will be using Xcode 6.1. By the time this book gets published, there might be a newer version of Xcode 6. If you wish to download the same version that I used to work with, you can create a

free Apple Developer Account, go to the **Downloads** section, and download the previous version.



Tip

Before downloading Xcode, make sure you are working on either Mac OS X Yosemite 10.10 or Maverick 9.4. Otherwise, Xcode 6.1 can't be installed on your machine.

Xcode is Apple's **Integrated Development Environment** (**IDE**). It is required for developing any sort of app for iOS or Mac OS X. It is not just an IDE; it is packed with a lot of tools and features, which makes it an integral part of any developer's arsenal. You can click on the **...More** button on Xcode's **App Store** page to see the different tools and features it has to offer. We will also look at some features of Xcode in the next chapter, when we cover the Xcode interface.

To install Xcode, click on the **Install** button under the icon. You will then be asked to enter your Apple ID and password. The installation will start soon after. Once you click on the **Install** button, the download will start in the launch pad, as shown in the following screenshot:



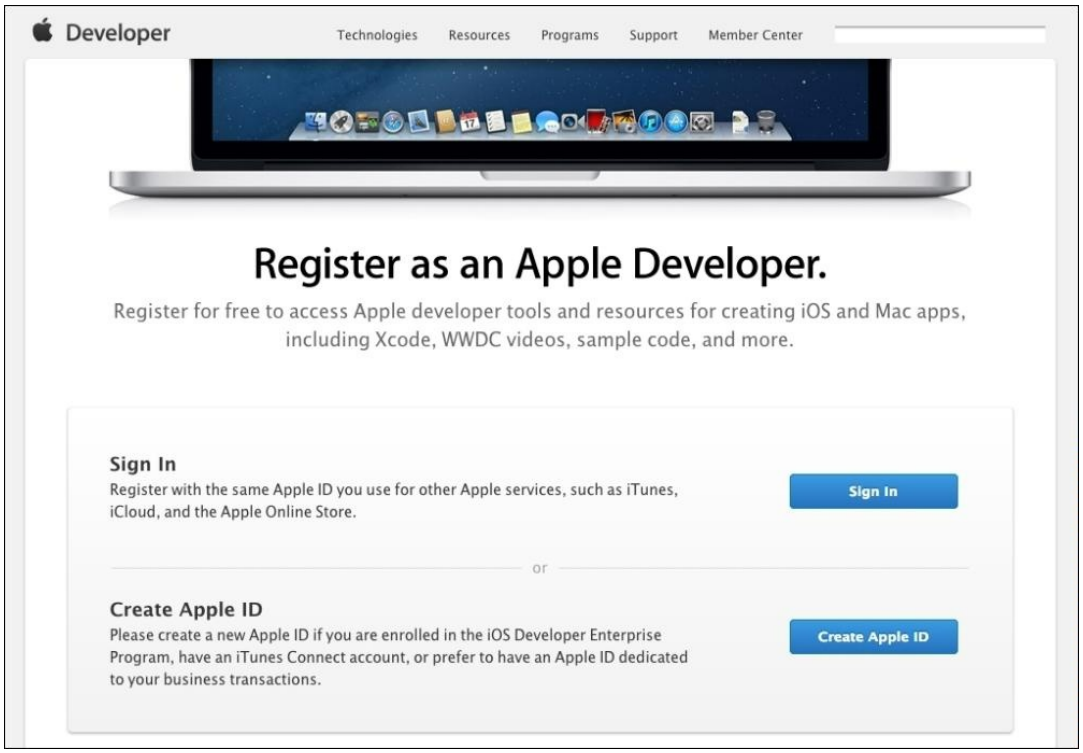
Once it is downloaded and installed, you can see it appear in your launch pad. You can click on the application's icon to launch the application.

Creating an iOS developer account

To create any app and publish it on the App Store for iOS, Mac OS X, or Safari, you need to be enrolled in a developer program. You can create a free developer account to access certain sections such as tutorials and downloads, but you won't be given access to the latest beta software. Also, to run and test the app or game on your device, you need to be enrolled for this developer program.

Since we do plan to create a small game for iOS and publish it on the App Store, we do need to enroll for the iOS Developer Program. I will assume that we don't have a developer account yet, so let's create a new developer account first, and then we will enroll for the iOS program.

To register as an Apple developer, go to https://developer.apple.com/register/index.action. The following page will be displayed:



If you already have an Apple ID, you can use it to log in. Otherwise, click on **Create Apple ID**. Here, you will be asked to fill in the information under the following headings:

- **Name**: Enter the following details here:
    - First name
    - Middle name

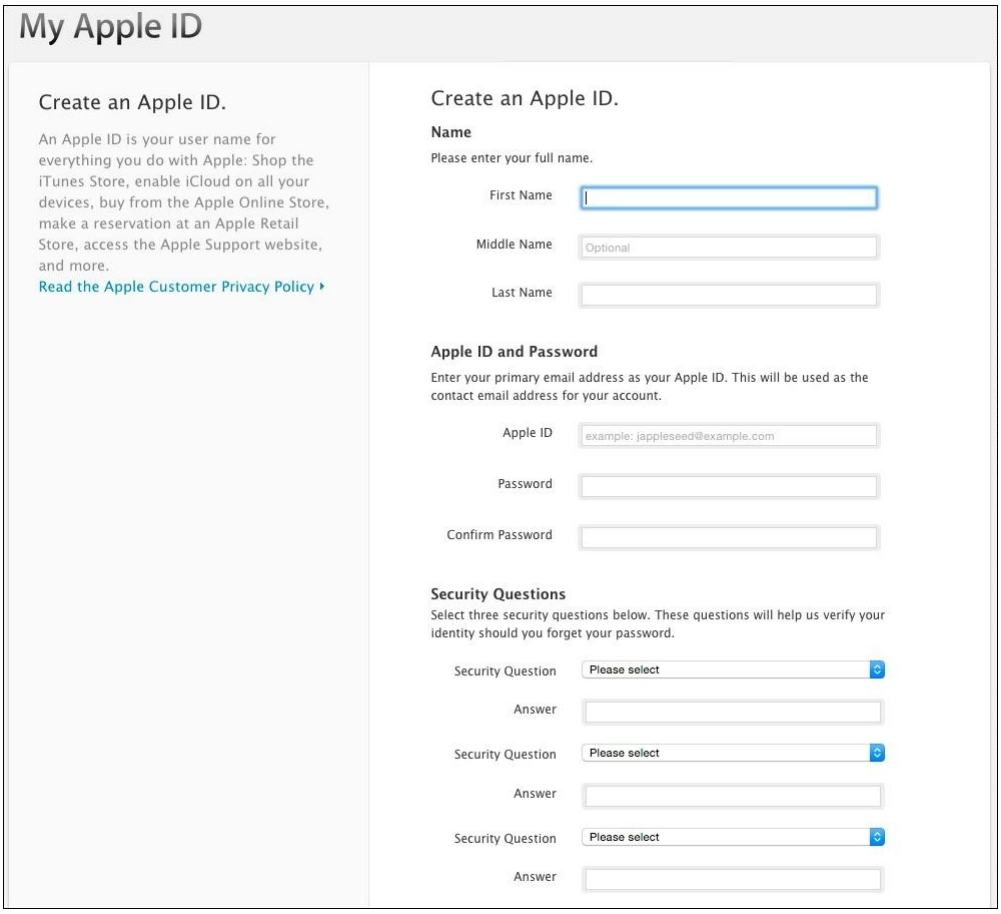
    - Last name

- **Apple ID and Password**: Enter a preferred Apple ID and password here. Keep the following pointers in mind:

    - **Apple ID**: Any e-mail address of your choice.
    - **Password**: The password should have at least eight characters; it should have at least one lowercase character, one capital letter, and one number; and it should not contain identical characters consecutively. Also, it shouldn't be the same as the account name.
    - **Confirm Password**: Type the same password that you typed in the **Password** field.

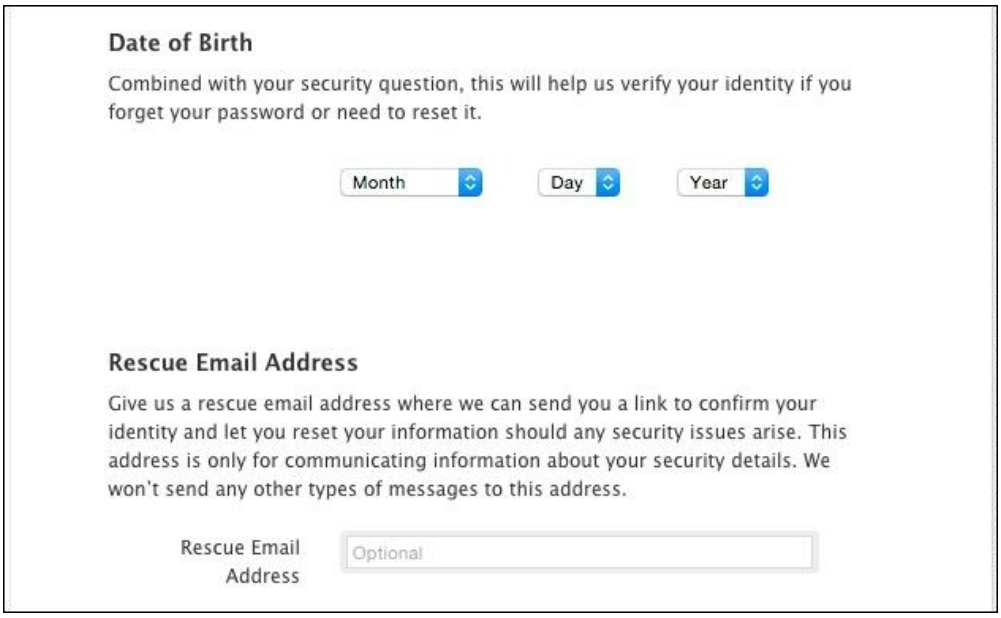
- **Security Questions**: If you forget your password, you will be asked to answer some questions to gain access to your account. You can choose them here. So, be careful when choosing these questions, and also make note of the questions and answers for future reference. There are three questions, and each question has multiple choices. So, choose a question that is most relevant to you and select answers that you can remember easily.

  All the headings mentioned until now can be seen in this screenshot:

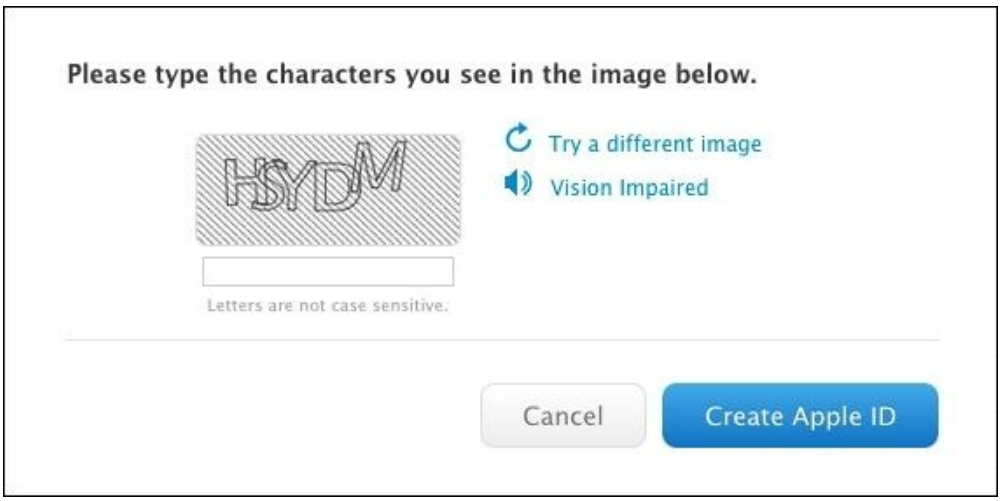


- **Date of Birth**: Enter your date of birth.
- **Rescue Email Address**: This is an alternate e-mail address through which you can be communicated with.

The preceding two headings can be seen in the following screenshot:

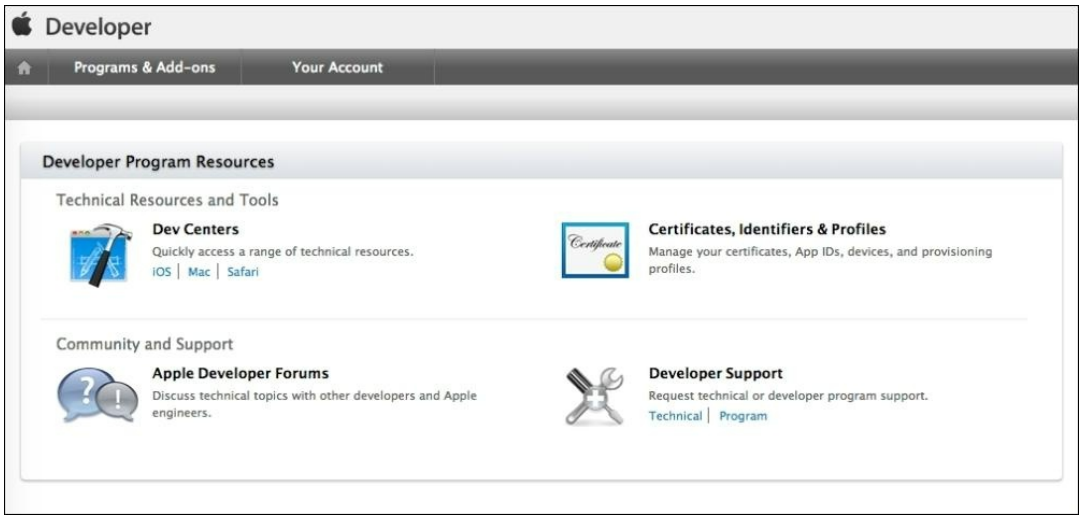


- **Mailing Address**: Enter your mailing address.
- **Preferred Language**: Choose the language in which you are most comfortable. If you ask any questions to the Apple support team, they will respond to you in this language.
- **Email Preference**: If you like to be up to date with the latest Apple news, software updates, and information on products and services, you can check the two check boxes below this heading.
- **CAPTCHA**: Type the CAPTCHA image in the box shown in this screenshot:



Now click on **Create Apple ID** to generate the Apple ID. If you've done everything successfully, congratulations!!!!! You are now an Apple developer.

Once you are in, you will be greeted with the following screen:



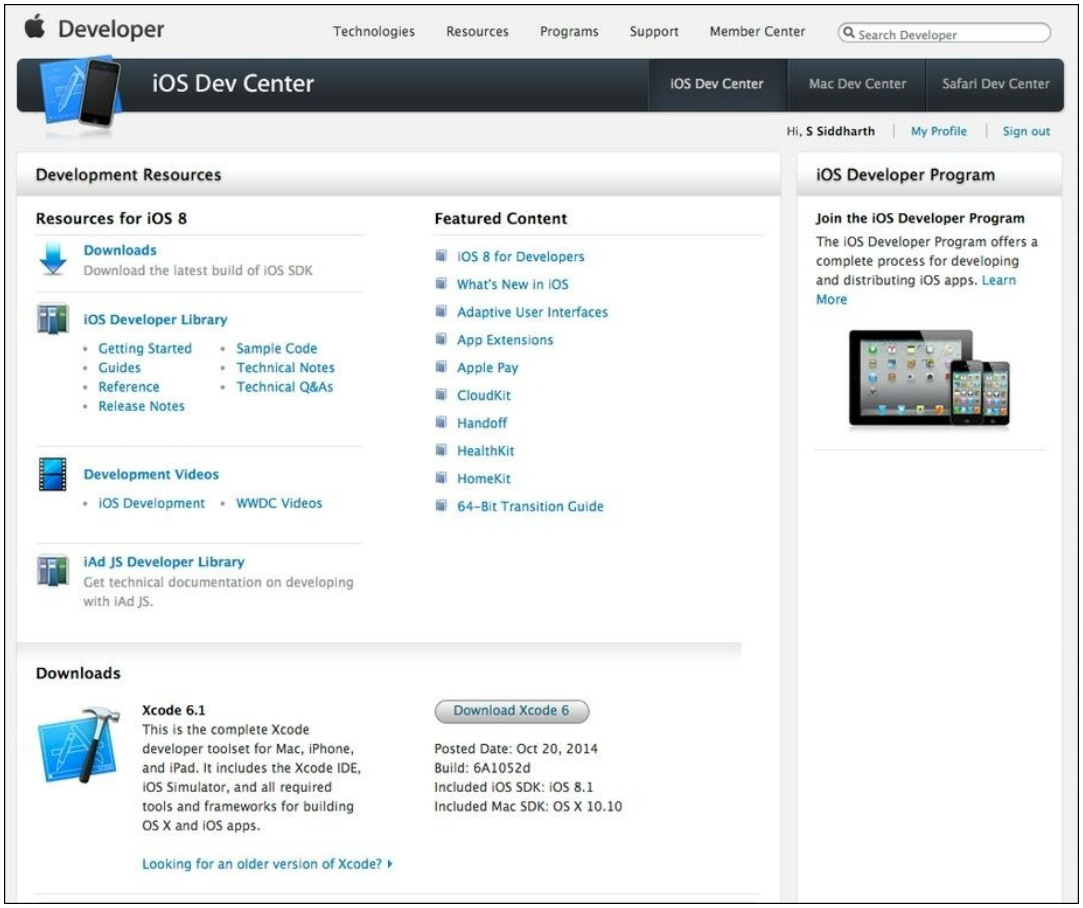
You will be spending most of the time in the **Technical Resources and Tools** section, though you have a strong developer community and developer support, which can be accessed at any time under the **Community and Support** section.

Under the **Technical Resources and Tools** section, you have two subsections: **Dev Centers** and **Certificates, Identifiers & Profiles**. In the **Dev Centers** subsection, you will find all the technical resources for the appropriate development platform. Through the **Certificates, Identifiers and Profiles** link, you can generate and manage your certificates, create provisioning profiles, and manage App IDs and development devices. We will go through this section when we create an app and wish to run it on our device.

Let's look at the **Dev Centers** section. If you click on **Mac**, you will see the links to the resources that develop apps for Yosemite, such as the link to the latest build of OS X. You can also get articles, sample code, guides, and so on that you can use to develop the app you always wanted to make for Yosemite. You also get a link to the development video that was shown in WWDC.

If you click on **Safari**, you will see a layout similar to what you saw earlier, with links to sample code, and the developer library that you can use to develop apps for Safari.

Now click on **iOS** in **Dev Centers**. You should be greeted with the following screen:



Here, you will find all the resources that are required to develop all kinds of apps for iOS. You can download the latest version of iOS SDK, or if you wish to download an older version of Xcode, you can do that from the **Downloads** link.

Similar to Mac Developer Library, you have an iOS Developer Library with links to how to get started, guides, sample code, references, and release notes. There are also links to videos on iOS development under the **Development Videos** section.

You can download Xcode, install it on your machine, and test your coding skills, but if you want to test your awesome app on a device and later publish it on the App Store, you need to register to the iOS Developer Program.

As I said earlier, you are required to be part of the program to be able to test the app on the device or publish it on the App Store. In the meantime, you can run the SpriteKit and SceneKit apps you make on the simulator.

So, if you wish, you can enroll once you are satisfied with how the game looks on the simulator.

That being said, a simulator is just that—a simulator. How the game works on the simulator shouldn't be taken to mean how the game will actually work on the device. The game will run slower on the simulator because the Mac processor performs two tasks: running your OS and also the simulator.

So, it is better to test the app on the device to get a better understanding of how it is going to finally run on the device, and the more devices you run it on, the better.

> ## Note
>
> Additionally, it should be noted that at the time of writing this book, apps/games developed using Metal cannot be run on the simulator, and they require an A7 or A8 chip to run on the device. If you try running them on the simulator, it will give an error.

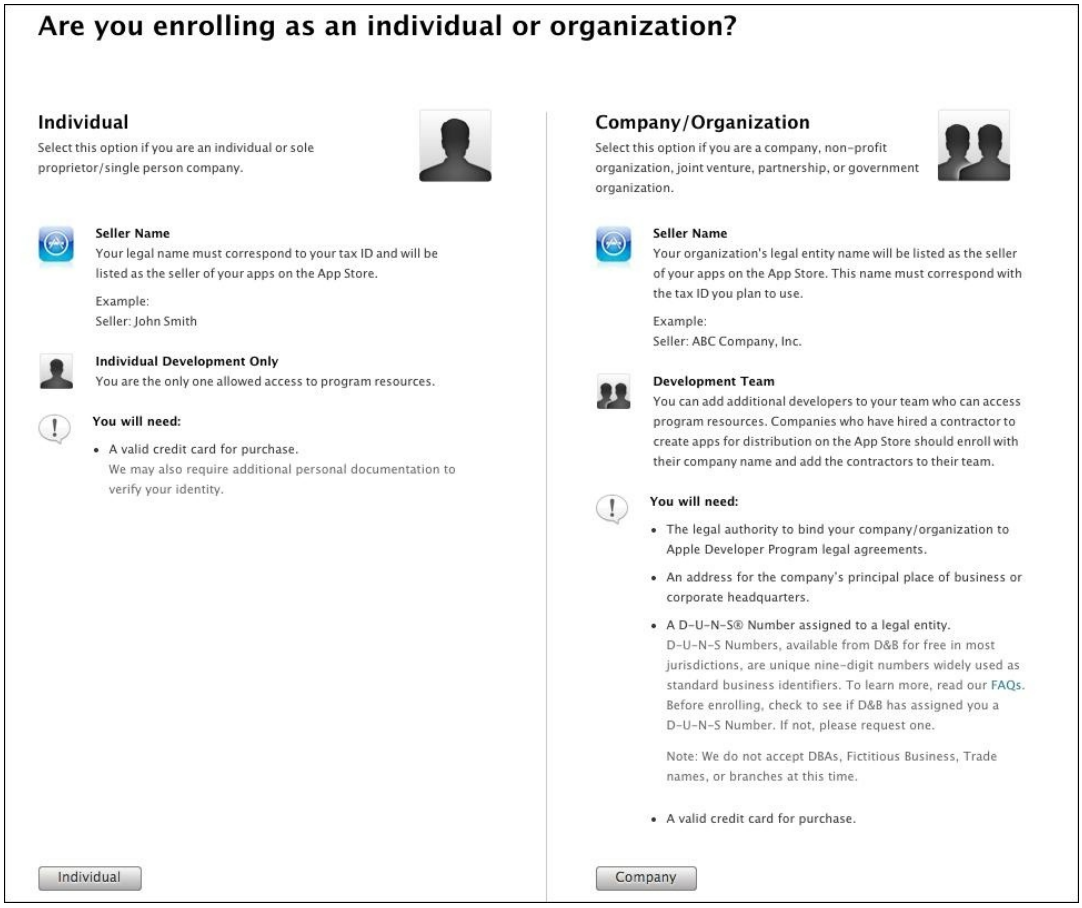
So, if you are ready to enroll in the iOS Developer Program, click on the **Learn More** link under the **Join the iOS Developer Program** heading on the right-hand side of the page you are currently at.

Let's get enrolled in the iOS Developer Program. Get your credit card and your attorney ready, and click on the **Enroll Now** button at the top to start the process. Next, click on the **Continue** button to go to the next screen. Click on **Continue** again, as you have already created an Apple ID. If you have still not created an Apple ID, click on **Create Apple ID** and follow the mentioned steps, then come back to this page, and continue with your new Apple ID. Once you click on **Continue**, you will be redirected to a page where you will have to choose whether you want to register as an individual or a company.

If you register as an individual, your name will be displayed as the seller on the App Store, and you won't be asked for any documentation for registration. If you register as a company, non-profit organization, joint venture, partnership, or government organization, you should select the option to register as an organization. To enroll as an organization, you will need additional documents such as the Tax ID, D-U-N-S number (which is available for free), and so on.

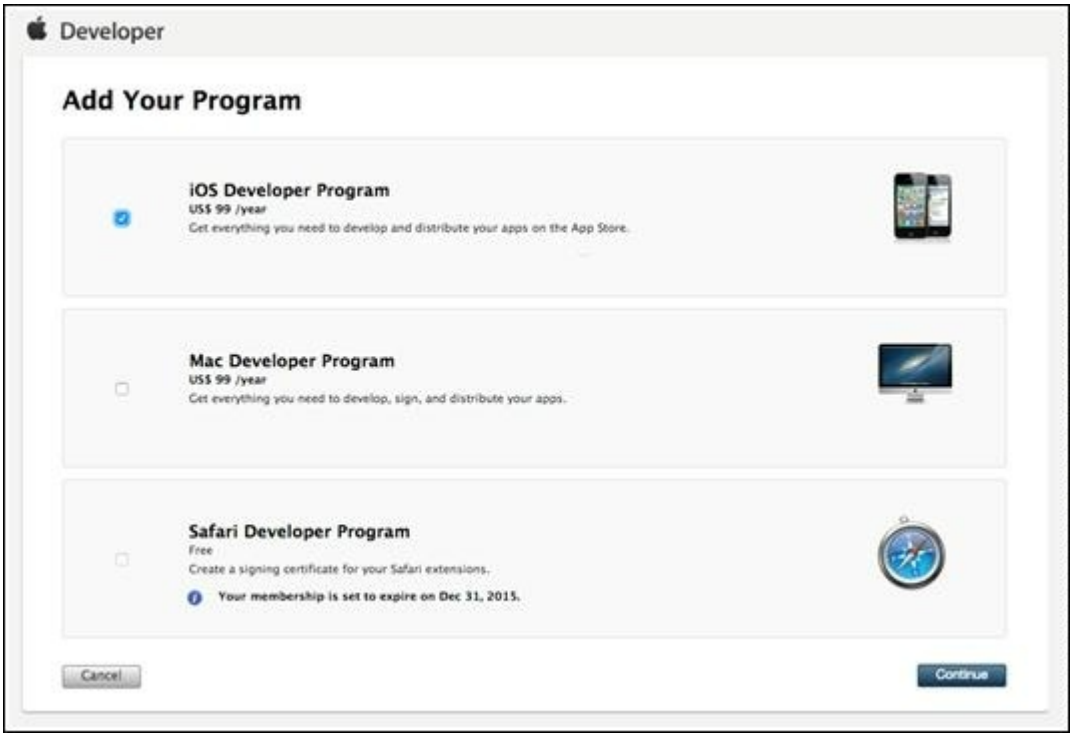
In the case of the organization, the name of the organization will be displayed as the seller on the App Store. You can also include additional developers as part of the team, unlike individual registration, in which only one individual can be added per account, that is, the person who is

enrolling. The following screenshot shows the screen where you can select whether you are registering as an individual or a company/organization:



For this book, we will be enrolling as an individual, so click on **Individual** in the bottom-left corner of the screen to proceed to the next screen.

On the next page, you will have to choose the program that you want to enroll for. Since we are going to enroll for the iOS Developer Program, we check the box to the left of **iOS Developer Program** and click on **Continue**, as shown in the following screenshot:



On the next page, you are required to agree to the Program License Agreement. Get your attorney over and ask them to read it. After their confirmation, click on the checkbox to agree to having read the agreement, and to being above the legal age. Click on **I Agree** at the bottom of the page to agree and move on to the next page.

Here, you have to enter your payment information. Provide your credit card and billing information and click on **Continue**. Now verify the information once again and click on **Place Order**.

Once you are done with this, you are a registered iOS developer. You will get a **Thank You** screen and also an e-mail confirmation. It will take up to 2 business days for the order to be processed, so in the meantime, we will take a sneak peek at Swift, SpriteKit, SceneKit, and Metal.

Introducing Swift

With Xcode 6, Apple has introduced a new scripting language called Swift. If you are coming from a JavaScript or ActionScript background, you will find yourself very much at home. But even if you have been using Objective-C for a long time, don't worry. You can still create games using Objective-C. Before creating any project, you will be asked in which language you want to create the app/game. Just select **Swift** and you are good to go.

In Chapter 2, *Swift Basics*, you will learn how to code in Swift, and we will see how it is different from Objective-C. When coding in Swift, we will start from the absolute beginning, with variables, control statements, loops, and advanced topics such as classes.

Introducing Playground

Playground is a file that you can use to test your Swift code and see immediate results of your code. Let's actually create a new file and take a look at it.

If you have downloaded Xcode from the developer portal instead of the Mac Store, double-click on the DMG file to open it. Once it is open, you can drag the Xcode app into your `Applications` folder to install it.

Once it is installed, drag the app into the dock, as we will be using it quite often. Once you have dragged it into the dock, click on it to open it. You will have to agree to the terms and conditions once it opens, so agree to them to proceed.

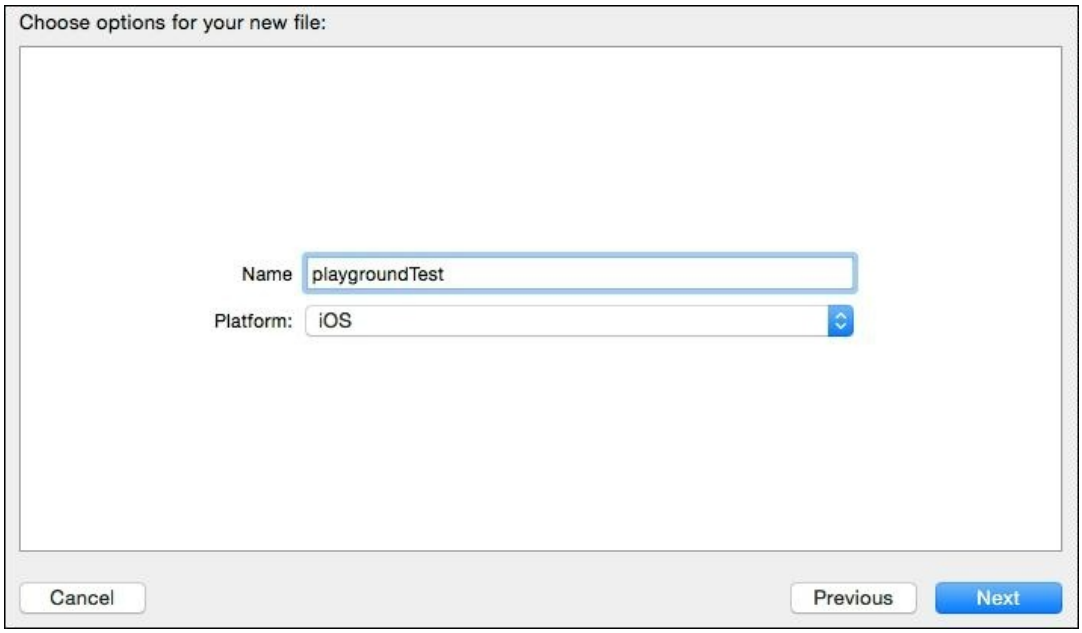
You should see the welcome screen on Xcode. Here, you can click on any one of **Get started with a playground**, **Create a new Xcode project**, or **Check out an existing project**. The details of these three options are as follows:

- **Get Started with a playground**: With a playground, you can test your Swift coding skills and hone them before using the playground to develop an app or game
- **Create a new Xcode project**: Whenever you want to create a new Xcode project, you will have to click on this button, and then you can also select the type of project you want to create
- **Check out an existing project**: If you are using a **source code manager** (**SCM**) to manage code, such as GitHub, SVN, or BitBucker, you can check out the project you have there, using this

These options can be seen in this screenshot:



For now, click on the **Get Started with a playground** button. Give a filename (I gave it the name `playgroundTest`), select the **iOS** platform, and click on **Next**, as shown in the following screenshot:



On the next screen, you will be asked where you want to create the project folder. To be organized, I keep all my projects in the `_Projects` folder in the `Documents` folder, under the specific game technology's folder name starting with an underscore (_). So in this case, I created a new `_Projects` folder in the `Documents` folder. Then I created a new folder called `_Playground`, selected it, and clicked on **Create**.

You will see that the `playgroundTest` file is created in the `_Projects/_Playground` folder in the `Documents` folder. Once the file is created, you will see the following window:



You will be coding in the left pane, and you will be able to see the immediate result in the right-hand-side pane. You can see that there is already some code written on the left and the result is shown on the right. Here, the `Hello, playground` string is assigned to a new variable called `str`, and the `str` variable's value is immediately logged on the screen to the right.

We will cover more information about playgrounds when you learn Swift in Chapter 3, *An Introduction to Xcode*, as all the coding practices that we will be doing will require the playground file.

Exploring SpriteKit

SpriteKit is a 2D game development framework that was first introduced in iOS 7 and Xcode 5. It is primarily used to create 2D games, so objects can be placed or moved only in the *x* and *y* coordinates. Using SpriteKit, you can create 2D games for both iOS and OS X.

If you have used Cocos2d, you will feel very much at home with the architecture and syntax of SpriteKit. Since the game will be mostly populated with sprites, it is called SpriteKit.

Since SpriteKit is a 2D game development framework, it provides all the tools that you require to create a complete 2D game from start to finish. You can create a MainMenu Screen, Gameplay Screen, and Options Screen. You can also create buttons on each of these screens. When the buttons are pressed, you can navigate between the screens. In the gameplay screen, you can add players, enemies, text to display the score, and particles such as smoke and explosion with the particle editor.

SpriteKit also includes a physics engine that performs all the physics-related calculations. All you have to do is include it in the scene, and you will see the objects in the scene interacting with each other automatically, according to the physics simulation. In addition, SpriteKit also includes an automatic texture atlas generator for better optimization of your game assets.

There are some classes in SpriteKit that are the basic building blocks for creating any game.

New features in SpriteKit

There are a lot of new cool features added in SpriteKit in Xcode 6:

- Graphics technologies:
    - Shaders
    - Lighting and shadows
- Physics simulations technologies:
    - Per-pixel physics
    - Physics fields
    - Inverse kinematics
    - Constraints
- Tools and improvements:

- SpriteKit Editor
- Integration with SceneKit

SpriteKit now includes shaders that you can use to create new and interesting effects in your game. You can also create light sources to cast real-time 2D shadows.

The already powerful physics engine of SpriteKit has been made even more powerful with the inclusion of per-pixel physics for pixel-perfect collision detection. With the addition of physics fields, we can create an *Angry Birds Space* clone in no time, and with inverse kinematics, it is easier to make joint movements of your 2D characters look more realistic. Along with all of this, you can also use constraints to control the movement or rotation of any physics object in the scene along any direction or angle.

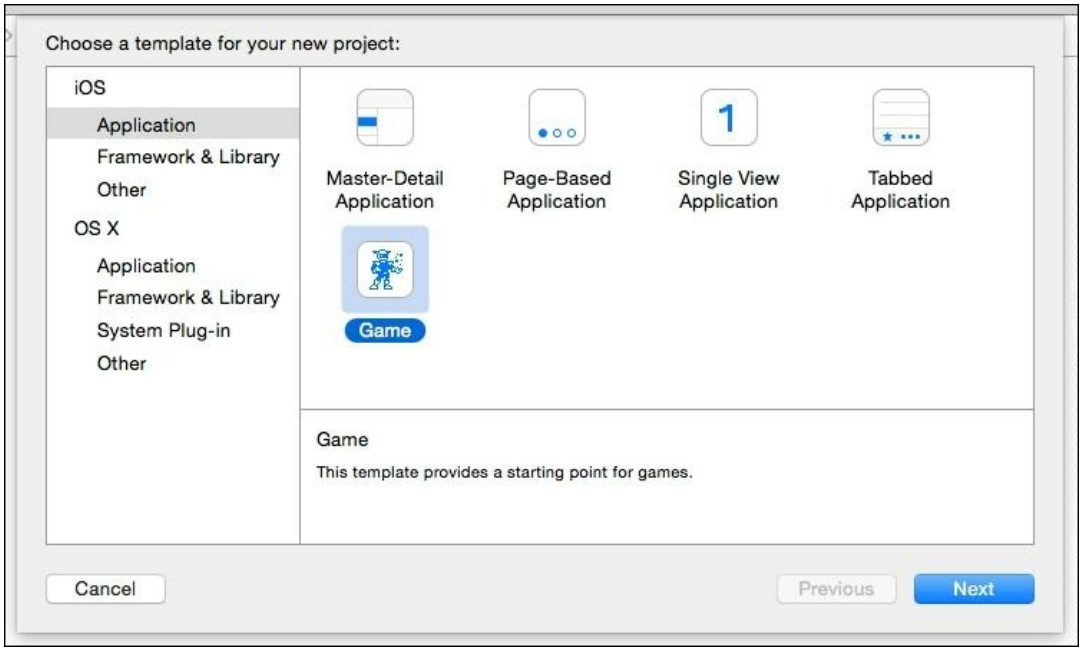
SpriteKit also includes a new editor that can be used to create a simple game without writing a single line of code. It can also be used to check for errors as a debugging tool.

SpriteKit can also be used with SceneKit. SceneKit is a 3D game development framework newly developed by Apple. If you want to create GUI elements in your 3D game, such as 2D buttons and radars, this can be achieved very easily using SpriteKit.

Looking at the default SpriteKit project

Let's look at the default SpriteKit project that gets created when we create a new project so that you can understand some of the terms that you will be using while creating the game.

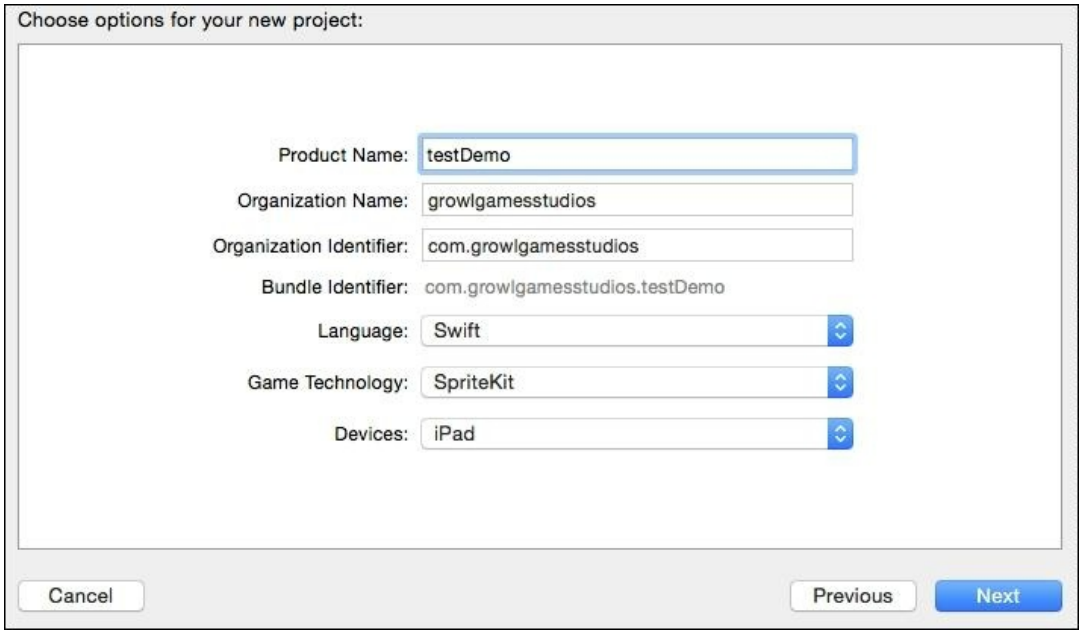
For now, we will create a new project. So, click on the **Create a New Xcode Project** button in the middle. Once you have clicked, you will see the following window:



In the left panel, you need to select the platform you wish to create the game for, iOS or OS X. Since we are going to be creating games for iOS in this book, we will select from the iOS section.

In **Applications**, you can select the type of application that you want to create: **Master-Detail Application**, **Page-Based Application**, **Single View Application**, **Tabbed Application**, or **Game**. Since SpriteKit, SceneKit, Metal, and OpenGL ES are part of a game application, we will select **Game**. Then click on **Next**.

In the **Choose options for your new project** window, you will have to fill the **Product Name**, **Organization Name**, **Organization Identifier**, **Language**, **Game Technology**, and **Devices** fields:



The details of the fields shown in the preceding screenshot are as follows:

- **Product Name**: When you create an actual project, the entry for this field is the name of the game that you are creating, such as `AngryBirds`, `CutTheRope`, and so on.
- **Organization Name**: Here, you can input the organization name for which you are developing your game. For the purpose of this book, you can input any name you want.
- **Organization Identifier**: This is very important, and I can't stress enough on how important it is. On the App Store, Apple will identify your app by only this. Whatever input is given here translates into the value for **Bundle Identifier**. This has to be unique, and no other app on the App Store can have the same package name. Although you can call it whatever you wish, the standard format used is the reverse format of the company website followed by the product name, like this: `com.<company name>.<product name>`.

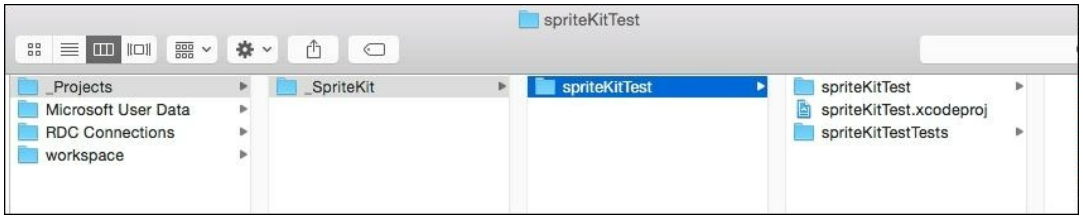
  If you don't have a company or a website, don't fret. You just have to make sure your package name is unique. If it is not, Apple will not accept the app. In such a case, you can try different package names, and later you have the option to change the package name in Xcode to whatever Apple approved as the package name.

- **Language**: From this drop-down list, you can select the language of choice in which you want to develop the game. You can choose between **Objective-C** and **Swift**. Since we will be developing the games in Swift in this book, we will use the **Swift** option here.
- **Game Technology**: From this drop-down list, you can select the technology that you want to use to develop the game with. You have the option of choosing SceneKit, SpriteKit, Metal, and OpenGL ES. As we are going to use SpriteKit, select it from the dropdown.

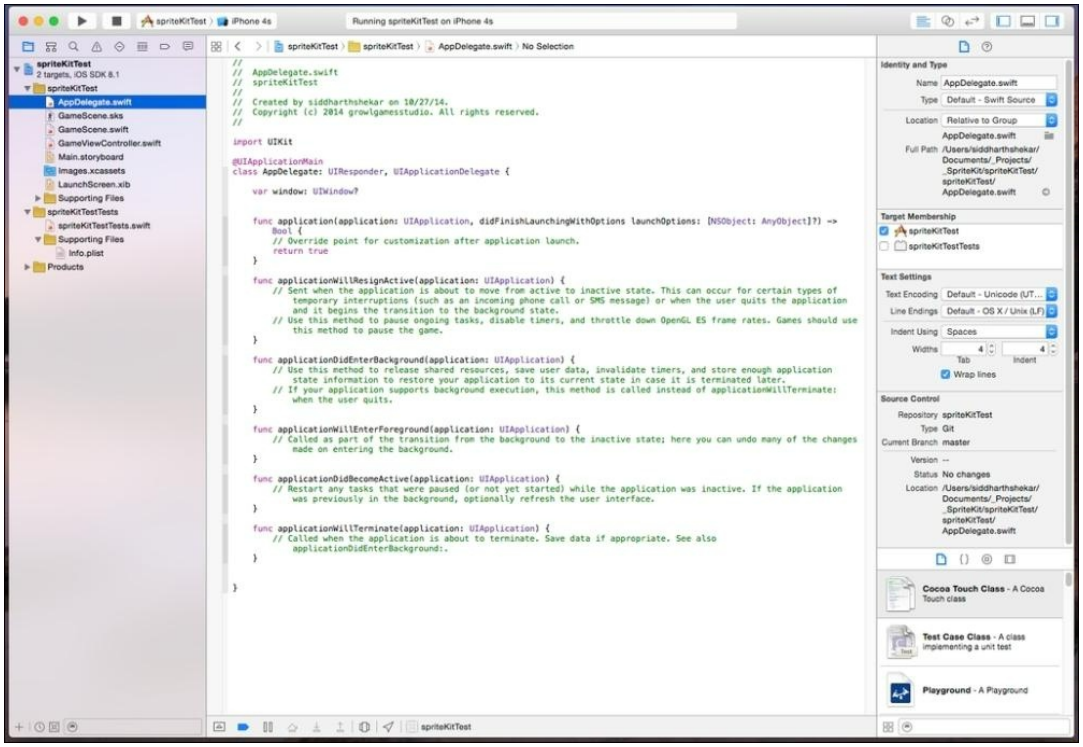
- **Devices**: From this drop-down list, you can select the device that you want to develop the game for. You can choose either **iPhone** or **iPad** if you want to develop for the iPhone or the iPad, or you can choose **Universal** if you want the game to run on both the platforms. You can choose **iPad** for now.

Click on **Next** to proceed. In the screen that follows, you will be asked where to create the project folder. For this, create a new folder called `_SpriteKit` in the `_Projects` folder. Then select this folder and click on **Create**.

You will see that the `spriteKitTest` project folder is created in the `_Projects/_SpriteKit` folder in the `Documents` folder, as shown in the following screenshot:

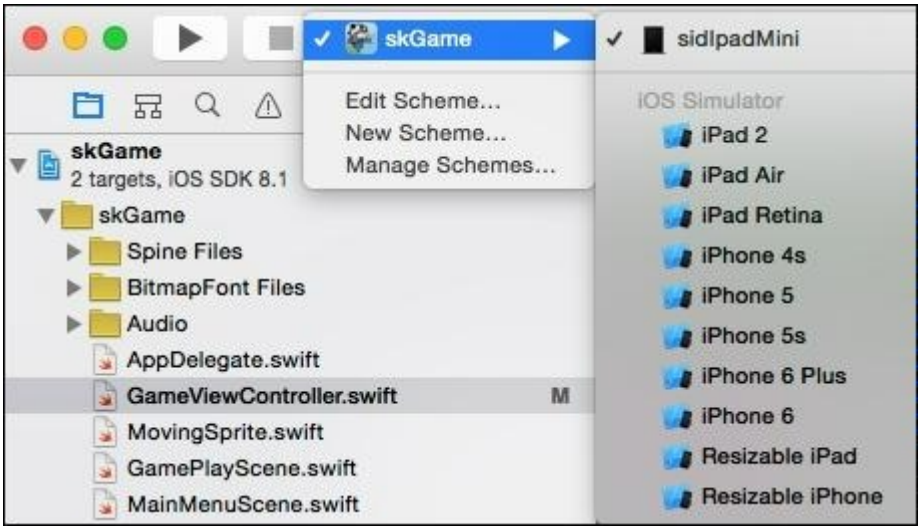


Now double-click on the `spriteKitTest.xcodeproj` file to open your project in Xcode. Your project should open up, as shown in this screenshot:

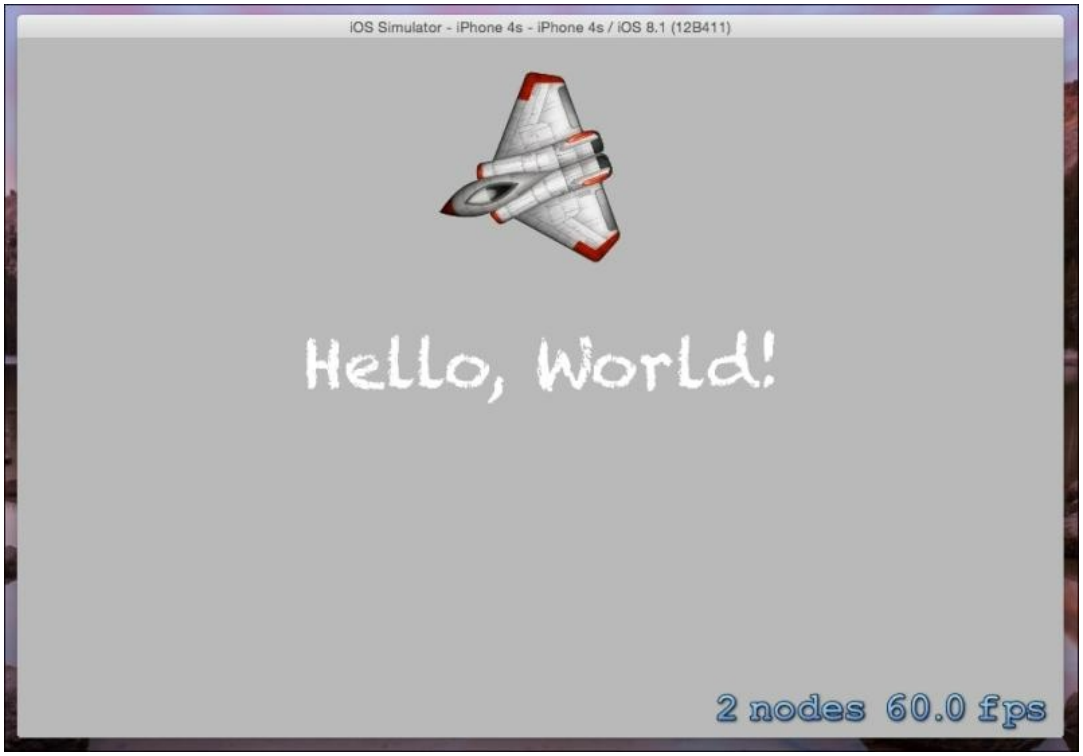


Xcode has a built-in simulator that can show you how the game or app will look in a specified device. You can select the device by clicking on the

name of the project to the right of the stop button at the top of the window, as shown in the following screenshot:



Click on the play button in the top-left corner to build the current project. The game will start compiling all of the code and will automatically launch the simulator. Once the simulator starts, give it some time to display the following screen:



Well, it won't look exactly like the preceding screenshot. For one thing, you won't see the image of the plane above `Hello, World`. Secondly, it will be in portrait mode, instead of landscape as shown in the preceding screenshot.

To change the view to landscape mode, click on the project name in the left panel. The middle panel will change and display the options for landscape, portrait, and so on. Uncheck the portrait option and stop and build the game again. Now the view will change from portrait to landscape whenever you build the game.

Next, how do you get the plane onto the screen? It is very simple; all you have to do for it to appear is to click anywhere on the screen. I clicked on the center of the screen, between the top and the place where `Hello, World` is shown, which is why the plane showed up there. To create more planes, all you have to do is keep on clicking wherever you want a new plane to appear. You will see that when you add more planes, the numbers on the bottom right of the screen also change, and the planes start rotating slower and slower. Why is this happening? And what are these numbers? Before that, let's first cover a few terms, classes, and functions that we need to familiarize ourselves with:

- `SKScene`: This is a class used to create scenes, for example, the MainMenuScreen, OptionsScreen, and GamePlayScreen. Each scene will contain sprites such as the player and buttons, which will populate the screens and help you to navigate to other screens. In the project on the left-hand-side panel, you will find a file called `GameScene.swift`. This is the scene that gets loaded as soon as the game is built.
- `SKSpriteNode`: As we saw earlier, each scene will be loaded with sprites or images. To load a sprite onto the screen, you have to use `SKSpriteNode`. In this scene, whenever you touch the screen, you create a sprite in the specific location. It gets the image name of `Spaceship`. This can be seen in the `touchesBegan` function in the same class.
- `SKLabelNode`: In order to display any sort of text on the screen, you need to use `SKLabelNode` to decide which font should be used to create the text. You also need to provide the text that needs to be displayed on the screen, along with the size and the position of the text to be displayed. Here, we see that `SKLabelNode` is used to display `Hello, World` on the screen. In `myLabeltext`, you can write whatever you want to display, in quotes, and build again to see whatever you typed get displayed on the screen.
- `SKAction`: These are used to modify nodes' parameters over a period of time. For example, you can scale an object up to twice its size over a 1-second duration, and then bring it back to its normal size. Or you can change the position of an object to make it move or rotate over a period of time. You can perform these actions together or one after the other, using SKAction. Here, as soon as the spaceship is created, an action is run on it, telling it to rotate by 180 degrees every second.

```swift
//
//  GameScene.swift
//  test
//

import SpriteKit

class GameScene: SKScene {
    override func didMoveToView(view: SKView) {
        /* Setup your scene here */
        let myLabel = SKLabelNode(fontNamed:"Chalkduster")
        myLabel.text = "Hello, World!";
        myLabel.fontSize = 65;
        myLabel.position = CGPoint(x:CGRectGetMidX(self.frame), y:CGRectGetMidY(self.frame));

        self.addChild(myLabel)
    }

    override func touchesBegan(touches: NSSet, withEvent event: UIEvent) {
        /* Called when a touch begins */

        for touch: AnyObject in touches {
            let location = touch.locationInNode(self)

            let sprite = SKSpriteNode(imageNamed:"Spaceship")

            sprite.xScale = 0.5
            sprite.yScale = 0.5
            sprite.position = location

            let action = SKAction.rotateByAngle(CGFloat(M_PI), duration:1)

            sprite.runAction(SKAction.repeatActionForever(action))

            self.addChild(sprite)
        }
    }

    override func update(currentTime: CFTimeInterval) {
        /* Called before each frame is rendered */
    }
}

```

- `touchesBegan`: SpriteKit has inbuilt overridable functions that can be used to register touches on the screen. There are four functions called `touchesBegan`, `touchesMoved`, `touchesEnded`, and `touchesCancelled`. You can use these functions in combination with each other to detect finger touches and create your own control schemes such as tap, swipe, and double tap.
- `update`: The `update` function is another overridable function that is provided by SpriteKit and gets called repeatedly throughout the game, depending on how often you set it to call itself. Usually, the `update` function gets called 60 times a second. We can use this function to update the position, check a collision, or update the score of the game. The `update` function starts getting called automatically once the scene gets initialized.

So now, knowing all that, let's answer the question asked earlier; what are the two values in the bottom-right corner of the screen?

The Nodes represent the number of nodes you have added to the scene. These nodes can be sprites, labels, and so on. This is why each time you add a new spaceship on the screen, you will see the node count increase. The more objects you add to the screen, the more the processing power required.

The number next to the node count is the calculation of FPS or frames per second. In the `update` function, we saw that the function gets called 60 times a second. So, 60 times in a second, the screen is wiped and redrawn again to create a new frame, and the position and rotation of each object will also be updated. So, if you add more objects to the screen, the processor has to do that much work more by drawing all the images on the screen. This is why you see a dip in the FPS; that is, the spaceship starts rotating slower and slower when you add more spaceships to the scene. Also, in this case, it needs to be taken into consideration that the new spaceships are also asked to rotate, so the processor has to do additional work calculating by how much the spaceships need to be rotated every second.

That's all for the introduction to SpriteKit and `GameScene.swift`. There are additional files on the panel that we will cover in the next chapter when we take a deeper look into Xcode. Let's look at SceneKit next.

Exploring SceneKit

SceneKit is a 3D game development framework. So, it can be used to create 3D games or apps for iOS and OS X. It was initially released in OS X 10.8, and now it is available on iOS 8. It is a high-level API built on OpenGL and OpenGL ES, and can be integrated with SpriteKit, as we saw earlier.

In SceneKit, as in any 3D game development framework, we need to provide the camera, lights, and objects for the scene so that the scene can be rendered from the viewpoint of the camera and be processed and displayed on the viewport of the device.



All objects need to be added to the nodes by creating one for every object you place, whether it is a camera, light, or an object.

You can either add a predefined object, such as boxes, spheres, tori, and planes and add texture to it; or import COLLADA files or Alembic files created in a 3D program, export it in `.dae` or `.abc` format, and import it into SceneKit. These can be opened in **Preview** so that you can take a look and check the file before importing it to the SceneKit project. Apart from the geometries; the animations, textures, lights/camera, and so on that you added to the imported scene will also be imported. You can also add 3D text and shapes to your game using SceneKit.

If you assign a texture to the object in the `COLLADA` file, the texture maps need to be imported along with the file. You can add textures to primitive shapes, 3D text, and complex shapes. You will also be able to modify the 3D text and shapes by extruding them to give them depth, or chamfering the corners, in code.

Custom objects and shapes can also be created by providing the position, texture, coordinates, and color for each of the vertices used to create the shape. You have full freedom over the polygon count; if you want smoother models, you can divide the polygons to get a smoother result—all through code.

SceneKit, along with a camera, also provides different light types such as **Ambient**, **Omni**, **Direction**, and **Spot light**. These can be attached to the nodes, making them easy to place and move around.

There is also an editor that can be used to view your scene and all the objects added to the scene. You will also be able to look at the properties of the individual objects and modify them if you wish, after importing them to the scene.

The game assets will be managed by the Asset Catalogue, which will optimize the assets at build time, similar to the texture atlas creator in SpriteKit.

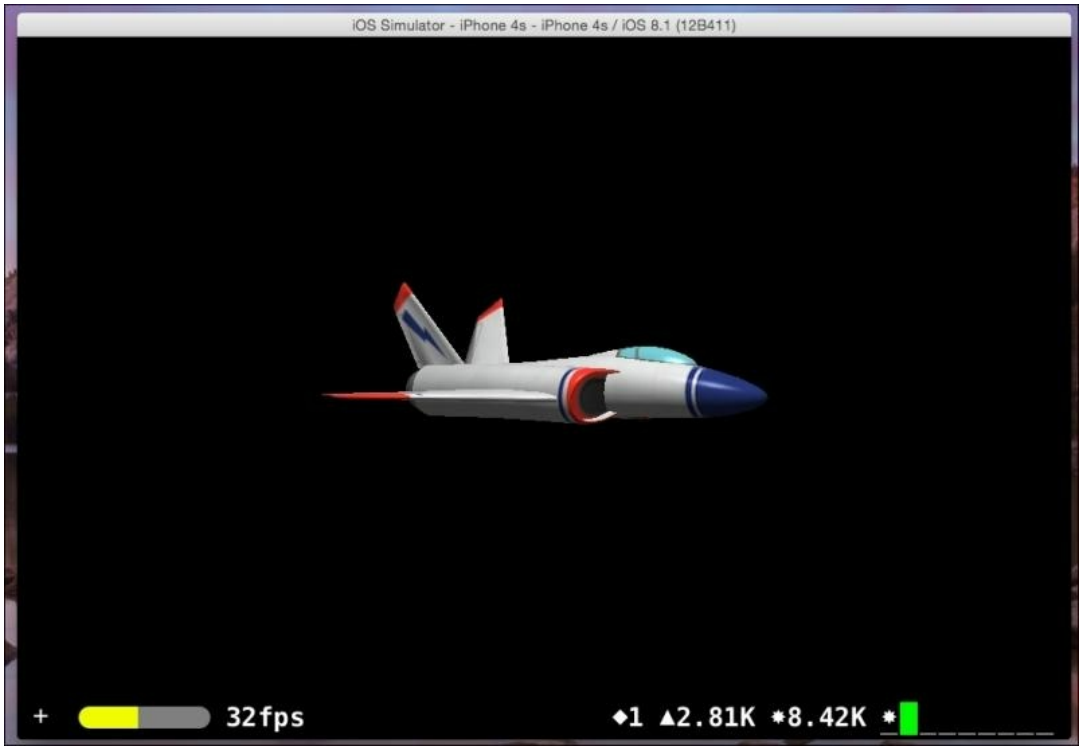
Similar to SpriteKit, SceneKit has actions that can be performed on objects to animate them. It also has a physics engine for physics simulations and collision detection. Like the physics engine in SpriteKit, you can add joints, constraints, and inverse kinematics to your objects and scenes.

Looking at the default SceneKit project

Similar to how you created a SpriteKit project, create a SceneKit project. The only difference when creating the project is that while selecting **Game Technology** in the drop-down list, select **SceneKit** instead of **SpriteKit**. Name the project `SceneKitTest`.

Once the project is created, double-click on the `SceneKitTest.xcodeproj` file. Once it opens, you should see a project structure similar to what you saw in the case of SpriteKit.

As we did earlier, you can click on the play button on the top of the window and select the simulator of your choice. Once the simulator loads, you should see a window similar to what is shown in the following screenshot. I have once again changed the view to landscape for the sake of convenience.



At the bottom, you get more debug information than in SpriteKit. In the bottom-left corner, you have the FPS counter, which is the same as in SpriteKit. As expected, the FPS will be less than SpriteKit projects. This is because we are now looking at 3D models, which include a higher number of vertices and polygons for the processor to calculate, causing the FPS to dip. This is only on the simulator; on the actual device, the FPS will be at 60.

In the bottom-right corner of the screen, the diamond shows the number of draw calls, which is similar to the nodes in SpriteKit. Since there is only one object on the screen—the fighter jet—you get one draw call. The more the objects added, the higher the draw call.

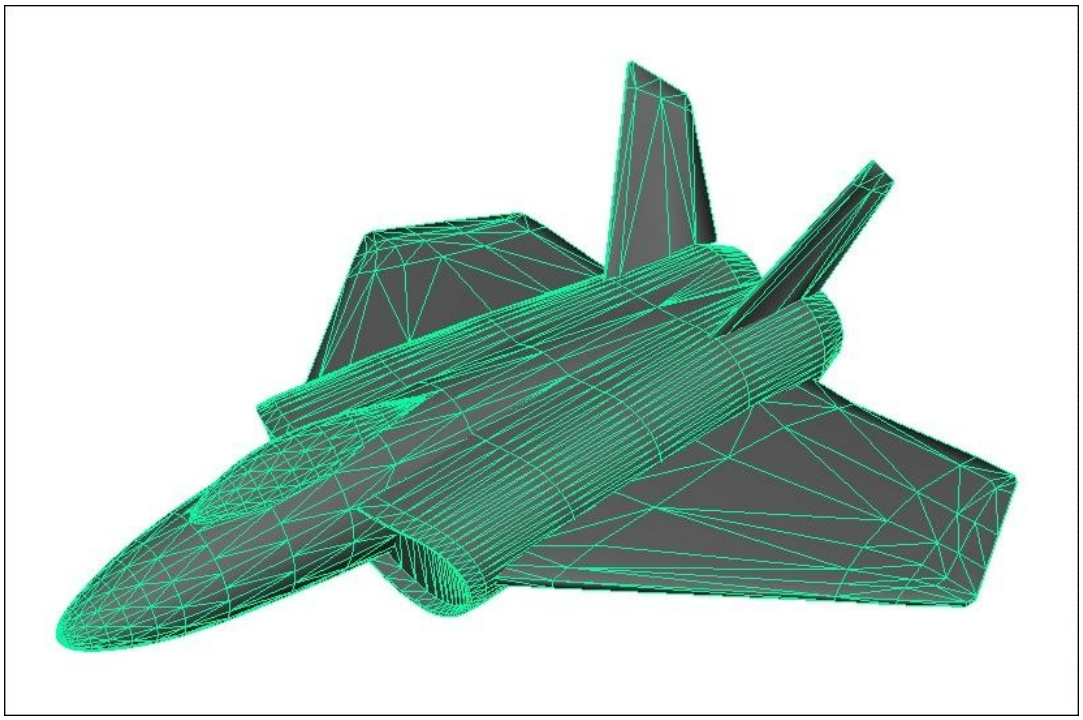
The triangle tells the number of polygons that form the object. The star, or the asterisk, shows the number of vertices that are present in the object. You can click on the **+** sign on the left to get additional information such as **Animations**, **Physics**, **Constraints**, **Particles**, **Delegate**, **Rendering**, **GL Flush**, and **2D**.

3D objects are made of vertices, and these vertices are joined by lines forming triangular or rectangular shapes called polygons. These in turn form the mesh, giving a shape to the object, the fighter jet in this case. Then a 2D image is painted over the shape, giving it the look of a fighter jet. In the left panel, you can open `art.acnassets`. This is where all the art assets of the game will be stored. Select it to look at the mesh and the image, or texture files that get placed on the mesh.

Understanding 3D objects

I have taken the `ship.dae` COLLADA file and imported it into a 3D program to show the mesh of the jet object. COLLADA files can be imported into any of the popular 3D software packages, such as 3DSMax, Maya, or Blender.

In the SceneKit project, we saw the object in all its textured glory. The scene had the textured object, along with the camera, and lights casting shadows. In the following figure, you can see the wire mesh view of the actual mesh of the object. This has been provided so that you can understand what a 3D polygon object is and how vertices make up the polygon to create a mesh of a 3D object.



The green lines are the lines that connect points to form the polygon surface of the object. You can see how these vertices are used to form triangles or polygons to create the desired shape, such as the wings, the cockpit, and the fuselage of the plane. The object is completely hollow. A texture is then pasted on it so that it looks as if it has had a fresh coat of paint put on it. You can also specify a material that can reflect light, making it shiny and reflective.

In SceneKit, you can import a 3D scene created in a 3D package, with the texture, lighting, animations, and camera completely intact. You can create it in your favorite 3D package and export it in the `.dae` format. You will be able to import it into SceneKit, like the jet.

If your game is very simple and doesn't require such complex shapes, then you can create a box, a sphere, and other primitive objects and shapes in SceneKit itself through code, and give them whatever color that you wish. However, you will still need to create a camera and lights so that the object will be visible in the scene.

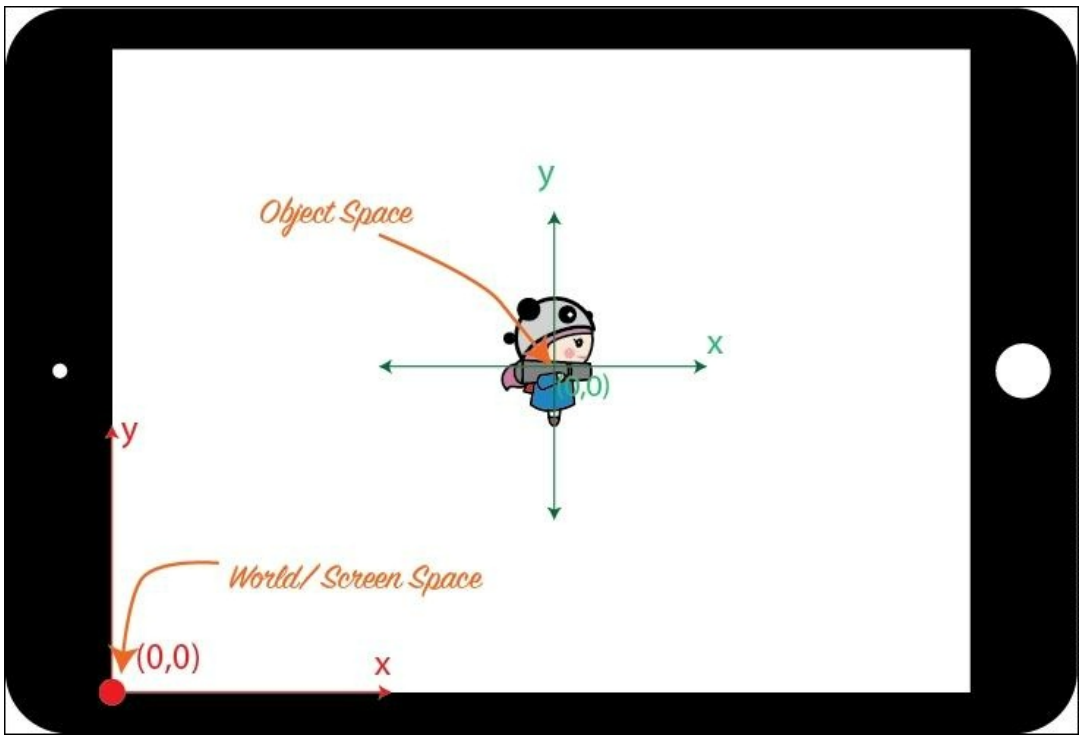
Going back to the simulator, you can click and drag the viewport to rotate the object around its center to take a better look at the object. You can also double-click on the screen to reset the view to its initial state. Other than creating a 3D scene, SceneKit also allows the touch interface so that the player can interact with the object. We will look at this in detail when we cover SceneKit in depth later in this book.

2D and 3D coordinate systems

In the case of 2D game development, we have only two coordinate systems to worry about. The first is the screen coordinate system, and the other is the object coordinate system.

In 2D, whenever we place an object on the screen, we always wonder how far the object is from the bottom-left corner of the screen. This is because the bottom-left corner of the screen, and not the center of screen, is the origin. Therefore, if you place a sprite without changing its position, it will be created in the bottom-left part of the screen. The screen origin or the `(0, 0)` position is in the bottom-left corner of the screen. If you want to place the sprite in the center of the screen, you need to add half the width and height to the position property, since everything is in respect to the bottom-left corner of the screen. This is called the Screen Coordinate system.

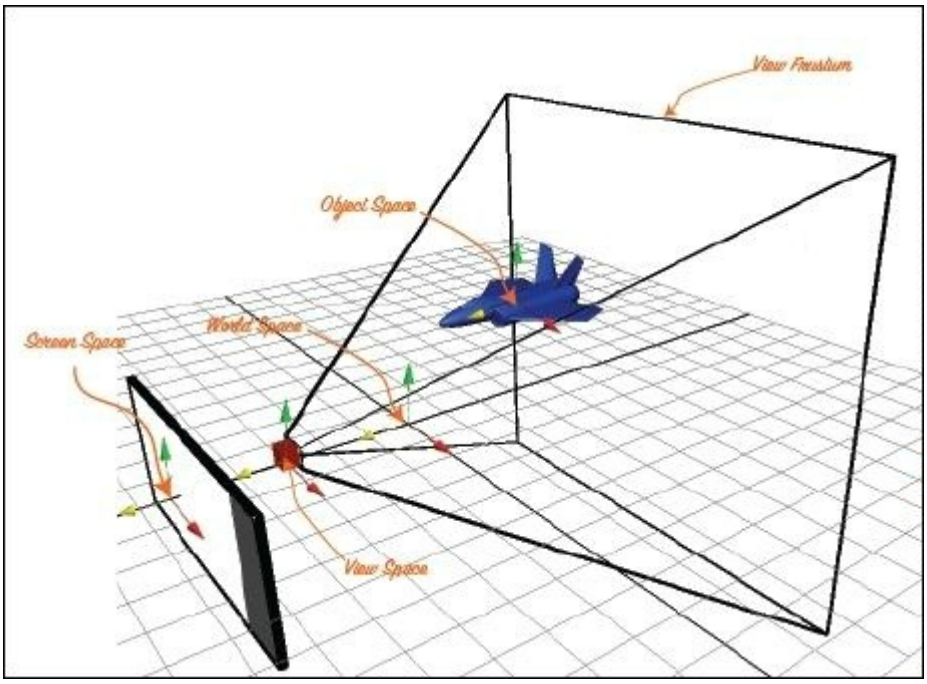
The object coordinate system refers to the sprite itself. The center of the sprite is at the center of the object, unlike the screen, which has its origin at the bottom-left corner. The center of the sprite is called the **anchor point**. When you rotate a sprite, it will rotate about its center because its origin is at its center. You can change the origin of the sprite by accessing its **Anchor Point** property.



In 3D game development, there are a couple of more coordinate systems. So, there are the World, Object, View, and Screen coordinate systems.

Other than being referred to as Coordinate Systems, they are also referred to as a space.

In the case of 2D, the World Coordinate system is the same as the Screen Coordinate System, but in 3D that is not the case. In the following diagram, imagine you are seeing the jet on your device:



The World space's origin is where the red, green, and yellow arrows originate from. The red arrow represents the positive *x* axis, green is the positive *y* axis, and yellow is the positive *z* axis. The World space's origin is at `0, 0, 0`.

The jet is placed within the World space. As with the sprite, the Object Coordinate System is at the center of the object.

The red box represents a camera looking towards the jet plane. The place where the camera is positioned is called a view, eye, or camera coordinate system. The view frustum represents the view limits of the camera. Anything placed outside this area won't be rendered by the camera. In graphics programming, this is called **clipping**.

Finally, whatever is seen by the camera has to be projected onto the screen of the device. This is called the Screen Coordinate System.

The basics of SceneKit

Let's take a brief look at how the scene was created in this project. Open the `GameViewController.swift` file. To create a scene in SceneKit, you have a separate scene creator, here is it called `SCNScene`, to create 3D scenes.

The `GameViewController.swift` file is responsible for the views. Any app that is created must contain at least one view so that it can be displayed on the screen. Whenever you open a new application, a view is created. Once the view is created, the first function that gets called is the `viewDidLoad` function, which starts executing whatever code is written in it:

```
//
//  GameScene.swift
//  test
//

import SpriteKit

class GameScene: SKScene {
    override func didMoveToView(view: SKView) {
        /* Setup your scene here */
        let myLabel = SKLabelNode(fontNamed:"Chalkduster")
        myLabel.text = "Hello, World!";
        myLabel.fontSize = 65;
        myLabel.position = CGPoint(x:CGRectGetMidX(self.frame), y:CGRectGetMidY(self.frame));

        self.addChild(myLabel)
    }

    override func touchesBegan(touches: NSSet, withEvent event: UIEvent) {
        /* Called when a touch begins */

        for touch: AnyObject in touches {
            let location = touch.locationInNode(self)

            let sprite = SKSpriteNode(imageNamed:"Spaceship")

            sprite.xScale = 0.5
            sprite.yScale = 0.5
            sprite.position = location

            let action = SKAction.rotateByAngle(CGFloat(M_PI), duration:1)

            sprite.runAction(SKAction.repeatActionForever(action))

            self.addChild(sprite)
        }
    }

    override func update(currentTime: CFTimeInterval) {
        /* Called before each frame is rendered */
    }
}

```

We create a new scene of the `SCNScene` type and load the `ship.dae` object. There's no need to load the material for the plane separately; it will automatically be assigned to the plane. As every scene first needs a camera, we create a new camera node and add it to the scene. Then the camera is positioned in the 3D space. So unlike SpriteKit, in which positions are specified in *x* and *y* coordinates, SceneKit needs its coordinates to be defined in terms of *x*, *y*, and *z*.

The origin is at the center of the scene. In one of the previous screenshots (the one which shows the world space), the red arrow shows the positive *x* axis, the green arrow denotes the positive *y* axis, and the yellow arrow shows the positive *z* axis. So, the camera—denoted by the red cube—is placed 15 pixels away from the object in the positive *z* direction. The jet object is placed at the origin.

Next, we create two light sources. First we create an omni light, and then we create an ambient light to illuminate the scene. If you don't add any light sources, nothing will be visible in the scene. You can try commenting out the lines, but due to this being where the light sources are added to the scene; you will see a black screen. The jet is still there, but due to the absence of light, you are just not able to see it.

Then we get the ship object from the scene and apply the rotation action to it, similarly to how it was done in SpriteKit. It is just that now, it is called `SCNAction` instead of `SKAction`. Also, you are providing the angle in all three axes, keeping the value for the *x* and *z* axes zero, and rotating the object in the *y* axis every second.

Then, a new `sceneView` variable is created, assigned with the current view and then, the current scene is assigned to the scene of `sceneView`. Then `sceneView` is allowed to control the camera, and show the debug information, such as FPS. You should also set the background color to `black`.

A new function is created, called `handleTap`, where resetting of the view on double tapping is handled.

The `GameViewControllerclass` class has other functions such as `shouldAutoRotate`, `prefersStatusBarHidden`, `supportInterfaceOrientation`, and `didReceieveMemoryWarning`. Let us look at each of them in detail:

- `shouldAutoRotate`: This function is set to `true` or `false` depending on whether you want to rotate the view if the device is flipped. If it is `true`, the view will be rotated.
- `prefersStatusBarHidden`: Enable this function if you want the status bar hidden. If you want it to be shown for some reason, you should disable this option.
- `supportInterfaceOrientation`: The device will automatically check which types of orientation are allowed. In this case, if it is an iPhone, it accepts all but the upside down orientation.
- `didReceiveMemoryWarning`: If there are images and other data that you haven't released, and if they are putting stress on the memory,

this function will automatically issue a warning saying that there is not enough memory and will close the app.

All of this would have been fairly complex, but don't worry. We will break this down to its basics when we cover SceneKit later in the book.

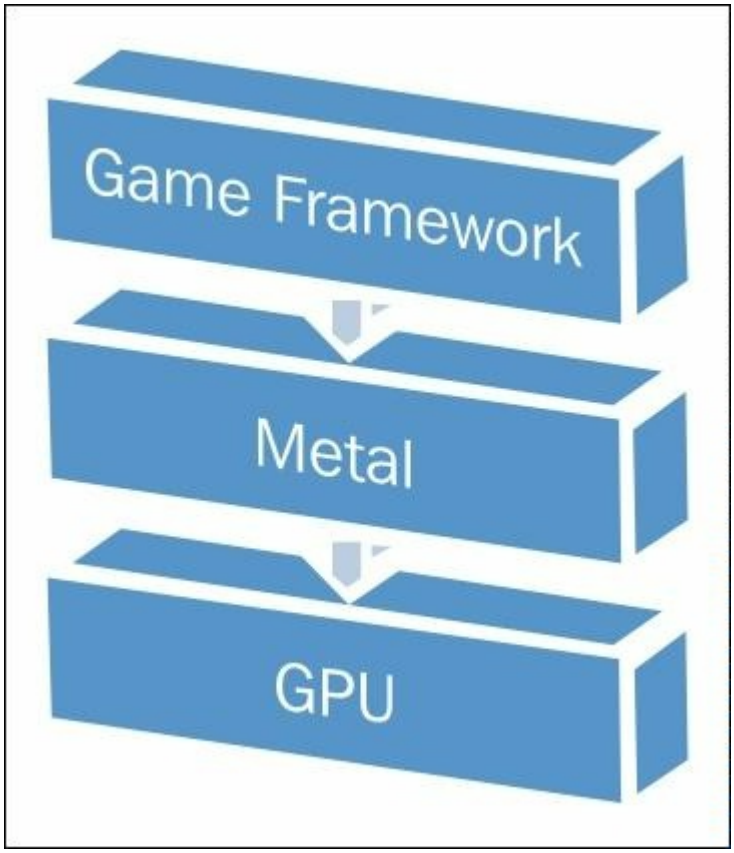
Introducing Metal

Metal is Apple's new low-level graphics API. With it, you can communicate with the GPU directly and conduct graphics, as well as other computational operations. Using Metal, you can create 2D and 3D apps or games by developing your own custom framework from the ground up, without relying on pre-existing frameworks such as SpriteKit and SceneKit.

Metal, like any other graphics API, has a graphics pipeline with programmable shaders. You are also able to allocate the memory, including buffer and texture objects. Metal also has it own shader language for compiling and applying the shaders.

But why do we need Metal when there is SceneKit? In SceneKit, SpriteKit, or any other game development framework, Metal first interacts with a graphics library, such as OpenGL ES, which in turn transfers the information to the GPU. Well! With Metal, you have the absolute power to directly communicate with the GPU and create your own framework as per your desire so that you can create a more optimized game.

A framework always sits above OpenGL or Metal, then communicates with the graphics processor, as shown in the following diagram. With Metal, you can eliminate one layer and talk directly with the GPU:



You also have the freedom to create your own tools as per your own specifications, from the ground up. Further, you can access the memory to get the maximum juice out of the processor.

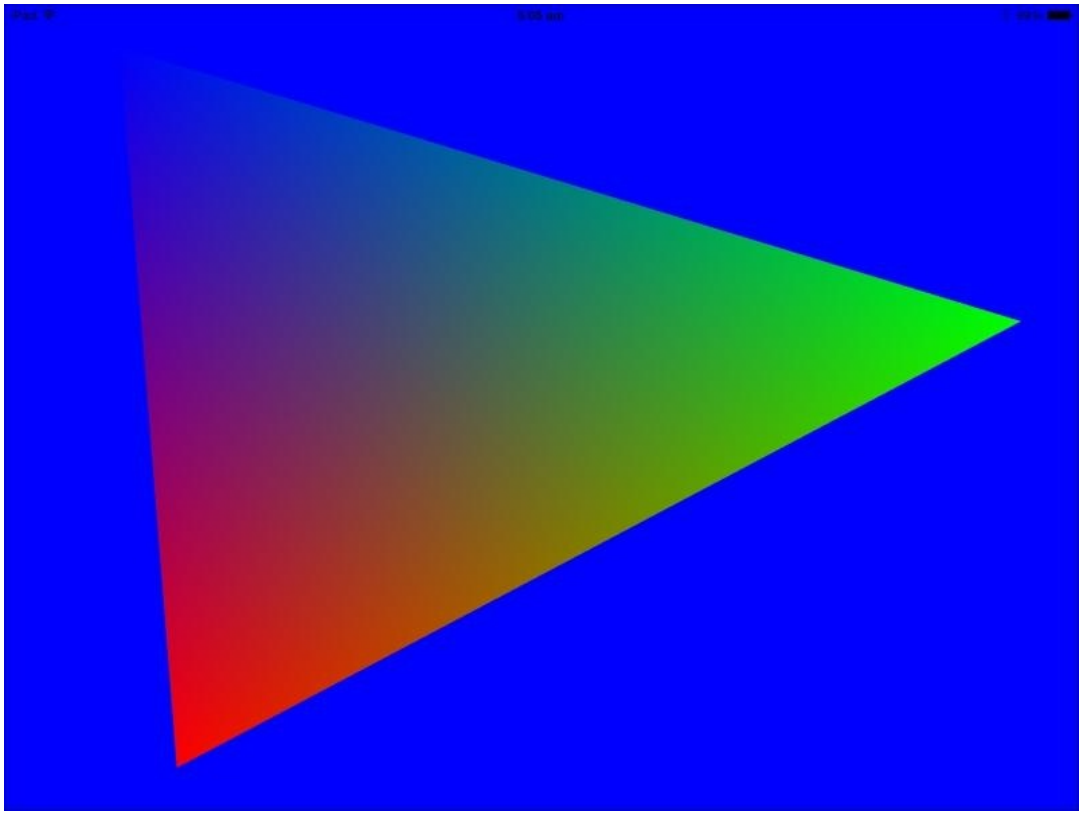
However, you might still say; "Well, OpenGL ES also provides all of this." That is true, but OpenGL ES is cross-platform and will work on other devices running many other operating systems. So, you can imagine the amount of extra material in OpenGL ES that is there to support all the devices and operating systems. This is where the difference between Metal and OpenGL ES lies.

Metal is very specifically written for the Apple range of devices and OS. In fact, at the time of writing this book, it is highly constrained to what Apple devices and OS are required to run it. You need an Apple device that has an A7 or A8 chip running iOS 8 to run games made with Metal. Moreover, if you want to test the game during development, you can't run it on a simulator, as it is not supported. You will require an actual device to run it, so you will require at least an iPad Mini 2, iPad Air, or iPhone 5S just to test your game.

With Metal, developers are seeing at least a 30 percent increase in performance as compared to the same game made with OpenGL ES running on the current Apple devices. So, if you are planning to create a 3D game just for the latest range of devices of Apple, it is highly recommended that you start creating your games using Metal.

You can create a Metal project and build it on your device to see the output. Since a lot of concepts will have to be covered to even start explaining all of the code on the screen—and since we will be covering these in detail later in the book—it is better that we go through the code later, after we understand the meaning of terms such as vertex buffer, frame buffer, and index buffer, and the difference between each of them.

If you have a device connected and have downloaded the developer license from the Apple Developer portal, you can click on the play button to build the app. Once the project is built, you will see the following screen:



To better understand Metal or OpenGL and appreciate the effort it actually takes to render even a triangle on the screen, you will need an understanding of what is called a **graphics pipeline**.

The graphics pipeline

If we look at any game development framework, we will realize that there are at least two functions that are present in almost every class. The first function is `init` or the `start` function, and the second function is the `update` function.

The `init` function or the start function is called once, at the beginning, to initialize all the variables of the class. On the other hand, the `update` function is usually called 60 times per second to update the position of the objects or detect collisions between objects.

Apart from initializing and updating the position, there is also another function that needs to be called along with the `update` function. Like a wiper on your windscreen, it needs to erase and redraw the content on the screen. This is called the `draw` function. In some frameworks, it is the third function, called `draw()`.

Every time this function is called, it is termed as a *draw call*. In SceneKit, we saw this word. But where is this function in SceneKit or SpriteKit? In both of these frameworks, similar to Cocos2d and ActionScript, there is something called a display list. Whenever you add an object using `addChild`, that object gets added to the display list.

A display list is like any array in which you can add or remove objects. The display list loops through all the objects that are added to it, and then draws all objects on the screen.

So, you don't have to worry about calling the `draw` function; you just add the object to be drawn by calling `addChild`. Otherwise—that is, if you don't want the object to be rendered—you can call the `removeChild` function to remove that object from the display list so that it won't be drawn to the screen.

The process of displaying things on the screen is known as a graphics pipeline. This is a step-by-step process of converting the vertex data and texture data that is provided to the GPU so that an object can be displayed and manipulated on the screen. Let's look at this process in detail.

Shown here are the stages in a render pipeline:



Let us look at each stage in detail:

- **Vertices**: You can provide either individual vertices with coordinates of each of the points, or the mesh itself. In the case of the jet, we provided the `.dae` file. The vertex information provided will contain the *x*, *y*, and *z* positions in 3D space. Apart from just position information, some of them may also contain information such as color, surface normal, and texture coordinate information. All of this information is stored in a sequence of memory space called a buffer, similar to an array.

  Pieces of vertex information, such as position and color, are fed to the GPU as attributes. Attributes can be imagined as a set of properties of a particular vertex. So, there are position attributes, color attributes, and so on. All of these attributes are provided to the GPU to start generating the object.

- **Generating primitives**: The vertices position provided in the position buffer are now connected to form small triangles, which in turn form the mesh of the object. These triangles are formed based on the sequential vertex information provided as an attribute.
- **Vertex/Geometry Shader**: The information is then passed to the vertex shader. Vertex shaders are programmable by using a shader language. The language is similar to C. Using this language, we can change the position, causing the object to move, scale, and rotate, just as our `update` function does, within the game loop.
- **Rasterization**: After knowing which vertex falls where, based on the locations of the polygons created by the vertices, the GPU starts drawing the pixels between the points.
- **Pixel/Fragment Shader**: As in the vertex shader, where you were able to do vertex modification, pixel shaders enable you to perform pixel-based operations. So, since this is also a shader, you know that it is programmable as well. Using pixel shaders, you can create effects such as changing the color and transparency of texture provided.
- **Testing and Mixing**: This is the stage where the GPU will check whether the pixel should be actually displayed on the screen. Suppose there is an object in front of the current object, and the current object should be only partially visible. The GPU will put only those pixels on the screen that should actually be visible, and will disregard the overlapping pixels.
- **Frame Buffer**: Buffer? So is this a memory space again? Yes. Before the image gets finally displayed on the screen, the whole image is first stored in a memory space called the frame buffer. When all of the computation is done and the final image is ready to be displayed on the screen, the data from the frame buffer is taken and then displayed on the screen.

If you don't understand all of this, it is completely fine. When we take a look at creating objects and displaying them on the screen in Metal, we will be walking through these steps practically, so don't worry about them.

Summary

There are a lot of basics and features of Xcode 6, and we covered them in this chapter. You saw how to become an Apple developer so that you can test the games that you are going to build on the device. Later, you will be uploading a game to the App Store.

We looked at how to create a playground file and project folders for different game technologies such as SceneKit, SpriteKit, and Metal, which we will be using in the forthcoming chapters. Finally, we looked at the render pipeline that is widely used to render stuff on the screen in any device.

In the next chapter, we will take a look at the interface of Xcode and the tools and features it provides.

Chapter 2. Swift Basics

Okay, let's begin. But before we dive into Xcode, you need to learn Apple's new programming language—Swift. For people coming from a JavaScript background, this will look very familiar. Apple has taken the best parts and practices from both scripting and low-level languages, and integrated some amazing features of its own to make Swift easy for anyone to get into, and start coding. As we go further, we will see that Apple has also added a few features that really make Swift shine compared to other languages.

As we saw in Chapter 1, *Getting Started*, we will be using Xcode's playground to do all the coding. Playground is a very versatile tool. Whenever you create a new file, you can start working on it immediately without any further setup. It has an inbuilt **Results** panel on the right that will compile your code in real time every time you make changes to the file. There is also an **Assistant Editor** window, which will give a graphical representation of the changes we make in the code.

In this chapter, we will start with the basics of the Swift language, from declaring variables to conditional statements, loops, arrays, functions, and classes. So, let's get started. Create a new playground file, name it whatever you like, and open it. If you have created a file in the first chapter, you will already have a file in the `_Playgrounds` folder that you can open.

The following topics will be covered in this chapter:

- Data types and operators
- Statements
- Arrays and dictionaries
- Functions
- Classes
- Optionals

Variables

If you have some experience in coding in other languages, you should know what a variable is by now. If not, then know that a variable is something that holds a value that can be changed at any time. This is why it is called a variable.

In Swift, you can define a variable using the `var` keyword, similar to JavaScript. So, if I create a new variable called `age`, then I type `var age`, and that is all; a variable is defined.

Veteran C coders will notice that I missed the semicolon at the end. Well, it is not required in Swift, but if you want to use the semicolon, you are most

welcome to do so. I do it out of habit and also because it is good practice. This is because after coding in Swift for a while, if you don't use colons and then use a C-based language, you will get errors all over the place.

But wait! There is an error. Swift is not able to figure out implicitly what data type `age` is; that is, is it an integer, a float, a Boolean, or a string? For Swift to implicitly assign a type, you have to initialize the variable. I can depict the age of a person as an `int` variable of value `10`, a `float` variable with the value of `10.0`, or as a `string` variable with the value of `Ten`. Depending on the variable type you assign, Swift will know that the variable is an `int`, `float,` or `string` type.

I am going to assign an `age` variable of value `10`. In the **Results** panel, you will see that the value of the variable is printed as **10**. This is the value that is stored in the `age` variable as of now. So, if you just type `age` in the next line, you will find the result displayed on the right side.

If you add `1` to `age`, you will see the result as **11**. This has not changed the value of `age`; only the line is evaluated and displayed. If you want to change the value of `age` to `11`, you can use the shorthand `age++` command to increment the value, as you do in Objective-C or any other C-based language.

Once a variable has been assigned a certain type, you cannot assign values of some other type. So now, if you try to assign `11.0` or `Eleven`, it won't be accepted and you will get errors.

Okay, we saw how to assign an `int` variable, but how do we assign the `float`, `string`, and `bool`? This can be done as shown in the following code snippet:

```
var height = 125.3
var name = "The Dude"
var male = true
```

What if I don't want to initialize a variable? We can tell Swift the type of the variable explicitly as well. We can tell Swift that a variable is of a certain type by telling it the variable type that we would like to assign. We do this by adding a colon and the type after the name of the variable, as shown in the following code snippet:

```
var age:Int
var height:Float
var name:String
var male:Bool
```

Note

Instead of the `var` keyword, you may have seen this new keyword called `let`, which is very common in Swift. If you don't want a variable's value to change throughout the code, then you should use the `let` keyword instead of `var`. This will also make sure that the value remains constant even if you change the value stored by `let` by accident. These are also known as constants in other languages, which you can define using the `const` keyword, but in Swift, you use the `let` keyword.

In the preceding example, you know that no matter what, `male` remains a male throughout his life. So, `bool male` will remain `true` and remain constant; it will never change (well, unless the guy is really not happy with his gender). Once it is set to `true`, we can't change it to `false` even by accident later on. So, the syntax will change to the following:

```
let age = 10
let height = 125.3
let name = "The Dude"
let male = true
```

Once again, if you are not explicitly giving the data type, you will still have to initialize the variable. So, you might think that you didn't change anything. The results panel still shows the same value as it did when the variables were `var`. But try changing the values now. Suppose, for example, you incremented the `age` variable; try doing it now and you are certain to get an error.

So, if you know that a variable's value is bound to change, it is better to use `var`. Otherwise, you are safer using `let` to save yourself a whole lot of bugs in your code resulting from this.

Now that you have understood how to declare and initialize the variables, let's use these variables to conduct some operations using operators.

Operators

Operators are used in computer languages to perform various operations on variables. It could be an arithmetic operation (such as addition), a comparison operation to check whether a number is larger or smaller than the other, a logical operation to check whether a condition is true or false, or an arithmetic assignment such as increasing or decreasing a value. Let's now look at each type in detail.

Arithmetic operators

Computers were initially made to perform numerical operations such as addition, subtraction, multiplication, and division. In Swift also, we have operators such as `+`, `-`, `*`, `/`, and `%`.

Let's declare two variables, `a` and `b`, and initialize them with values `36` and `10`. If you want to make a single-line initialization in Swift, you can do it as shown in the following snippet of code:

```
var a = 36, b = 10
```

You can also use the semicolon to separate them, as shown here:

```
var c = 36; var d = 10
```

This is useful if you want to initialize variables of different types. Now let's use operators between the variables. The result should be `46`, `26`, `360`, `3`, and `6`. Notice that the result of dividing an `int` variable with an `int` variable, is an `int` variable, instead of a `float` variable such as `3.6`:

```
a+b // 46
a-b // 26
a*b // 360
a/b // 3
a%b // 6
```

To get the exact result, we will need to type cast the variables. Type casting is done very similarly to how it is done in other languages; the type that you want to convert the variable to is added prior to the variable in brackets, as shown here:

```
Float(a)/Float(b) //3.5999999
```

Comparison operators

Logical operators are used to check whether a variable is equal to, less than, greater than, less than or equal to, or greater than or equal to the other variable:

```
a == b //false
a<b //false
a>b //true
a<=b //false
a>=b //true
```

Logical operators

Similar to other languages, you can use the `&&` sign to check for the logical AND condition and `||` to check the OR condition between two operations. These two are used to check whether both expressions satisfy their conditions or only either of the conditions is true:

```
a==b && a > b // false
a==b || a > b // true
```

In the previous example, we know that `a` is not equal to `b` but `a` is greater than `b`. So, the first statement in the preceding example is `false` (because both its parts aren't `true`), but the second statement holds `true` because at least one condition is `true`.

Arithmetic increment/decrement

Just as we increased the age in the first example by adding a `++` sign at the end of the variable, we also assign a `--` sign to decrement the value of the variable.

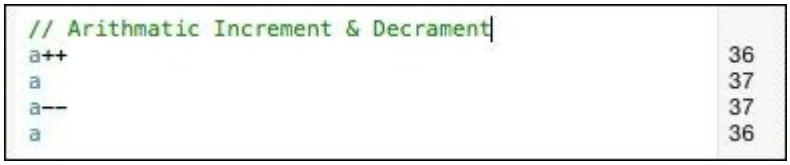
> **Tip**
>
> Keep in mind that `a++` is different from `++a`. They are called post- and pre-increment operators respectively.

In the following example, we start with a variable, `a`, with an initial value of `36`.

If you do `a++`, it is called post-increment. It will first display the result and then increment the value of `a`.

So, if you do `a++`, which should have incremented the value of `a`, the result will still show up as `36`, suggesting that the value has still not been

incremented yet. Yet, immediately in the next line if you ask it to output the value of `a`, it will show up as `37`.



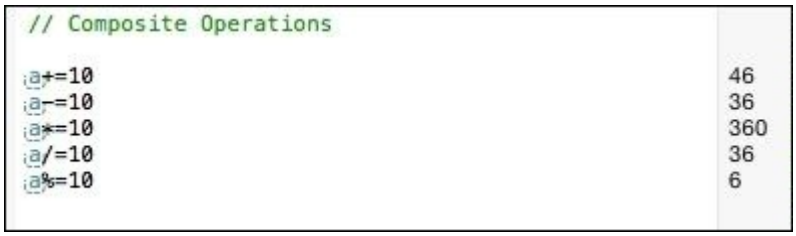
If you do `++a`, it is called pre-increment. It will increment the value first and then show the output. So in this case, it will show `37` in the same line where you incremented the value of `a`, and if you ask it to show the value of `a` in the next line, it will again show `37`, like this:



This is just to keep in mind for the future, in case you encounter any bugs in the code and wonder why the correct value of your expression is not getting displayed.

Composite operations

We can also perform composite operations to increment, decrement, multiply, or divide by a much higher number, like this: `a+=10`, `a-=10`, `a*=10`, `a/=10`, and `a%=10`, which are shorthand ways of doing `a = a+10`, `a = a-10`, `a = a*10`, `a = a/10`, and `a = a%10` respectively. So, if you replace `a+=10` with `a = a + 10`, it will still give the same result. This is done for the sake of optimization and readability of code.



Statements

Statements are of two kinds in any programming language—decision making statements and looping statements.

Decision-making statements

Decision making statements are of the following types: `if`, `if else`, `else if`, and `switch`. This is very standard in any programming language. Let's look at the `if` statement first and see how the syntax differs from other C-based languages.

The if statement

In Swift, the `if` statement is written like this:

```
if a > b {

    println("a is greater than b")

}
```

Immediately, the guys using C will be like, "Blasphemy!! No brackets!!" Yes, in Swift for the sake of simplicity and readability, you don't have to use the brackets. If you want to, you can still use them, and you won't get any compile errors.

But you absolutely must enclose the statement in the curly braces. Even if it is a single line statement, it is absolutely mandatory. If you don't, you will get compile errors for sure.

> **Note**
>
> Here's a small note about logging code on the screen: we will be using this feature quite extensively when we start developing the game to make sure that the program is doing exactly what we want it to do. So, we will be logging statements to check for logical errors.
>
> In Objective-C, we would use something such as `NSLog(@"Print Stuff to Screen")` or `NSLog(@"My age is: %d", age)`. In Swift, this is done a bit differently. For one, you don't need to put the `@` sign in front of any string to log it out. Secondly, to print the values, we have to use `\()` with the variable in the bracket:

```
println("\(a) is greater than \(b)")
```

```
//logging text

println("\(a) is greater than \(b)")
```

```
"36 is greater than 10"
```

The same can be applied to strings:

```
let x = "Dude"
let y = "Name"
println("\(x) is my \(y)")
```

```
let x = "Dude"
let y = "Name"

println("\(x) is my \(y)")
```

```
"Dude"
"Name"

"Dude is my Name"
```

The if else statement

Similar to the `if` statement, the `if else` statement is written as shown here:

```
if a < b {

    println("\(a) is smaller than \(b)")

} else {

    println("\(a) is greater than \(b)")

}
```

Instead of checking whether `a` is greater than `b`, we are checking otherwise, and now it will print **30 is greater than 10** by entering the `else` statement.

The else if statement

Similar to the `if` statement where we didn't put the brackets around the condition, in `else if`, we are not required to put the brackets around the condition:

```
if a < b {

    println("\(a) is smaller than \(b)")

} else if a > b {

    println("\(a) is greater than \(b)")
```

```
}
```

So here, we're checking whether `a` is greater than `b`, then print the following statement (from the code snippet) instead of checking for just the `else` bit.

Conditional expressions

Instead of taking 10 lines of code for checking such a simple statement, you can use the conditional expression statement to do the work in one line. This is more popular if you are just checking for `if-else`:

```
a > b ? a : b
```

Here we check whether `a` is greater than `b`. If `a` is greater than `b`, then the expression evaluates to `true` and the output becomes `36`. Otherwise, the output becomes `10`.

The switch statement

Switch statements in Swift are a little different from the way they are in other languages. Like the `if` statement, the brackets around the variable, or expression, are not required, but that's not the only difference.

All statements need to print some value, or some condition needs to be checked. You cannot have an empty case; that will give an error. Secondly, as all statements will be evaluated, a break is not necessary at the end of every line. In other C-based languages, all lines will be evaluated and executed, unlike the `switch` statement in Swift, where only the valid cases will be executed. Finally, the cases need to be exhaustive. This means that there needs to be a default at the end so that if none of the cases match, then a default value or statement gets thrown.

Let's look at an example of switch-case:

```
var speed = 30

switch speed {

case 10 : " slow "
case 20 : " moderate"
case 30 : " fast enough"
case 40 : " faster "
case 50 : " fastest "
default : " value needs to >= 10 or <= 50"

}
```

Here, a new variable called `speed` is created. Based on the value of the speed, the system will print whether the value is slow or fastest. If `speed` is not a number, or the value is not between `10` and `50`, the default message will be printed. So in this case, as the value matches with `case 30`, it will print **fast enough**.

Instead of fixed values for case statements, you can also provide a range of values for which the statement will be `true`. So for example, if the value is between `0` and `10`, you would want the output to be **slow**. You can do so by giving the range from `0` to `10` as `0…10` in the first case. It is important that you put three dots between the values because they form an operator.

So now, if you change the value of speed to anything between `0` and `10`, the output will be **slow**. Similarly, the range can be used with other case statements:

```
switch speed {

case 0...10 : " slow "
case 20 : " moderate"
case 30 : " fast enough"
case 40 : " faster "
case 50 : " fastest "
default : " value needs to >= 10 or <= 50"

}
```

Looping statements

Looping statements are used to execute a particular block of code infinitely, for a said number of times, or until a certain condition is satisfied. As in any other language, we have the `while`, `do while`, and `for` loops, and we have a `for each` loop as in C# and C++11.

The while loop

In a `while` loop, we first give a condition. If the condition holds `true`, the block following the code will keep on executing.

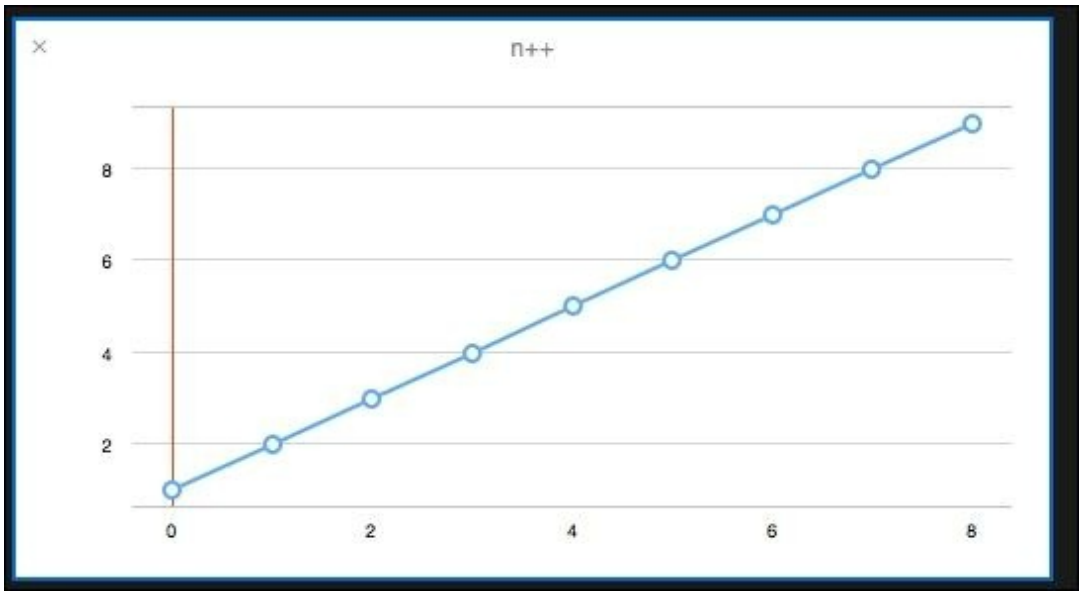
For this example, we create two variables, `n` and `t`. We set `n` equal to `1` and `t` equal to `10`. In the condition, we increment the value every time the code is executed:

```
var n = 1, t = 10

while n < t{

n++
```

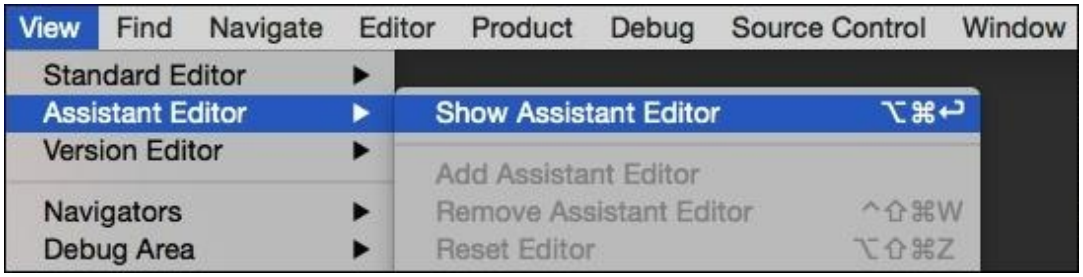
```
}
```

Once again, there is no need to put the parentheses around the condition. But you will see that, in the results panel, nothing printed except nine times.

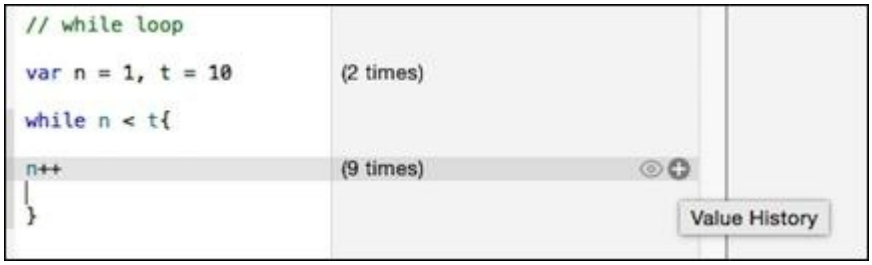
There is a + icon, and there is another icon next to it. Pressing it will open a new pane. This is called **Assistant Editor**. In it, you will see a graph. This shows the increment of the value of `n` over the period of time the loop was running. It starts from `1` and goes all the way to `9`, since we told the loop to run only while the value of `n` is less than `10`. You can move the mouse arrow onto the nodes to know the value at each node. Alternatively, there is a scroll bar to scroll to the graph.



You can also open **Assistant Editor** in Xcode by going to **View** | **Assistant Editor** | **Show Assistant Editor**:



Click on the value history icon to open the graph:

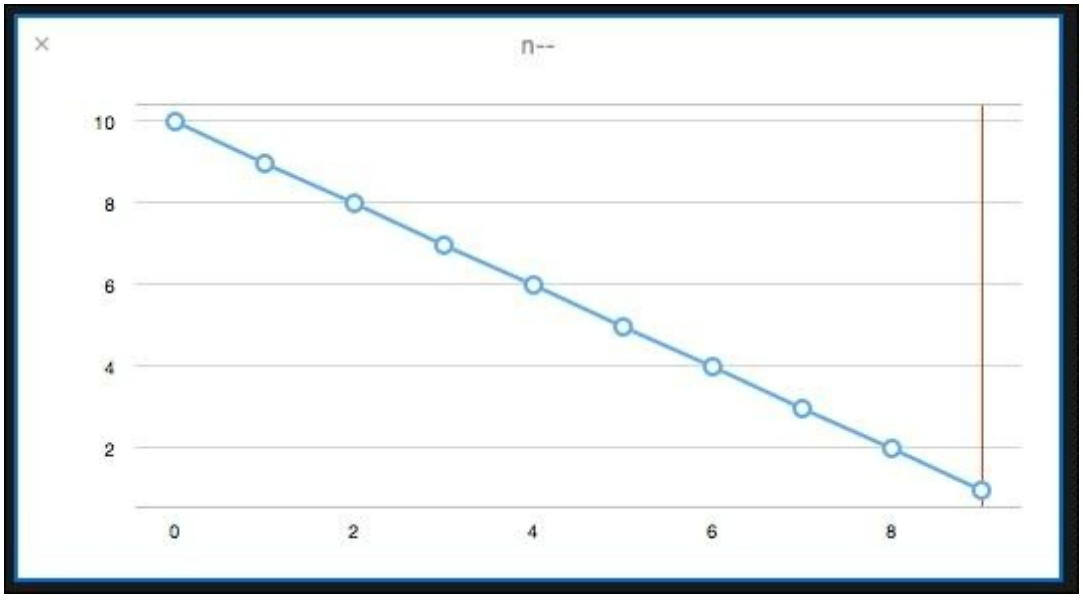


The do while loop

Similar to the `while` loop, the code block of the `do while` loop gets executed while the condition is `true`. However, in this case, the block executes at least once and then the condition is checked.

Here, we are reducing the value of `n` by 1 every time it enters the loop and checking whether the value is greater than `0`. If it is, then the code block gets executed. The value of `n` from the previous piece of code is `10`, so it starts from `10` and goes back to `1`. Again, this is plotted in a graph, which can be seen by clicking on the icon on the results panel:

```
do{
    n--
} while n > 0
```



Tip

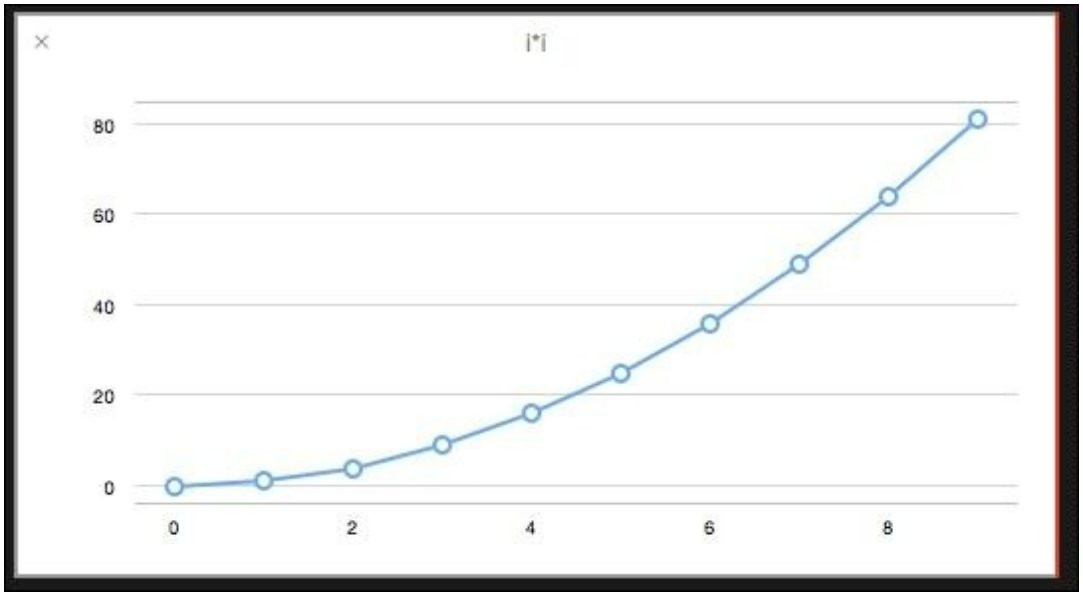
Be carful to make sure that the condition will be met. Otherwise, it will lead to an infinite loop, causing your system to become unstable.

The for loop

The `for` loop can be written as shown here. Even without the brackets, it will work fine:

```
for var i=0; i < 10 ; i++ {

    i*i

    }
```

Here, instead of adding `1` every time, we multiply the value of `i` to itself to get a curved line in the graph, ranging from `0` to `81`.



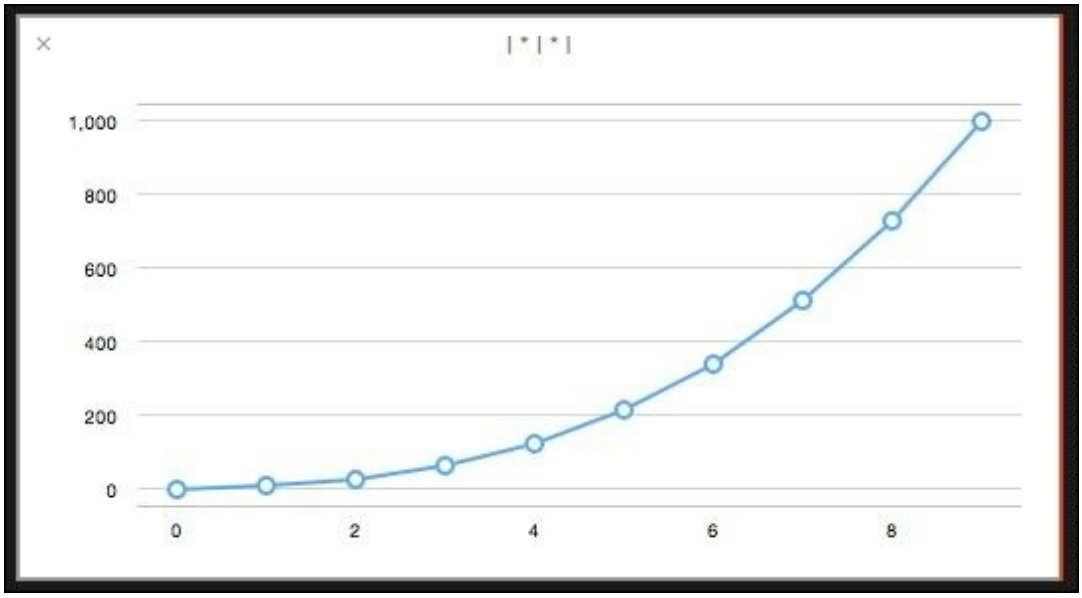
The for in loop

Like the `for each` loop in other languages, you have a `for in` loop in Swift. It goes through each item in the loop and executes whatever has been stated by the code. Here, unlike other languages, it is not necessary to specify the type of variable. Swift will automatically understand the variable type according to the list of items you want it to loop through. For example, looping through a list of integers can be written as follows:

```
for l in 1...10{

    l * l * l
```

```
}
```

Here, even though we never mentioned in the code that `l` is an integer type, Swift automatically understood it, according to the list of values we gave in the code. Since the type of the list given is an `int`, the value of `l` automatically gets assigned the type `int`, and the output is provided:



Additionally, while looping through an index of an array, we would want to start from `0` and go to the last but one value. The same can be achieved using `..<` instead of the `…` operator when specifying the range. So, in the preceding example, if we have to go from `0` to `9`, we can rewrite the code as follows:

```
for l in 0..<10{
    l * l * l
}
```

This is obviously not limited to numbers; we can loop through any data type. For example, we can loop through the characters in a string like this:

```
for c in "string"{
    println("character \(c)")
}
```

Here again, it was never explicitly mentioned that the type of the variable is `char`, and yet Swift was able to implicitly judge that `c` should be of the `char` type. In the console, each character will be displayed as follows:

```
character s
character t
character r
character i
character n
character g
```

Arrays

An array is a consecutive block of memory that holds a certain type of data. It can be a predefined data type, such as `int`, `float`, `char`, `string`, or `bool`, or it can be user-defined. Also, arrays are zero-based, which means that the first object in the array is `0`.

Arrays can be either of the `var` or `let` type. If we create an array of the `var` type, then we can change, add, or remove the objects from the array similar to how we have `MutableArray` in Objective-C. If you want an `ImmutableArray` array in Swift, instead of `var`, use the `let` keyword. Arrays in Swift can be declared and initialized as follows:

```
var score = [10, 8, 7, 9, 5, 2, 1, 0, 5, 6]

var daysofweek = ["Monday", "Tuesday",
"Wednesday"]
```

Now, if we want to declare an array and initialize it later, we can declare it as shown in the following snippet of code:

```
 var score : [Int]

var daysofweek : [String]
```

Here, you will have to provide the data type that the array will be storing. It is very similar to defining a regular variable; it's just that the type here is surrounded by square brackets. Now we can initialize it as we did earlier:

```
score = [10, 8, 7, 9, 5, 2, 1, 0, 5, 6]

daysofweek = ["Monday", "Tuesday", "Wednesday"]
```

To get an item at an index, you can use the variable name with the index number in square brackets:

```
Score[0] // output: 10

Score[5] // output: 2
```

Looping through arrays

To loop through an array, we can use the `for in` loop, as we saw in loops. Here, instead of passing the range, we provide the range itself, and as

before, the data type present in the array is automatically determined by Swift:

```
for myScore in score {

myScore

}


for day in daysofweek{

    println("\(day)")

}
```

Adding, removing, and inserting objects into arrays

To add items to the array, you can use the `append` method. So, to add values to the `score` or `daysofweek` array, we do it as shown in the following code snippet. This will add the object at the end of the array:

```
score.append(10)
daysofweek.append("Thursday")
```

You can remove the last object from the array by calling this:

```
score.removeLast()
```

If you want to remove an object at a particular index, you can use the following line:

```
score.removeAtIndex(5)
```

This will remove the element at the fifth index, which is `2` here. To insert an item at a particular index, we will have to provide the value and index at which the item has to be inserted:

```
score.insert(8 ,atIndex: 5)
```

Important array functions

Apart from functions such as `append`, `remove`, and `insert`, there are other built-in functions that we will be using quite often in games. The first is the `count` function, which tells us the number of elements present in the array:

```
score.count
```

The other function is `isEmpty`, which can be used to check whether an array has any items present in it or not:

```
score.isEmpty
```

Dictionary

Also known as a map or hash table in other languages, a dictionary—like arrays—is a data collection type. However, unlike arrays—in which each element in the array can be accessed only by an index number—in dictionaries, we will provide keys with which we will be able to access the element at a particular index. The values of `keys` and `values` can be `int`, `float`, `bool`, or `string` values. You cannot have duplicate keys, but you can obviously have duplicate values in the dictionary.

For example, let's look at the FIFA's country code list:

Code	**Country**
AFG	Afghanistan
ALB	Albania
ALG	Algeria
ASA	American Samoa
AND	Andorra
ANG	Angola
AIA	Anguilla
ATG	Antigua and Barbuda
ARG	Argentina

Code	Country
ARM	Armenia

The code in the left columns can be termed as **keys** and the **Country** column on the right represents the **value**. So, when I want to refer to Argentina, I can simply call ARG and Swift will automatically understand that I am referring to Argentina. Here's another example: when we say, "get the value for the `ATG` key," the program will know that we are referring to Antigua and Barbuda.

The advantage of a dictionary is that the data doesn't have to be sorted. If we know that the key-value pair exists in the dictionary, irrespective of where it is in the list, we can get the value by asking for it by the key.

This syntax is similar to arrays. The only difference is that instead of giving one type of variable, we have to provide two during declaration, the first being the type of the key and the second being the type of the value.

Once again, you don't have to provide the types explicitly if you are already aware of the types of both `key` and `value`:

```
var countries = ["AFG": "Afghanistan",
                 "ALB": "Albania",
                 "ALG: Algeria"]
```

The key-value pair can be of any combination. This means that you can have a `key` variable of type `int` and `value` of type `string`; `key` of type `string` and `value` of type `string`, as shown in the example; or `key` of type `string` and `value` of type `int`; and so on. You cannot change or provide a different type after declaring the dictionary keys and values with a certain type. Also, you cannot add a key of type `int` if you've already declared that the keys will be of type `string`.

We can also explicitly tell the types of the key-value pair, as shown here:

```
var states:[String: String]
```

This tells us that the keys will be of type `string` and the values associated with those keys will also be of type `string`. If we want to store the population for a set of keys, it can be done as follows:

```
var population: [String: Int]
```

Adding and removing objects from the dictionary

To add a new key-value pair to an existing dictionary, we can simply run one of these two lines of code:

```
countries["GER"] = "Germany"
countries.updateValue("Netherlands","NED")
```

Beware, however, because if the key already exists, it will overwrite the current value it holds.

If you want to remove a key-value pair, you can use either of the following statements to delete it:

```
countries["ALB"] = nil
countries.removeValueForKey("AND")
```

Now, if we try to access the key-value pair, we will notice that it has removed not only the value stored by the key, but the key as well.

Looping through items in the dictionary

Once again, we can use the `for-in` loop to access the data stored in each key-value pair, but instead of just one value, we will have to provide two values in parentheses, separated by a comma for separation of the key and value:

```
for (code, country) in countries{

println("\(code) is the code for \(country) ")

}
```

The console output will be as follows:

```
GER is the code for Germany
AFG is the code for Afghanistan
ALG is the code for Algeria
NED is the code for Netherlands
```

Dictionary functions

Similar to arrays, the dictionary also has inbuilt functions to determine the number of key-value pairs present in it. It also has an `isEmpty` function,

which can be called on the dictionary to know whether it contains any key-value pair at all:

```
countries.count
countries.isEmpty
```

Functions

A function is a block of code that is used to perform a certain task. It can be used to assign a block of reusable code that you can call again and again without rewriting the code every time to perform that task.

Simple functions

Functions in Swift are written as follows:

```
func someFunction(){

Println(" performing some function ")

}
```

So here, before a function, we will need to use the `func` keyword, followed by the name of the function, the parentheses, and the open and closed curly braces.

Calling a function here is very similar to any other language, that is, the name of the function followed by parentheses:

```
someFunction()
```

Passing a parameter

Now, to perform some task on a variable passed to a function, we first need to pass a parameter into the function. This can be done as follows:

```
func printText(mtext: String){

    println("Print out \(mtext)")

}
```

Here, we have to provide the input variable type within the parentheses. We cannot provide an integer here when the function is expecting a string, as it takes the type explicitly.

To perform the function, call the function and, within the parentheses, include the text that you want to perform the function on:

```
printText("Hello Function")
```

It should be noted that the value passed to a function is a constant by default. This means that even though we don't specify whether `mtext` is a variable or constant, it will remain a constant. So, you will not be able to make any modifications to it within the function.

If your code requires the type to be a variable instead of a constant, you need to specify this while creating the function:

```
func printVarText(var mtext: String){

    mtext = "Text Changed"

    println("Print out \(mtext)")

}
```

Passing more than one parameter

We can obviously pass more than one variable. Here, you will need to separate the variables types with a comma, like this:

```
func calcSum(a: int, b: int){

let sum = a + b

println("The sum of the numbers is: \(sum) ")

}
```

Here, we pass two numbers, `10` and `15`, to the function. We perform their addition, store the value in a constant called `sum`, and print it to the console by calling the following function:

```
calcSum(10, 15)
```

Returning a value

For a C++ user, this might look a little strange. Also, people not comfortable with pointers might freak out looking at the syntax of a function returning a parameter as it uses the pointer operator in Swift. Don't fret! This has absolutely nothing to do with pointers. It is just a way to show that this function will return a data type, and that's all.

The function is written as usual, but at the end, we specify the return type after the parentheses by typing the dash and greater than sign, and then typing the return type.

Here, we are performing a multiplication using a function that takes in two `int` values and returns an `int` value. We calculate the value by performing the multiplication operation, store the value in a constant called `mult`, and then return the value.

The function is called, and the result is stored in a variable and then printed to the console:

```
func mult(a: Int, b: Int) -> Int{

let mult = a * b

return mult

}

let mVal = mult(10, 20)

println("The Multiplied valued is = \(mVal)")
```

Here, the console will output the multiplied value—`200`.

Default and named parameters

If we want to assign a default value to the parameters in the function, we can certainly do it, as shown here:

```
func defMult(a: Int = 20, b: Int = 30) -> Int{

let mult = a * b

return mult

}

let dVal = defMult()

println("The Multiplied valued is = \(dVal)")
```

Here, we are assigning `20` and `30` as default values. We simply call the function to get an output of `300`.

But what if we want to modify only one value and keep the other value as default? In that case, we will assign a value by name, while calling the function. So, if we want the value of `a` to be `80` instead of the default value, we will do it like this:

```
println("The Multiplied valued is = \(defMult(a:
80))")
```

If we wish to change both the values, we can also do that, as follows:

```
println("The Multiplied valued is = \(defMult(a:
80, b: 50))")
```

Returning more than one value

In Swift, we can return more than one value using tuples.

Note

Tuples are variables that can hold two values, like an array. In fact, a variable with three values in it is called a 3-tuple.

A tuple can be initialized as follows:

```
var person : (int, string)
```

Individual values can be initialized as shown here:

```
person.0 = 23
person.1 = "The Dude"
```

You can also name your variables for convenience. Then you can access the names to assign values instead of the index:

```
var person2:(age:Int, name: String)
person2.age = 23
person2.name = "The Dude"
```

In Swift, you will be able to pass the actual value, perform some action on the variable, and then return the variable. When returning two values, we will need to provide the return type of both values, separated by a comma:

```
func getAreaAndPerimeter(a: Int, b: Int) ->(Int,
Int){

  let area = a*b

let perimeter = 2a+ 2b

return(area, perimeter)
```

```
}
```

So, in the preceding function, we take in two integers and calculate the area and the perimeter of a rectangle. Then we return both the values:

```
let value = getAreaAndPerimeter(40 ,80)
```

The value is stored in a constant called `value`, and to output the values to the console, we use the dot operator. It is used with `0` to get the first value and with `1` to get the second value:

```
println("Area is = \(value.0) and Perimeter is =
(value.1)")
```

This is a bit cumbersome, as we will have to remember that the first value returned is the area and the second value is the perimeter. But this can also be fixed quite easily in Swift. Just as we name the values while passing, we can also name the returned values. Then, instead of using index values, we can use the name itself to access the values:

```
func getNamedAreaAndPerimeter(a: Int, b: Int)
->(area: Int, perimeter: Int){
        let  area = a * b
        let  perimeter = 2 * a + 2 * b

        return (area, perimeter)
}

let namedvalue = getNamedAreaAndPerimeter(80 ,100)

println("Named Area is = \(namedvalue.area) and
Named Perimeter is = \(namedvalue.perimeter)")
```

Classes

Classes are quite easy to create in Swift compared to Objective-C or C++. There is no need for separate files such as an interface file and then an implementation file. Also, there is no property keyword used to define properties. All Swift files end with an extension of `.swift`.

Properties and initializers

We will have to use the `class` keyword when creating a class:

```
Class Character{

var name = "The Dude"

var health = 100

}
```

Also notice that there is no semicolon after the closing brace. To instantiate a variable of the `character` type, you can use the following line of code. There is simply no need of `alloc` as was the case in Objective-C:

```
var theDude = Character()
```

The `name` and `health` properties can be accessed using the dot operator:

```
theDude.name

theDude.health
```

To initialize the variable, we will need to use the `init` function. This is the constructor for the class, and it doesn't require the `func` keyword, which is required when creating any other function. So, if you want to initialize the `name` and `health` variables in the `init` function, you have to do it as shown in the following code snippet. Notice that you will need to explicitly provide the variable type for the `name` and `health` variables in this case:

```
class Character1{

    var name: String

    var health: Int

    init(){

      name = "The Dude"
```

```
        health = 100

    }
}
```

We can instantiate and access the properties of the class in the same way as before. We can definitely create as many custom initializers as we want. These will also use the `init` key word, and you can pass the type of variable in the parentheses and assign the value to the property:

```
class Character2{

var name: String
var health: Int

    init(){

        name = "The Dude"
        health = 100
    }

    init(name: String){

        self.name = name
        self.health = 100
    }

}
```

Notice that here, while assigning the variable passed to the property, we execute `self.name` to tell the code that we are assigning the name of the passed variable, to the `name` property of the class.

Also, while passing the value, we have to mention that we are passing it to the `name` argument:

```
var hero = Character2(name: "Hero")

hero.name
hero.health
```

Custom methods

We can obviously create custom methods in a class. Here, suppose we want the character to take some damage after being hit. For this, we will create a new method, as shown here. Add this block before the closing bracket of the class:

```
    func takeDamage(damage: Int){
```

```
        self.health -= damage

    }
```

Methods in a class are defined like any other function. Here, we are defining a method called `takeDamage`, and it takes in a variable called `damage` of type `int`. Now we can call this function on the `hero` variable we instantiated earlier, and pass `10` for the hero to take 10 points of damage. If we call the `health` property again, we will see that the hero's health has dropped by 10 points:

```
hero.takeDamage(10)

hero.health
```

Now let's add one more property to the class and call it `armour` of type `int`. Let the initial armor level of the player be `10`. Each time the player takes a hit, his armor also goes down:

```
func reduceArmour(damage: Int, armour: Int){

    self.health -= damage
    self.armour -= armour
}
```

We add a new function called `reduceArmour`. In it, we take in an additional parameter called `damage` and reduce the `armour` property of the class by that amount:

```
hero.reduceArmour(10, armour: 2)
hero.health
hero.armour
```

Now, when we want to call the method, we have to explicitly mention that the second value passed is for `armour`. When we call the `health` and `armour` properties, we can see that `health` is reduced by 10 again and `armour` of the hero is now reduced by 2.

Inheritance

Like any other object-oriented language in which we wish to reuse or extend an existing class, Swift also allows inheritance. To inherit from another class, we will need to use a colon and specify the name of the class we want to extend while defining the current class.

Tip

> Swift doesn't allow multiple inheritance.

Suppose we want to create a `Mage` class, and the mage has a property called `magic` so that she can do some "magic" damage along with regular damage:

```
class Mage: Character2{

    var magic: Int

    override init(name: String){

        self.magic = 100
        super.init()

        //self.name = name
        //self.health = 60
        //self.armour = 15
    }

}
```

Here, we create a new class called `Mage` and inherit it from the `Character2` class. We will need the `override` keyword to tell the code that we are overriding the `init` function of the `Character2` class, which is the superclass.

In Objective-C, we would usually call `super.init` first, but in Swift, we will need to initialize the `magic` property first, as this property doesn't exist in the superclass. So, we initialize it first and then call the `super.init` method.

If we create a new variable for a character called `vereka`, which is of type `Mage` and name her `vereka`, we will see that the name and other values assigned are of the `Character2` superclass instead of the name that we assigned:

```
var vereka = Mage(name: "Vereka")

vereka.name
vereka.health
vereka.armour
vereka.magic
```

For the new name to be assigned, we have to initialize it again after we call `super.init` in the `init` method of the `Mage` class. So, we uncomment the lines of code in the `Mage` class where we assign the name and the new

`health` and `armour` values to the `Mage` class. Now, the correct `name`, `health`, `armour`, and `magic` values will be displayed.

Access specifiers

In Swift, there are access specifiers that can be used to encapsulate classes, properties, and methods. Access specifiers provide read and write access to variables. Both the usual `public` and `private` access specifiers are present in Swift.

When creating a class, if a variable of a function is set to `public`, then any class can access that variable and modify it, but if it is set to `private`, then other classes cannot access that variable.

Apart from public and private access specifiers, there is also an additional specifier called `internal`. If you have used encapsulation in Objective-C or C++, you will be pleased to know that public and private variables work pretty much the same in Swift, and `internal` replaces the `protected` encapsulation type.

In Swift, the `internal` specification is the default specification type, unlike C ++, where the default specification is private if not specified.

Even though we didn't specify the access specifier in the `Character2` class, we were still able to access its variables and functions in the `Mage` class, as they are internal and can be accessed by the child class:

```
public class myCharacter {
         public var name: String
         private var age:Int
         var speed:Int

        public init(){
          self.name = "The Dude"
          self.age = 100
          self.speed = 20
      }
}
```

The `myCharacter` class and `name` variable are public. The `age` variable is private and speed is assumed to be internal because it is not specified otherwise.

Optionals

Optionals are a new data type in Swift. In all the cases covered so far in this chapter, we have always initialized a variable, whether we initialized it inline or in an initializer in a class. If we don't initialize the variable and start using it, we get an error saying that we haven't initialized it yet. The quick fix is to initialize it to `0` or `""`.

Suppose this variable is `score` and we have initialized its value to zero to get rid of the error. Now we try to retrieve the last score stored in GameCenter. For some reason, there is no Internet connection or Wi-Fi, and so we are not able to retrieve that player's last `score`. If we display the score on the screen, it will say **0** because that is what is stored in score. The player will be very confused and frustrated, as they are sure that the score was a lot more than zero in their last attempt. Now, how will we tell the player that we weren't able to retrieve the score because they forgot to pay the Internet bill?

In other words, how do we tell the program that there is no value in `score`? We can't even equate score to `nil` because we will get an error saying that it is not of type `nil`. For this, we put a question mark after the type. This means that if we assign a value, it will be of type `int`; otherwise, it should be treated as `nil`.

Now, at the time of displaying the score, we can check whether the value is `nil`. If it is, then we can print a message saying there is no Internet connection, and if `score` holds a value, we can print that as well:

```
var score:Int?

if score != nil {

    println("Yay!! Your current score is
\(score)")

}else{

    println(" No internet! Pay your bills on time
")
}
```

The preceding code will print from the `else` block, as the value is not assigned—it is still `nil`.

Now, if we assign a value, say `score = 75`, it will print the value but with the optional keyword along with the value in parentheses. This is because, since the type we specified is optional, it is letting us know the data type.

For it not to show the optional keyword while logging out, we need to "unwrap" the optional type. This is done by following the `score` variable in the `if` block with an exclamation, as shown in this line of code:

```
println("Yay!! Your current score is \(score!)")
```

The optional keyword will now disappear when logging the score.

You might not use optionals in the games you create, but if you do have to, you will at least not be surprised when you see one of them.

Summary

In this chapter, we saw some basics of the Swift language. This should get you up to speed with the language so that from the next chapter onwards, you will have a good idea of the syntax and will know exactly what is happening in the code. You might also revisit the first chapter now and look at the code from the SpriteKit and SceneKit project files to see whether you are able to make any sense of the code.

As the chapter says, this was just a basic introduction to the Swift language. This language is quite similar in certain aspects and quite different in others when compared to other languages. When we come across any of these differences in the following chapters, I will be pointing them out so that you will be well aware of the errors and will know how to get around them and understand them better.

In the next chapter, we will dive into Xcode, look at it in depth, and start from the basics of Xcode.

Chapter 3. An Introduction to Xcode

In the last chapter, we saw the Playground tool. It's a simple tool to test your code, but if you want to actually create an application and run it on a device or publish and distribute it through the Mac or iOS App Store, you need to create an Xcode project.

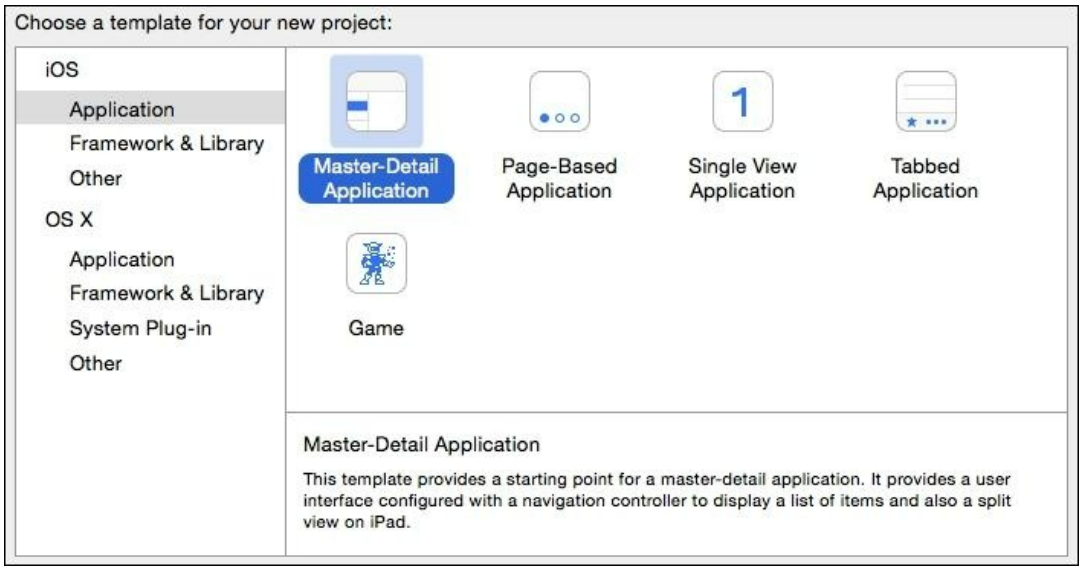
In this chapter, we will take a tour of Xcode. We will take a detailed look at the interface, create a very basic "Hello World" application, and make it run on the simulator and later on the device.

We will be covering the following topics extensively in this chapter:

- Xcode application types
- The Xcode interface
- Running the app on the device

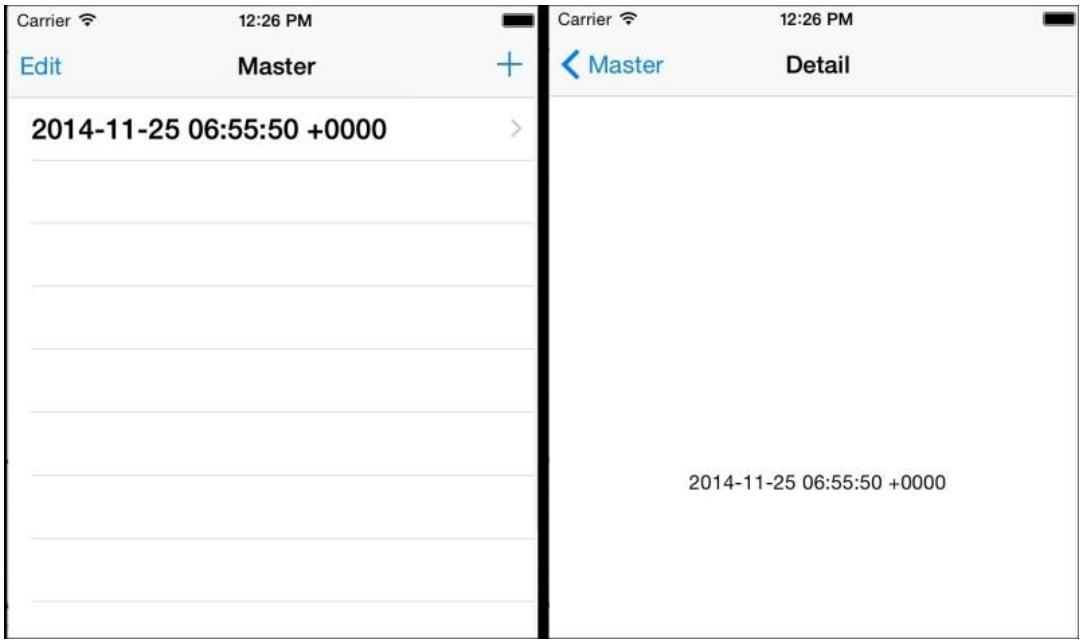
Xcode application types

Xcode comes with some predefined application types for creating some specific types of applications. They are the **Master-Detail Application**, **Page-Based Application**, **Single View Application**, **Tabbed Application**, and **Game** types, as shown in the following screenshot. We have already seen what is in the **Game** type, so let's look at the other types:



Master-Detail Application

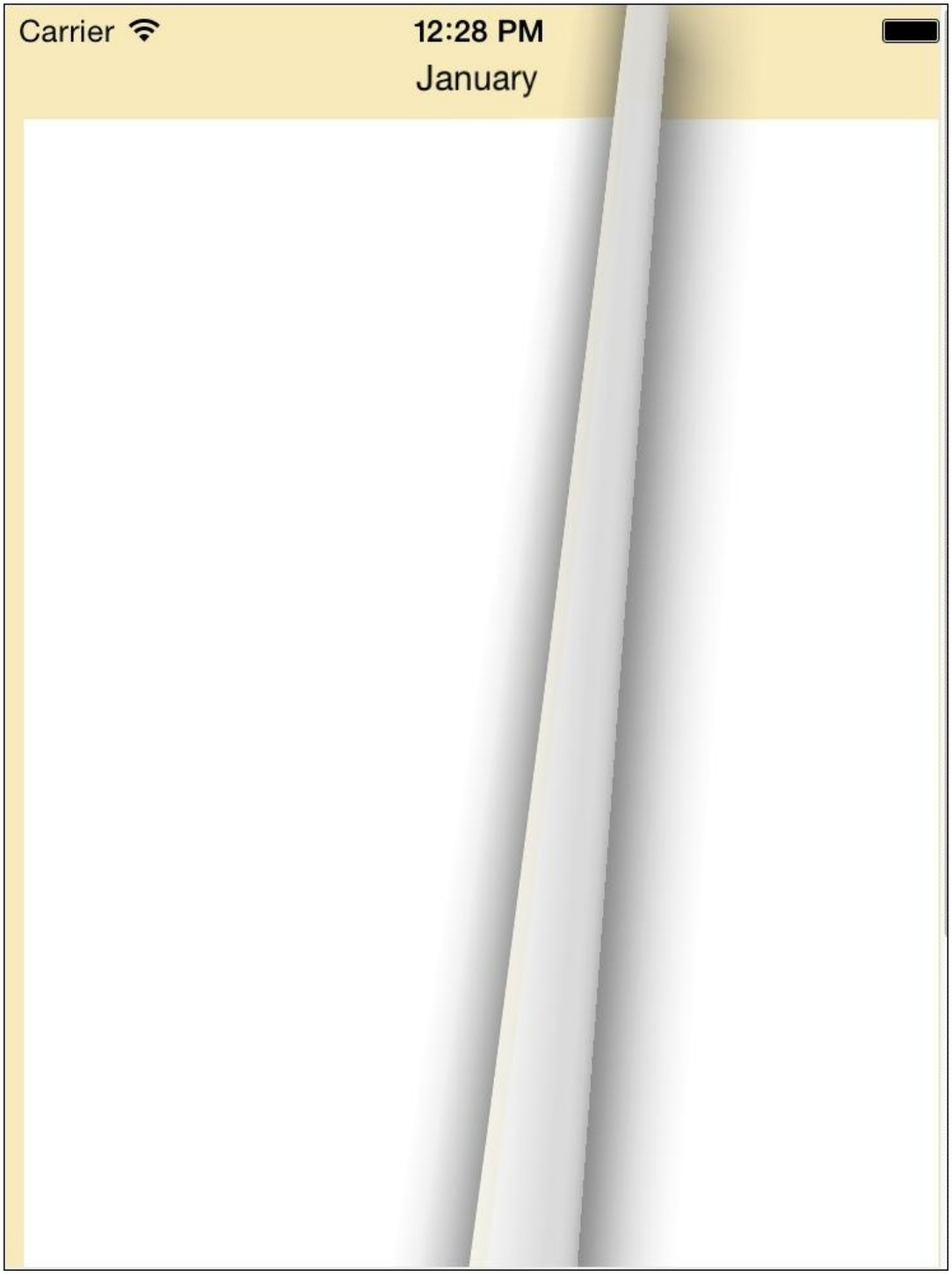
If you want to create a very basic application type, such as the *Notes* application, for the iOS, you should choose this template and then modify it to create your application. There is a **Master** list (shown in the following screenshot), and when you click on the each item on the list, it will show the details of the selected item from the list. There is a customizable **Master** button in the top-left corner of the **Detail** view. Clicking on it will take you back to the **Master** screen:



You can create the project in a way similar to how you created the **Game** application in the first chapter: click on the icon for the type of application, and then select the name of the project, its location, bundle ID, and platforms that you want to create the app for.

Page-Based Application

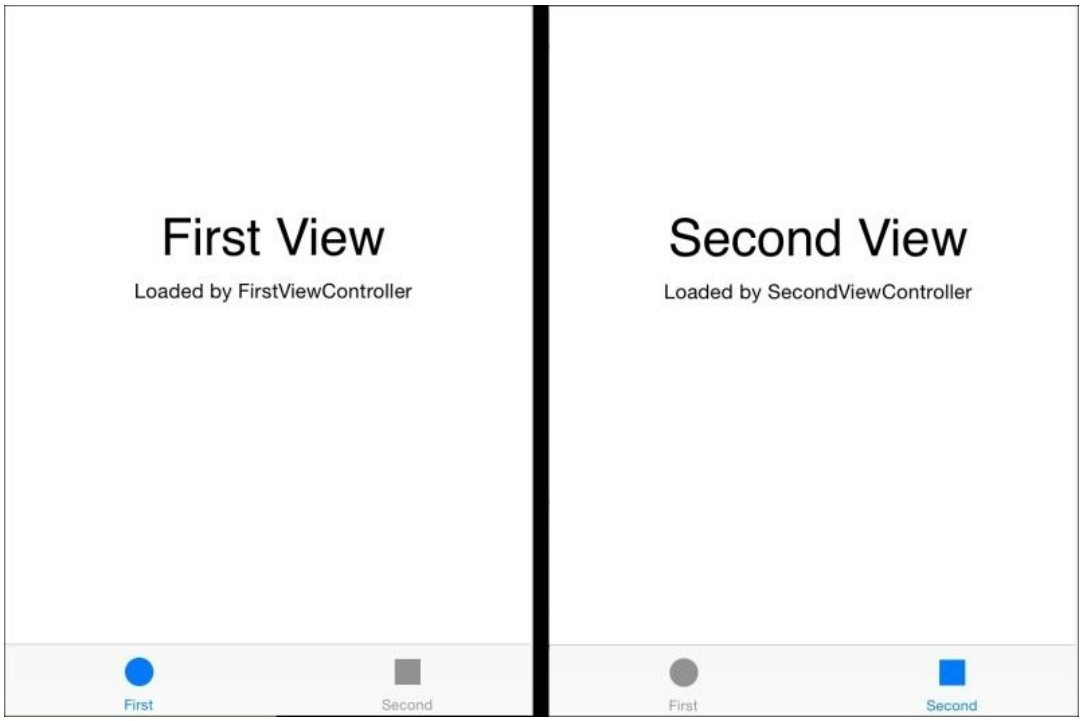
If you want to create an application that resembles a book with a page-flipping effect, you need to choose the **Page-Based Application** template. You can specify the book cover and then add content to the pages to create a flipbook application. You can swipe your fingers to the left to turn to the next page, or swipe to the right to go back to the previous page.



Tabbed Application

The **Tabbed Application** template will have a row of buttons at the bottom of the screen, and clicking through it will show you different views. A **Tabbed Application** template is similar to the way the tabs work when opening the *Music* app on your device. When the Music app is opened, the tab displayed is **Playlists**. The other tabs are **Artists**, **Songs**, **Albums**, **Genres**, **Compilations**, and **Composers** (the last tab). The **Playlist** tab shows the playlists that you have stored, but if you click on the **Artists** tab,

it will show a list of songs that has been sorted by artists. If you build the default application, you will have two tabs: **First** and **Second**. Clicking on the tab at the bottom will open the respective screens, like this:

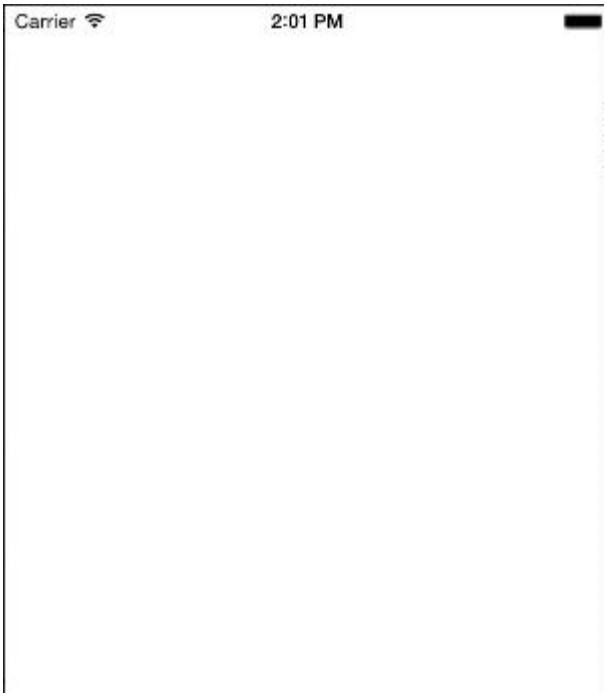


Single View Application

The **Single View Application** template is the most basic one. If your application doesn't adhere to any of the previous application template types, then you can use this to create your own applications.

For this chapter, we will start from the basics, so we will create a Single View Application. Click on **Xcode** to open it and then on **Create a new Xcode project**. Under **iOS**, select **Application** on the left panel. Then, select **Single View Application** and create it.

When you create a project and run it, the simulator will open up to show a blank screen, as shown in the following screenshot. When I said **Single View Application** templates are the most basic, I wasn't kidding. You will get a single view controller and no buttons to navigate. The advantage here is that you will see some basic classes that Apple already provides for creating the app from scratch, and then you can add items of your own to make it your app.

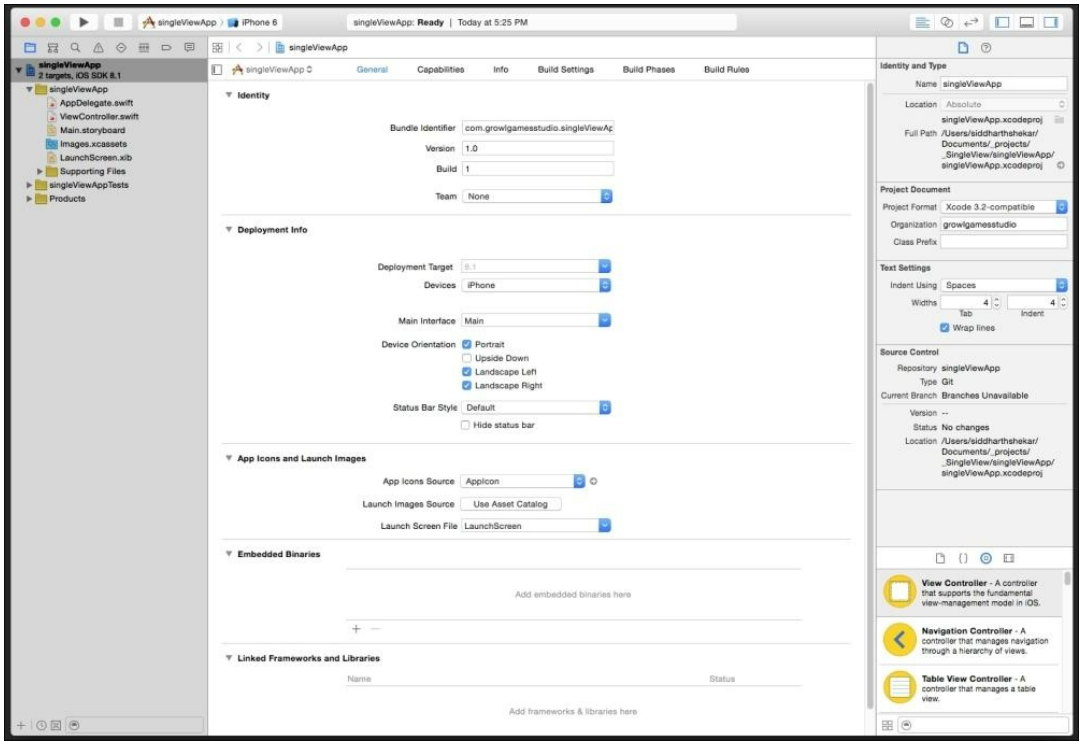


You should save this blank project, as we will be referring to it when we cover the interface of Xcode in the next section. So go ahead; name the file and save the project in the Mac.

The Xcode interface

If you are continuing from the previous section, then open the single view application you created in the previous section. Otherwise, you can create a new Single View Application project. As we will be spending most of our time looking at this window for the rest of the book, let's get a detailed understanding of its layout.

Open the project and you will be greeted with the window that is shown in the following screenshot. The top area of the **Project** view is called a toolbar. Below the toolbar on the left is the **Project Navigation** Panel. Right under in the middle is the **Editor** panel, and to the right of that is the **Utilities** panel. There is one more panel, the **Debug** panel, that might not be open by default, but we will activate it later:

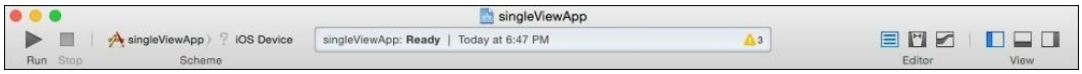


There is a lot of information in each of the toolbars and panels. Let's look at them individually.

The toolbar

The top of the Xcode project is the toolbar. Next to the maximize button on the left, the play and stop button are for running and stopping the application. Next to these buttons are the schemes in which you can select the target app and the device that you would like to run the application on, whether it is an actual device or a simulator.

At the center of the toolbar is the Activity View. This shows the status of the application at any given time. When you build a project, it will show the progress of the build, displaying the various stages of the build process.

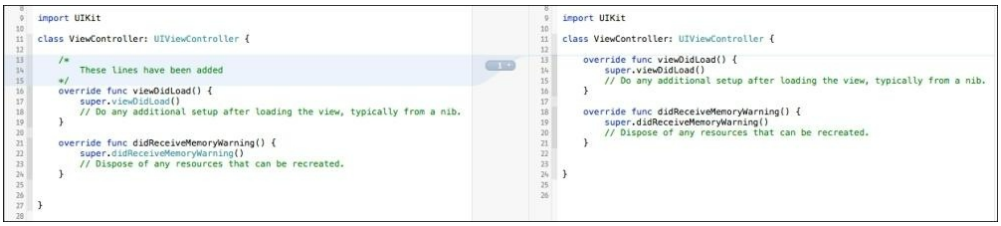


To the right of the Activity View, there are two sets of three buttons. The first set of three buttons on the right is used to change the **Editor** panel according to your needs:

- **Standard Editor**: This is activated by default whenever a new project is created. This setting is used when coding, debugging, or navigating through the project. We will be using this panel almost all the time.
- **Assistant Editor**: Click on this button and another panel will open up to the right of **Standard Editor**. This is called **Assistant Editor**. It will provide more information, depending on what is clicked on in **Standard Editor**. In the following example, in the navigation panel, I clicked on the `Main.Storyboard` file. The **Standard Editor** changes to display the objects in the file, and the **Assistant Editor** shows the contents of the class associated with the View Controller as it is selected in the **Standard Editor**.



- **Version Editor**: When selected, the **Standard Editor** is again split into two panels, and we will be able to see and compare the changes in the current and previous versions of the code. In the example shown in the following screenshot, I made some changes to the `ViewController.swift` file. The comment added is highlighted on the left panel, and the right panel shows where these changes are being made to the file. This is obviously very powerful when using source code management for your project. Source Control Management is already included in Xcode and is initiated when a new project is created.



The three buttons to the right of the editor are used to hide and show the **Navigator**, **Debug**, and **Utilities** panels. These panels can be shown or hidden as per your requirement or convenience. For example, to take the preceding screenshot and get more screen real estate, I hid the utilities panel by clicking on the rightmost button.

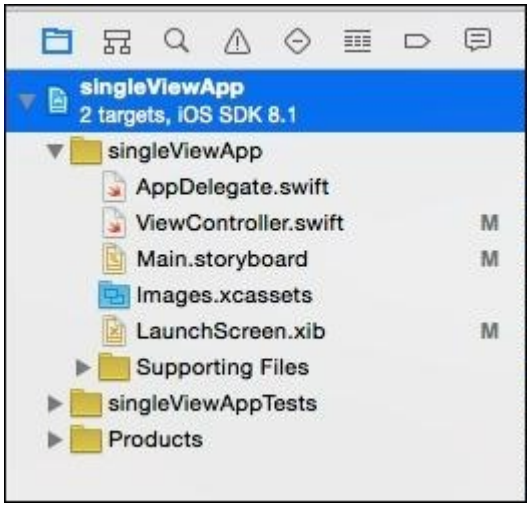
That is all for the toolbar. Let's look at the **Navigation** panel next.

The Navigation panel

Below the toolbar on the left is the **Navigation** panel. It has eight tabs. From left to right, they are **Project Navigation**, **Symbol**, **Find**, **Issue**, **Test**, **Debug**, **Breakpoint**, and **Report Navigator**. Clicking on any tab will activate it. By default, the **Project Navigation** tab is selected.

The Project Navigation tab

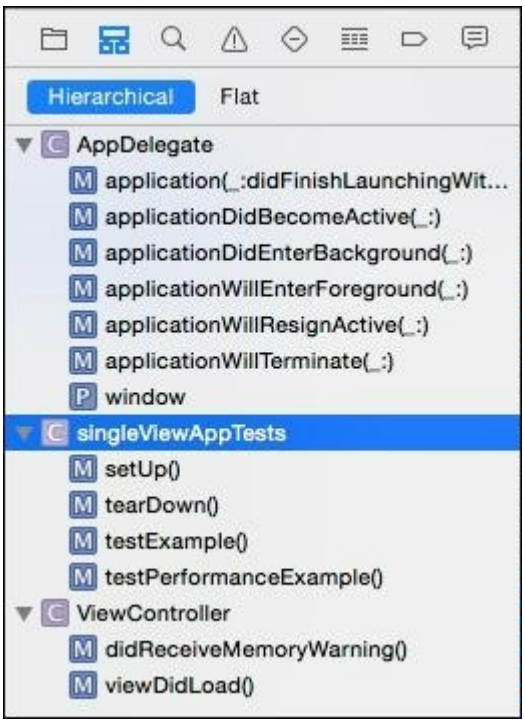
The **Project Navigation** tab shows the content of the files that you have in a project, including code files and asset files. Later, if you add frameworks to the games, those will also be displayed is this panel, making it easier to navigate through. As we saw earlier, you can edit code by clicking on a code file such as a `.swift` file, and in the editor window, you can add, remove, and modify code. If you have image assets and three-dimensional meshes, those files will also be shown here. By clicking on the images and three-dimensional files, you can view the content, but you will not be able to modify it. For that, you will need to open the file in applications such as Photoshop to edit images and three-dimensional packages such as `Maya` or `3dsmax` to edit the three-dimensional geometry:



You can organize a set of files using folders (called **groups** here). Keep in mind that this is meant just to organize files in **Project Navigator**. Creating a new group or folder in the **Navigation** panel won't create a folder in the project directory in the system.

The Symbol Navigator tab

The next button to the right is the **Symbol Navigator** tab. You can access all the symbols such as functions, methods, properties, classes, structs, enums, and variables in the project.

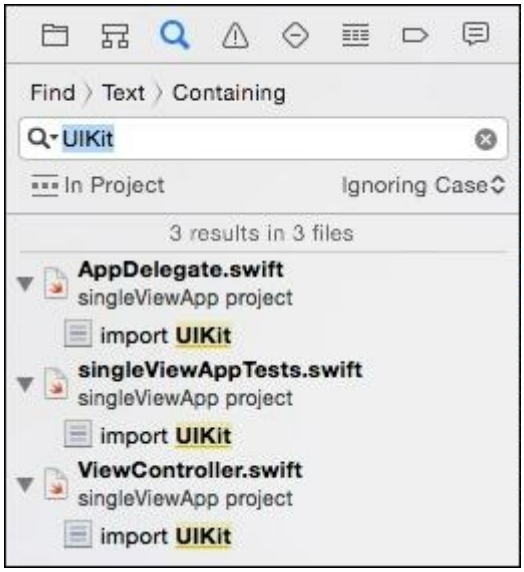


C denotes that it is a class, **M** stands for method, and **P** stands for property. There are other notations also, but these are the most basic notations you will usually come across.

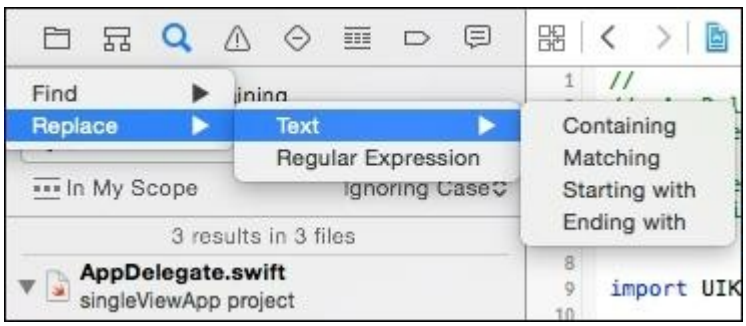
So, in the `AppDelegate` class, there are six methods and a property called `window`.

The Find Navigator tab

The **Find Navigator** tab is used to search for phrases or files in the project by name. In the following example, I am trying to search for the term `UIKit`, and it looks as if it is imported in the three classes:

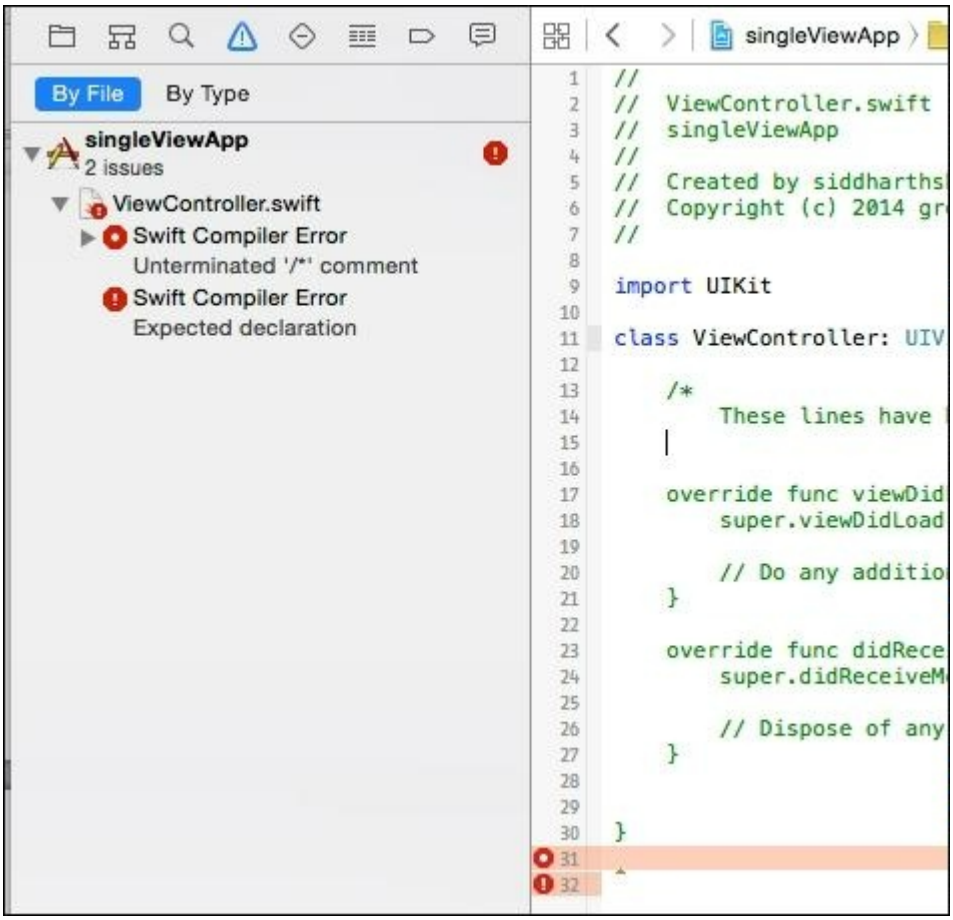


You can also choose to search for the keyword in the current score, or create a new scope to look in other places. Also, by clicking on the **Find** drill-down arrow, you can choose to find or replace commands on text that contain, match, start with, or end with your search term.



The Issue Navigator tab

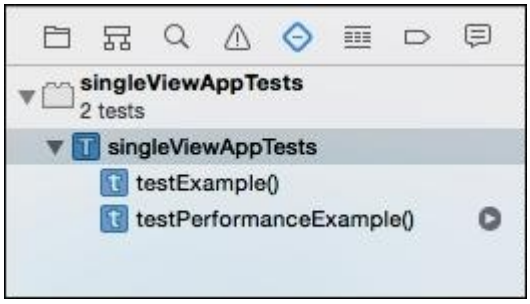
The errors and warnings list, count, and description will be displayed in the **Issue Navigator** tab whenever your project has build errors. You can click on the items in the list to show the location and file where the error exists:



In the preceding example, I forgot to close the comment section in the `ViewController` class, so I get an error after building the project telling me that in `ViewController.swift`, there is an undetermined `/*` comment. In some cases, the error message may not be very explicit, but at least, you will get a hint of what might be causing the error.

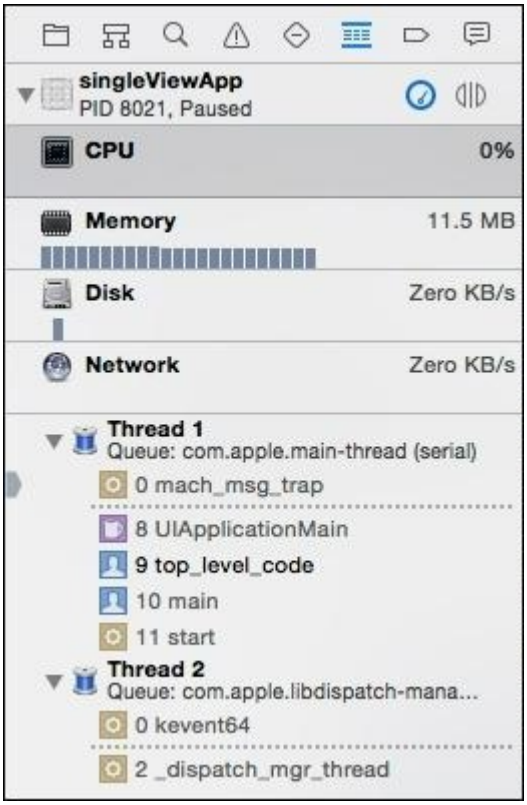
The Test Navigator tab

If you are using unit tests on the project, then the results of the test are displayed in the **Test Navigator** tab. Unit testing is a topic in itself, and is unfortunately beyond the scope of this book. To know more about unit testing, you can look at Apple's official documentation for more information at https://developer.apple.com/library/ios/recipes/xcode_help-test_navigator/Recipe.html#//apple_ref/doc/uid/TP40013329-CH1-SW1.



The Debug Navigator tab

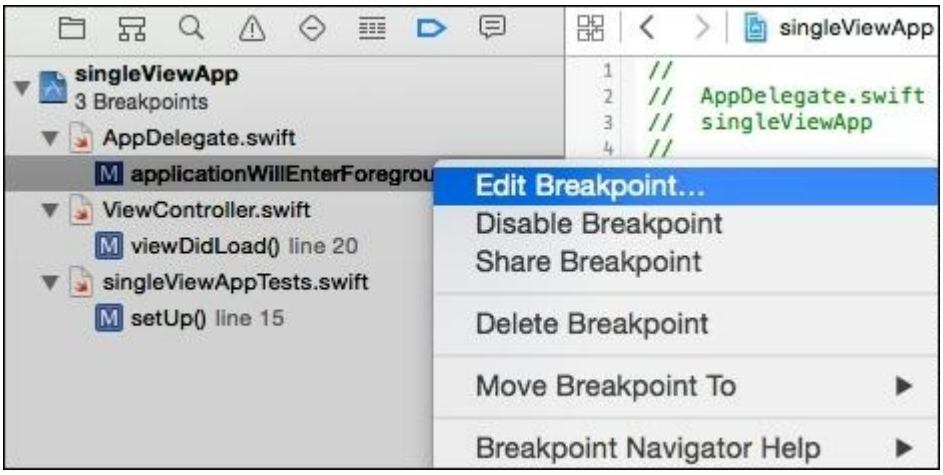
The **Debug Navigator** tab is used for debugging and showing how optimized your code is. This navigator is active only when an application is running. While an application is running on the simulator or on a device, we can get information regarding CPU, memory, disk, and network usage. Apart from general system information, you also get a stack of methods and functions that were called in the corresponding order. Clicking on the methods and functions will open them in the editor panel.



There are two buttons to the right of the running app. The first button on the left lets you hide or show the gauges, and the button on right lets you select whether you want to view the **Process by thread**, **Process by Queue**, or **View by UI Hierarchy**.

The Breakpoint Navigator tab

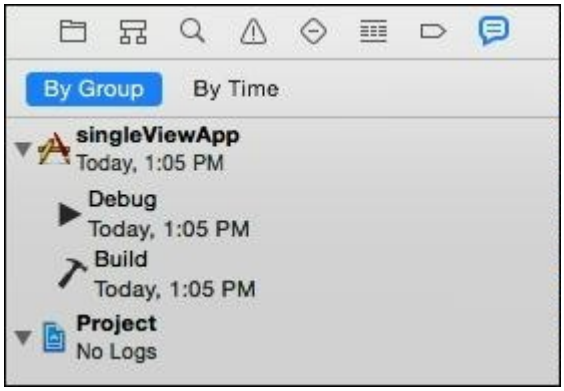
The **Breakpoint Navigator** tab shows the location of all the breakpoints we've added to the files in the project. The location of the breakpoints is shown class- and method-wise. It will also show the line number on which the breakpoint was added.



By right-clicking on the any breakpoints, you can edit, disable, share, delete, or move it.

The Report Navigator tab

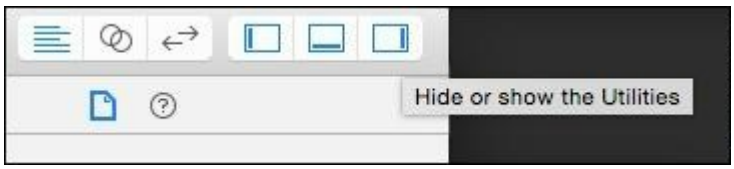
The **Report Navigator** tab shows the history of the recent builds and logs of the project, along with a timestamp:



This concludes the section on the **Navigation** panel.

The Utility panel

The top part of the **Utility** panel is context-sensitive, depending on what kind of file is clicked on in the **Navigation** panel. The bottom part of the **Utility** panel has four tabs for the different library types used to drag-and-drop certain library-specific objects onto the **Editor** panel. Sometimes, the top part gets hidden under the bottom part of the panel, but you can drag the bottom part of the utility to reveal the content hidden under it.



The **Utility** panel is better understood once we jump through the project and look at each file individually.

The Single View Project

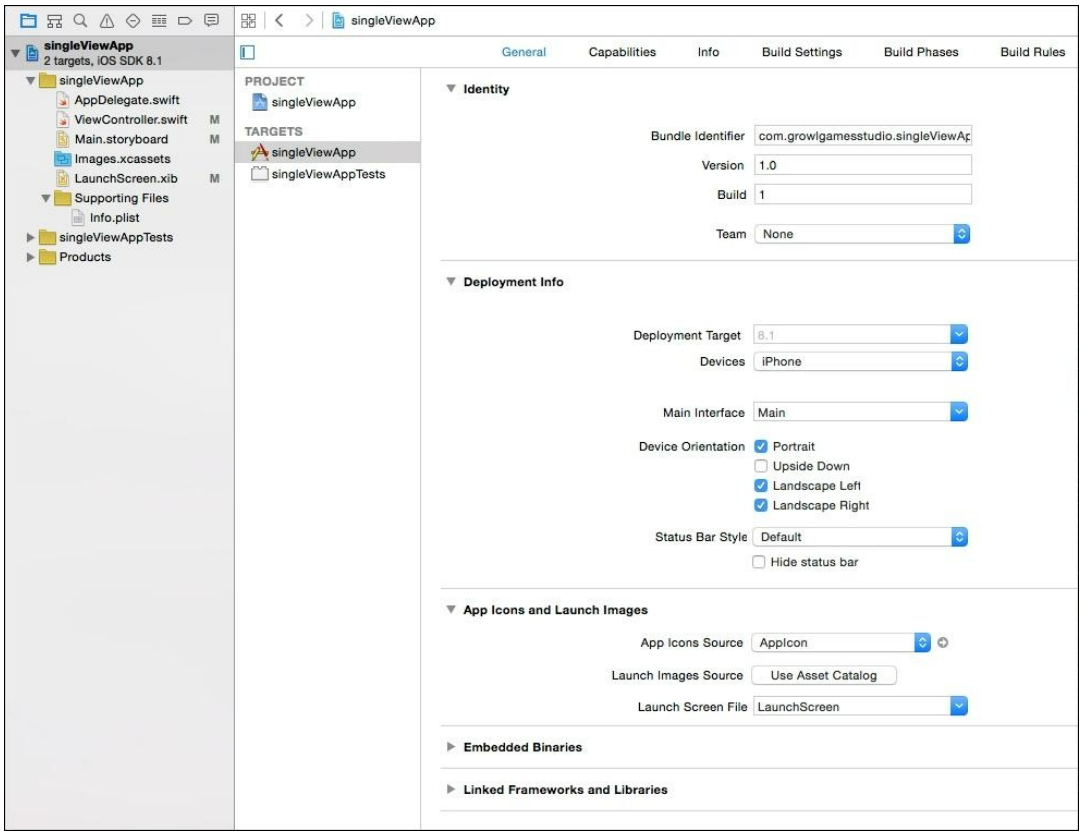
Go back to the **Project Navigator** tab. This is the first tab from the left in the **Navigation** panel. Let's look at the files in the **Project Navigator** tab. The **Project Navigator** tab shows the project root and under it, all the files associated with the project. Let's look at the project root first.

The project root

When you click on the root of the application that is at the top of the tab, you will see the **Editor** panel change to display six tabs: **General**, **Capabilities**, **Info**, **Build Settings**, **Build Phases**, and **Build Rules**. Most of the time, you will be concerned with the first three tabs: **General**, **Capabilities**, and **Info**.

General

This tab contains the basic information about the app. We had a brief look at it in Chapter 1, *Getting Started*, when we changed the orientation of the game. This tab has five subsections, as shown in the following screenshot:

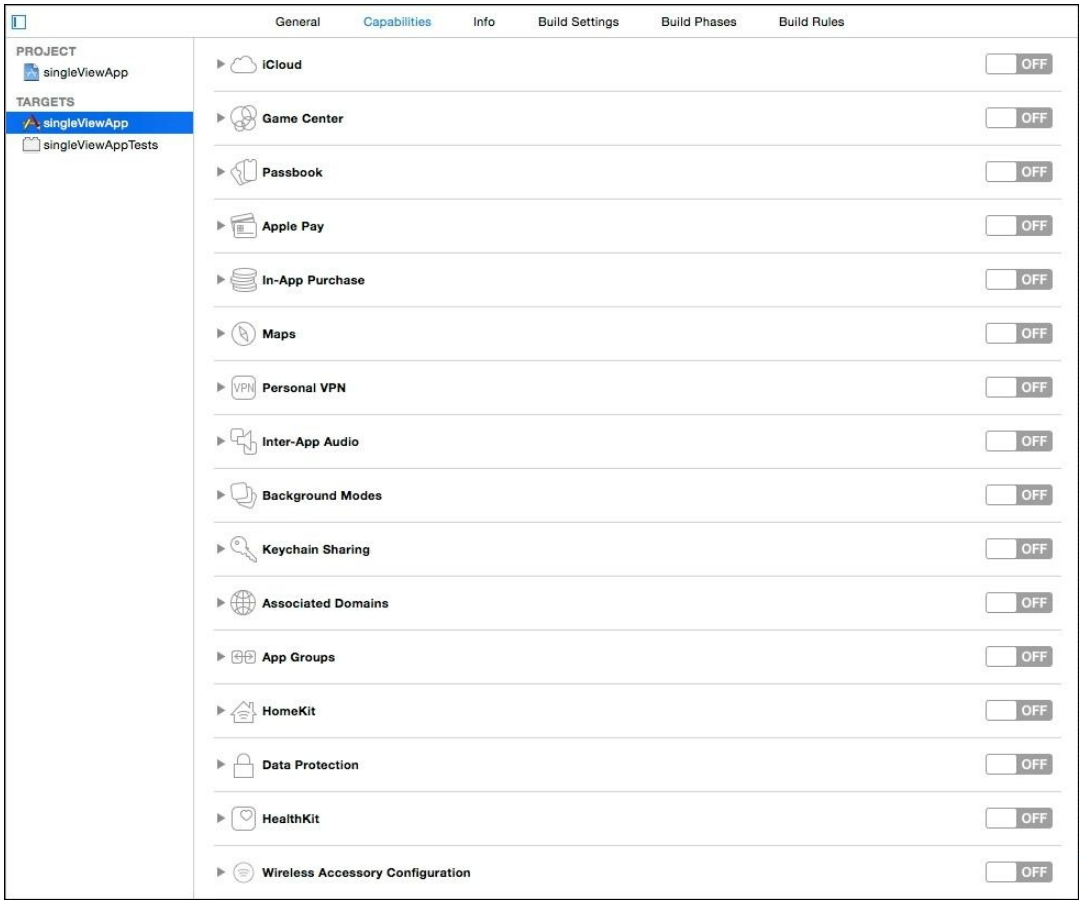


The preceding screenshot has the following sections:

- **Identity**: This shows the bundle identifier, version number of the app, build number of the app, and team. We will look at the team when we deploy the app on the device, as it requires some steps to get it.
- **Deployment Info**: Here, you can select the value of **Deployment Target**. By default, we are targeting iOS 8.1 devices here, but if we want our app to be compatible with previous version, we can use 7.0. However, we must make sure we are not using any APIs from 8.1, in which case the app will give build errors. We can select the device that we want to target, whether we want to target the iPhone or iPad, or make a universal app. The main interface file is the first file that will be called when the application has finished loading. The files here should have the `.storyboard` extension, so here we are calling the `Main.storyboard` file when the application launches. We can also change the device orientation here and select the value for **Status Bar Style**, and also hide the status bar by clicking on the check box.
- **App Icons and Launch Images**: The source file for the app icons is provided here. The `AppIcon` file is selected here. If you are wondering where this file is, it is in the `Images.xcassets` folder. We will use this file to assign the icons for different iOS versions and devices. If you want an image to be displayed at launch of the application, you can create an asset catalog file to display the launch images for different devices and iOS versions, similar to icons. We can also specify a launch screen file, which will be displayed while the application is launching. Here, the `LaunchScreen` file will be displayed during the launch. The file extension for the `LaunchScreen` file, which was in `.nib` format previously, should be in `.xib` format. Even though the extension has changed, they are still referred to as NIB files.
- **Embedded Binaries**: This shows any binaries that are embedded in the project.
- **Linked Frameworks and Libraries**: This displays the list of frameworks and libraries included in the project.

The Capabilities tab

This shows the Apple services that the application uses. To use a service, you will need to activate it by turning it **ON** on the right-hand side. Games usually use services such as GameCenter, In-App Purchases, and iCloud integration for cloud saves.



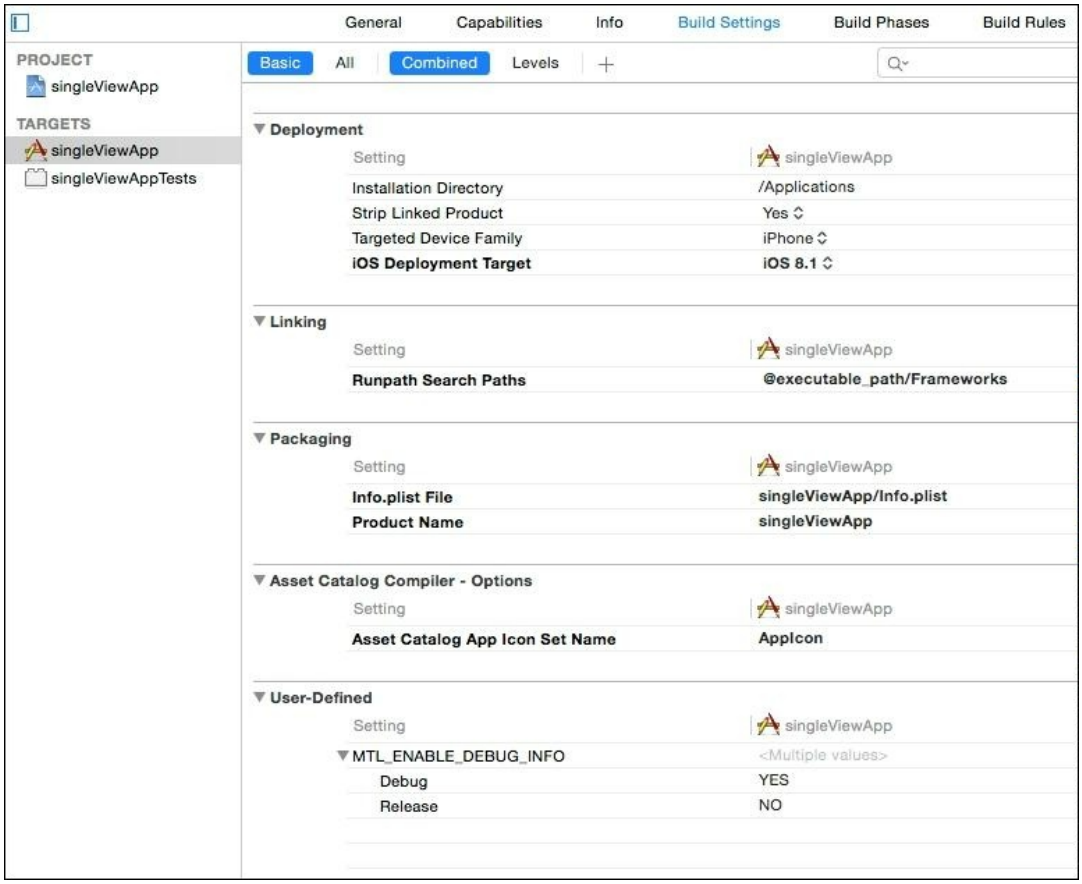
The Info tab

The **Info** tab contains information regarding the version number, build number, storyboard file base name (which is the storyboard file to be loaded at start of the app), app name, and some other information. All of this information is loaded from the `info.plist` file, which is located in the `Supporting Files` folder of the project. There are some other things such as document types, exported UTIs, imported UTIs, and URL types, which, for the most part, you won't be dealing with during the course of this book.



The Build Settings tab

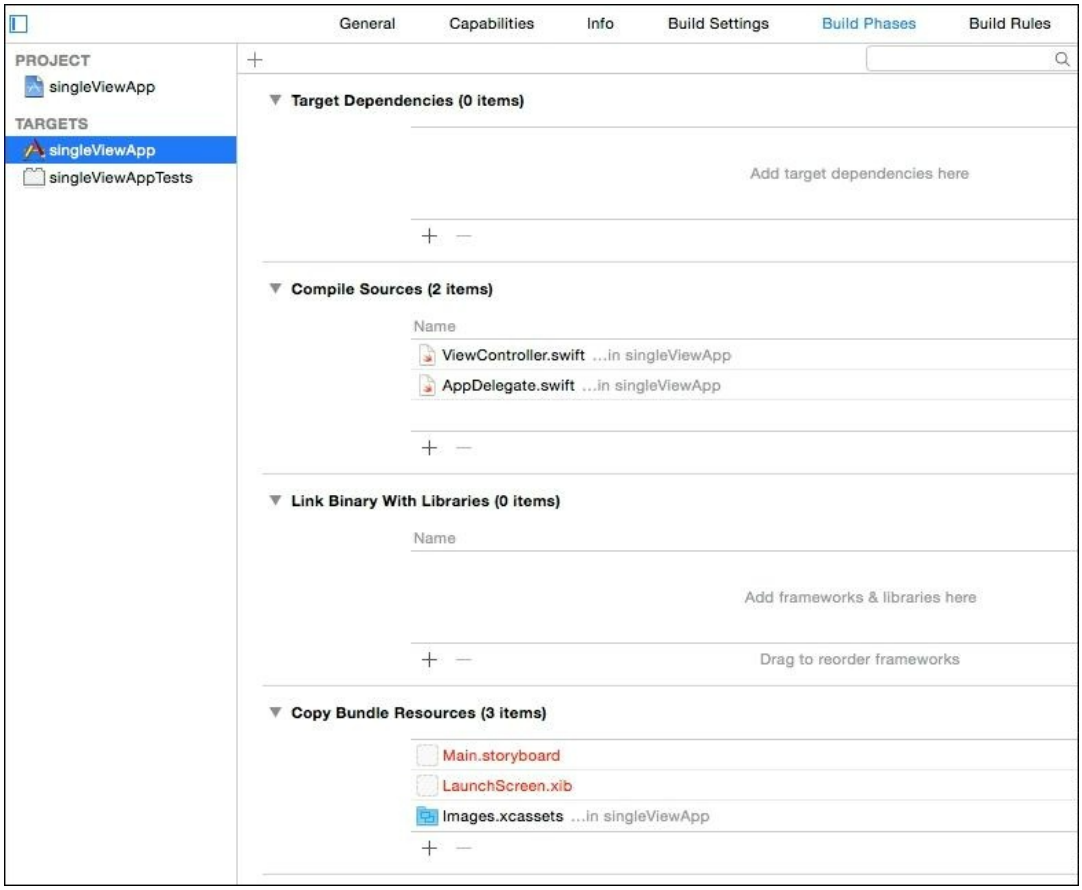
The basic view of the **Build Settings** tab shows information regarding deployment, such as the target device and iOS version, location of the path for frameworks, packaging info such as location of the `info.plist` file and **Product Name**, **Asset Catalog App Icon Set Name**, and other user-defined settings.



A more detailed look can be taken by clicking on the **All** tab instead of **Basic**. This will give information about the setting for the architectures supported, locations and options for the build, and code signing, which we will cover when we are ready to test the app on the device and deploy it on the App Store. It has further settings and information regarding the kernel module, linker, and compiler.

The Build Phases tab

The **Build Phases** tab shows the target dependencies added, list of source files, libraries added, and asset resources added.



This tab is sort of important, in the sense that when you want to add frameworks to your project, you will have come to this tab to include them. Also, sometimes when you get build errors, you might want to check whether all the required source files are actually in the **Compile Sources** list, as this can be the reason for build errors. Build errors can be introduced once your project becomes bigger as you might end up deleting some source files.

The Build Rules tab

You will probably never be required to change anything here, as it is mostly used if you want a specific file type to compile in a certain way. To define a custom process for a certain file type, you just have to create a new build rule.



Now that you have taken a look at the project root, you can click on the small triangle button to the side of the project root to open the project tree, if it is not already open.

Under the created project, there are three groups. The first is given the same name as the project. The second group holds the files for the project test files, so it will always have the project name and be suffixed with `Tests`. The third is the **Products** group, containing the `.app` file and the `Tests` files. For the majority of the time, we will be dealing with the files in the first group, that is, the group names after the name of the application. The first folder is where all the code, assets, and project-related files for the project should exist. You can create subfolders like the `Supported Files` folder to organize your project better. So, you can get a `Classes` folder that contains all the classes for the project, and an assets folder in which you can put your images, icons and three-dimensional objects.

Let's look at each of the files of the main project folder in detail.

The project folder

The main project folder contains the `AppDelegate.swift`, `ViewController.swift`, `Main.Storyboard`, `Images.xcassets`, and `LaunchScreeb.xib` files; and the `Supported Files` folder.

As soon as we open the `AppDelegate.swift` file, we see that the `AppDelegate` class inherits from the `UIResponder` and `UIApplicationDelegate` classes. Both of these classes are part of the UIKit. The UIKit framework is most important, as it is required to create and manage any iOS application. This framework provides basics elements such as window and view creation.

The `UIResponder` class is responsible for handling events such as detecting touch and motion events. For touch events, it has `touchesBegan`, `touchesMoved`, `touchesEnded`, and `touchesCancelled`, which are called when a finger touches the screen. There are three motion events: `motionBegan`, `motionEnded`, and `motionCancelled`:

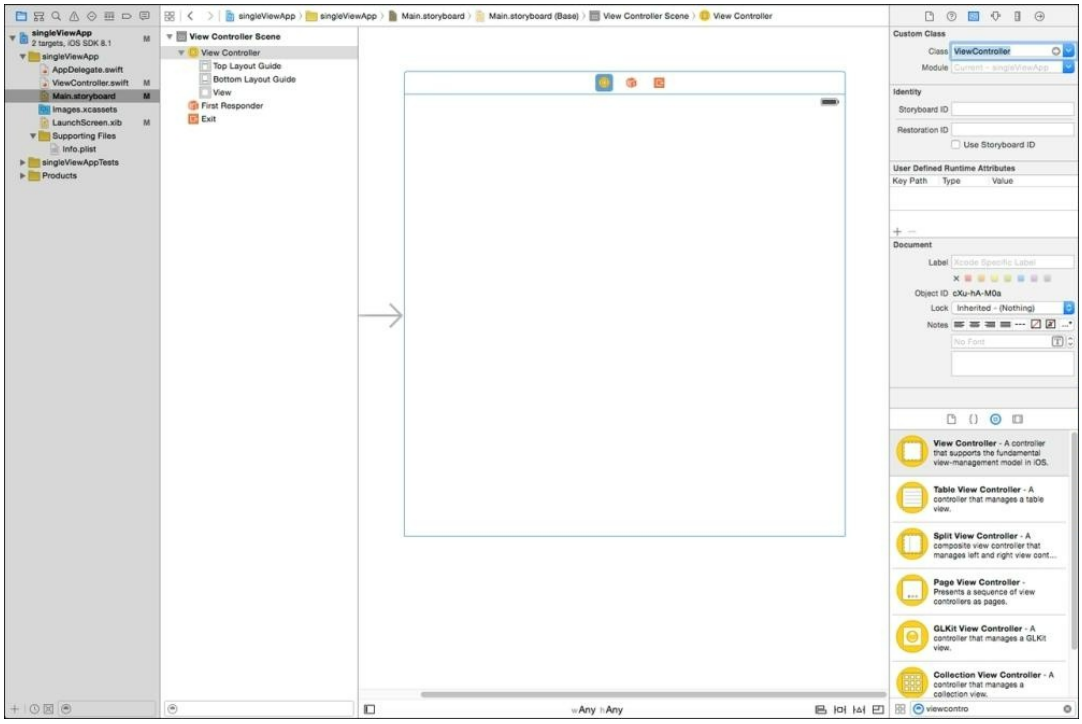
- `UIApplicationDelegate`: This is a subclass of `UIResponder` that responds to events during the life cycle of the application. The functions in the `AppDelegate` class are from `UIApplicationDelegate`, which will be called when any of those functions are triggered by the application.
- `applicationdidFinishinLaunchingWithOptions`: This is the function that gets called when the application has finished launching. It returns `true` when the application is launched.
- `applicationWillResignActive`: This is called after the application has just become inactive and is about to go to the background. It is triggered when you either receive a phone call or press the home button to switch apps. It is used to disable timers or pause any update function.
- `applicationDidEnterBackground`: When the application has gone to the background and is no longer active, this function gets called. You can save the user score and make the application ready, if it is about to be terminated.
- `applicationWillEnterForeground`: Once the application sitting in the background is selected and is about to be active again, this function is called. Since the user didn't terminate the application, you can restore the values to the previous state here.
- `applicationDidBecomeActive`: Now the application is fully active again, so you can resume the game and unpause and resume the update function.
- `applicationWillTerminate`: This is the last function that will get called before the application fully terminates, so you might want to save the game and release all objects.

There is an optional variable created, called window, and it is of the UIWindow type. Each application created is inside a window, and this window variable will give access to the current window. If needed, you can assign the current window to this variable to gain access to the properties of the currently running window, such as the size of the window.

Before we look at `ViewController.swift`, you need to get an understanding of `Main.Storyboard`, as the `ViewController.swift` class is called through `Main.StoryBoard`.

The `Main.storyboard` file is automatically called right after the application has finished launching. This is done automatically by Xcode as a standard protocol, and therefore, you won't see it getting called in the

`applicationDidFinishLaunching` function in the `AppDelegate` class. We have already called it in the **General** tab of the project root. Let's now look at the file in detail and see what it does. For this, you need to understand what storyboards are in iOS:



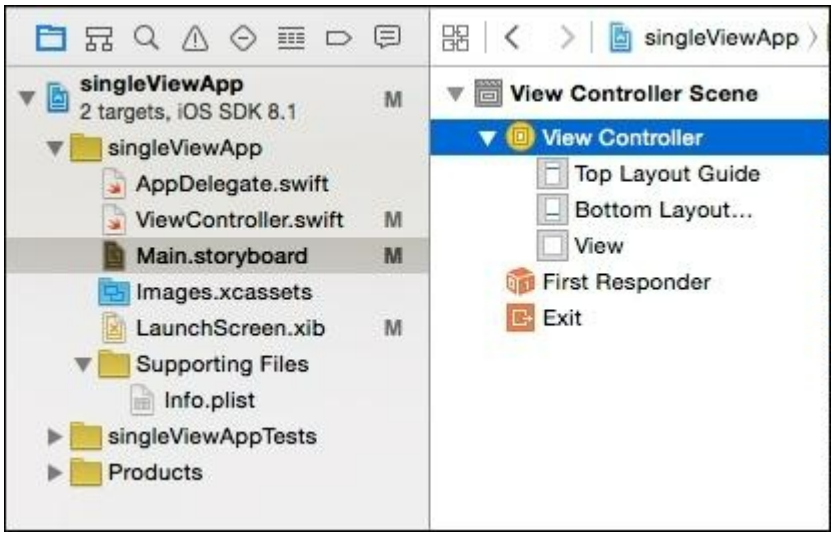
A storyboard is a part of iOS Interface Builder or UITookit. It is used to create a user interface without writing any code. You can add buttons, texts, or sliders to your app. Also, by clicking on the button, you can make the app change views by creating a link between the current view and the next view. You can create a chain of such views to create the screen flow for your application.

Every storyboard needs a ViewController to start the application with. ViewControllers are like screens that create the building blocks of the app. You will be linking the ViewControllers in the storyboard to develop the screens for any app you want.

The ViewController is associated with the custom class `ViewController.swift` file. This can be seen in the **Utility** panel by clicking on the **Identity Inspector** (third tab from the left). In the custom class, in the `class` field, the `ViewController` class is specified. We will cover more on **Utility** panel later.



If you are not able to see the ViewController tree, you can click on the small box icon at the bottom-left corner of the **Editor** panel to open it. The **View Controller Scene** has **View Controller**, **First Responder**, and **Exit**.



By clicking on the triangle next to the ViewController, you will see that it contains three more items: **Top Layout Guide**, **Bottom Layout Guide**, and **View**. The top and bottom layout show the top and bottom ends of the view. The top starts after the battery indicator icon and the bottom ends at the bottom of the view. These are more like guides to let you know the limits that you need to be working within. The **View** item is the entire area the user will be able to see when the ViewController opens.

The **First Responder** item is the first object that you interact with in the ViewController. This will send the message to the UIResponder. Each time you click on a button, interact with a slider, or enter the text field, **First Responder** gives information on what you are interacting with at that time. So, if you click on a button, the responder will know immediately that you are clicking on a button.

Exit is used when you want the user to be sent to a different ViewController. This can be the previous ViewController from which the user came to the present scene, or it can be some other ViewController. This is not all that important at this stage as long as you understand what a ViewController is.

We have seen that `ViewController.swift` gets called through the `ViewController` object in the `.storyboard` file. Inside the `ViewController.swift` file, there are two functions: one is `ViewDidLoad`

and the other is `didReceieveMemoryWarning`. In the `.storyboard` file, we saw that `ViewController` contains the view. When the view gets created, the `ViewDidLoad` function gets called. This means that the ViewController can display whatever is in the view and it is ready for any interaction that the user might have with the view.

The `didReceieveMemoryWarning` function gets called exactly when that happens. This function will get called either when you have too many apps open on the device, or if your program is not releasing the objects that it created earlier from the memory. The latter shouldn't be a problem because Swift has its own garbage collection, so you have to worry only about releasing memory.

`Images.xcasstes` will contain all the image assets for the project. As of now, the only file in it is the `AppIcon` set. In this file, all the different iOS versions and resolutions of the icon images are shown with placeholders.

Here is an example of the icon set for the game *pizZapMania* that was created. The value mentioned under each set is the base dimension. The base for the first icon in the top left is 29 points, so to create icons for iOS 5 and 6, we will need two icons: one at **1x**, meaning 29 x 29 pixels, and another twice that size. Similarly, icons will have to be created for all the base values and multiplication factors mentioned for a particular device.



While looking at the **General** tab of the project root, we saw that the `LaunchScreen.xib` file is called at the launch of the app. So what is this XIB file?



The XIB file is called the NIB file (just as `.xib` was previously `.nib`) or NeXTSTEP Interface Builder. As of now, this file is XML-based and `n` is replaced by `x`, but it is still called a NIB file. So, when someone refers to a NIB file, they mean an XIB file. According to Apple, storyboards are the way to go forward as they support multiple view controllers and a `.xib` file can have only one. Since in this case the application and the launch screen both use a single view controller, it is the same.

The `Launchscreen.xib` file adds two labels upon the creation of the project, by default. The first label is the name of the app, placed at the center of the view, and the other is placed at the bottom of the view to display the copyright information. Click on either of the text to see the **Utility** panel light up like a Christmas tree. Let's finally look at the **Utility** panel.

The Utility panel (Redux)

The top part of the **Utility** panel contains six inspector tabs, and the bottom part has four library tabs. Let's first look at the inspector tabs one by one.

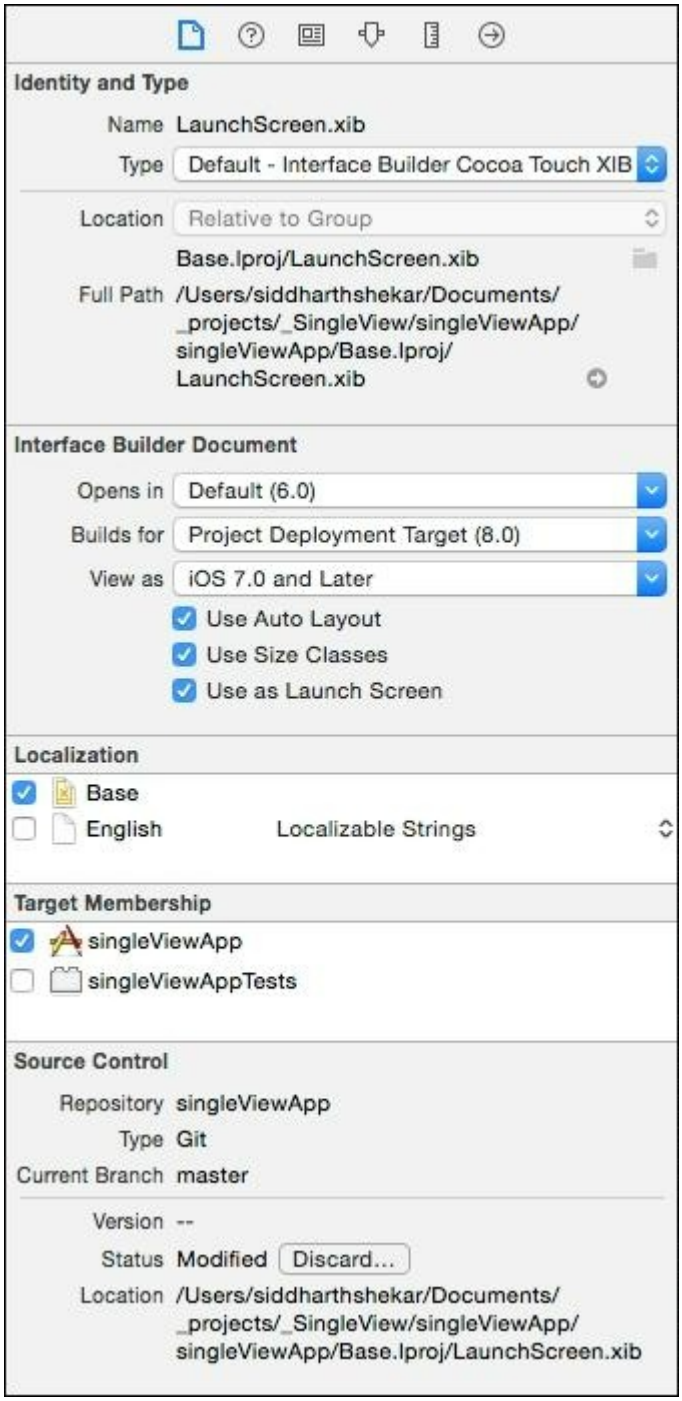
The inspector tabs

Let's start from the leftmost tab, the **File Inspector** tab, and then we will look at the **Quick Help Inspector**, **Identity Inspector**, **Attributes**

Inspector, **Size Inspector**, and **Connections Inspectors** tabs. The main tab to remember is **Attributes Inspector**. We will just glance at the rest.

The File Inspector tab

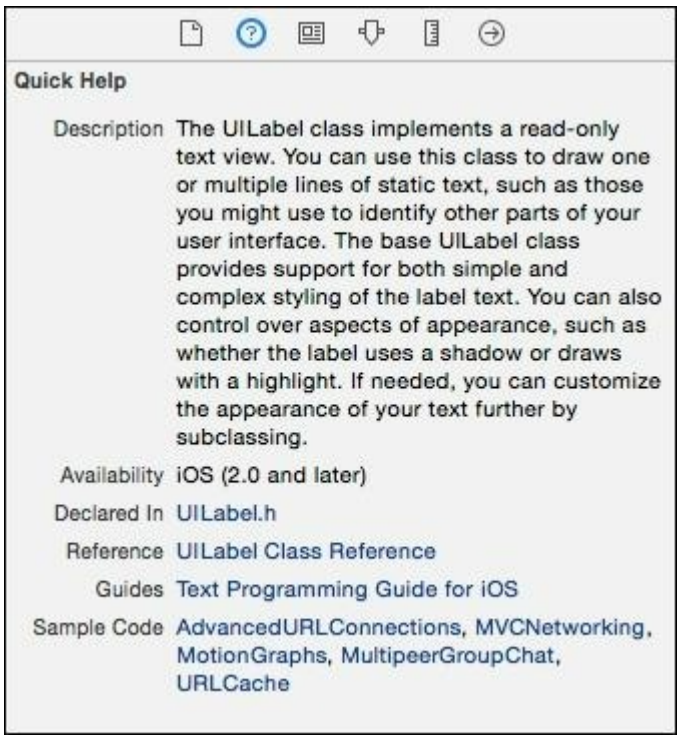
The **File Inspector** tab gives information about the parent file that the current object is attached to. Here, the label is attached to the `LaunchScreen.xib` file and is located in the `Base.lproj` directory on the Mac.



It shows the Xcode version that can open it, the build target version, and the iOS version supported by the document. It also has details regarding **Source Control**, such as the repository name, type, and branch.

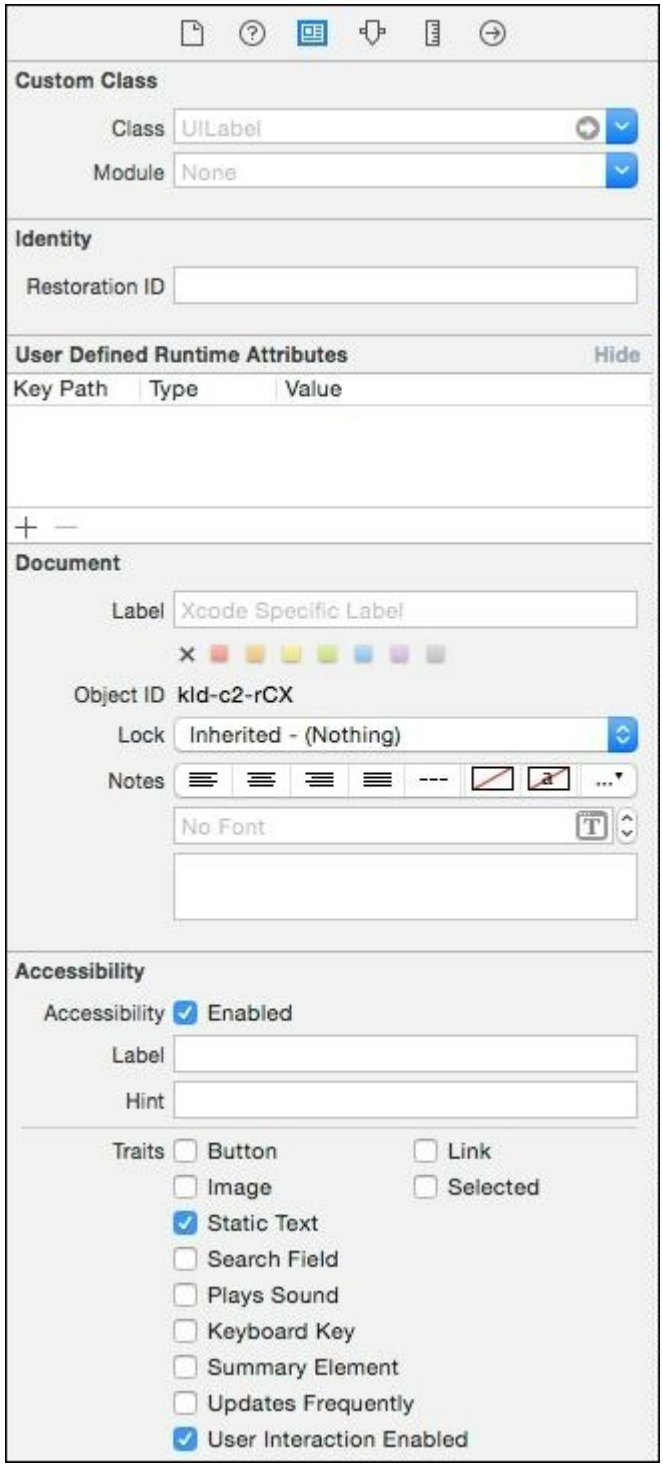
The Quick Help Inspector tab

The **Quick Help Inspector** tab provides a brief description, features, and capabilities of the item selected. Here, since we selected the label, it shows the parent class of the label, which is `UILabel`. It also provides other details, such as **Availability**, which tell us which iOS versions support this.



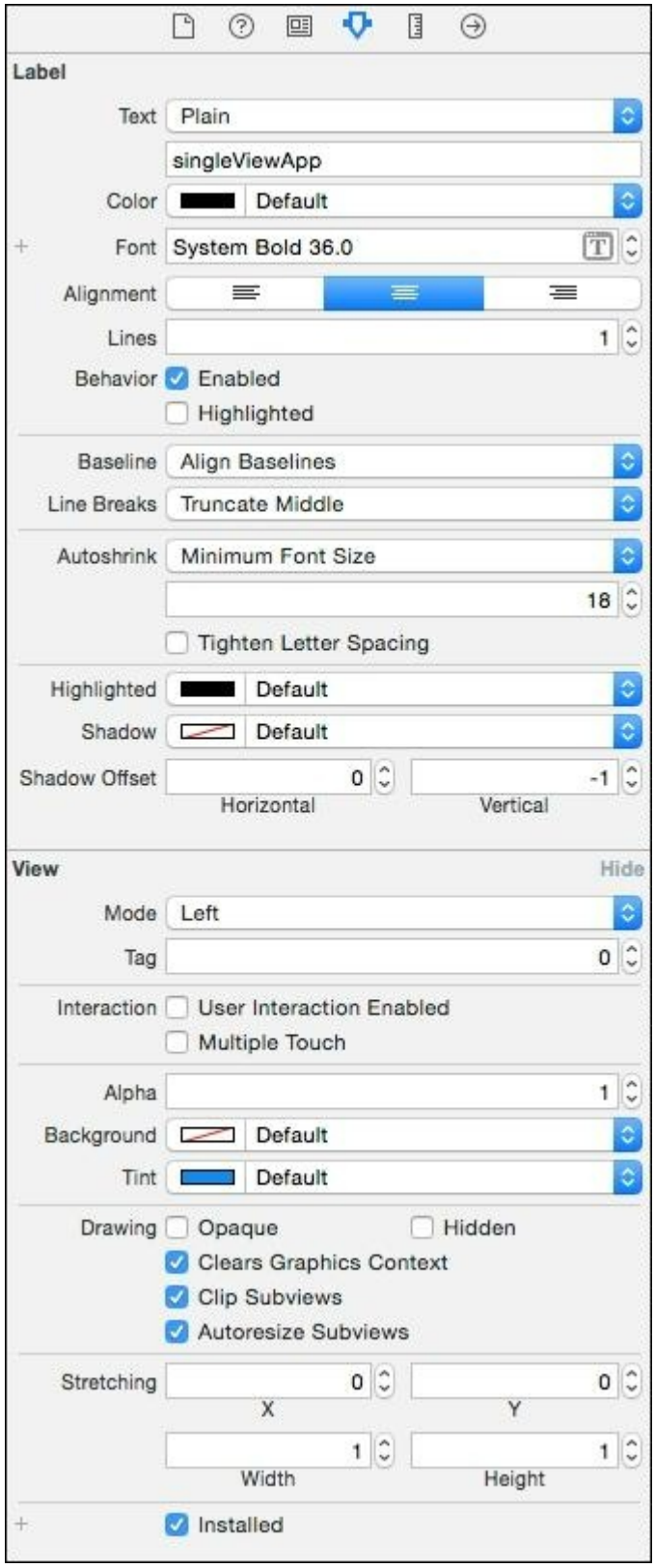
The Identity Inspector tab

The **Identity Inspector** tab helps in assigning and managing metadata for any object, which in this case will be the text label selected. It also shows the custom class attached to it. Here, it is expecting a class that is inherited from the `UILabel` class.



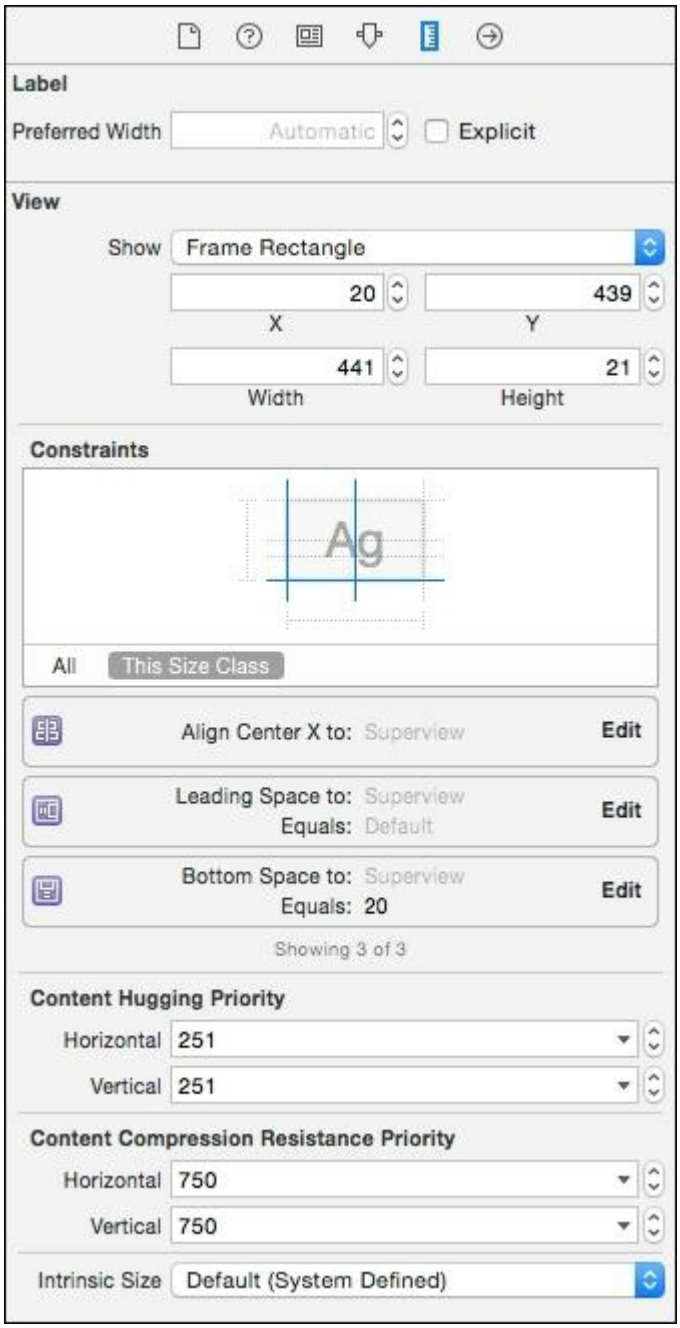
The Attributes Inspector tab

This is where you can edit the properties of the object selected. So here, we can change the look, color, position, search, and text—as we can for a page of a word document.



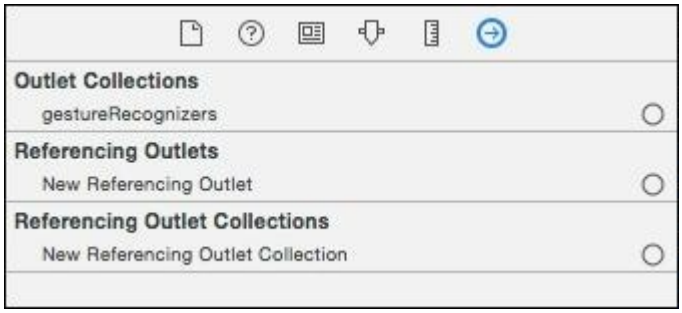
The Size Inspector tab

The **Size Inspector** tab helps you position the object, set the size of the object, and add constraints for further assistance in positioning the object correctly.



The Connections Inspector tab

As explained earlier, we can connect the ViewControllers to create the screen flow and transition from one ViewController to the next. In the **Connections Inspector** tab, we can see which objects are connected to which outlet.

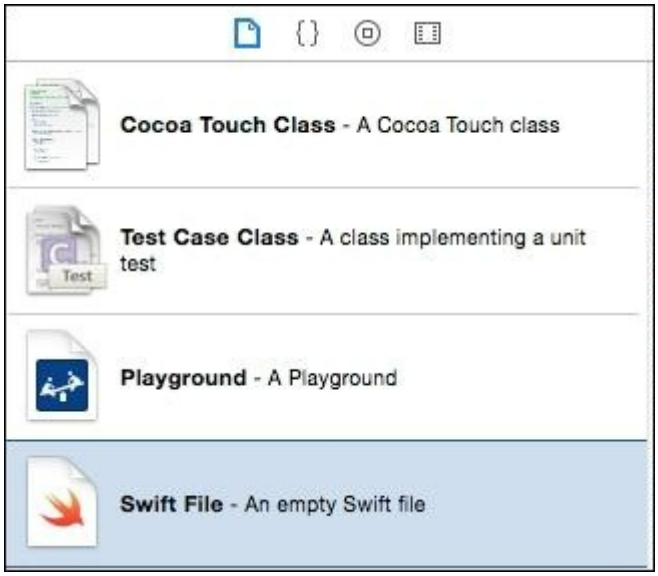


The library

The library contains four tabs: **File Template**, **Code Snippet**, **Object**, and **Media Library**.

The File Template tab

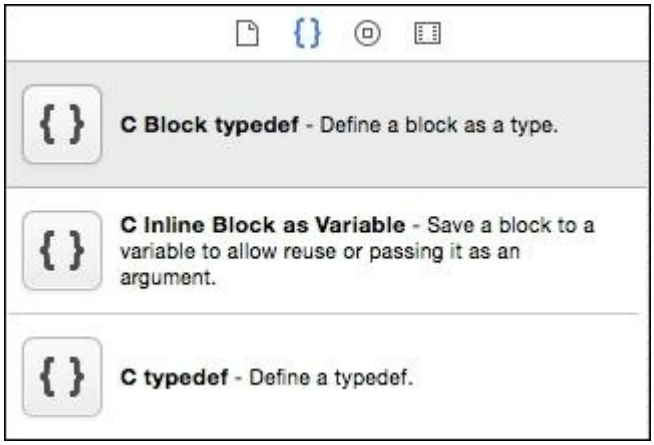
If you want to create a new file, you can either navigate to **File** | **New** or simply drag the required file type into the project navigator.



Here, if we want to, we can drag the **Swift File** template into our project, and once we release the left mouse button, Xcode will ask us to save the file. Then we will be able to rename the file and it will be included in the project.

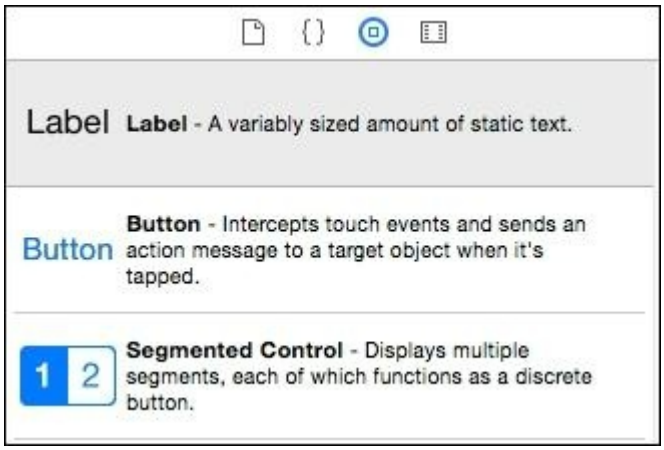
The Code Snippet Library tab

These contain a collection of code blocks that can be dragged and dropped into your file to remove rework.



The Object Library tab

In the object library, you have predefined objects such as **Text**, **Label**, and **Button** that can be dropped into the ViewController.



The Media Library tab

This tab contains all media, such as **Movies**, **Audio**, and **Images**, that you have added to the library and dragged into your app. Currently, no media have been added, so this tab is empty.

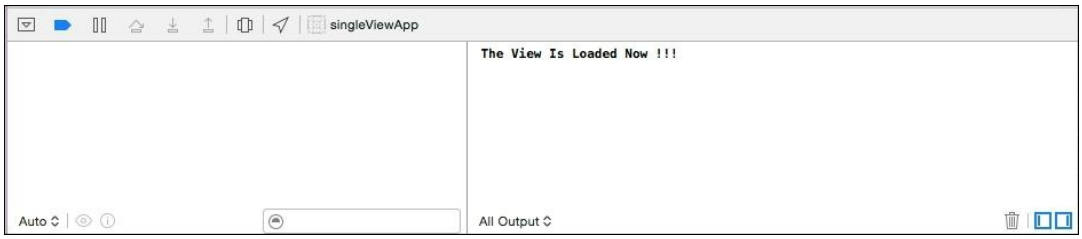


This concludes the section on the **Utility** panel. Now that we have covered the other files and looked at the **Utility** panel, let's look at the last folder in the project hierarchy.

The `Supported Files` folder/group contains only an `info.plist` file, which consists all of the necessary information about the app. It will contain details such as the name of the app, bundle name, version number of the app, and other basic information about the app to Xcode. Mostly, we will be making changes to this file only when we upload an update to an application that is already live on the store, and we would want the version number of the latest build to be 1.1 instead of 1.0. We will cover this when we publish the app on the App Store in the last chapter.

The Debug panel

There is one more panel that we haven't covered yet, and that is the **Debug** panel. This can be activated by clicking on the middle button from the three-button set to the right of the toolbar and left to the **Utility** panel. Once you click on it, the panel shown in the following screenshot will open up. You can click on it again to hide it.



The **Debug** panel or area has two subsections. To the left is the **Variables** view, and to the right is **Console**. Whatever you log using the `println` function will get displayed here. For example, in the ViewController class in the `viewDidLoad` function, I logged this:

```
println(" The View Is Loaded Now !!!")
```

When I ran the application, the message popped up in the console. So, similar to how we logged information in the playground, we can do it in Xcode, but to see it, you obviously need to run the application—unlike playground.

Now that we have this awesome app, we can see how to run it on the device.

Running the app on the device

All this time, we ran the app on the simulator. Making the app run on the simulator is relatively easier—select the simulator to run the app on, click on the play button, and that's it! To run the app on a device, however, you'll need to perform a couple of steps.

First, we need to get the **Developer** certificate and install it. So, go to the iOS developer portal at https://developer.apple.com, click on **Member Center**, and type in the login name and password we created in the Chapter 1, *Getting Started*:

1. Under **Developer Program Resources**, click on **Certificates, Identifiers & Profiles**.



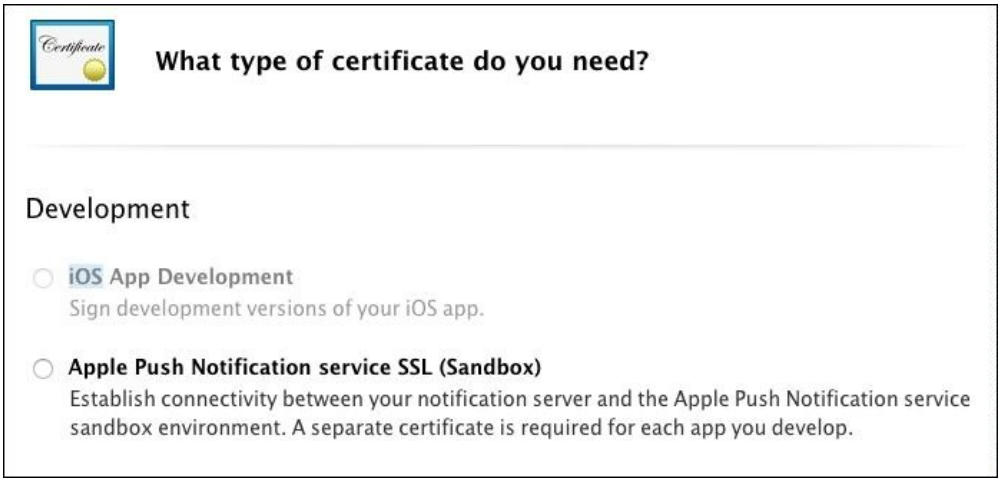
2. Under **iOS Apps**, click on **Certificates**.



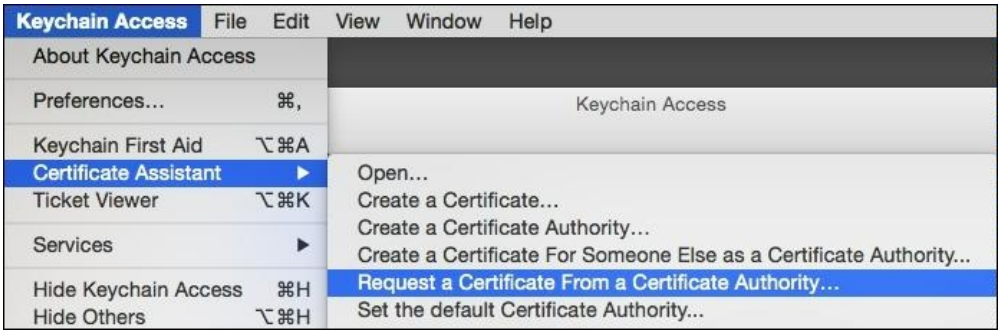
3. Under **Certificates**, click on **Development**. In the top-right corner, click on the **+** sign next to the search button.



4. Next, click on **iOS Development** once the page loads and then click on **Continue**. Then, we have to create a certificate signing request.



5. On your Mac, open *Keychain* by clicking on *Launchpad*. Once the *KeyChain* app is opened, go to **CertificateAssistant** and click on **Request a Certificate From a Certificate Authority...**.



6. In the popup, type the e-mail address that you used to create the Apple Developer ID. Then type a common name for reference, select **Saved to Disk**, and click to continue to select the location where you

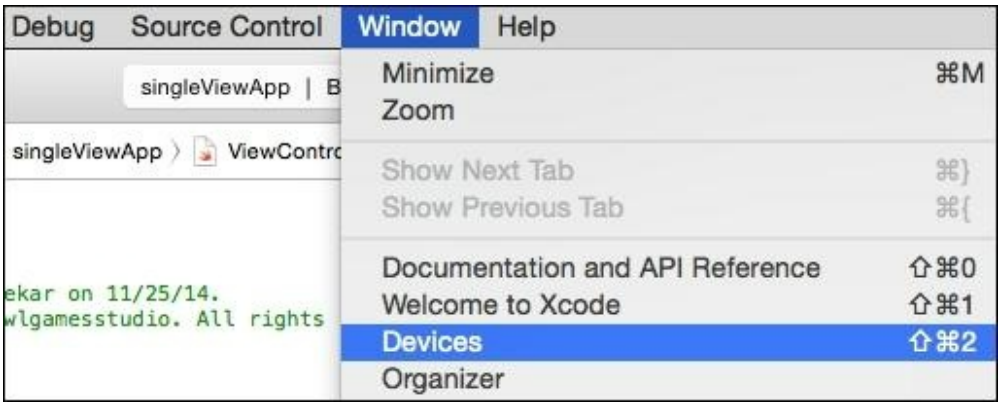
want the certificate to be downloaded. You can save it on the desktop for now, but later, move it to a safer place.



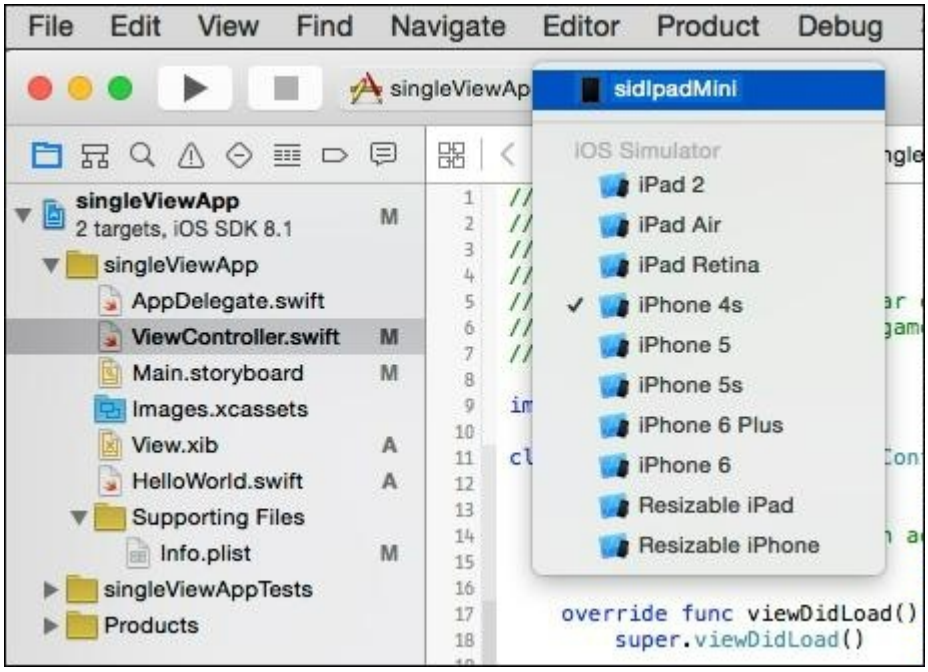
7. Go back to the iOS Developer portal and click on **Continue**. Now, you have to upload the CSR file that you saved on the Mac in the previous step. Click on **Choose file**, navigate to the desktop where you downloaded the file, select it, and click on **Generate**.
8. Now your certificate is ready. Click on **Download** to download the file.
9. Double-click on the downloaded file to install it. Then, click on **Done**.
10. Now, open Xcode, go to **Preferences**, click on **Accounts**, and then click on **View Details**. In the top part, it should show the iOS Development meaning that the certificate is installed.

    This certificate is only for development. For distribution, you will have to download a distribution certificate, which we will do once we are ready to publish our game.

11. Now, plug in your device to Mac. Xcode will automatically assume that you will use the device connected for development and register it for you. You can see all the registered devices. Open Xcode and, from the top bar, select **Window** and then **Devices** to show all the registered devices. You can have up to 100 devices registered to test your application.



12. Now, in the top-left corner next to the play button, from where we previously used to select the simulator, we can select the device.



13. There's one final step: we still have to select the team to make the app run on the device. So, go to the **Project Navigator** tab and select the project root. In the Editor, under team, scroll down to select the team.



Now, if everything goes well, you should be able to run the app on the device. Congratulations!!!!

Summary

In this chapter, we looked at the interface of Xcode. I hope that you now have a good idea about the different panels and how to access, activate, and deactivate them. I intentionally haven't covered the **File**, **Edit**, and **View** menus because these are somewhat similar to other applications. If needed, we will go through some of them on a need-to-know basis.

As an exercise, you can try and change the color of the view, or change the height and type in the `.storyboard` file. You can also make similar changes in the NIB file and the app a little more colorful.

Now that you know how Xcode works, we will jump straight into game development using SpriteKit. Once we have mastered 2D space, we will look at the 3D space in SceneKit.

Chapter 4. SpriteKit Basics

After an entire chapter of theory, we have finally reached the chapter where we will be creating a game. I am sure that this is a moment you have been waiting for and your fingers are aching to write some code and make a game.

In this chapter, you will create a small and basic game using SpriteKit. We will see how to create the main menu of the game, and you will learn how to transition from the main menu scene to the gameplay scene, where all of the gameplay code will be written.

In the gameplay scene, we will add sprites, such as the background and hero, first. We will then create a small physics engine to make the hero move around. Then, we will add in the enemies, and move them too. Next, we will make the hero and the enemies shoot at each other. We will detect collision between the hero's rockets and the enemies, and between the enemies' bullets and the hero. For each enemy the hero shoots, we will get one point, but if any of the enemies go past the left of the screen, it will be game over. If the current high score is greater than the previous score saved, then your current score will be saved as the new high score. Once the game is over, the player can click on the button to go back to the main menu to start the game. I hope you are excited! Let's finally jump in.

The topics covered in this chapter are as follows:

- Introduction to SpriteKit and SKScene
- Adding a main menu scene and a gameplay scene
- Adding and moving the Hero sprite
- Creating interactivity with touches
- A simple physics engine
- Spawning enemies
- Firing hero rockets and enemy bullets
- Collision detection
- Scoring and game over conditions
- Displaying, saving, and retrieving the score

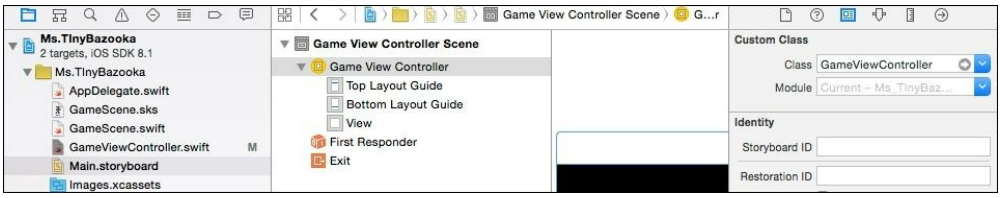
Introduction to SpriteKit and SKScene

We have already seen in Chapter 1, *Getting Started*, how to create a SpriteKit project. Just to jog your memory, I will show you once again how to create a project. Click on **Xcode**, and then on **Create a new Xcode Project**. Then, on the left-hand side panel, navigate under **iOS**, and then under **Application** and select **Game**. Then click on **Next**. Give a new name

to the project. Select the language as **Swift**, the game technology as **SpriteKit**, the device as **iPad**, and then click on **Next**. Select the location where you want the project folder to be created and click on **Create**.

You will see that the majority of the project structure remains similar to the SingleView project we saw in the previous chapter. We have the `GameScene.sks`, `GameScene.swift`, and `GameViewController.swift` files:

- `GameScene.sks`: This is a serialized `SpriteKitScene` file. This is used to create SKScenes visually without writing code. So, for example, you can drag-and-drop images and design them as buttons, and by clicking on them, you can make them perform different functions. But since we will be writing it all in code, we won't be using this file to create the interface of the game.
- `GameScene.swift`: This is inherited from SKScene. SKScenes are the building blocks of games. This class is called once the application view is loaded. You can create SKScene files to create the main menu Scene, gameplay scene, options scene, and so on. In fact, we will later rename the `GameScene.swift` file to `MainMenuScene.swift` and create a new scene called `GamePlayScene`, where will write our gameplay code.



- `GameViewController.swift`: This class is similar to the `ViewController.swift` file we saw in the previous chapter. In the `Main.Storyboard` file, you will see that there is a `GameViewControllerScene` file instead of `ViewController`, but the structure is very similar to it. If you click on **GameViewController**, you can see that it calls the `GameViewController` class on the **Utility** panel under the **Identity** inspector. Refer to the preceding diagram.

Now, open the `GameViewController.swift` file. You will see some new functions and some older functions that we saw in `ViewController.swift`. You will also notice that SpriteKit has been imported. We will need to import SpriteKit in all the classes in which we want to use its features. All classes and objects that are part of SpriteKit start with the prefix `SK`, so SpriteKit Scenes are `SKScene`, sprites are `SKSpriteNode`, and so on.

A `SpriteKitNode` or `SKNode` is the basic building block required for creating any content in SpriteKit, but unlike an `SKScene` or `SKSpriteNode`, it doesn't draw any visual content. However, both `SKScene` and `SKSpriteNode` are child classes of `SKNode`. So, if `SKScene` is the building block of any game,

`SKNode` is the basic building block of SpriteKit itself. A detailed explanation is provided under the SpriteKit section in Chapter 1, *Getting Started*.

After importing the SpriteKit, we see that an extension of SKNode is created with a class function called `unarchivedFromFile`, which takes a string and returns an SKNode. The following function is used to load the `.sks` file we saw earlier:

```
extension SKNode {
    class func unarchiveFromFile(file : NSString)
-> SKNode? {
        if let path =
NSBundle.mainBundle().pathForResource(file,
ofType: "sks") {
            var sceneData =
NSData(contentsOfFile: path,
options: .DataReadingMappedIfSafe, error: nil)!
            var archiver =
NSKeyedUnarchiver(forReadingWithData: sceneData)


 archiver.setClass(self.classForKeyedUnarchiver(),
forClassName: "SKScene")
            let scene =
archiver.decodeObjectForKey(NSKeyedArchiveRootObjectKey)
as GameScene
            archiver.finishDecoding()
            return scene
        } else {
            return nil
        }
    }
}
```

After the extension, we see the actual class of `GameViewController`. It is still inheriting from `UIViewController`. Similar to the `ViewController.swift` file, the first function called here is `viewDidLoad`, which is the function that is called as soon as the view gets loaded. The `super.viewDidLoad` function is called, which calls the `viewDidLoad` of the parent class. Then, the `GameScene.sks` file is loaded using the extension that was created earlier. The `if let` statement checks if the object scene is empty or not. If it is not empty, then the code in the `if` block will be executed.

Note

`extension`: In Swift, you can add functionality to an existing class. In the preceding case, we are adding a new function called `unarchivedFromFile` to the `SKNode` class, which unarchives files and

returns an SKScene. This function is used to unarchive the `SKS` file in the `viewDidLoad` function in the following code.

`if let`: This checks if the object scene is empty or not. If it is not empty, then the code in the `if` block will be executed.

`as`: This operator is used to downcast SKView since it is actually a subclass of `UIView`.

```
    override func viewDidLoad() {
        super.viewDidLoad()

        if let scene =
GameScene.unarchiveFromFile("GameScene") as?
GameScene {
            // Configure the view.

            let skView = self.view as SKView
            skView.showsFPS = true
            skView.showsNodeCount = true

            /* Sprite Kit applies additional
optimizations to improve rendering performance */
            skView.ignoresSiblingOrder = true

            /* Set the scale mode to scale to fit
the window */
            scene.scaleMode = .AspectFill

            skView.presentScene(scene)
        }
    }
```

A new variable, `skView` is created and the current view is assigned to it by type casting it as an SKView since the root view of `GameViewController` is an SKView.

Then, the `showsFPS` and `showsNodeCount` properties of the scene are set to `true`, which will show the **FPS** and **Node Count** on the bottom-right of the screen.

The `ignoreSiblingOrder` property is set to `true`, meaning that if one or more objects are at the same depth, then it won't prioritize between them and all objects will be drawn at the same depth.

The value of **Z order,** or depth order, decides which object is at the front and which object is at the back of the screen. The object with the smallest **Z** value is kept at the back of the screen and the object the with highest value is the closest to the screen. If no Z order value is assigned to an object, SpriteKit will assume that this object is above the previous object added.

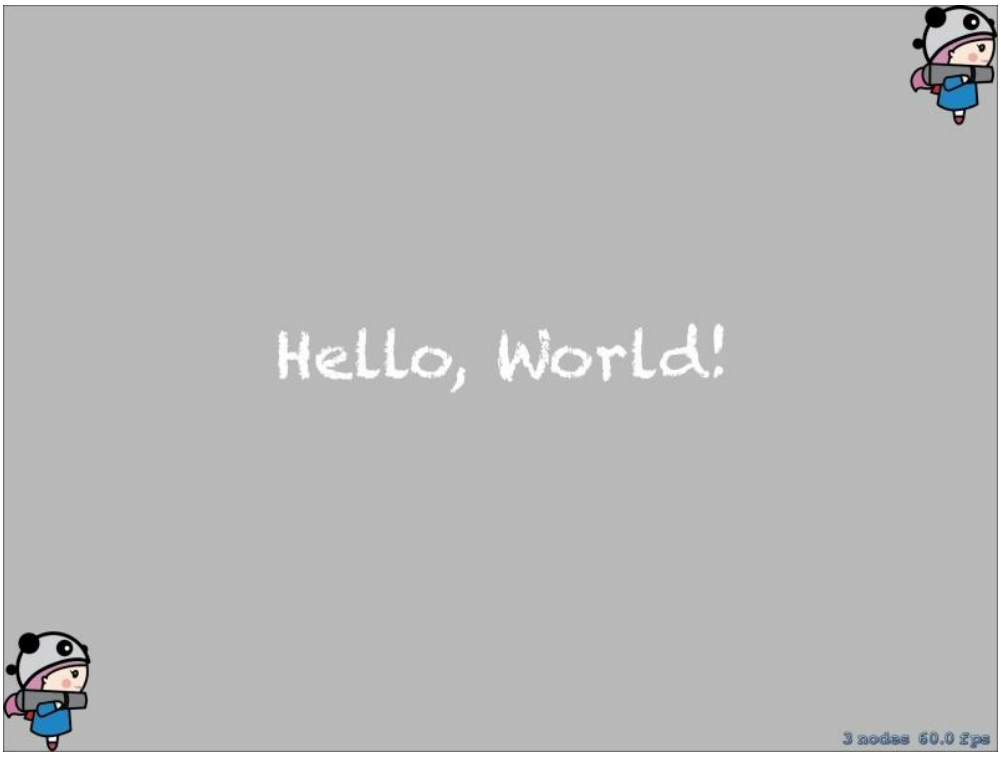
That is why in all games, the background is added first so that it is at the lowest Z order and other objects such as the hero are added next. If you were to add the hero first and then the background, the hero would be at the lowest Z order and the background image that covers the whole screen would be above it. You might think that there is something wrong with the code or SpriteKit since the hero is not being displayed only the background. The fact is that the hero is there but he is behind the background. So, be careful about Z orders as this may lead to bugs or unexpected results in games.

After setting the order, we can set the `scaleMode` property of the scene. Here, by default, it has been set to `AspectFill`. There are four modes: `AspectFill`, `Fill`, `AspectFit`, and `ResizeFill`.

- `AspectFill`: This is the default mode when you create a new project. In this scale, both *x* and *y* scale factors are calculated and the large-scale factor is chosen to fill the view and maintain aspect ratio of the image. This will lead to cropping of the scene.

  Let us create a project and place characters at the top-right and bottom-left corners, and observe what happens when we move from the landscape mode to the portrait mode.

  In the landscape mode, both characters are displayed as they should be. One at the bottom left and the other at the top-right corner of the screen, as shown in the following screenshot:



But in the portrait mode, they have gone out of bounds of the screen, as shown in the following screenshot:



- `Fill`: Both *x* and *y* axes are scaled to fill the view. The view is the region that is shown once you click on the view in the `Main.Storyboard` file. The aspect ratio of the image will change both in terms of the width and height, to fill the view.

  If we do the same test with `.Fill,` once again in landscape mode, the images appear to be normal, but in the portrait mode, the two images are in their respective locations, but they are squashed to fit, as shown in the following screenshot:



- `AspectFit`: Instead of the upper scale factor, the lower scale factor will be chosen to maintain the aspect ratio of the scene. This may lead to letterboxing of the scene, but all the content of the scene will be displayed and will be visible in view.

  With this mode, once again everything looks fine in the landscape mode, but then in the portrait mode, the image is not at all squared; the whole scene is scaled down to fit to the width of the screen. This will cause letterboxing on the top and bottom of the screen, as shown in the following screenshot:



- `ResizeFill`: The scene is not scaled at all. It is just resized to fit the view. The images will maintain the original size and aspect ratios.

  Here, since the aspect ratio and scale are maintained, the bottom-left image is shown at the right position but the top-right image goes out of bounds of the screen, as shown in the following screenshot:



Looking at the preceding screenshots of the four modes, we can see that not all sizes fit all. You need to tinker with the scale mode to best suit the needs of your game. Since our game is primarily designed to be played in the landscape mode, we will just disable the portrait mode. So, in the main project node, disable the **Portrait** mode and **Upside Down** by unchecking it in the **General** tab, as shown in the following screenshot:



We will also be using `ResizeFill` as we will be providing separate images for Retina and NonRetina assets of the devices, so that aspect ratio is not affected, resulting in nice full-screen images instead of cropped or scaled images. So, in the `GameViewController` class, change the scale mode to `ResizeFill` as follows:

```
                /* Set the scale mode to scale to fit
the window */
                scene.scaleMode = .ResizeFill
```

Finally, the scene is loaded and presented using the `presentScene` function of the `skView` object.

Once the `GameScene.swift` file is presented, the `didMoveToView` function is called, which, as we saw in Chapter 1, *Getting Started*, shows the `SKLabelNode` label showing the **Hello, World!** text, and each time you click on the screen, the `touchesBegan` function gets called, and at the location of the touch, a SKSpriteNode is created and an action is run on it.

There are three new functions: the `shouldAutoRotate` function, which is set to `true`, and will rotate the view if the device is rotated; the `supportedInterfaceOrientation` function, which checks for the orientation and aligns view accordingly; and the `prefersStatusBarHidden` function, which hides status bar elements, such as network and battery indicators on the top of the screen. You can enable or disable them depending upon your needs.

We will now change the `GameScene` to start making our game.

Adding a main menu scene

Let us make some changes to the `GameScene` class to make it our main menu scene:

1. Rename the file to `MainMenuScene.swift` by selecting the file in the project hierarchy in the project navigator.
2. Change the name of the class in the file to `MainMenuScene`.
3. Delete all the lines of code inside `didMoveToView`.
4. In the `touchesBegan` function, delete the code related to the adding of the sprite and the running of the action upon it.
5. Delete the `update` function as it is not required for the main menu scene. If required, we will add it later.

The `MainMenuScene.swift` file should look like the following code snippet, as we deleted all the code from the `didMoveToView` function and modified the `touchedBegan` function:

```
import SpriteKit

class MainMenuScene: SKScene {

    override func didMoveToView(view: SKView) {

    }

    override func touchesBegan(touches: NSSet,
withEvent event: UIEvent) {
        /* Called when a touch begins */

        for touch: AnyObject in touches {
            let location =
touch.locationInNode(self)

        }
    }
}
```

Delete the `GameScene.sks` file from the project hierarchy by moving it to **Trash**.

We also need to make some changes to the `GameViewController.swift` file as well.

1. Delete the extension created for SKNode
2. Remove the `if let` scene line and the opening and closing bracket since we will be calling the `MainMenuScene` class directly through code.

3. Replace the above line with `let scene = MainMenuScene(size: view.bounds.size)`. The SKScene constructor takes the size of the screen, so here we get it from the `bounds.size` property of the view.
4. Change `.AspectFill` to `.ResizeFill`.

The rest of the file can remain the same. Now the `viewDidLoad` function should look like the following code:

```
    override func viewDidLoad() {

        super.viewDidLoad()

            let scene = MainMenuScene(size:
view.bounds.size)

            // Configure the view.

            let skView = self.view as SKView
            skView.showsFPS = true
            skView.showsNodeCount = true

            /* Sprite Kit applies additional
optimizations to improve rendering performance */
            skView.ignoresSiblingOrder = true

            /* Set the scale mode to scale to fit
the window */
            scene.scaleMode = .ResizeFill

            skView.presentScene(scene)
    }
```

Let's start adding content to the main menu scene next.

In the `MainMenuScene.swift` file in the `didMoveToView` function, we will first add the background image, then we will add a label that will display the name of the game, and then we will finally add the play button, which, if clicked, will launch `GamePlayScene` and start the game.

To add the background image, add the following code:

```
let BG = SKSpriteNode(imageNamed: "BG")
BG.position = CGPoint(x: viewSize.width/2, y:
viewSize.height/2)
self.addChild(BG)
```

We create a constant variable called `BG` and assign an image set named `BG` to it. Then, we position this image. For positioning the image, we need the size of the view. It is very simple to get the width and height of the view. We create a new constant called `viewSize` of type `CGView` and assign

`view.bounds.size` to it. So, add the following at the start of the `didMoveToView` function:

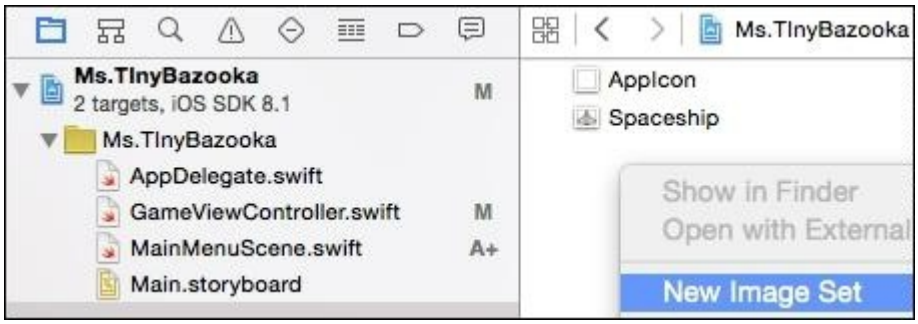
```
let viewSize:CGSize = view.bounds.size
```

We can now set the position of `BG`. To set the position, we assign `BG.position` equal to half of the width and half of the height of the size of the view. Whenever we need to assign or create a new `CGPoint` variable, we have to call CGPoint, and in the brackets provide the `x` and `y` values separated by a comma. The `x` value needs to be prefixed with `x` and then a colon and, similarly, the `y` value needs to be prefixed with `y` followed by a colon.

For the background to get displayed, we will have to call the `addChild` function on self, and pass in the background created.

If you run the game now, it will give errors as we have still not assigned an actual image to the project. For this, go to the `Resources` folder of this chapter and copy all the assets onto the desktop. This will contain all the assets that will be used in this chapter.

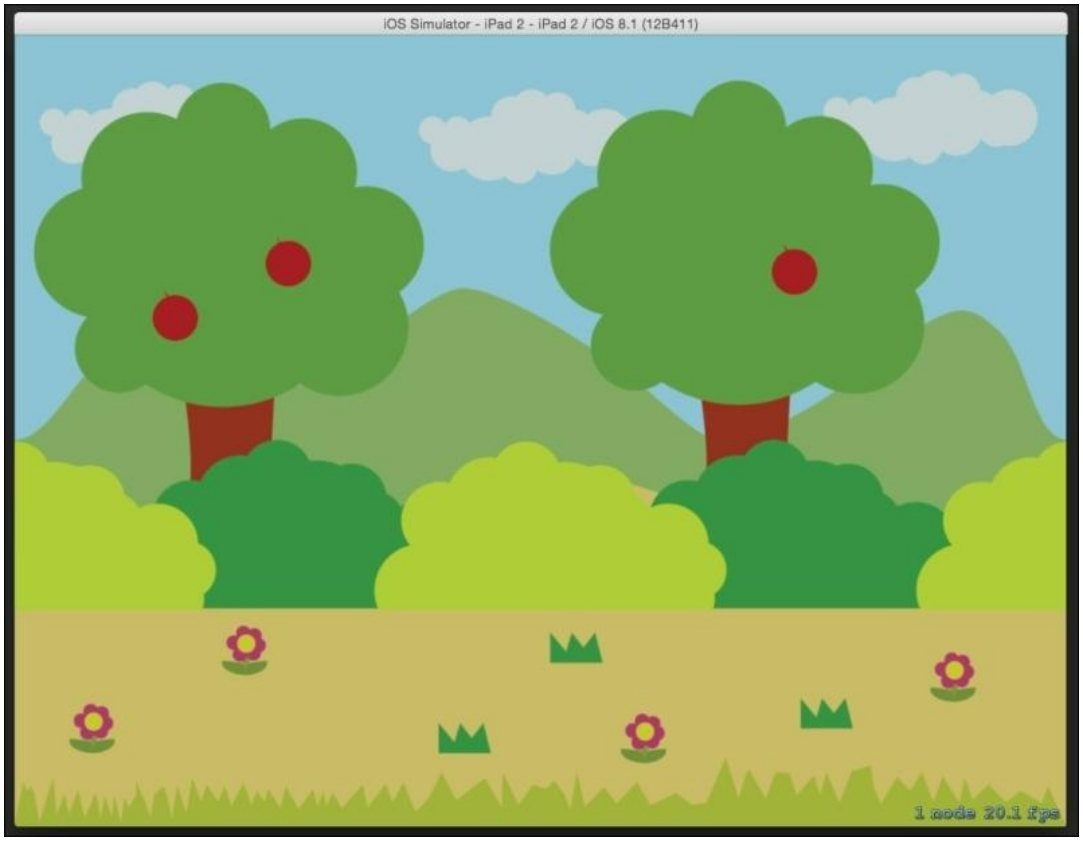
Now, go to the `Images.xcassets` file in your project navigator, and right-click on the panel and select **New Image Set**, as shown in the following screenshot. A new file called `Image` will be created. Select and rename it to `BG`. When we call `BG` while creating the sprite for the background, we are actually referring to this file. So, if you name it incorrectly, the code will give errors.



The file has placeholders for **1x**, **2x**, and **3x** images. Since we are making the game for the iPad, there are only two resolutions we have to worry about; the 1024 x 768 and 2048 x 1536 resolutions. Our BG is also of the same two resolutions. In the `Resources` folder, look for image files `Bg.png` and `Bg2.png`. Drag `Bg.png` to the **1x** box and `Bg2.png` to the **2x** box, as shown in the following screenshot:



Now you can run the application and the screen will display the background image in all its glory. On the simulator, you can select **iPad2** or **iPadAir**, and you will see the image will fill the entire screen, as shown in the following screenshot. Make sure you are running in the landscape mode.



Next we will add the label to display the name of the game. Labels are used to display text onto the screen. Add the following code to display the label right after we added the BG to the scene:

```
let myLabel = SKLabelNode(fontNamed:"Chalkduster")

myLabel.text = "Ms.TinyBazooka"

myLabel.fontSize = 65
```

```
myLabel.position = CGPoint(x: viewSize.width/2,
y: viewSize.height * 0.8)

self.addChild(myLabel)
```

We create a new constant called `myLabel` and call the constructor of `SKLabelNode`. It takes the name of the font we want to use to create the text, so we pass in `Chalkduster`, which is one of the default fonts in Mac. In `myLabel.text`, we pass in the actual text that we want to display. Next we assign the size of the font, its position, and add it to the current class as a child. To create text, we don't have to create an image set but we have to have the font in the system as it is automatically taken from the system's font directory.

Let us create the play button next. In the `Resources` folder, you will find `playBtn.png` and `playBtn2.png`. Similar to how we created an image set for `BG`, create one for the play button by naming the file `playBtn`. Drag the `playBtn.png` image to **1x** and `playBtn2.png` to **2x**.

For all the assets in the `Resources` folder, you will find two of each, one with the filename and the second one ending with a `2` at the end. Make sure, henceforth, that the regular filename asset is assigned to **1x** and one with the `2` at the end of the file is assigned to **2x**.

Now appropriate images are assigned to the `playBtn` image set. Add the following code right under where we added the code for the label:

```
let playBtn = SKSpriteNode(imageNamed: "playBtn")

playBtn.position = CGPoint(x: viewSize.width/2,
y: viewSize.height/2)

self.addChild(playBtn)

playBtn.name = "playBtn"
```

The `playBtn` image set is also a regular SKSpriteNode, so similar to how we added `BG`, we will assign the `playBtn` image set to the `playBtn` constant. Position it at the center of the view and then add it to the view.

In addition to what we do usually, I have also assigned a name to the `playBtn` constant so that we can refer to it later, if needed. It is not necessary that you assign a string; you can even assign an integer value if you wish. It should be named something that you find easy to remember and associate the constant with.

Now, if you build and run the project, it should look like the following screenshot:



Next, we will add code in the `touchesbegan` function to check whether the play button was pressed. In the `touchesbegan` function, we first check whether any object was touched on the screen, and then, if it was touched, we get the location of the touch. After getting the location, we add the following code:

```
        let _node:SKNode = self.nodeAtPoint(location)

        if(_node.name == "playBtn"){

            let scene = GamePlayScene(size: self.size)
            self.view?.presentScene(scene)
    }
```

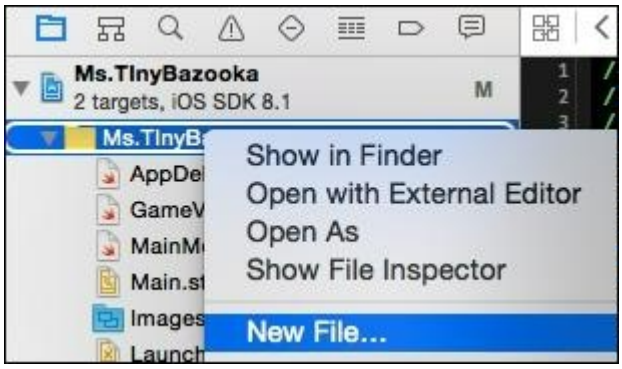
We create a new constant called `_node` of type `SKNode` and get the node that is at the touched location. We then check whether the name of the node pressed is `playBtn`, if it is pressed, then we create a constant named `scene` and assign `GameplayScene` to it, and then present the scene like we presented `MainMenuScene` in the `GameViewController` class. Since we have not created `GamePlayScene`, you will get an error. Don't worry, we will be creating it in the next section.

The question mark after `self.view` checks if the view is empty or not. If it is empty, it will give an error, but since the view exists, it won't give an error.

Let us create the gameplay scene so that we don't get errors saying that it doesn't exist.

Adding a gameplay scene

All this time, we have been modifying the files already included with the base project. Now we will create a new file in the project. Right-click on the base project folder and click on **New File**:



In the left panel, select **iOS** and select the Swift file, and then click on **Next**. It will ask for the name of the file, call it `GamePlayScene` and click on **Create**. This will create an empty Swift file.

Add the following code in it. This is the basic structure required whenever you create a new scene file:

```
import SpriteKit

class GamePlayScene: SKScene {

   required init?(coder aDecoder: NSCoder) {
        fatalError("init(coder:) has not been
implemented")

} // required init

    override init(size: CGSize){

        super.init(size: size)

    } //init function

} //class end
```

We first import SpriteKit, and then we create the class using the `class` keyword followed by the name of the class and inherit from `SKScene`.

Then we have the `required init` function after that. Since superclass `SKScene` implemented it, it has to be included in all the subclasses. This is a

requirement, so there is no avoiding it, but we won't be using it for anything as we will be using the regular `init` function.

The regular `init` function of an SKScene takes in the size of the view. Then we have to make sure we call the `super.init` function and pass the size of the view in it as well.

That's it, and we are ready to add some gameplay code in this class. You can check in the `MainMenuScene.swift` file that there are no errors and code is building properly.

In the `GamePlayScene.swift` file, first we have to create a global variable for the `viewSize`. So, between the class and the required `init` function, add `let viewSize:CGSize!` to make `viewSize` a global variable. Also, we use `let` instead of `var` as we know that the size of the view won't change in the middle of the game.

Since we are not initializing the constant here, we have to use an exclamation mark at the end to tell Swift that we will initialize it and we know that the type that we will initialize will be `CGSize`.

Initialize `viewSize` equal to the size that got passed in the `init` function. Add the following line after `super.init` is called:

```
viewSize = size
```

Adding a background and a hero

We are going add the background first, so we can copy and paste the same code from the `MainMenuScene` into the `init` function right after where we initialized `viewSize`:

```
let BG = SKSpriteNode(imageNamed: "BG")
BG.position = CGPoint(x: viewSize.width/2, y:
viewSize.height/2)
self.addChild(BG)
```

Next we will add the hero sprite. Similar to how we created the `BG` image asset, create a new asset called `hero` and assign `hero.png` and `hero2.png` to the **1x** and **2x** slots.

Next we want the hero to be a global variable aswell, as we will need to refer to her outside of the `init` function. So, right under where we created the `viewSize` property, add the following line of code at the top of the class:

```
let hero:SKSpriteNode!
```

Next, in the `init` function after where we added `BG`, add the following code:

```
hero = SKSpriteNode(imageNamed: "hero")
hero.position = CGPoint(x: viewSize.width/4, y:
viewSize.height/2)
self.addChild(hero)
```

Here, as usual, we assign the image set `hero` to the constant, assign the position, and add it to the scene.

Similar to how we positioned the background, we position the hero, but instead of adding the hero in the center, we place her at a distance of one fourth the `viewSize` from the left of the screen.

Updating the position of the hero

Next, let's update the position of the hero. Let us add gravity to the scene so that she starts falling down once the game starts. For updating her position, we will use the `update` function. The `update` function gets called as soon as the class gets created and it gets called 60 times a second. So, add the `update` function to the class as follows. Call a `updateHero()` function in it, we will define this function shortly:

```
override func update(currentTime: CFTimeInterval)
{
```

```
    updateHero()
}
```

Create a new global constant after the `let hero` line called `gravity` of type `CGPoint` and initialize it with `x` value `0` and a `y` value of `-1` as gravity only affects in the negative *y* direction:

```
let gravity = CGPoint(x:0.0, y: -1.0)
```

We will also create a new function called `updateHero` in which we will write all the code to update the hero's position. Create this function under the `update` function and don't forget to call this function in the `update` function, otherwise the hero's position won't be updated.

```
func updateHero(){

    hero.position.y += gravity.y

}
```

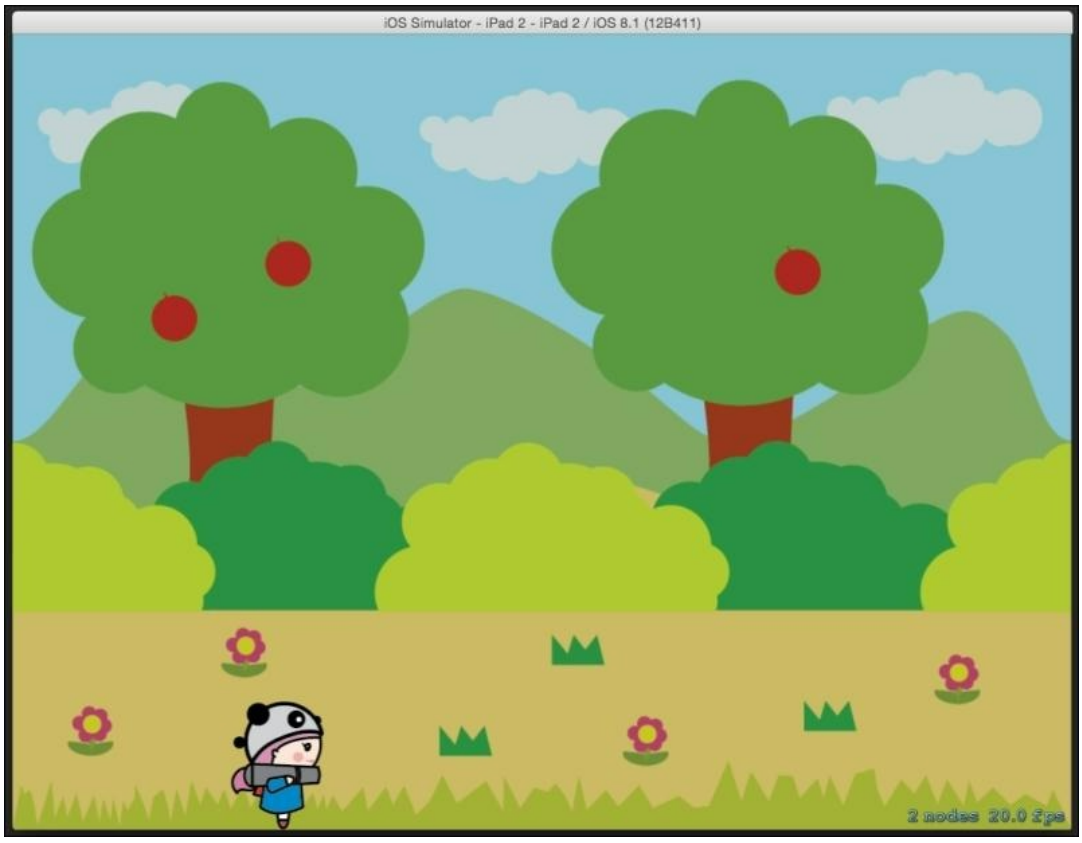
In the `updateHero` function, we are decrementing the `y` position of the hero in each update. Eventually, she will fall through the bottom of screen. To make her stay within the bounds, we check whether she is about to go beyond the screen and place her back at the bottom edge of the screen. To do this, add the following in the `heroUpdate` function right under where we decrement her position:

```
if(hero.position.y - hero.size.height/2 <= 0){

        hero.position.y = hero.size.height/2

}else if (hero.position.y + hero.size.height/2
>= viewSize.height){

        hero.position.y = viewSize.height -
hero.size.height/2
}
```

In the first `if` block, we check whether the bottom of the hero has crossed the bottom of the screen. If so, then we place the hero's origin at half her height from the bottom of the screen.

In the `else if` block, we check whether the top of the hero has crossed the top of the screen. If so, then we place her at half her height from the top of the screen.

Now, if you build and run the game and press play, the hero will be at one fourth of the distance from the left of the screen and will stop once she reaches the bottom of the screen.



Adding player controls

We will now add player controls by using the `touchesbegan` function. If the player taps the left side of the screen, the hero get pushed up and will then start falling again due to gravity, and if the player taps the right of the screen, the hero will fire rockets.

For detecting touches, add the `touchesBegan` function under the `update` function as follows:

```
override func touchesBegan(touches: NSSet,
withEvent event: UIEvent) {
  /* Called when a touch begins */

  for touch: AnyObject in touches {
       let location = touch.locationInNode(self)


  }
```

```
}//touchesBegan
```

This is obviously the same function that we used in `MainMenuScene` to detect touches on the play button. Since we are just going to be checking the location of the touch, we don't require the object we touched, for now at least.

To detect which side of the screen was tapped, add the following code under where we get the location of the touches in the `for in` loop:

```
  if(location.x < viewSize.width/2){

        println("[GamePlayScene] touchedLeftSide
")


   }else if(location.x > viewSize.width/2){

         println("[GamePlayScene]
touchedRightSide ")

   }
```

We check whether the `x` value of the touched location is less than half of the width of the screen. If so, then we print out that the left side of the screen was touched, otherwise we check whether the `x` value of the touched location was greater than half of the width of the screen, then in that case, we can confirm that the right of the screen was touched.

Now, to push the hero up in the air, we give her a small thrust each time the player touches the left of the screen. Add a global variable called `thrust` of type `CGPoint` and initialize both `x` and `y` values to zero as follows:

```
var thrust = CGPointZero
```

Note that we are using `var` and not `let`, as the value of `thrust` will change over a period of time. Also, `CGPointZero` is just short for `CGPoint(x:0, y:0)`. They will both do the same thing, so it is just a matter of convenience and preference.

In the `touchesBegan` function, right after we checked if the left part of the screen is tapped, add the following line:

```
thrust.y = 15.0
```

And in the `updateHero` function, change the line `hero.position.y += gravity.y` to the following:

```
thrust.y += gravity.y
hero.position.y += thrust.y

println("Thrust Y Value: \(thrust.y)")
```

Now, whenever the left side of the screen is touched, the hero will be pushed up by 15 points and then she will start falling down after reaching the highest position. Log the value of `thrust.y` as follows to see how it works:

```
[GamePlayScene] touchedLeftSide
Thrust Y Value: 14.0
Thrust Y Value: 13.0
Thrust Y Value: 12.0
Thrust Y Value: 11.0
Thrust Y Value: 10.0
```

Once the screen is tapped, the `y` value of `thrust` that was initially set to `0` is set to `14`. It is not 15 since we are subtracting 1 from it due to gravity. Then, at each update, the `y` position of the hero is slowly decreased until it becomes zero, and gravity will start acting again and start pulling the hero down.

One thing you will notice is that when the hero is at the bottom of the screen and you apply an upward thrust, the hero doesn't immediately start moving up. The answer to why this is also visible in the console output of `thrust.y`. As the gravity is added to the `thrust.y`, its value becomes huge and the small thrust of 15 has to overcome this value to make the hero move up again. To solve this, we have to set the value of `thrust` back to zero once the hero touches the top or the bottom of the screen. So, in the `updateHero` function, where we check whether that she touched the top or bottom of the screen, add the following line in both the `if` and the `if else` blocks after setting the hero's position:

```
thrust.y = 0
```

We will next add the rockets that fire when the right side of the screen is tapped. For this, we will create a new generic class so that we can use it later for creating enemies and enemy bullets.

As we created the `GamePlayScene.swift` file, create a file called `MovingSprite`, and in this file, add the following code:

```
import SpriteKit

class MovingSprite{

    let _sprite: SKSpriteNode!
```

```
    let _speed : CGPoint!

    init(sprite: SKSpriteNode, speed: CGPoint){

        _sprite = sprite
        _speed = speed

    }//init



    func moveSprite(){

        _sprite.position.x += _speed.x
    }

}//class
```

In this class, we import SpriteKit and then create the definition of the class. We create two global constants for holding the reference of the SKSpriteNode and CGPoint objects we will be passing into the constructor. The SKSpriteNode will hold the sprite that we will be passing in, and `CGPoint` will hold the speed with which we want the sprite to move by.

In the `init` function, we assign the objects passed in to the local objects we have created. We have added one more function called `moveSprite`. This will move the sprite with the speed that was assigned to it. That is all for this class for now. We will revisit and modify this class when we add the enemy and bullets class.

For creating the rockets, create a new function called `addRockets` in the `GamePlayScene` file. In it, we add the following code to create the rockets:

```
func addRockets(){

let rocketNode: SKSpriteNode =
SKSpriteNode(imageNamed: "rocket")
rocketNode.position = CGPoint(x: hero.position.x
+ hero.size.width/2 + rocketNode.size.width/2,y:
hero.position.y - rocketNode.size.height/2)

   self.addChild(rocketNode)

      let speed: CGPoint = CGPoint(x: 10.0, y:
0.0)
      let rocket: MovingSprite =
MovingSprite(sprite: rocketNode, speed: speed)

}
```

In the `addRockets` function, we first create a constant called `rocketNode` of type `SKSpriteNode` and assign a rocket from the `imageset`. So, create a new image set and name it `rocket`. In the `Resources` folder, you will find `rocket.png` and `rocket2.png`, which you can assign to the **1x** and **2x** slots, respectively, in the file.

Next we set the position of the rocket. Since we want the rockets to appear to be coming out of the bazooka, instead of spawning in at the position of the player, we place it at the front end of the bazooka. So, for the `x` position, we get the player's position, and then add half of the width of the player to it and also add half of the width of the rocket itself to it. For the `y` position, we get the y position of the hero and subtract half of the height of the rocket from it. We then add it to the scenes display list.

Next we create an object called `rocket` of the type `MovingSprite` and assign the speed with which we want to move the sprite and pass in the `rocketNode` we created earlier. For assigning the speed, we create a new constant called `speed` of type `CGPoint` and assign `10` and `0` to the `x` and the `y` values, respectively, so that whenever we call the `moveSprite` function of the class, the position will update in the *x* direction according to the value provided.

In the `touchedBegan` function, where we checked if the right side of the screen was clicked, add the `addRocket` function to create the rocket every time the right side of the screen is tapped:

```
    override func touchesBegan(touches: NSSet,
withEvent event: UIEvent) {
        /* Called when a touch begins */

        for touch: AnyObject in touches {
            let location =
touch.locationInNode(self)

                if(location.x < viewSize.width/2){

                    println("[GamePlayScene]
touchedLeftSide ")

                    thrust.y = 15.0

                }else if(location.x >
viewSize.width/2){

                    println("[GamePlayScene]
touchedRightSide ")

                    addRockets()
                }
```

```
            }
        }//touchesBegan
```

Now, if you build and run, and tap on the right side of the screen, the rockets will be created, but they are not moving. To move the rockets, we have to add each rocket we created, into an array, and on each rocket we have to call the `moveSprite` function to actually move the sprites.

For updating the position of the rockets, first we have to create an array to hold all the rockets. This array needs to be a global variable so that we can easily access it. So, right under where we added `var thrust = CGPointZero` at the start of the class, add the following line of code right under it. We create an array called `rockets` to hold objects of type `MovingSprite` and we are using `var,` as it is a mutable array, meaning we will be adding and removing objects from it during the course of the game.

```
var rockets:[MovingSprite] = []
```

Next, create a new function called `updateGameObjects` right under where we created the function for updating the hero, and add the following code:

```
func updateGameObjects(){

for(var i:Int = 0; i < rockets.count; i++){

      rockets[i].moveSprite()
      var sprite: SKSpriteNode =
rockets[i]._sprite
if((sprite.position.x - sprite.size.width/2) >
                              viewSize.width){

            sprite.removeFromParent()
            rockets.removeAtIndex(i)
      }
}
}
```

You might be thinking, why are we using a `for` loop instead of a `for in` loop? Well this is because once the rocket goes off screen, we have to delete the object, and for removing objects from the array in Swift, we require the index of the object to be removed and the `for in` loop doesn't have that feature.

So, we create a regular `for` loop starting from `0` and go through each object in the `rockets` array by incrementing the index by 1 every time. We call the `moveSprite` function on the ith object of the array. Next, for convenience, we get the `spritenode` from the index so that we can perform some checks on it. We check whether the left edge of the rocket sprite is beyond the width of the screen, and, if so, then we remove the sprite from its parent

node, which is the `GamePlayScene` as this is where we will add `addChild` in the `addRockets` function. Then, we remove the object at the current index in the rockets array by calling the `removeAtIndex` function of the array and pass in the current index.

Finally, we also need to add the objects to the array to delete it. So, in the `addRocket` function, add `rockets.append(rocket)` at the end of the function to add the rocket to the `rockets` array:

```
    func addRockets(){

        let rocketNode: SKSpriteNode =
SKSpriteNode(imageNamed: "rocket")
        rocketNode.position = CGPoint(
                    x: hero.position.x +
hero.size.width/2 +
                rocketNode.size.width/2,
                    y: hero.position.y -
rocketNode.size.height/2)
                    self.addChild(rocketNode)

        let speed: CGPoint = CGPoint(x: 10.0, y:
0.0)
        let rocket: MovingSprite =
   MovingSprite(sprite: rocketNode, speed: speed)

        rockets.append(rocket)
    }
```

Lastly, don't forget to call the `updateGameObjects` function in the `update` function right below where we call the `updateHero` function.

Now, when you build and run the game, you will be able to tap on the left-hand side of the screen to boost up the player and then tap on the right-hand side of the screen to shoot the rockets.

Also, take a look at the node count on the bottom right of the screen. Each time a new rocket is created, the node count will increase, and since the rockets are removed once they are off screen, the count will also decrease every time a rocket is removed from the scene.

You can also log the `rockets.count` to check how many rockets there are in the rockets array by adding the following in the `update` function:

```
println(" rockets count: \(rockets.count)")
```



Adding enemies

For the hero to be a hero, we need villains. So, we will add the enemies now. Similar to how we created the `addRocket` function, create a new function called `addEnemy`.

Also, create a new array called `enemies` just after the `rocket` array in the global variables of the `GamePlayScene` class. This adds a new array that will manage enemies, like the following:

```
var enemies:[MovingSprite] = []
```

You should now have something similar to the following in your variables declarations:

```
let viewSize:CGSize!
let hero:SKSpriteNode!
let gravity = CGPoint(x:0.0, y: -1.0)
var thrust = CGPointZero
var rockets:[MovingSprite] = []
var enemies:[MovingSprite] = []
```

We can now update all the enemies, update their positions and remove them when they leave the screen.

Create a new image set called `enemy` and add the `enemy.png` and `enemy2.png` to the file.

Unlike the rockets that spawn at the nozzle of the bazooka, the enemy will spawn from the right of the screen and move toward the left of the screen. They will also be spawning at different heights on the screen. It won't be challenging to the player if all the enemies spawned from the same position. So, we will create a random number, based on which we will decide at what height the enemy will be created.

Create the `addEnemy` function as follows:

```
func addEnemy(){

   var factor = arc4random_uniform(4) + 1
   var fraction = CGFloat(factor) * 0.20
   var height = fraction * viewSize.height

   println("enemy height: \(factor), \(fraction),
\(height)")

   var enemyNode:SKSpriteNode =
SKSpriteNode(imageNamed: "enemy")

enemyNode.position = CGPoint(x: viewSize.width +
 enemyNode.size.width/2, y: height)

self.addChild(enemyNode)

enemyNode.name = "enemy"

   let speed: CGPoint = CGPoint(x: -5.0, y: 0.0)

var enemy:MovingSprite = MovingSprite(sprite:
enemyNode, speed: speed)

enemies.append(enemy)
}
```

For creating a random number, we use the inbuilt function `arc4random_uniform`. This function takes in a value and generates a random number from `0` to one less than the number. So, in this case, since we have passed in `4`, it will create a number from `0` to `3`. Since we want a random number from `1` to `4`, we add `1` to it at the end. We assign this value to a variable called `factor`.

Then, we typecast this variable to `CGFloat` so that we can get a fraction value. Then, multiply this value by 0.20 and store it in a new variable called `fraction`. To finally get the random height at which the enemy needs to be created, we multiply the fraction with the height of the view, and assign it to a variable called `height`.

This way the enemy will be created at 20, 40, 60, or 80 percent of the height of the screen. We can't spawn the enemy at 0 percent or at 100 percent of the height since then, either the top or the bottom part of the enemy wouldn't be visible, because the anchor point for sprites are at the center of the sprite.

Now, similar to how we created the rocket, we create a new called `enemyNode` of type `SKSpriteNode` and assign the enemy image set to it. We have to place the enemy just beyond the right of the screen, so we get the width of the screen and add half the enemy width to it. For the height, we give the random height at which the enemy needs to be spawned in and add the enemy sprite to the scene. Finally, we will name the `enemyNode` sprite as `enemy` as we will need it later.

Next, since we need to create an instance of the `MovingSprite` class and provide the `enemySprite` node and speed, we will create a new `speed` object. Since this time we want the enemy to be moving in the negative *x* direction, we provide the value of `-5` in the `x` direction for speed, keeping the `y` value as `0` as we don't want the enemy to be moving in the *y* direction. Then, we create a new `MovingSprite` object called `enemy` and provide `enemySprite` and `speed` to it. At the end, we append the newly created enemy object to the `enemies` array so we create an array called `enemies` at the top, similar to how we created rockets for the hero.

Now we have to update through the enemy objects in the array and call `moveSprite` on the enemy to make it move in the negative *x* direction. We also need to make sure that we remove the enemy sprite from the parent class and then remove the enemy object from the `enemies` array. To do this, we add the following code under where we update the player rockets in the `updateGameObjects` function:

```
        for(var i:Int = 0; i < enemies.count; i++){

            enemies[i].moveSprite()

            var sprite: SKSpriteNode =
    enemies[i]._sprite

            if((sprite.position.x +
    sprite.size.width/2) < 0){

                sprite.removeFromParent()
```

```
                enemies.removeAtIndex(i)
            }
        }//update enemies
```

Similar to how we updated the hero rockets, we create a `for` loop and then call the `moveSprite` function on all the objects. Create a `sprite` node for convenience. Now, instead of checking if the object left from the right end of the screen, as the enemy is moving, the negative *x* direction, we check whether the right edge of the enemy is beyond the left of the screen, and if so, then we remove the sprite from the parent and then remove the object from the current index in the `enemies` array.

Since we are already calling `updateGameObjects` in the `update` function, we aren't required to add it again. But we should call the `addEnemy` function after every couple of seconds to spawn the enemy.

For actually spawning the enemy, we can use an action to call the `addEnemy` function after whatever duration we want. To do this, in the `init` function, add the following under where we added the hero to the scene:

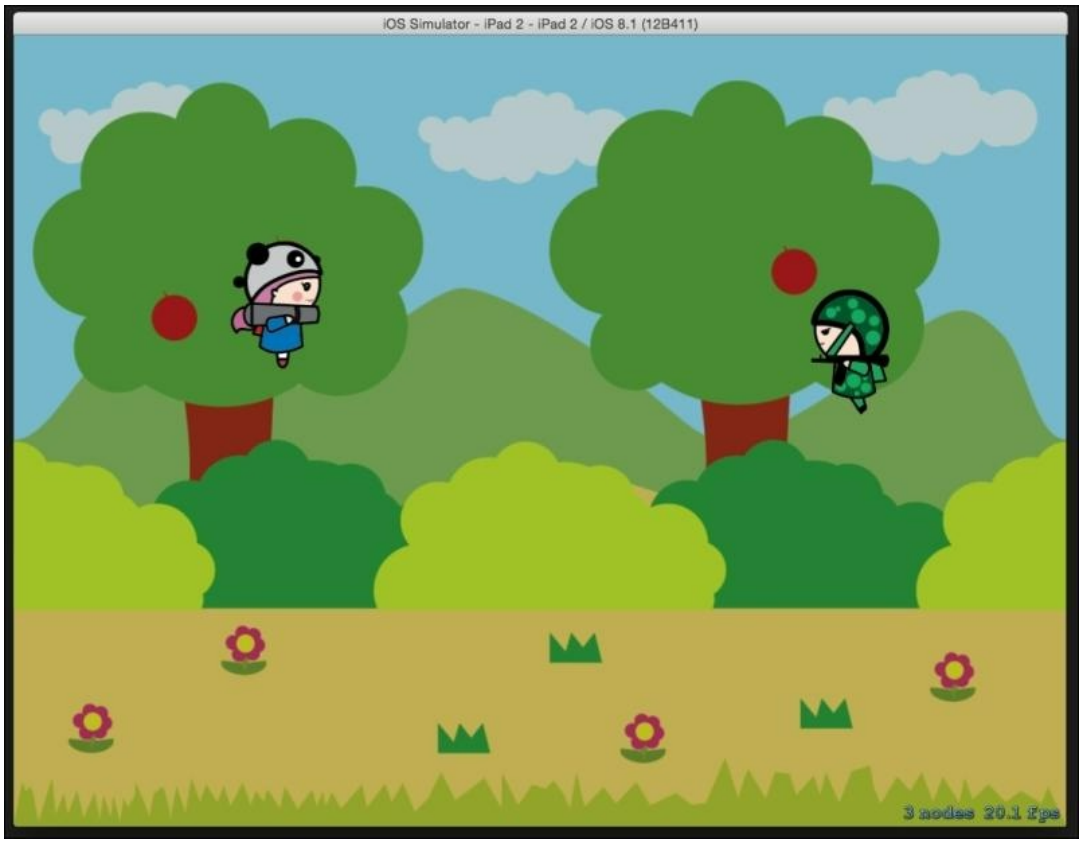
```
//spawn enemies after delay
let callFunc = SKAction.runBlock(addEnemy)
let delay = SKAction.waitForDuration(3.0)
let sequence = SKAction.sequence([callFunc,delay])
let addEnemyActiom =
SKAction.repeatActionForever(sequence)

self.runAction(addEnemyActiom)
```

First we create a few actions. All actions are of the type `SKAction`. The first action is `runBlock` in which we provide the function that we want to call, which is `addEnemy`. We assign this action to `let` called `callFunc`. Next we create another action called `waitDuration` and assign `3.0`, which is 3 seconds, and assign it to `delay`. The third action is a `sequence`. A `sequence` action lets you perform actions one after the other. So, here we first give `callFunc` and then `delay`. The square brackets indicate that the sequence is an array, so we can create a sequence of however many actions we want to call, by adding it to the array and then passing it into the sequence. In this sequence, the `callEnemy` function will be called first and then the action will wait for 3 seconds.

The last action is the `repeatActionForever` action and here we pass in the sequence action so that the sequence is called over and over.

Finally, we run the action on the current scene and provide `addEnemyAction`, which will eventually call the `addEnemy` function until we tell it to stop running the action.

Build and run now to see enemies popping up from the right of the screen, getting updated, and then once they leave the scene, getting deleted from the scene.



Adding enemy bullets

OK. The enemy seems to be carrying a rifle of some cartoony sort but he doesn't seem to be doing anything with it. Let's make him use it. We will make the enemy shoot bullets.

Create an image set called `bullet` and assign `bullet.png` and `bullet2.png` to the **1x** and **2x** slots, respectively, in `Images.xcassets`. Also, create a new global variable called `bullets` of type `array` at the top of the `GamePlayScene` class as follows:

```
var bullets:[MovingSprite] = []
```

For spawning the bullets, we will be using the `movingSprite` class. But we have to make some changes to it so that as soon as the enemy is created, he starts firing away with the rifle.

So, open the `movingSprite` class and add the following in the `init` function right after we initialize the global variables. Remember that we added a

name to the enemy sprite when we create it. This will come into use in the following:

```
if(_sprite.name == "enemy"){

let shootActiom =
SKAction.repeatActionForever(SKAction.sequence([SKAction.runBlock(shootBu

_sprite.runAction(shootActiom)
}
```

Here we first check the name of the sprite passed in, and if it is the enemy, then we create an action similar to how we created the spawning of the enemies in the `GameplayScene` class. The difference being that, instead of creating a separate variable for each action here, we are just creating one single action called `shootAction` and calling all the actions inside it.

So, basically, we are calling a `shootBullet` function, which we will create in the same class, that will be called every 3 seconds.

After we create `shootAction`, we call the action on the sprite so that it can start calling `shootAction`.

We will define the `shootBullet` function as follows. This can be added right under where we added the `moveSprite` function.

```
func shootBullet(){

   let _gameplayScene = _sprite.parent as
GamePlayScene
   _gameplayScene.addBullets(_sprite.position,
size: _sprite.size)

}
```

This function, in turn, will call a function called `addBullets` in the `GameplayScene`. Since we have added the hero, rockets, and enemy in the gameplay scene, it is better if we also add the bullets into the same scene, as it will be easier for us to cycle through the objects when checking for collision.

To get an instance of the `GameplayScene`, we will create a local constant called `_gameplayScene` and use the `.parent` property of the `SKSpriteNode` to get the parent class on to which the sprite was added. Since we added the enemy in `GameplayScene`, it will return `GameplayScene`. We still need to typecast it so that we use the `as` operator and typecast it to `GamePlayScene`.

Now, since we need to position the bullets properly, the same way we positioned the rockets for the hero, we will need to provide the position and

size of the enemy object while creating bullets. Assuming that, we will create a function called `addBullets`, we will call this function on the `GamePlayScene` by providing the position and size of the enemy sprite.

Now, let's go to `GameplayScene` and create the `addBullets` function as follows:

```
func addBullets(pos:CGPoint, size: CGSize){

   let bulletNode: SKSpriteNode =
SKSpriteNode(imageNamed: " bullet")

var newPos = CGPoint(x: pos.x - size.width/2 -


 bulletNode.size.width/2,
                        y: pos.y -
bulletNode.size.height)

   bulletNode.position = newPos
   self.addChild(bulletNode)

   let speed: CGPoint = CGPoint(x: -10.0, y: 0.0)
   let bullet: MovingSprite =
MovingSprite(sprite: bulletNode, speed: speed)
   bullets.append(bullet)

}
```

You should be quite accustomed to adding objects in SpriteKit now. Like the rockets, we create a `SKSpriteNode` called `bulletNode` and assign the bullet image set it. You know the drill.

We then create a new position, which will be at the left end of the enemy sprite. So, we take the current position of the enemy and subtract half the width of the enemy and half the `bulletNode` from the `x` position. For the `y` position, we get the `y` position of the position and subtract the full height of the bullet. In the next step, we assign this position to the position of `bulletNode`, and then we add it to the current scene.

We create a new speed variable and assign the speed with which we want the bullet to move. We create a new constant called `bullet` and provide the `bulletNode` and speed to it.

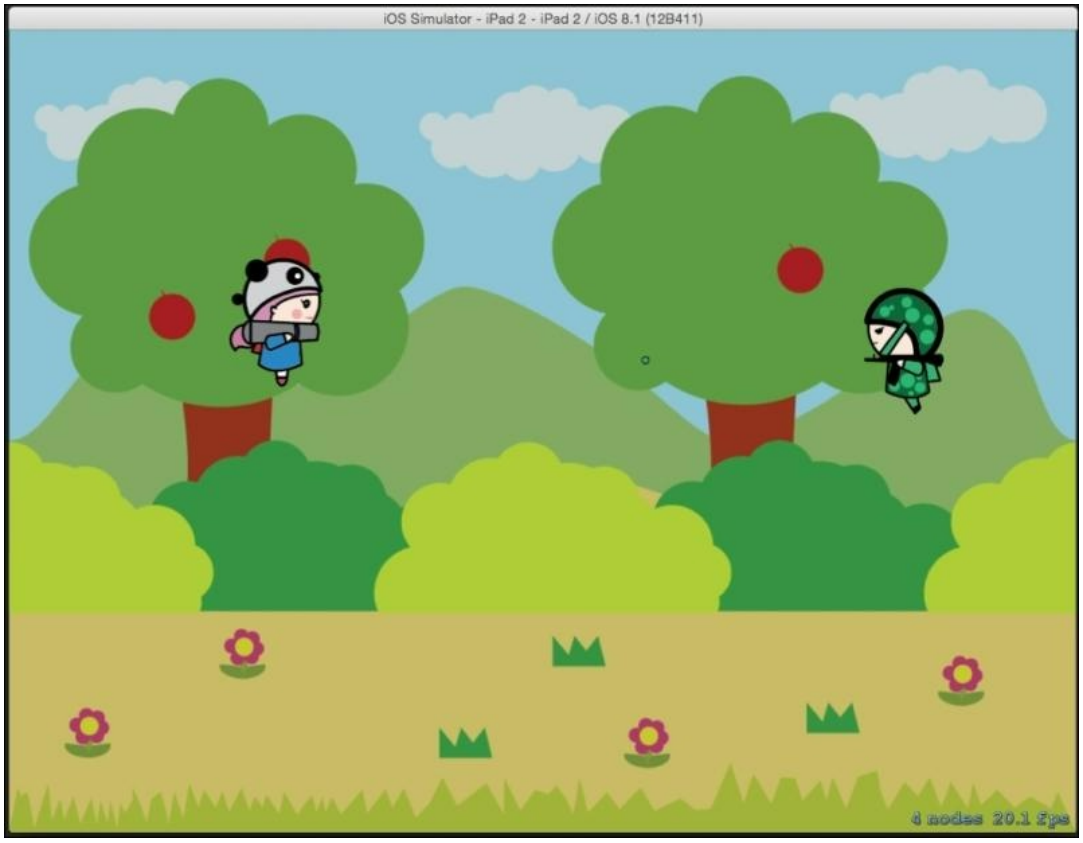
A bullets array needs to be created to append all the bullets created. So, create a new array called `bullets` that takes in `MovingSprite` and adds it to the top of the class.

After the bullet is created, append it to the `bullets` array.

We also need to update the position of the bullets and check so we can delete them once they have left the screen. So, similar to updating the enemy, we need to add the following code in the `updateGameObjects` function to update the bullets:

```
for(var i:Int = 0; i < bullets.count; i++){

    bullets[i].moveSprite()

    var sprite: SKSpriteNode = bullets[i]._sprite

    if((sprite.position.x + sprite.size.width/2)
< 0){

        sprite.removeFromParent()
        bullets.removeAtIndex(i)
    }
}
```

So, here we go through the bullets in the array, call the `moveSprite` function, and if the sprite has gone beyond the left of the screen, we remove it from the parent and remove the object in the current index.



Now we have all the elements needed to go to the next stage of development, that is, to check for collisions between the hero's rockets and enemies, and between the enemy's bullets and the hero. Based on

collision, we will be doing the scoring and checking for the game over condition.

Collision detection

Collision detection in 2D games is done using the `intersectsNode` function of the sprite class itself. We can check whether the current sprite is overlapping the other sprite. In the `intersectsNode` function of the current node, we pass in a node that we want to check for collision with the current node. If there is an intersection, the function will return `true`, if not, it returns `false`.

For checking collision, we will first check the collision between the enemy's bullets and the hero. If there is collision, then the game is over. Then we will check collision between the hero's rockets and the enemies. If we detect a collision, then we have to update the score.

Also, if the enemy goes beyond the left of the screen, the game is over, so we will call the `GameOver` function in this case also.

For checking collision, create a new function called `checkCollision` in the `GameplayScene` and add the following code:

```
        //Hero and Bullets

        for bullet in bullets{

            var sprite = bullet._sprite

            if(sprite.intersectsNode(hero)){

                GameOver()
            }
        }
```

As we wont be needing to know the index numbers, we will just use the `for in` loop to check collision.

Here we go through all the bullet objects in the `bullets` array. First, we assign the sprite of the bullet object to a local sprite variable. Then, we will call the `intersectsNode` function on the sprite to check whether it intersects the hero sprite. If it does intersect, then the `GameOver` function is called.

The `intersectsNode` function just takes the sprite of the node and checks if the box surrounding this sprite intersects with the bounding box of the sprite provided for it. If there is some overlapping, it will return `true` if the collision has occurred, or `false` otherwise.

Call the `checkCollision` function in the `update` function right after we update the hero and the game objects, as shown in the following code:

```
    override func update(currentTime:
CFTimeInterval) {
        /* Called before each frame is rendered */

            updateGameObjects()
            updateHero()
            checkCollision()

    }
```

Next, let's create the `GameOver` function. Once the game is over, we want to stop updating the hero and game objects, and stop checking for collision.

Also, create a global bool at the top of the class called `gameOver` and set it to `false`. It should be a `var` and not `let` as we will change it in the `GameOver` function:

```
var enemies:[MovingSprite] = []
    var bullets:[MovingSprite] = []
    var gameOver = false
```

Add the `GameOver` function to the `GamePlayScene` class right after the `update` function as follows:

```
func GameOver(){

   gameOver = true

   self.removeAllActions()

   for enemy in enemies{

       enemy._sprite.removeAllActions()
   }
}
```

Once the game is over, we set the `gameOver` bool to `false`. Next we call `removeAllActions` on the current class so that enemies will stop spawning, and then we also call the function on all the enemies that are currently present on the screen, so that the bullets stop spawning.

We also need to call the `GameOver` function when any of the enemies go beyond the left of the screen, so to update the position of the enemy, call the `GameOver` function in the `if` condition in the loop, as follows:

```
for(var i:Int = 0; i < enemies.count; i++){

        enemies[i].moveSprite()

        var sprite: SKSpriteNode =
enemies[i]._sprite

        if((sprite.position.x +
sprite.size.width/2) < 0){

            sprite.removeFromParent()
            enemies.removeAtIndex(i)

            GameOver()
        }
}//update enemies
```

In order to stop updating the hero and game objects and stop checking for collision, once the game is over, enclose the three functions in an `if` condition, as shown in the following code, where we check whether the `gameOver` Boolean value is `false` or not. If it is `false`, then the functions will get called, else it will be bypassed and won't call the functions.

```
override func update(currentTime: CFTimeInterval)
{
   /* Called before each frame is rendered */

   if(!gameOver){

        updateGameObjects()
        updateHero()
        checkCollision()

   }
}
```

Next, if the game is over, the player shouldn't be able to fire rockets or make the hero thrust up. So, basically, we have to disable the player controls once the game is over. So, in the `touchesBegan` function, enclose where we check which side the screen is tapped in the `if` condition, checking if the game over condition is met or not, as shown in the following code:

```
override func touchesBegan(touches: NSSet,
withEvent event: UIEvent) {

/* Called when a touch begins */

   for touch: AnyObject in touches {

        let location = touch.locationInNode(self)
```

```
        let _node:SKNode =
self.nodeAtPoint(location)

        if(!gameOver){ //if game is not over check
for touches

            if(location.x < viewSize.width/2){

                println("[GamePlayScene]
touchedLeftSide ")

                thrust.y = 15.0

            }else if(location.x >
viewSize.width/2){

                println("[GamePlayScene]
touchedRightSide ")

                addRockets()
            }
        }
    }
 }
```

Keeping score

We are still not done with the `checkCollision` function. We still need to keep track of the score. For this, we have to check the collision between the hero's rockets and the enemies. So, in the `checkCollision` function, add the following:

```
for(var i:Int = 0; i < rockets.count; i++){

    var rocketSprite = rockets[i]._sprite

    for(var j:Int = 0 ; j < enemies.count; j++){

        var enemySprite = enemies[j]._sprite


 if(rocketSprite.intersectsNode(enemySprite)){

            enemySprite.removeFromParent()
            rocketSprite.removeFromParent()

            rockets.removeAtIndex(i)
            enemies.removeAtIndex(j)

            score++

        }
    }
}
```

We are using the `for` loop as we will need the index of the objects in the loops here. We loop through all the rockets in the scene and check the collision with all the enemies in the scene by looping through the `enemies` array. If any of the rockets collides with the enemy, then we remove both the rocket and the enemy sprite node from the scene, and also remove the rocket and enemy from the array.

Finally, create a new global variable `var` called `score` of type `int` and initialize it equal to zero at the top of the class, as shown in the following. We increase the score by 1 after checking the collision, to keep track of the score.

```
    var bullets:[MovingSprite] = []
    var gameOver = false
    var score:Int = 0
```

We can log the score in the console to check whether the score variable is actually increasing. But how will the player know how much he/she has scored?

Displaying the score

For showing the score, we will use a `SKLabelNode` and assign the value of the score every time the score is changed. Since we will need to access this variable in the `checkCollision` function, it has to be a global variable. So, create a variable called `scoreLabel` of type `SKLabelNode` along with the other global variables at the top:

```
var scoreLabel: SKLabelNode!
```

In the `init` function, right after we added the hero, we can add the following lines to initialize the `scoreLabel` variable:

```
scoreLabel = SKLabelNode(fontNamed:"Chalkduster")
scoreLabel.text = "Score: 0"
scoreLabel.fontSize = 45
scoreLabel.position = CGPoint(x:
viewSize.width/2, y: viewSize.height * 0.9)
self.addChild(scoreLabel)
```

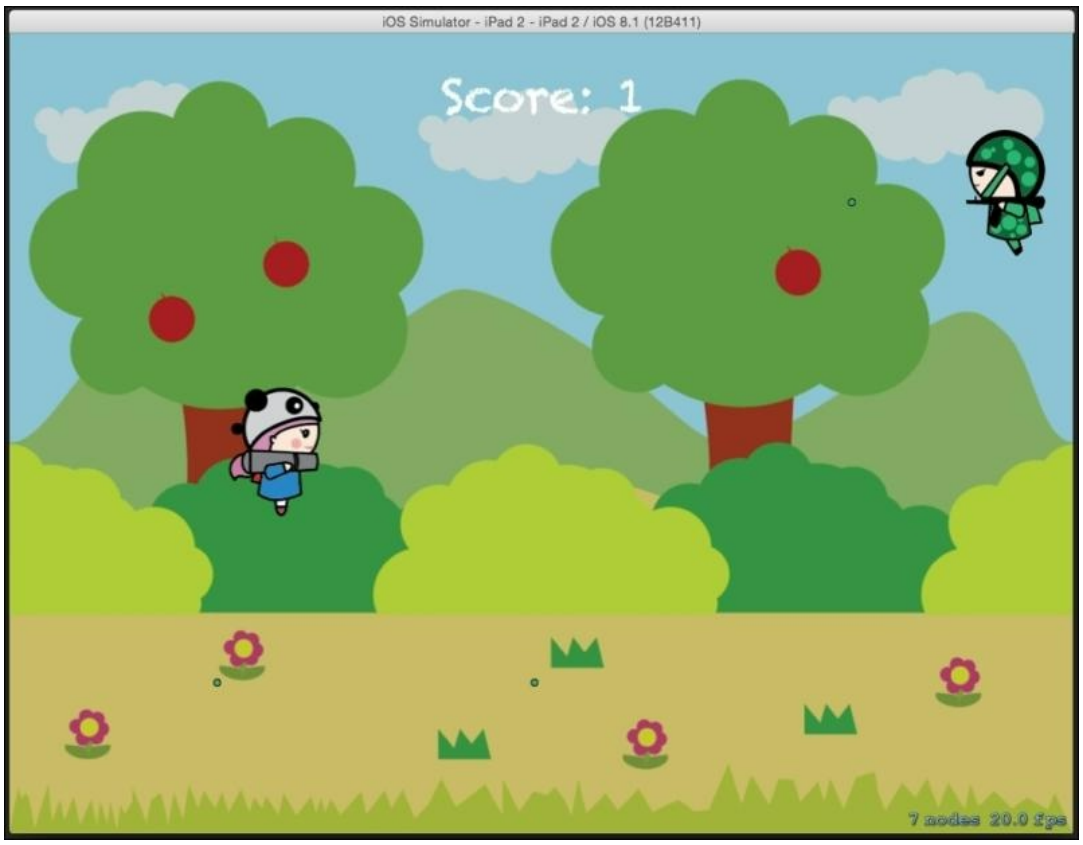
We give a font name to use as the font, which is `Chalkdust`. Then we assign the actual text to be displayed. We will later change the value of this text, depending on what the score variable is later in the `checkCollision` function. We set the size to `45` and place the `scoreLabel` at 90 percent of the height of the display, so that it is at the top of the screen, and place it at the center of the width of the screen. Finally, we add the `scoreLabel` to the current scene.

If you run the game now, the text that we assigned will be displayed, but the score won't update. For the score to update, we have to change the text and assign the actual value of the score variable we created earlier.

So, right after we increment the score in the `checkCollision` function, add the following to increment the score text of the label:

```
rockets.removeAtIndex(i)
enemies.removeAtIndex(j)

score++

scoreLabel.text = "Score: \(score)"
```

Similar to how we log variables to the console, we assign the score value to the string and then pass it to the text property of `scoreLabel`. Now, if we build and run the game, it should display the current score, as shown in the following screenshot:



Displaying the game over screen

Once the game is over, we have to display **GameOver!** to the player and add a button so that the player can go back to the main menu.

In the `GameOver` function, add a label called `myLabel` after we called the function to stop all the actions:

```
let myLabel = SKLabelNode(fontNamed:"Chalkduster")
myLabel.text = "GameOver!"
myLabel.fontSize = 65
myLabel.position = CGPoint(x: viewSize.width *
0.5, y: viewSize.height * 0.65)
self.addChild(myLabel)
```

We add the **GameOver!** text of font size `65` so that it is easily visible to the player and place it slightly above the center of the screen to make space for the main menu button that we will be adding next.

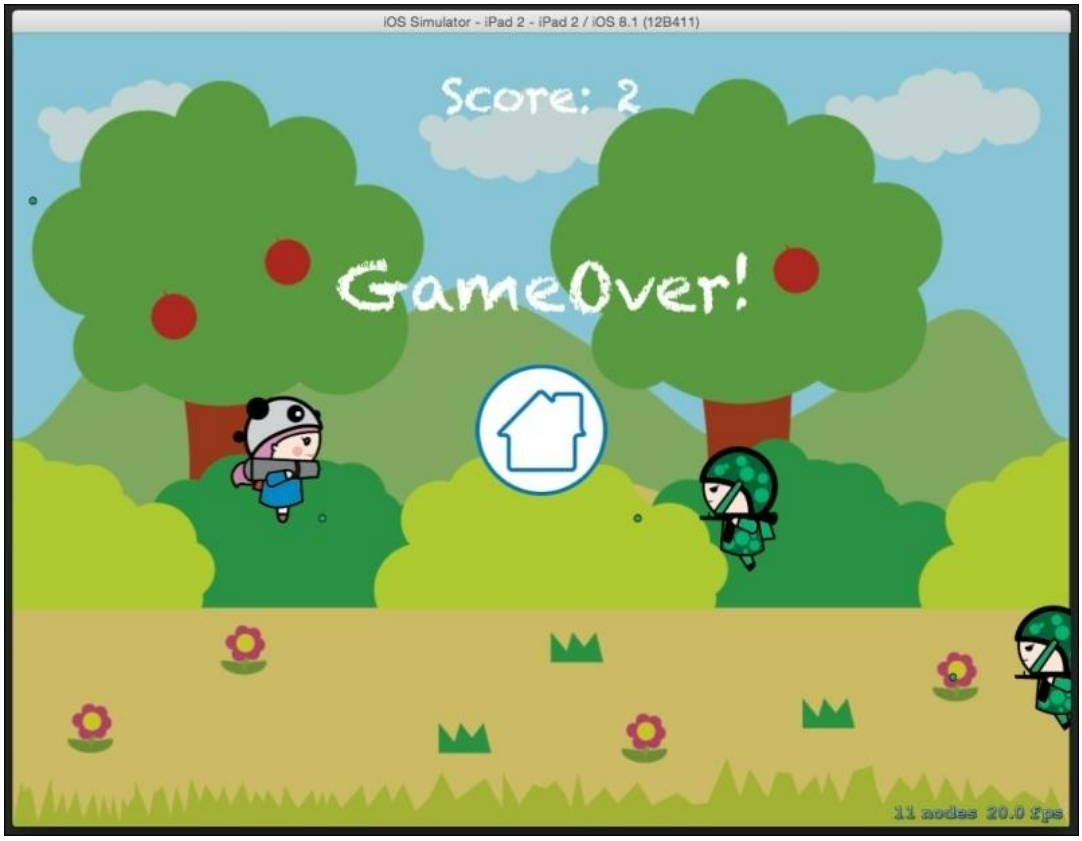
Adding the main menu button

Next, in the `GameOver` function, a SKSpriteNode called `menuBtn` is created, and we pass the `menuBtn` image set to it. For creating the image set in `images.xcassets`, `homeBtn.png`, and `homeBtn2.png` are included in the `Resources` folder. We place it at the center of the screen. We also give it a name so that we can refer to it later in the `touchesBegan` function so that if it is pressed, we can call some function on it:

```
let menuBtn = SKSpriteNode(imageNamed: "menuBtn")
menuBtn.position = CGPoint(x: viewSize.width/2,
y: viewSize.height/2)
self.addChild(menuBtn)
menuBtn.name = "menuBtn"
```

Next, in the `touchesBegan` function, after we checked if the game is over or not, add an `else` block and add the following code to replace the current scene with the `MainMenuScene`:

```
else{ // else check whether main menu button is
clicked

     let _node:SKNode = self.nodeAtPoint(location)

     if(_node.name == "menuBtn"){

            let scene = MainMenuScene(size:
self.size)
            self.view?.presentScene(scene)

      }
}
```



Similar to how we made the play button interactive, if the game is over, we check whether the player touched the screen. If the location that he tapped contains the node name `menuBtn`, we create a local object called `scene` and assign the `MainMenuScene`, and then we replace the current scene with the `MainMenuScene`.

As far as gameplay goes, we are done with it. Now let's add a feature that allows us to save the high score, so that we can challenge the player to beat it, to increase replayability of the game.

Saving the high score

For saving the high scores, we can use the `NSUserDefaults` property. Here we can use a key and assign a particular value to it which the device will store in its memory, so that we can retrieve later. The best thing about this is, we can retrieve and rewrite the value stored in it currently, to some other file. So, here we will store the high score in `GamePlayScene` and later the value stored in the key, in the `MainMenuScene`.

Since it is a dictionary, you can store integer, floats, and string. In this case, since the high score will always be an integer, we will get and store an integer for the key. The key here is a string and the value that is stored is an integer.

For retrieving the high score value, add the following code after we added `menuBtn` in the `GameOver` function:

```
var currentHighScore =
NSUserDefaults.standardUserDefaults().integerForKey("tinyBazooka_highscor
```

Since no value is stored currently in the key, it will return zero.

For congratulating the players on getting a new high score, create a new `SKLabelNode`, right after we added the `currentHighScore`, and call it `highScoreLabel`, as shown in the following code:

```
var highScoreLabel =
SKLabelNode(fontNamed:"Chalkduster")
highScoreLabel.text = ""
highScoreLabel.fontSize = 45
highScoreLabel.position = CGPoint(x:
viewSize.width * 0.5, y: viewSize.height * 0.30)
self.addChild(highScoreLabel)
```

We create the label with font `Chalkduster`. We set the initial text value to a blank so that we can change it later depending on whether the player beats the high score or not. We set the text height to `45`, and place it below the button and then add it to the scene.

Next we check the value stored in `currentHighScore` against the current `score` and see if the `currentHighScore` value is greater than the current `score`, as shown in the following:

```
if (score > currentHighScore) {

NSUserDefaults.standardUserDefaults().setInteger(score,
```

```
                        forKey:
"tinyBazooka_highscore")

NSUserDefaults.standardUserDefaults().synchronize()

highScoreLabel.text = "New High Score: \(score) !"


} else {

    highScoreLabel.text = "You can Do Better than
\(score)"

}
```

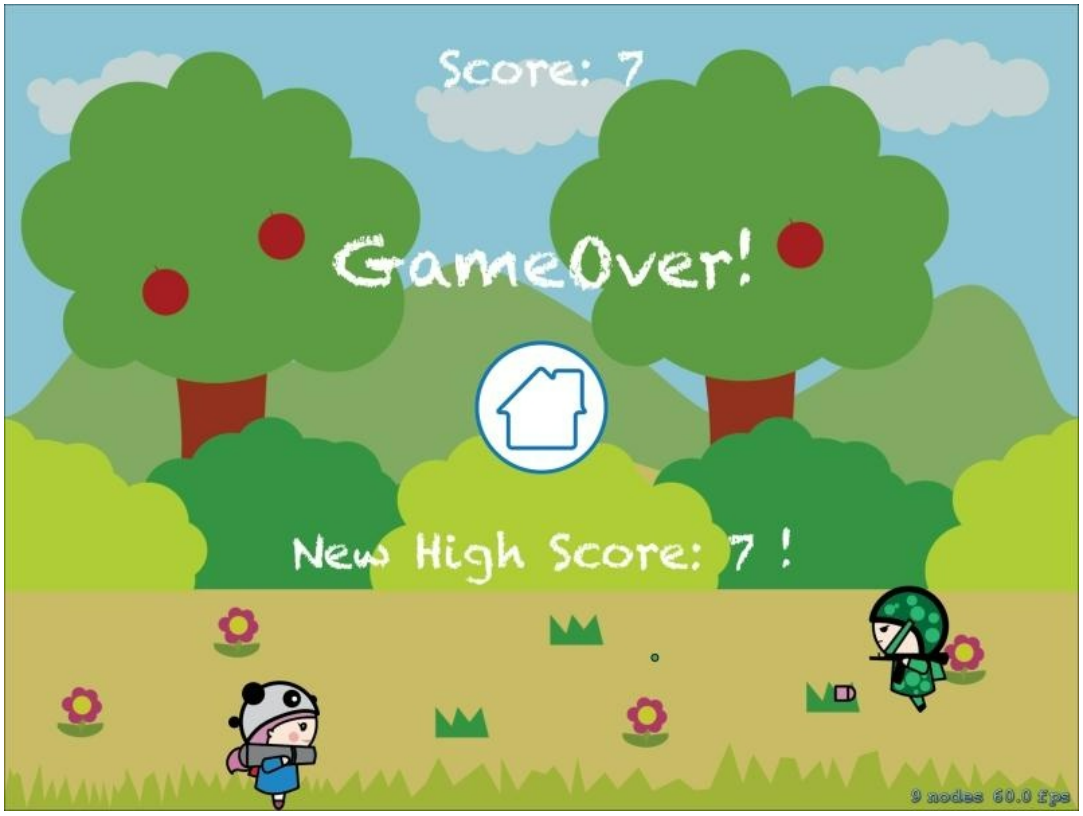
If the score is higher than the current high score, then we call the `setInteger` function of `standardUserDefaults` and assign the new high score that is the current score and the key to store it against. When assigning keys, make sure it is unique.

To save the data to the device, we have to call the `synchronize` function. If we fail to call this, once you close the application, the data will be lost, so make sure to call this function each time you store the value against a key.

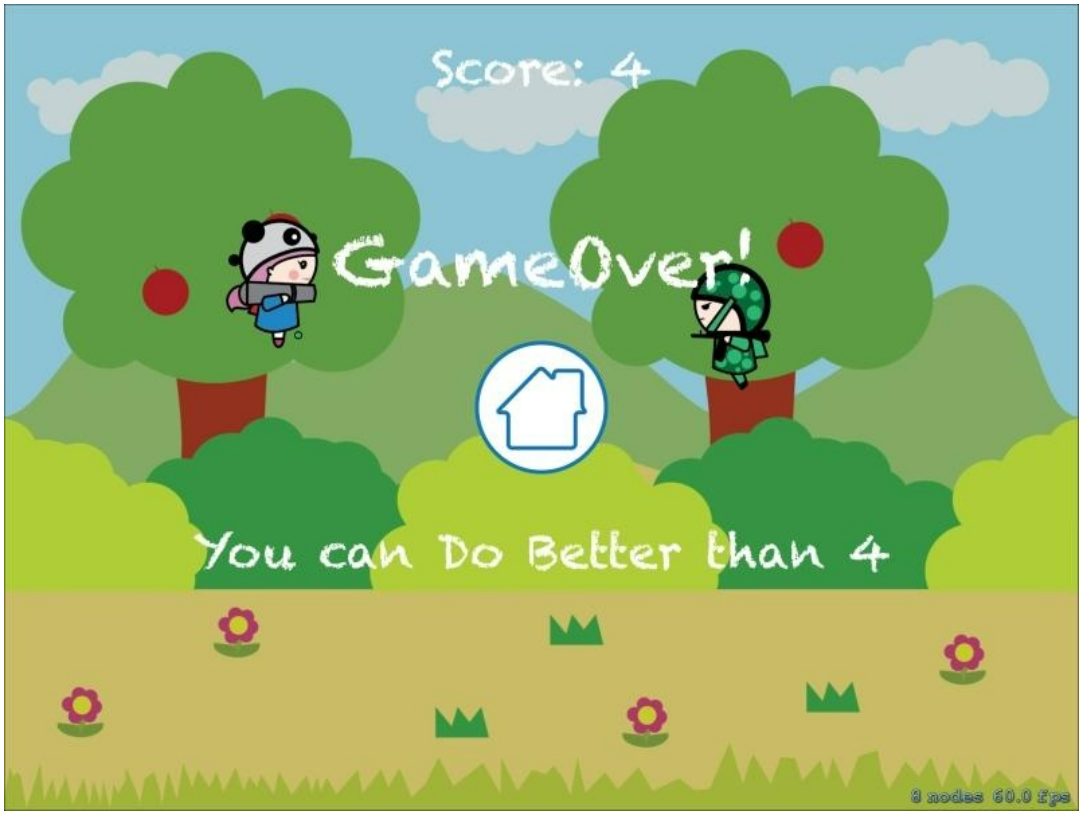
We then congratulate the player on getting a new high score by changing the text property to show the current score, which is the new high score.

If the player didn't beat his current high score, then in the `else` statement we change the text property of the `highScoreLabel` to show that he can do better than the current score so that it will motivate him to play the game again and beat his previous high score.

In the following screenshot, I scored **7**, which is okay I think. With a little bit of practice, I think I will be able to do better. It is also just to demonstrate that the code works fine, and if the current score is greater than the high score, then it shows the new high score.



If the current score is less than the current high score, then you will see the following screen. Here I got a **4**, which is less than the current high score saved in memory that is **7**.



Resetting the high score count

What if the player wants to reset their high score to zero? We can do this by adding a button in the main menu and resetting the value of the key to zero.

Open the `MainMenuScene.swift` file, and in the `didMoveToView` function where we created the play button, create one more SKSpriteNode called `resetBtn`. Place this button at three-fourth of the width of the screen and name the sprite `resetBtn`. We will use the name to check whether the player clicked the play button or the reset button.

The assets for the reset button are in the `Resources` folder, so create a new image set called `resetBtn` and drag-and-drop `resetBtn.png` and `resetBtn2.png` to the **1x** and **2x** placeholders, respectively, as shown in the following:

```
let resetBtn = SKSpriteNode(imageNamed:
"resetBtn")

resetBtn.position = CGPoint(x: viewSize.width *
0.75, y: viewSize.height/2)

self.addChild(resetBtn)

resetBtn.name = "resetBtn"
```

Now, in the `touchesBegan` function where we checked for `playBtn`, add an `else if` block and check whether `resetBtn` is pressed, like this:

```
else if (_node.name == "resetBtn"){

NSUserDefaults.standardUserDefaults().setInteger(0,
forKey: "tinyBazooka_highscore")

NSUserDefaults.standardUserDefaults().synchronize()

}
```

If the reset button is pressed, we set the value of the key by which we have been getting the value, to zero and call the synchronize function again so that this value is stored in the system.

Now, as one last thing, let's show the player the current high score when the application opens, so create a new global variable called `currentHighScoreLabel` of type `SKLabel` at the top of the `MainMenuScene` class, as shown in the following code:

```
class MainMenuScene: SKScene {
```

```
var currentHighScoreLabel: SKLabelNode!
```

Next, we add the following in the `didMoveToView` function right after where we added the reset button:

```
var currentHighScore =
NSUserDefaults.standardUserDefaults().integerForKey("tinyBazooka_highscor

currentHighScoreLabel =
SKLabelNode(fontNamed:"Chalkduster")

currentHighScoreLabel.text = "Current High Score:
\(currentHighScore)"

currentHighScoreLabel.fontSize = 45
currentHighScoreLabel.position = CGPoint(x:
viewSize.width * 0.5, y: viewSize.height * 0.20)

self.addChild(currentHighScoreLabel)
```

We first get the current high score stored, and then assign the value to the text. The rest of the code is similar to the code except that the `y` value for the position is multiplied by 0.2 instead of 0.3, as the play button is pretty big.

We also need to change the text once the high score is reset, so in the `else if` block where we check whether the reset button is pressed, under the place where we added code to reset the value for the key, add the highlighted lines shown here:

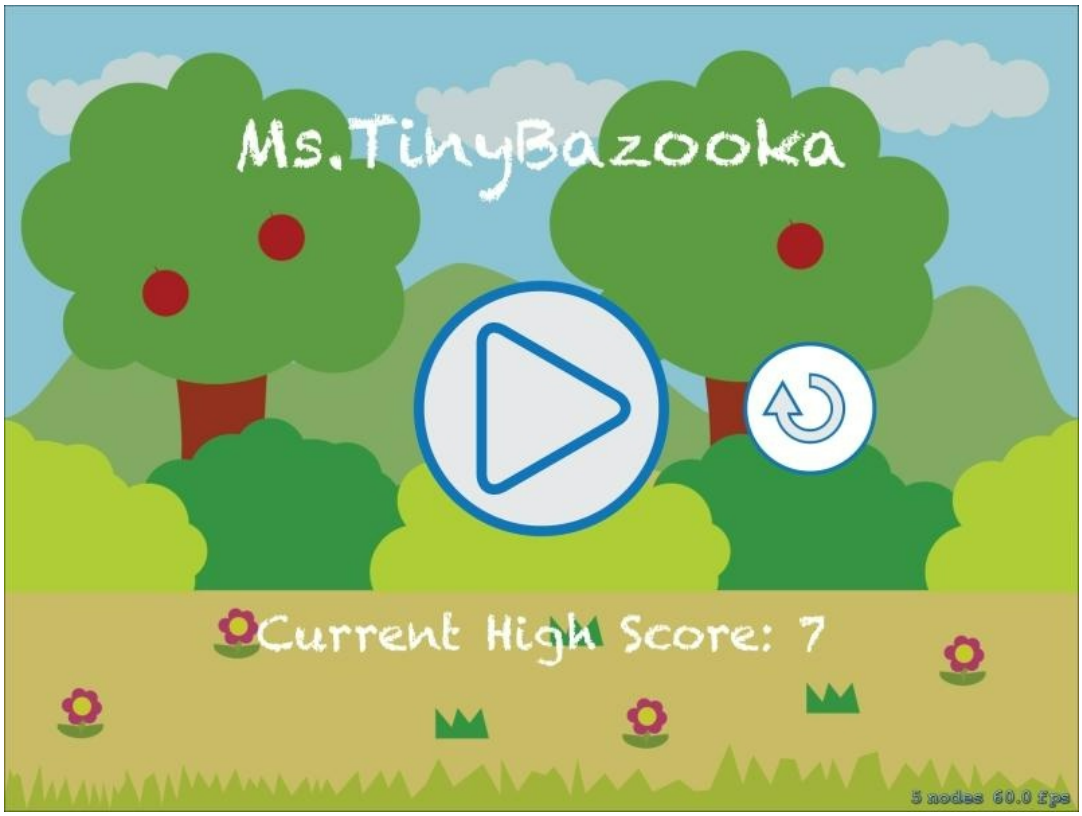
```
else if (_node.name == "resetBtn"){


 NSUserDefaults.standardUserDefaults().setInteger(0,
forKey: "tinyBazooka_highscore")

NSUserDefaults.standardUserDefaults().synchronize()

     var currentHighScore =
NSUserDefaults.standardUserDefaults().integerForKey("tinyBazooka_highscor
     currentHighScoreLabel.text = "Current High Score:
\(currentHighScore)"

}
```

Here, we get the stored value again just to check whether the value that we stored earlier is actually reflected. Then we set text of `currentHighScoreLabel` to this value.

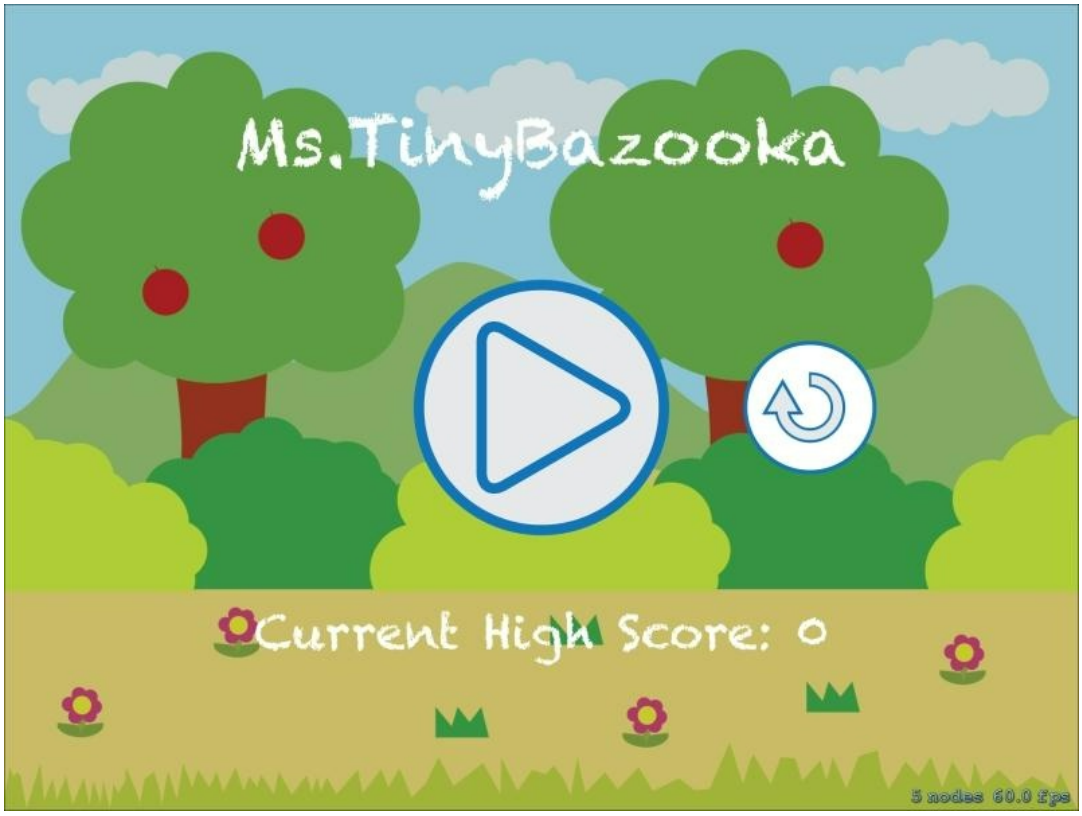
We could have just set the text as `Current High Score: 0` and it would have still worked without a problem, but this way, we would not have been sure whether the value for the key was actually set to zero.

Now, if you run the code, you can see the current high score displayed at the main menu screen when the game starts. You can also verify that the value of the high score is actually set to zero once you reset it.

So, in the following screenshot, the main menu scene, once again, is showing my current high score, even though this code was added after achieving my new high score:



In the next screenshot, we can be sure that the value of the current high score changes once the reset button is pressed:



So this is all for this chapter. Save this file and keep it, as we will need it again in the next chapter.

Summary

This was a pretty long chapter and we actually made a complete working game in it. We created the main menu and gameplay scenes using SKScenes and added interactive buttons to move between the scenes.

You learned how to import assets into the project, add them onto the screen, and make them move around and interact with each other. You also saw how to remove objects from the screen and add and update the score.

Finally, you saw how to store and retrieve the current high score on the device. However, we are not done with the game. We still need to add animations, particle effects, and background music and sounds effects to the game to make it come alive. Hope you guys are looking forward to it.

In the meantime, you can practice and try to beat my high score.

Chapter 5. Animation and Particles

In the previous chapter, we created a basic game. In this chapter, we will make the game more lively and the characters a little more believable. Instead of just static images, we will add animation. We will also look at SpriteKit's inbuilt particle effects creator.

For character animation, we will look at how SpriteKit creates animations and also at an external tool called **Texture Packer**, developed by Code'n'Web. We will see how it simplifies the process of animation. We will also be covering a concept called **sprite sheets**, which is fundamental and is used to optimize a game's performance.

We will first look at how SpriteKit gets an inbuilt sprite sheet generator and makes the animation process simpler. Then we will look at Texture Packer, which simplifies the process even further. So let's see what a sprite sheet is.

The topics covered in this chapter are as follows:

- Sprite sheet animation
- Basic SpriteKit animation
- Texture Packer
- Creating a hero sprite sheet
- Animating the hero
- Particle systems
- The particle designer
- Creating a particle effect
- Adding a particle system to the game

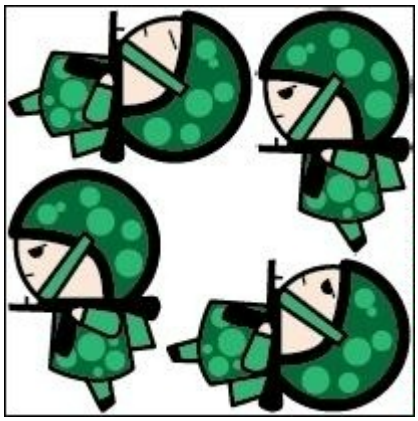
Sprite sheet animation

Until now, what we were doing was using a single file for the player, enemy, rocket, and bullet. Every time we create a new rocket, the game goes and gets the memory, and finds and retrieves the image. The process is repeated every time a bullet is created. This is okay for a simple game such as ours, but later, when we start creating more complex games with a lot more enemies and bullet types, the process will become very taxing on the device, and the performance of the game will be affected.

To solve this problem, sprite sheets are used. A sprite sheet contains all the images that we will use in the game in a single file, instead of 10 images sitting at 10 different memory locations. The sprite sheet image file will also be accompanied by a data file, which will contain the location and size of each of the images located in the sprite sheet. At the start of the game, the sprite sheet image and data file are loaded into the cache once. Then,

every time the rocket or the bullet is called, the game knows where the sprite sheet is and simply loads the image from it.

The sprite sheet file needs to be as compact as possible, so images might be rotated to make it more compact. The data file will keep track of this, and when creating the frames in game, it will make the image upright again.



When animating in the game, each of the frames for the corresponding animation will be stored in an array and made to loop at a particular predefined speed.

Fortunately, in SpriteKit, all you have to do is provide the frames. At runtime, SpriteKit automatically creates a sprite sheet, which we can use in the game. We will use SpriteKit's inbuilt sprite sheet creator to create the enemy animation. Later, we will create the player animation using Texture Packer.

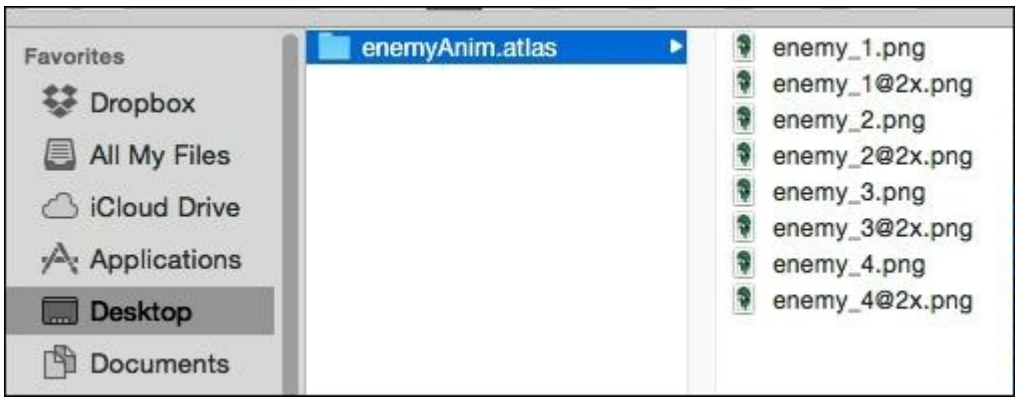
Basic SpriteKit animation

To create the enemy animation, we first have to give the images to SpriteKit. Since we have to provide **1x** and **2x** images for each frame, we could create four image sets; name them `enemy1`, `enemy2`, `enemy3`, and `enemy4`; and then drag **1x** and **2x** for each set. Although it is absolutely possible to do it this way, it is highly tedious. There is an alternate, less tedious way of doing it. While naming each frame, for the **1x** image of a frame you can add the numbers `1`, `2`, `3`, and `4` at the end of the file to show the frame names. For the **2x** version of a frame, you need to add `@2x` at the end to tell SpriteKit that this file is twice the size of the original version of the image. So, for the first frame, the **1x** file will be `enemy_1.png`, and the filename for the **2x** version of the frame will be `enemy_1@2x.png` (here, I am using an underscore because the regular enemy image we used in the previous chapter was already named `enemy1.png`).

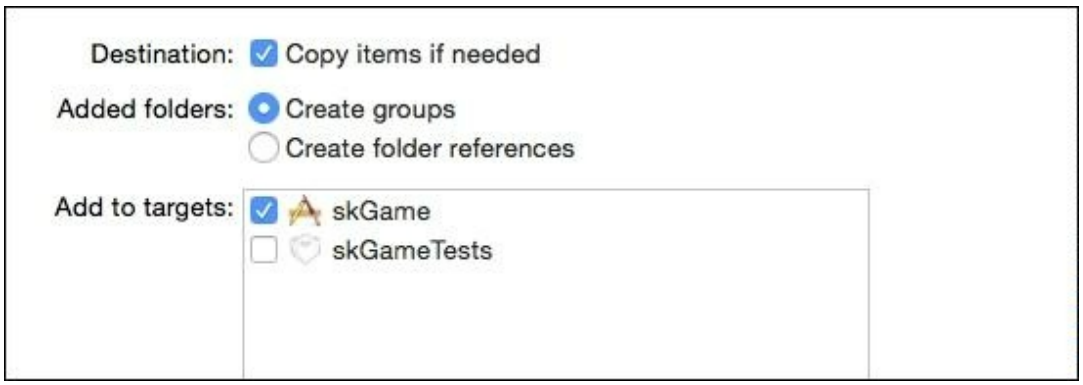
The `@2x` bit is a keyword, so make sure that you don't name your image regularly with `@2x`, as it will cause some unexpected outcomes. This naming convention should be used only if you have a **1x** version of an image. To tell SpriteKit which image is the **2x**, version you add `@2x` at the end of the file to show that this image is twice the size of the other one.

So, for creating the enemy animation, we will have four frames of animation. Thus, we will have an image for each frame. Additionally, for the **2x** version, we will need four more images that are twice the size, with the filename ending with `@2x`.

The frames for enemy animation are provided in this chapter's `Resources` folder. Get all the images and place them in a new folder on the desktop. Now, to tell Xcode that a sprite sheet has to be created from the images provided, click on the folder in which all the images are present and rename it to `enemyAnim.atlas`. This is very important; if you don't do this, then the sprite sheet won't be created. The folder should appear as shown in the following screenshot:



Drag and drop the folder into the project folder. When the window opens up, as shown in the following screenshot, make sure that the **Copy items if needed** box is checked and the current project is the target:



Click on **Finish** to continue. Now the folder will be inside the project, and we can start adding the code to animate the enemy character. Instead of using a static image when the enemy is created, we will change the `addEnemy` function, as follows.

First, let's check whether what I said earlier is correct. If we replace the enemy variable with the name of the image of the first frame, the game should still work. So, in the place where we create an enemy, instead of passing the enemy image set, we will pass the first frame of the animation to the `enemyNode` variable. Change the line for creating the enemy node to the following in the `addEnemy` function:

```
var enemyNode:SKSpriteNode =
SKSpriteNode(imageNamed: "enemy_1")
```

This won't animate the character because it is still taking only the first frame of the animation and displaying it. But at least, we can be sure that the frame is loading properly. You don't have to add the extension when passing the name, so `.png` is not required. Neither do you have to say `@2x` to load the image for a higher resolution, as SpriteKit will automatically take the absence of `@2x` to mean the **1x** resolution, and get the `@2` image file's name for **2x** resolutions.

Now let's go ahead and load the other frames so that we can animate the enemy. After the `addChild(enemyNode)` line, add the following lines of code in the `addEnemy` function:

```
let textureAtlas = SKTextureAtlas(named:
"enemyAnim.atlas")
var textureArray:[SKTexture] = []

for(var i: Int = 1; i <= 4; i++){
```

```
 textureArray.append(textureAtlas.textureNamed("enemy_
\(i)"))

}

let animation =
SKAction.animateWithTextures(textureArray,
timePerFrame: 0.2)
let animate =
SKAction.repeatActionForever(animation)

enemyNode.runAction(animate)
```

First, we load the sprite sheet in a constant called `textureAtlas` using the **texture atlas** class in SpriteKit. Wait! What is a texture atlas? Well, it is just another name for a sprite sheet. You can call it either, but they mean the same thing.

After loading the sprite sheet, we create an array to store all the textures, or the frames of the animation.

Since we have four animation frames to load, namely `enemy_1`, `enemy_2`, `enemy_3`, and `enemy_4`, we create a `for` loop and iterate from `1` to `4`. Like any array, we append the four files to the `textureArray` variable we created. We assign each texture using the `textureNamed` function of the `SkTextureAtlas` class. Similar to how we log things on the screen or change text dynamically, we use the `\()` operator to provide the names of the four files. Once the images are stored in the array, we create an action so that we can run through the frames at a certain speed. The `animateWithTextures` function of SKAction takes in an array of textures and a duration for which each frame should be displayed on the screen. So here, we give the `textureArray` variable storing all the textures, and give 0.2 or 200 milliseconds as the time for which each frame will be displayed. However, this will run the animation only once. For the animation to run again and again, we use the `repeatForever` action. So, we create a new constant called `animate` and store the `repeatForever` action in it by passing the animation to it. Finally, we run the action on the `enemyNode` variable. Now you can build and run to see the enemy character getting animated.

That's all pretty good, but there is an easier way so that we don't have to create separate sets of frames for the **1x** and **2x** resolutions for the game. Moreover, while creating the array of textures, we need to know beforehand how many frames need to be looped through in the animation. In the case of the enemy, we know that there are four frames in the animation, so we looped from `1` to `4`. What if we are not aware of this information? It may lead

to errors or the animation looking clunky if we loop less or more than the total number of frames of the animation.

Therefore, to animate the player, we will be using the Texture Packer way of animating.

Exploring Texture Packer

Texture Packer is a very popular piece of software that is used by industry professionals such as Disney, Zynga, and WG Games to create sprite sheets. You can download it from https://www.codeandweb.com/.



This is similar to how we created the images earlier when creating enemy animations. To create sprite sheet animation using Texture Packer, you also have to create the individual frames in Photoshop or Illustrator first. I have already made them and have each of the images for the individual frames ready.

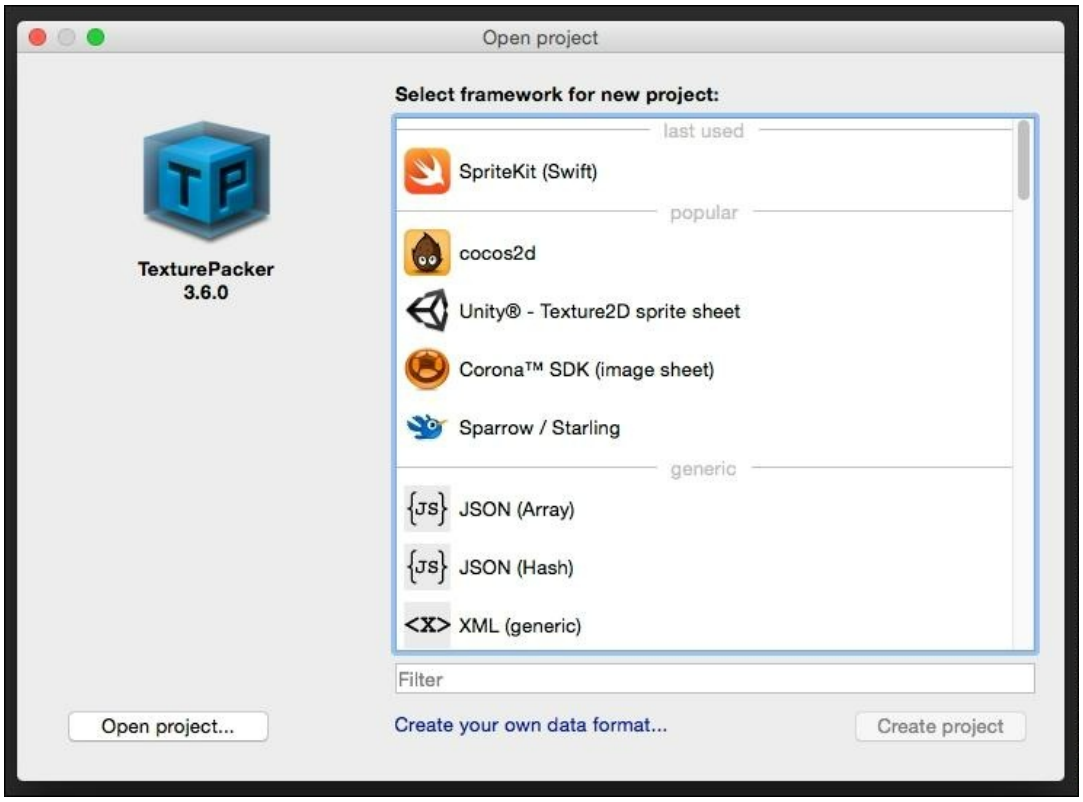
You can use the trial version of Texture Packer to follow along the tutorial. When downloading, choose the version that is suitable for your operating system. Fortunately, Texture Packer is available for all major operating systems, including Linux.

Tip

Once you have downloaded Texture Packer, you get three options: you can click to try the full version for a week, purchase the license, or click on **Essential version** to use a trial version. In the trial version, some

professional features are disabled, so I recommend trying the professional features for a week.

Once you click on the option, you should see the interface, as shown in the following screenshot:



Here, you can either open an existing project by clicking on the **Open Project** button in the bottom-left corner of the screen, or select the framework for which you wish to create the sprite sheet.

As you can see, Texture Packer supports a wide range of frameworks and formats, including Cocos2d, Unity, Corona, Swift, and many more. We need to select **Swift** from the list and click on **Create Project**.

Note

Texture Packer has three panels; let's start from the left. The left panel will display the names of all the images that you have selected to create the sprite sheet. Here, you can drag and drop individual images or full folders containing your assets. The middle panel is a preview window. It shows you how the images are packed. The right panel gives you options to choose where you would like to store the packed texture and data file to be published, and what the format of the packed image should be. The layout section gives a lot of flexibility to set up individual

images in the texture packer. Finally, we have different modes for optimizing the sprite sheets.

Let's look at some of the key items in the settings panel on the right:



Data

Under **Data**, we define all of the information regarding the data file to be exported. This includes **Data Format**, **Atlas Bundle**, and **Swift Class file**. The explanation of each of these fields is as follows:

- **Data Format**: As we saw earlier, each exported file creates a sprite sheet that has a collection of images and a data file that keeps track of the positions on that sprite sheet. The data format usually changes depending on the framework or the engine you selected. Since we selected SpriteKit initially and selected **Swift** as the language, the format is of type `swift`. Suppose we were using the Objective-C language. Then there is a separate option for that as well. So, be mindful while selecting the format, otherwise you will have to start over if you wish to develop the sprite sheet for another format.
- **Atlas Bundle**: This is the location where you want the exported image and data file to be saved. So, you will get a `.png` image file and a `.plist` data file containing information about the sprite sheet once the file is published.
- **Swift Class file**: Along with the image and the data file, Texture Packer will also create a helper class containing information that will

be imported along with the image and data file to make the animation code even simpler.

Texture

In the **Texture** section, we will specify the sprite sheet image's file details. This includes details regarding **Texture Format**, **Png Opt Level**, and **Pixel format**. The explanation of these fields is as follows:

- **Texture Format**: The default is set to `.png`, but other formats are also supported. Apart from PNG, you can also use the PVR format. This format is used for data protections because it is easier to copy data from regular PNG files. Also, PVR formats provide superior image compression. However, be aware that it can be used only on Apple devices.
- **Png Opt Level**: This is used to set the quality of the PNG file.
- **Pixel format**: This sets the RGB format to be used. Usually, you would want this to be set at the default value.

Layout

Here, we specify the layout of the sprite sheet image. The following fields can be seen in the **Layout** section:

- **Max Size**: You can specify the maximum width and height of the sprite sheet, depending on the framework. Usually, all frameworks allow up to 4092 x 4092, but this mostly depends on the framework, so check the maximum size the framework allows before creating the sprite sheet.
- **Size constraints**: Some frameworks prefer the sprite sheets to be in the POT format (or the powers of 2), that is, 32 x 32, 64 x 64, 256 x 256, and so on. If this is the case, then you need to select accordingly. Otherwise, you can choose **Anysize**.
- **Scaling variants**: This is used for scaling up or scaling down the image. If you are going to be creating images for different resolutions, such as **1x**, **2x** and **3x**, then this option allows you to create resources, depending on the different resolutions you are developing the game for. Moreover, there will be no need to go into the graphics software, shrink the images, and pack them again for all the resolutions individually.
- **Algorithm**: This is the code logic that will be used to create sprite sheets, and it will make sure that the images are packed in the most efficient manner. In the Basic version, you will have to use **Basic** from the drop-down menu, unlike the Pro version, where you can choose **MaxRects**.
- **Multipack**: If the PNG image file exceeds the maximum size, then Texture Packer will automatically create an additional sprite sheet and

data files for the images that it wasn't able to incorporate into the previous sprite sheet.

Sprites

Here, we specify any special treatment for individual sprites in the sprite sheet. There is one field in this section:

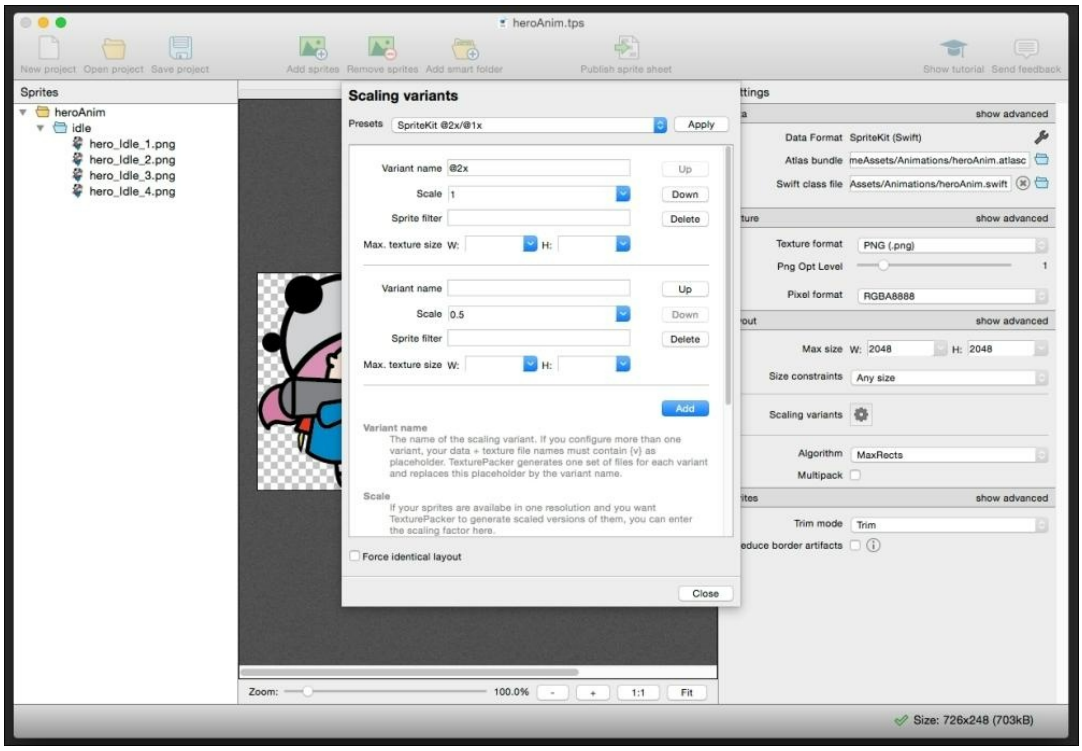
- **Trim mode**: Removes the extra alpha surrounding each image making the sprite sheet more compact and thereby deceasing file size even further.

Creating the hero spritesheet

There is a folder called `heroAnim` in the `Resources` folder for this chapter. This folder contains another folder, which contains the individual frames for the hero's idle animations. Drag and drop the `heroAnim` folder into the left panel of Texture Packer.

You will see the folder structure of the `heroAnim` folder here. Under it, you will see the `idle` folder where you will see the individual files for the frames. In the preview pane, you will see the preview of the sprite sheet that will be created from the images provided.

In the **Layout** section, click on the **Scaling variants** button. From the drop-down list in the pre-sets, select **SpriteKit @2x/@1x**. This will automatically create sprite sheets for both **2x** and **1x** resolutions by scaling down the image by 50 percent and saving it for the **1x** mode.



Next, under the data heading, select the location where you want the sprite sheet image and data file to be saved. The files will be saved in the location inside a folder with the `.atlasc` extension, and this folder will contain the image and data files. Next, select the location for the class file to be saved. You can save it to whichever location you want, but make sure that you remember where you have saved it because it will be required later.

Also make sure that you save the current file at some location so that if you want to make some changes to the file later, you will be able to open it and

make the changes with ease. One thing to note is that if you change the location of the `heroAnim` folder, then the reference will be lost and you will have to reimport the image. Keep the images, sprite sheet, class, and Texture Packer file in a separate folder so that all of the relevant data for the file is in the same directory.

Finally, to create the sprite sheet, click on the **Publish Sprite Sheet** button at the top. Now we can animate the hero.

Animating the hero

To animate the hero in the game, drag the `.atlasc` folder and the Swift class file into the project. Then, in the `init` function, add the following code right after adding the hero to the scene:

```
let heroAtlas = heroAnim()
let heroIdleAnimArray = heroAtlas.hero_Idle_()
let animaiton =
SKAction.animateWithTextures(heroIdleAnimArray,
timePerFrame: 0.2)
let animate =
SKAction.repeatActionForever(animaiton)
hero.runAction(animate)
```

And that's all! You can build and run the game to see the hero getting animated.

Here, we first create a reference to the Swift class that was created in Texture Packer. Next, we create a constant called `heroIdleArray` and assign the array for the idle animation that is already created in the `heroAnim.swift` class. If this wasn't there, then we would have had to manually create an array and store the frames, as we did for the enemy. The next three steps are exactly the same as how we created the animation for the enemy. We create an animation constant, pass the array and the delay that we want for each frame, then create another action to repeat the animation over and over, and finally run the animation of the hero.

So, we see that we didn't have to create two sets of images, as Texture Packer created them for us, and we didn't have to create an array for the idle animation either. This becomes even more important if we later have to create a run, walk, jump, or attack animation. We can't be creating an array each time. In fact, in other frameworks, that's what needs to be done if we want to create different animations. This process is so much easier using Texture Packer for SpriteKit.

Let's look at the Swift class created by Texture Packer so that you can get a better understanding of what is happening and how the array is created. So, open the `heroAnim.swift` file:

```
// Sprite definitions for 'heroAnim'
// Generated with TexturePacker 3.6.0
//
// http://www.codeandweb.com/texturepacker
// ---------------------------------------

import SpriteKit
```

```
class heroAnim {

    // sprite names
    let HERO_IDLE_1 = "hero_Idle_1"
    let HERO_IDLE_2 = "hero_Idle_2"
    let HERO_IDLE_3 = "hero_Idle_3"
    let HERO_IDLE_4 = "hero_Idle_4"


    // load texture atlas
    let textureAtlas = SKTextureAtlas(named:
"heroAnim")


    // individual texture objects
    func hero_Idle_1() -> SKTexture { return
textureAtlas.textureNamed(HERO_IDLE_1) }
    func hero_Idle_2() -> SKTexture { return
textureAtlas.textureNamed(HERO_IDLE_2) }
    func hero_Idle_3() -> SKTexture { return
textureAtlas.textureNamed(HERO_IDLE_3) }
    func hero_Idle_4() -> SKTexture { return
textureAtlas.textureNamed(HERO_IDLE_4) }


    // texture arrays for animations
    func hero_Idle_() -> [SKTexture] {
        return [
            hero_Idle_1(),
            hero_Idle_2(),
            hero_Idle_3(),
            hero_Idle_4()
        ]
    }


}
```

I named the class `heroAnim` while creating the file in Texture Packer, so that is why the name of the class is the same as what I named it. Secondly, you will notice that the four images that we used to create the sprite sheet are named `hero_idle_1`, `hero_idle_2`, `hero_idle_3`, and `hero_idle_4`. So, at the start of the class, it automatically creates constants for the four filenames equal to the string names.

Then the class creates an `SKTextureAtlas` constant called `textureAtlas`. This texture atlas is created from the `heroAnim.atlasc` folder. So, although the name is the same as that of the class, this is the sprite sheet file; don't get confused.

After getting the texture atlas, four functions are created to get the four images stored in the texture atlas with the constant names defined earlier.

Then a new function is finally created, which adds the four images into an array and returns the array. This function is named similar to how we named the image files that were used to create the animation. This makes it convenient to know the name of the function to call when assigning the array while creating the animation action. Also remember that all of this is done automatically in Texture Packer.

Apart from the idle animation, if we have two images for the run cycle and call them `heroRun1.png` and `heroRun2.png`, then the function that will return the `run` array will be called `heroRun()`. An easy way to remember this would be to replace any number with open and closed brackets to get the function name and the array of frames.

Tip

It is also important to note this: while naming your images required for creating the frames for animation, make sure you use the same naming convention for all the frames, just like the case of naming the frames. For example, while naming frames for animation, I usually name the files in the form of `name_action_number`. It can also be like `nameActionNumber`, as we named the run animation example, but make sure you are consistent.

It is also important to make sure that the numbering of the files is in the order in which you want the animation to be played. This is because when the animation is played, frame 1 will be played first, then frame 2, then frame 3, and so on. If your image is incorrectly numbered, the animation will be played in that order and it might look a little funny when it gets played.

These are the only things that you need to be careful about. If you are consistent in naming and numbering your files, then correctly creating animations in Texture Packer is a breeze. Texture Packer does most of the dirty work for you and leaves no room for error as it mostly automates the process for you.

Now let's look at how to create particles in a game.

Particle systems

A **particle system** is a collection of sprites or particles. Each particle system has an emitter from where the particles will be created. A particle system also determines the behavior of the particles in the system. Hence it can be said that a particle is the smallest entity that creates the particle system.

A very easy example of a particle system is Rain. **Rain** is a particle system in which each rain drop is a particle and a cloud has a lot of emitters from where the droplets, or particles, are created.

We create a particle system instead of creating individual particles because with a particle system, we can create different kinds of effects using the same particle. For example, we saw Rain, which is a particle system. What if we wanted another effect, such as water coming out of the faucet? Here, the particle is the same—a water droplet—but a rain droplet behaves differently. When water is falling from the faucet, each drop falls with a force and is created with a single emitter—the faucet outlet. So, we can change the particle system to have one emitter and give the particles an initial downward force. In this way, we will have the same particle behaving differently, instead of coding the system from scratch again.

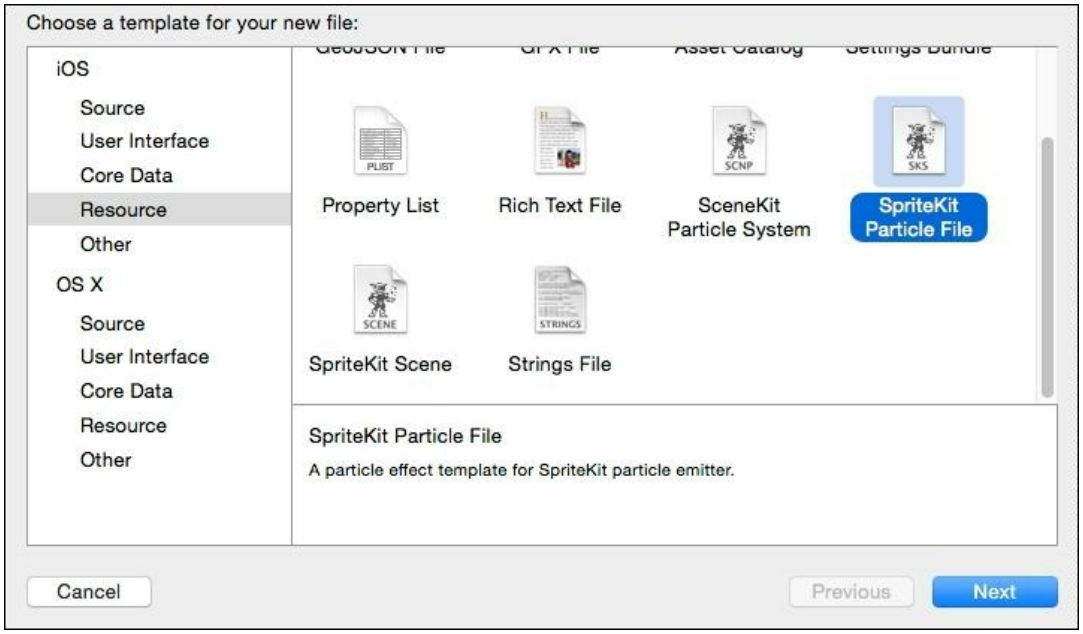
In SpriteKit, as in any other framework, each particle is an image that is controlled by a particle system that has one or more emitters. An **emitter** controls the spawning, movement, and destruction of the particle system.

For rendering the particle system, a **SpriteKit particle file** (`.sks`) is created. It can contain a particle system of any size, and allows for rotation and scaling of the entire particle system.

Designing particles

SpriteKit has an inbuilt particle designer. This designer has a pretty good user interface that can be used to create your own particle systems. SpriteKit also includes a number of default particle systems that are already included and can be created by selecting the particle system that you want for your game from the drop-down menu.

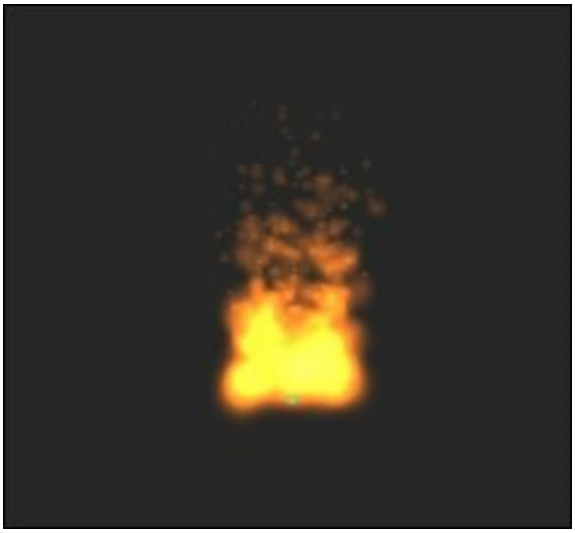
To create a new particle system, go to **File** and select **New File**. You can also create a new file by right-clicking on the root folder of your project. In the **iOS** panel on the left, select **Resource**, then select **SpriteKit Particle File**, and click on **Next**.



Now, from the drop-down menu, you can select from the eight default particle systems that come included in SpriteKit. You can select from **Bokeh**, **Fire**, **Fireflies**, **Magic**, **Rain**, **Smoke**, **Snow**, and **Spark**:



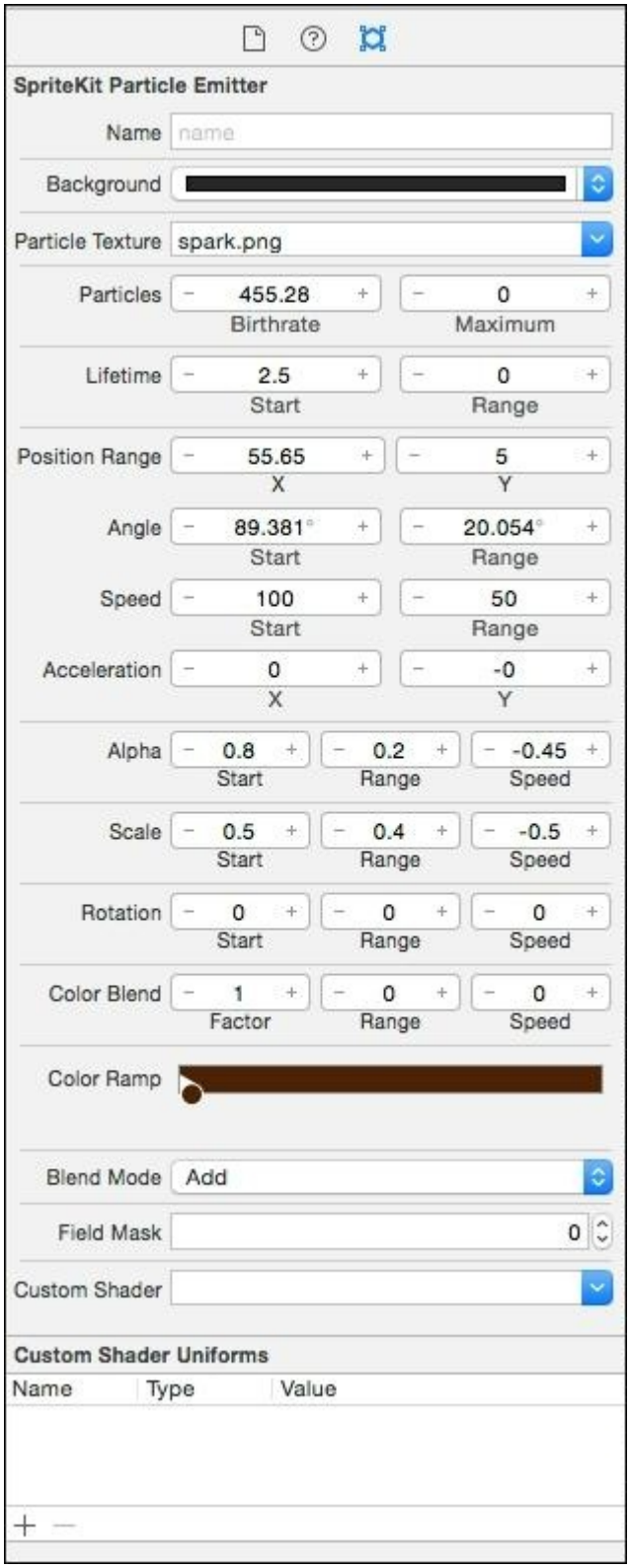
For this example, I selected **Fire**. Once you have selected the particle system of your choice, click on **Next**. We will have to give the particle system a filename that we can call it by later when we want to create that effect in the game, so I named it `fireParticle`.



Once you're done, you will see a `fireParticle.sks` file created in your project and the fire particle burning bright in the editor panel. By default, it is created in the center of the screen. Click anywhere in the editor panel and the fire particle system will be moved to that location. You can also click and move the mouse around to see that the particle system moves along with the mouse arrow.

You will also see that the **Utilities** panel has changed and a whole bunch of variables with changeable parameters have appeared. You can change these parameters to create your own custom particle system.

Let's go through each of these basic parameters so that you can understand what each of these variables do and how, by changing each of the parameters, the behavior of the particle system can be changed.



Name

If we want to refer to the particle system by name in code, we can give a name here so that we can refer to it later. Similar to how we gave a name to the `enemyNode` to check whether the node passed to the `movingSprite`

class was the enemy, and then we perform a certain function based on that information.

Background

The **Background** parameter sets the color of the background. Changing it doesn't affect the particle. It is purely for the purpose of visibility. If your particles are black, you can change the background to white so that you can see clearly how the particle looks and behaves.

Particle Texture

The **Particle Texture** parameter is the texture or image that will be displayed for each particle. Currently `spark.png` is used as the texture. You can change this to the enemy, bullet, rocket, or hero image if you want to.

Particles

The **Particles** parameter controls the rate at which particles are emitted and the number of particles you want the emitter to emit. To control the rate, you can increase or decrease the birth rate parameter. We can decrease the rate of emission to decrease the value of the birth rate, or if we want the particles to be emitted faster, we can increase it. To cap the number of particles, we change the maximum value to the number of particles we want the emitter to emit. If we want the emitter to continuously emit particles, we keep the value at 0.

Lifetime

When a particle is created, **Lifetime** decides for how long it stays on the screen before getting deleted. Here, every particle stays on the screen for 2.5 seconds. **Range** is used to bring in some randomness in the particles' behavior. Suppose we change this range value to 1. Then some particles will be on the screen for 2.0 seconds while others will be there for 3.0 seconds, before getting deleted. So, the random value created is plus or minus half of the range value in addition to the initial value.

This is how **Range** works: it takes the first value then gets a value by either adding or subtracting half of the range value from it so that it will look as if each particle has a different lifetime, as in life, not all particles behave in the same way.

Position Range

The default position at the beginning is the center of the screen. By looking at the **Position Range** keyword in the name of the variable, you might have guessed that the value that we are inputting is range value. Here, the `x` value is `55.65`, which means that when a particle is being spawned, it

spawns anywhere between `-27.825` to `+27.825` in the *x* direction from the center. The emission point is denoted by a small green dot on the editor view. The `y` value is `5`, which means that from the center, the particle will be generated anywhere between `-2.5` and `+2.5` in the *y* direction. If you change the `X` and `Y` values to `0`, you will see all particles getting emitted from the green dot.

Angle

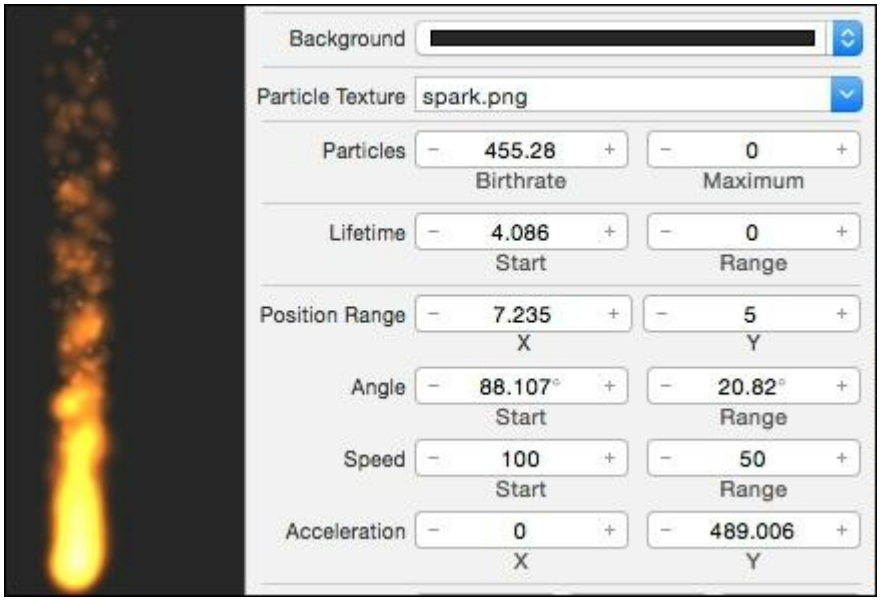
The **Angle** parameter determines the angle at which the particles are created. Since we want the flames to start moving up in this case, the angle is approximately 90 degrees. You can change this value to `45` as well, which will make it look as if there was some wind acting on the fire. To add some randomness to the initial angle, the range is kept at `20`. Otherwise, all particles will be going straight up, which will look very unnatural.

Speed

This is the speed at which the particles will start moving when created. Here, they start moving at an average speed of 100. There is a range of 50, which means that some will move at the minimum speed of 75 and others will move at the maximum of 125.

Acceleration

We can accelerate a particle in the *x* and *y* directions, for example, in the case of a jet engine or a comet. To create this effect, you have to change the `X` position's range to `5` and increase the `Y` acceleration to around `500`.



Alpha

This defines the level of opacity or transparency of each particle. If the value of **Alpha** is zero, then the particle will be completely transparent, while `1` means that it will be completely visible. There is also a **Range** value that you can specify. The **Speed** parameter determines the rate at which the **Alpha** value of each particle changes per second. So, it starts visible as soon as it is created, and over a period of time, it becomes transparent as the value of **Alpha** is reduced.

Scale

Similar to **Alpha**, the **Scale** value ranges from `0` to `1`. At `0`, the image is completely invisible; and at `1`, it is at its original size. Thus, at `0.5` the object will be half the size; and at `2.0`, it will be double the size in both the `x` and `y` directions. Here, the object has a start value of 0.5 and a range of 0.4. So, the initial start size of any particle will be between `0.3` and `0.7`. Since **Speed** is `-0.5`, it will slowly become smaller over a period of time after it has spawned.

Rotation

As soon as a particle is created, we can make it rotate by giving it a start value and a range to generate random speeds of rotation. We can also increase or decrease the speed of rotation over a period of time by changing the **Speed** parameter.

Color Blend

Color Blend is used to blend one color with another. Here, the initial factor is `1` and **Range** is `0`, so we are using the original color that is assigned. Since we are not color blending, we have kept the **Speed** value as `0`. We can change the speed to `-0.125` to see that the color slowly blends to white, and the colors become lighter at the end of the particle's life cycle.

Color Ramp

Here, we can specify the color of the particle to be generated. The particle texture image is always kept white so that we have the liberty to change the color of the object in code whenever we want. So here, even though the texture or image color is white, the color of the flame is orange.

We can also assign different colors to the particle at different stages of its life.



The other four variables—**Blend Mode**, **Field Mask**, **Custom Shader**, and **Custom Shader Uniform**—pertain to shaders and shader programming, which is beyond the scope of this book. Using shaders, you can create custom effects and behavior for the particles.

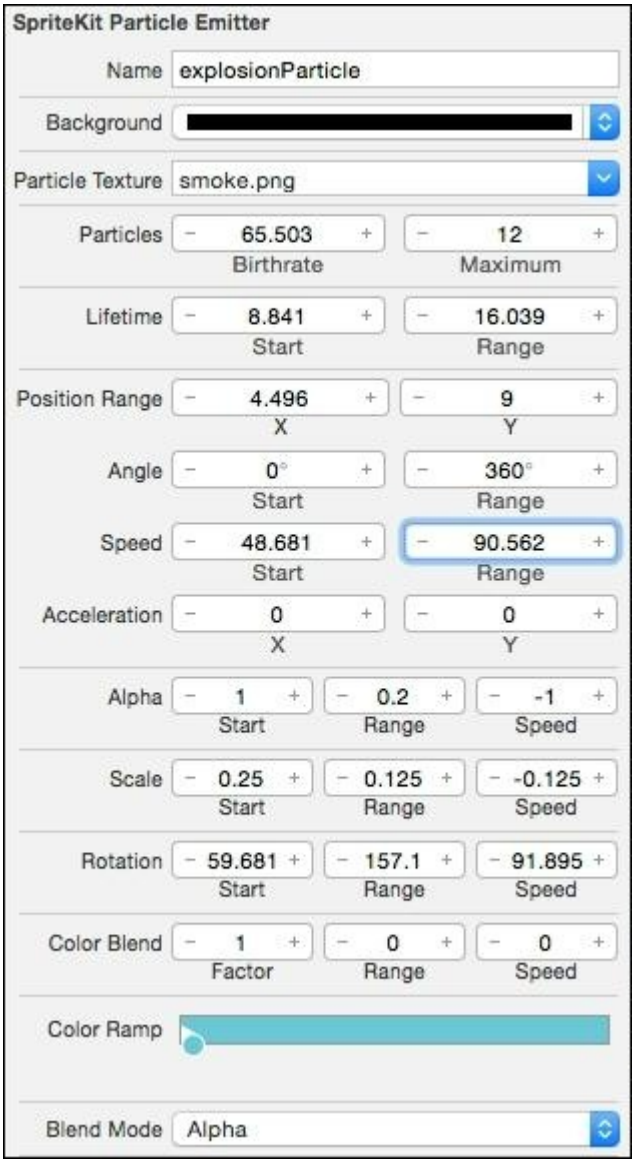
You can play around with **Blend Mode** a little if you know how it works, as it is very similar to what you see in Photoshop. If you know Photoshop, you will be familiar with **Add**, **Subtract**, **Multiply**, **Screen**, **Replace**, and **Alpha**. You can select each of these and see the effect it has on the particle system.

That is all of the information required for now for designing our own particle system for the game. Let's now create the particle system that we will be using in the game.

Creating particle effects

For the game, we will be creating a very simple explosion particle system, and it will be shown every time the hero fires a rocket. We will be using our own custom sprite for the image. So, go to the resources folder for the chapter and drag and drop the `smoke.png` image into the project. Don't forget to add it as a copy, and make sure that the current game is the target.

Now create a new particle system, name the file `explosionParticle`, and save it to your project. You can use fire, smoke, or any of the default particle systems. It doesn't matter, as we will be changing the values to our specification anyway.



In the preceding screenshot, you can see the parameters of the particle system I created using the default particle as a base.

I named it `explosionParticle` just in case I have to refer to the particle system in code. Then I kept the background as default, as I was still able to see the particle quite clearly.

I changed the texture to the smoke image file that was imported earlier, and replaced the default image with it. For **Birthrate**, I kept the value at approximately `65`; you can change it as per your needs. I kept the maximum number of particles at `12` so that after creating 12 particles, the emitter will stop creating any new particles.

The **Lifetime** parameter is kept at around `8` and **Range** at around `16`. You might say that some particles might be destroyed as soon as they are created. Well, that is true and it is also true about smoke particles in general. There are some particles in smoke that we don't even get to see when a gun is fired, so the behavior will be realistic even though it looks cartoony.

The **Position Range** parameter is approximately `4` along the `x` direction and `9` along the `y` direction. This is just to create the particle at a random position around the initial position specified so that all particles don't look as if they were emerging from the same point.

The **Angle** parameter is kept between `0` and `360`, as we want the smoke particles to move in all directions around the point at which they were created.

The **Speed** parameter of each particle at the start is kept at approximately `48`, and the **Range** is about `90`. This will make some particles move slower and others a lot faster. In fact, some particles may not move at all or move very slowly, making the behavior more realistic once again.

The **Acceleration** parameter is kept at `0` for both the **X** and **Y** directions, as it is not required, but you can tinker with these values to see whether you like the effect.

As we want the smoke particle to fade slowly, we assign an initial value and range, and reduce the value of the alpha by slowly reducing its value by increasing the **Speed** parameter to `-1`. Since the size of the image is huge, I scaled it to `0.25` to make it smaller and gave it a range so that the size is randomized. I changed the speed value to `-0.125` so that the size of each particle slowly reduces over a period of time.

The particle needs to rotate as soon as it is created, so I gave it an initial value and **Range**. I increased the **Speed** parameter so that the particle rotates faster over a period of time.

As I am not color blending, I kept the factor at `1` and the range and speed at `0`. I changed the color of the particle to a shade of light blue to make it look cartoony. Black would have been a little too dark and more realistic.

Finally, I changed **Blend Mode** to **Alpha** because I didn't want additive blend mode. You can make the required changes to your particle file and press *command* + *S* to save the file.

Adding a particle system to the game

The file particle system is now ready to be called in to the game. So, open the `GameplayScene.swift` file and, in the `addRocket` function, add the following code right after where we added the rocket to the scene:

```
let explosionParticle = SKEmitterNode(fileNamed:
"explosionParticle")

explosionParticle.position = CGPoint(x:
hero.position.x + hero.size.width/2 + 10, y:
hero.position.y - 5)

self.addChild(explosionParticle)
```

Here, we create a new constant called `explosionParticle` of the `SKEmitterNode` type, and pass the filename for the particle system we created earlier. Whenever you want to create a particle system, you will have to use `SKEmitterNode`.

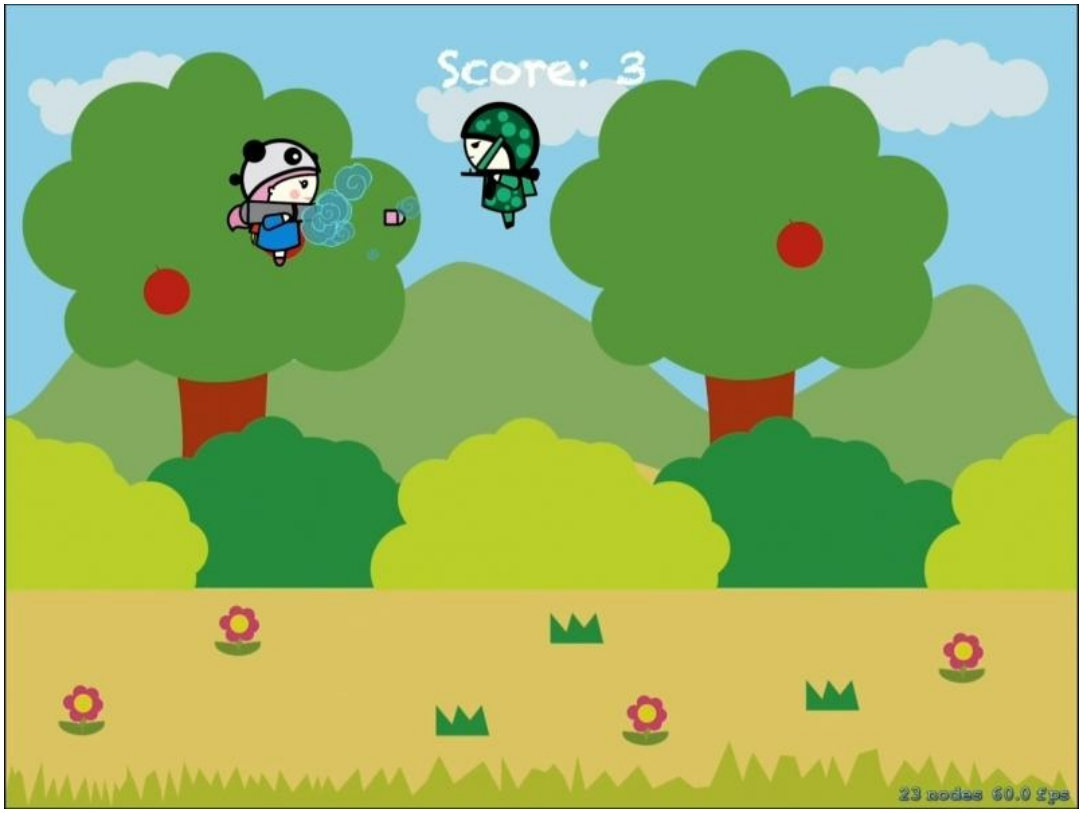
Next, we place the particle system at the nozzle of the hero's bazooka. This is similar to how we placed the hero's rockets. Finally, we add the particle to the scene.

We don't have to worry about removing each particle, as the particle system will take care of that automatically. As each particle reaches the end of its lifetime, it is automatically removed from the scene.

You can build and run the game now, but you will see that the particle system takes a bit of time to create the particles. To make it look as if it is getting created instantaneously, we will add the following highlighted code after adding the particle to the scene:

```
self.addChild(explosionParticle)
explosionParticle.advanceSimulationTime(0.25)
```

This line will fast forward the simulation to 0.25 seconds after creating the particle so that it looks as if the particles are getting created instantaneously. Build and run the game again to see the particles in action.



There are some additional parameters and properties present in the particle designer, such as `advanceSimuationTime`, that can be called and modified only in code. But it is better to design the majority of the particle through the designer and fine-tune it in code later when it is added to the scene.

Now, if you want, you can create the particle system using code and define the parameters individually through the code, as shown in the following snippet, which will give the same result:

```
let explosionParticle = SKEmitterNode()
explosionParticle.particleTexture =
SKTexture(imageNamed: "smoke")
explosionParticle.particleBirthRate = 65.5
explosionParticle.numParticlesToEmit = 12
explosionParticle.particleLifetime = 8.841
explosionParticle.particleLifetimeRange = 16
explosionParticle.particlePositionRange =
CGVector(dx:5.0, dy: 9.0)
explosionParticle.emissionAngle = 0
explosionParticle.emissionAngleRange = 360
explosionParticle.particleSpeed = 48
explosionParticle.particleSpeedRange = 90
explosionParticle.xAcceleration = 0
explosionParticle.yAcceleration = 0
explosionParticle.particleAlpha = 1.0
explosionParticle.particleAlphaRange = 0.2
explosionParticle.particleAlphaSpeed = -1.0
explosionParticle.particleScale = 0.25
```

```
explosionParticle.particleScaleRange = 0.125
explosionParticle.particleScaleSpeed = -0.125
explosionParticle.particleRotation = 60
explosionParticle.particleRotationRange = 60
explosionParticle.particleRotationSpeed = 5.0
explosionParticle.particleColorBlendFactor = 1.0
explosionParticle.particleColorBlendFactorRange =
0
explosionParticle.particleColorBlendFactorSpeed =
0
explosionParticle.particleColor = UIColor(red:
0.455, green: 0.784, blue: 0.835, alpha: 1.0)
explosionParticle.particleBlendMode =
SKBlendMode.Alpha

explosionParticle.position = CGPoint(x:
hero.position.x + hero.size.width/2 + 10,
            y: hero.position.y - 5);
self.addChild(explosionParticle)

explosionParticle.advanceSimulationTime(0.25)
```

These are literally the same variables and parameters. The only difference is that here, it is in code format. Obviously, creating the particles using the designer is more convenient.

There are additional commands that can be used on the particle system that are available for your convenience at Apple's developer portal. You might want to go through it and experiment if you are interested.

Summary

In this chapter, we saw how to use actions and sprite sheets to create animation in the game. We looked at SpriteKit's inbuilt sprite sheet generator, and we used a professional tool called Texture Packer to create a sprite sheet. We also saw how easy it is to generate sprite sheets and animations using this professional tool.

In addition to that, we saw an introduction to SpriteKit's particle designer, and created and implemented a particle system in the game.

There is still something missing in the game, however, and that is sound and font customization. We will be looking at this in the next chapter, which includes how to add the final touches to the game.

Chapter 6. Audio and Parallax Effect

Adding background music and sound effects in SpriteKit is very easy, as it is just a one-line code that you need to call. There are a few important things that need to be considered while adding the effects; for example, will the scene change after asking SpriteKit to play a sound? And what format should be used for the sound file?

In this chapter, we will be adding background music to the game and sound effects for the moments when the rocket gets fired, the enemy gets hit by a rocket, a button on the screen is pressed, and the game ends. We will also look at a free application that can be used to convert a sound file from one format to the other. In the end, we will add a nice scrolling effect to the background, for extra fun.

Here are the topics that are covered in this chapter:

- Audio file formats
- Downloading and installing Audacity
- Converting the audio format
- Adding sound effects
- Adding background music
- Parallax background—theory
- Implementing the parallax effect

Audio file formats

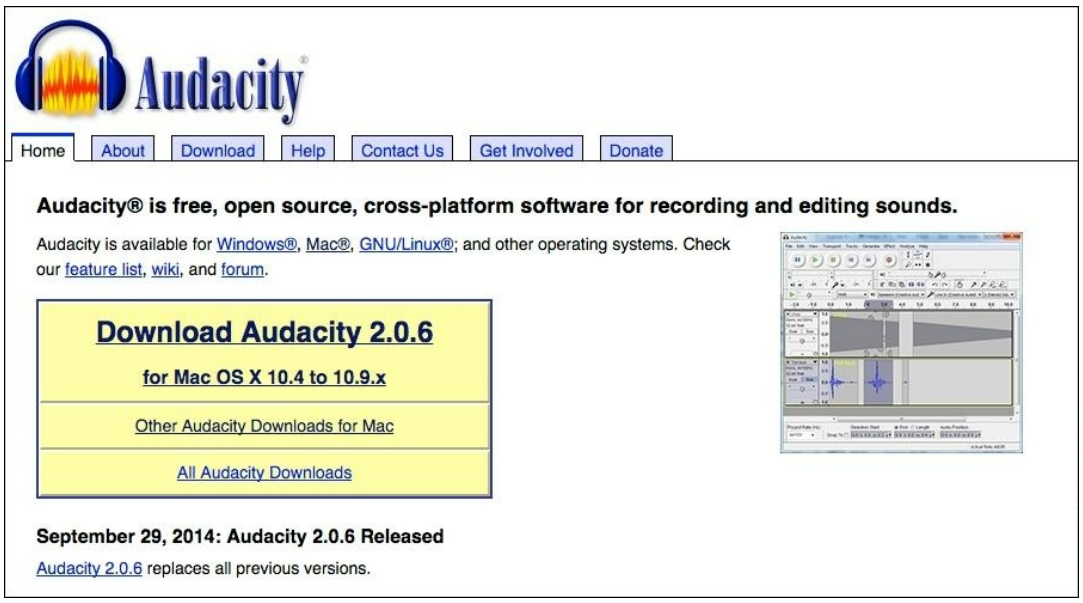
SpriteKit allows us to use the OSX standard `.caf` audio file format, but it also supports `.mp3` files. In theory, you can use any of these formats to play audio or effects in the game. All you will need to do is change the name of the file called when asking SpriteKit to play the file. But for the sake of optimization, we will be using an `.mp3` file format. The reason is that `.mp3` files are a lot smaller in size compared to `.caf` files. Take the example of the background music file, for instance; the `.caf` file size is 5.2 megabytes, compared to the `.mp3` file, which is just 475 kilobytes—roughly 10 times less than a `.caf` file, and the player experience won't be any different.

The more audio files you add, the more it will add to the bundle size, which will drastically increase the size of the game and take longer for the player to download and play the game. For a small game, such as ours, it might not matter much, but when you start making bigger games with more sound effects and background music, this is something to keep in mind.

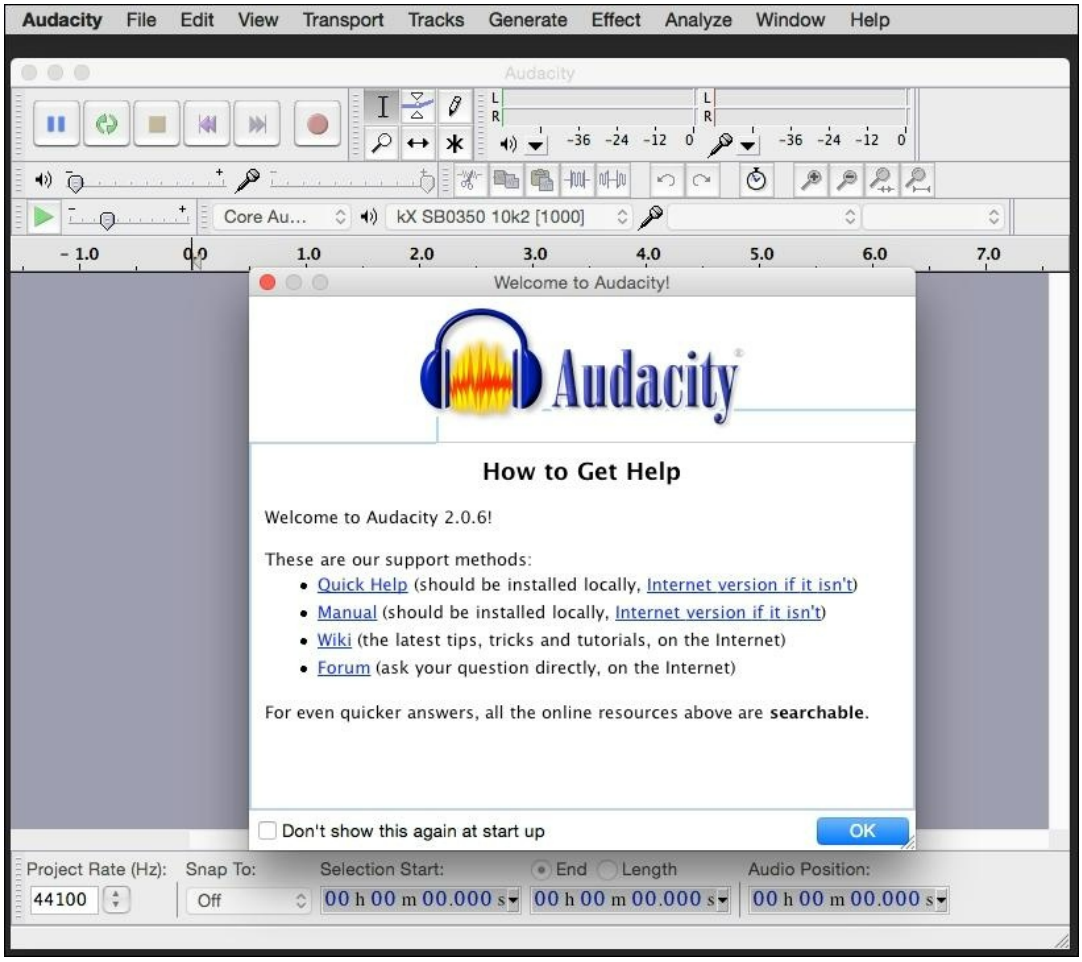
Although `.mp3` is a very popular format, there might be times when you will need to convert audio file formats to `.mp3`. In the next section, we will see exactly how to do this.

Downloading and installing Audacity

Audacity is a free cross-platform audio recording and editing software. You can download the latest version from http://audacity.sourceforge.net/. Although Version 2.06 says it is for Mac OS X 10.4 to 10.9.x, it will also work for Yosemite.



Download the DMG file, double-click on it, and drag the Audacity folder into your **Applications** directory. From the launch pad, click on the **Audacity** application to open it.



You can click on the **Quick Help**, **Manual**, **Wiki**, and **Forum** links to understand more about the software and its features. Click on **OK** to continue.

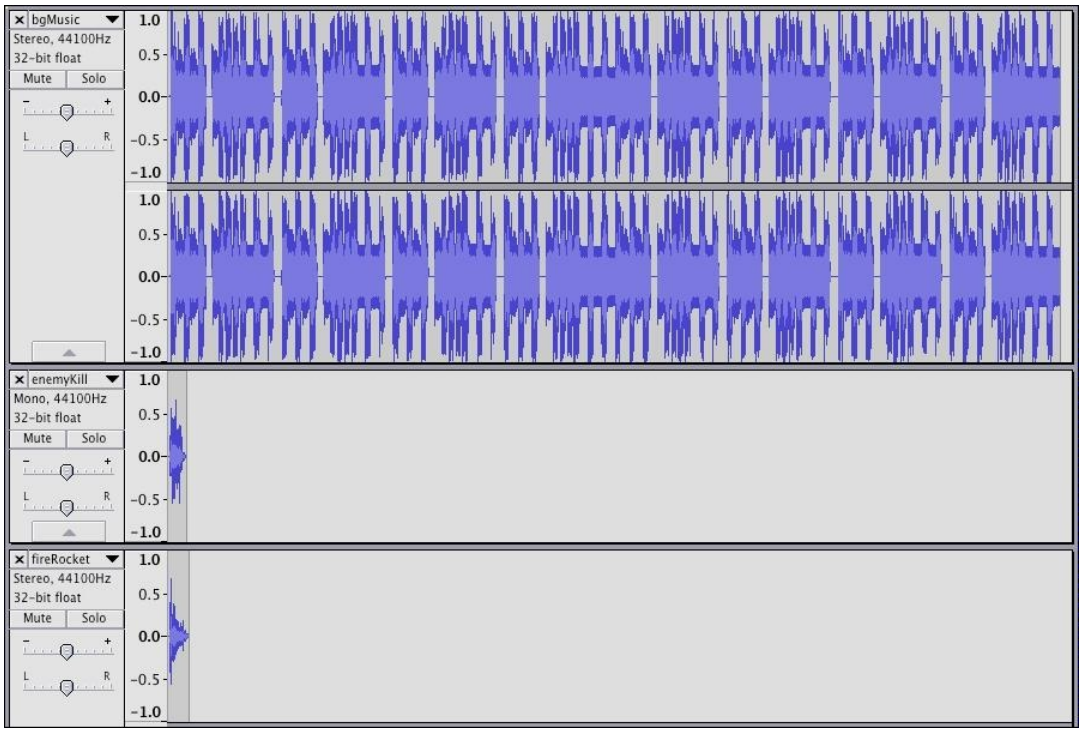
I am not a sound engineer, so I am really not the right person to tell you or explain what each of the headings does. All I can really say is that if you know a thing or two about sound programming or engineering, you can really do magic with this free and open source application. For example, we can reduce the size of a file by lowering the frequency of a file from 44 kHz to 22 kHz, as shown in the following screenshot, or reduce the format from two channels or stereo to mono.

To change the frequency, click on the down arrow next to the name of the file, select **Set Rate**, and choose **22050 Hz** instead of **44100 Hz**:


bgMusic
1.0
Name...
Move Track Up
Move Track Down
Move Track to Top
Move Track to Bottom
Waveform
Waveform (dB)
Spectrogram
Spectrogram log(f)
Pitch (EAC)
Mono
Left Channel
Right Channel
Make Stereo Track
Swap Stereo Channels
Split Stereo Track
Split Stereo to Mono
Set Sample Format
Set Rate
8000 Hz
11025 Hz
16000 Hz
22050 Hz
44100 Hz
48000 Hz

Converting the audio format

To convert the format to MP3, navigate to **File** | **Import** | **Audio**. In the `Resources` folder for this chapter, you will find a folder called `Game Audio Files`. Inside this folder, there is another folder called `caf`. Within this folder, you will have all the audio files in the `.caf` format. Select all the files in the folder and click on **Open**. Now all the files will be imported to the current project.



Once all the files are imported, to convert them to MP3 files, you will need to install the lame library from http://lame.buanzo.org/#lameosxdl. I know it has a very corny name, but it really works. Download the first link, which says **For Audacity 1.3.3 or Later**. Download the DMG file, open the package, and install it.

For FFmpeg/LAME on Mac OSX click below:

If Audacity does not detect FFmpeg, download the ZIP option, extract the files inside to a well known folder, then open Audacity, go to Library Preferences and configure it to search on the well known folder you extracted the files to.

For Audacity 1.3.3 or later on Mac OS X 10.4 and greater (Intel or PPC),and Audacity 1.2.5 on OS X 10.4 and later (Intel)

Lame_Library_v3.98.2_for_Audacity_on_OSX.dmg (ZIP version here)

Now go back to Audacity and navigate to **File** | **Export Multiple**. The following window will open:



In the window that opens, select the format as **MP3 Files**. For the export location, I created a new folder called `mp3` in the `Game Audio Files` folder. For the name of the file, select the first choice so that it will keep the same label name as the file we are giving it. You can check or uncheck the **Overwrite existing Files** box as per your preference.

> ## Tip
>
> Keep in mind that if you check this box, all the files with the same name with be overwritten in future, so be careful.

Click on **Export** to start the process. Then click on **OK** for all the files that are getting exported. You will get a confirmation that all the files were converted successfully, like this:



Now, if you check out the `mp3` folder, you will see that all the files have been converted to the `.mp3` format.



With the files ready, we can add sound and effects to the game.

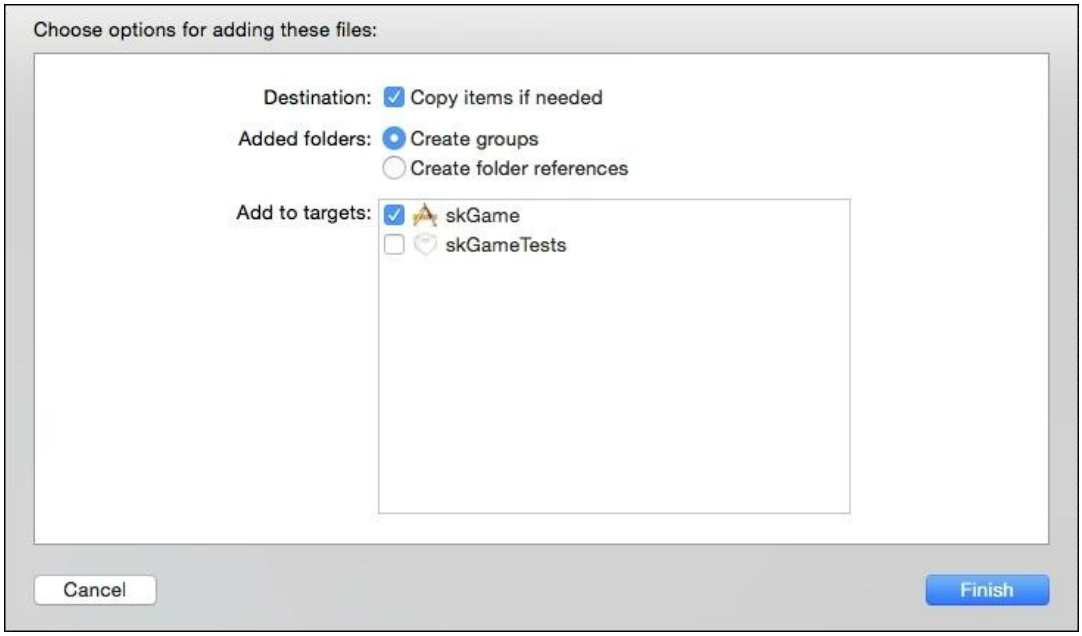
Adding sound effects

As I mentioned earlier, adding sounds and effects in SpriteKit is very simple. It is really one line of code that needs to be added to create the desired sound effect.

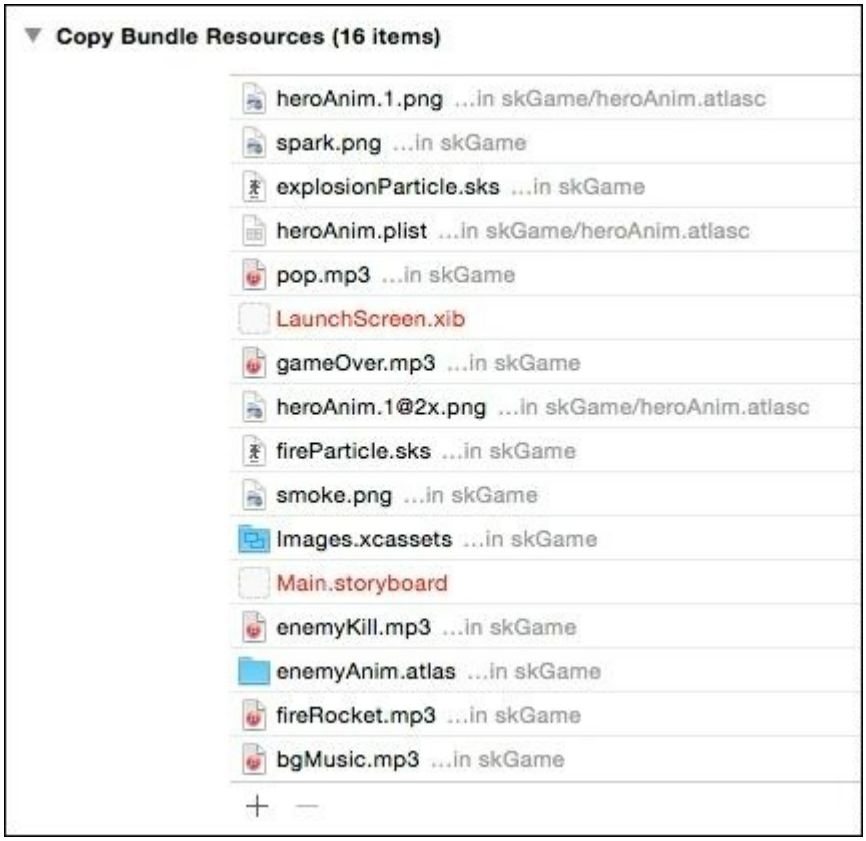
First, we will import all the files to the project. So, select all the `.mp3` files in the folder you created earlier, and drag them into the project.



Make sure that you check **Copy items if needed** and your project is selected as the target.



Also, make sure that the files exist under **Copy Bundle Resources in the Build Phases**, as the files sometimes don't get copied, even after you select the project as the target. If the files don't exist here, then you will get a build error saying the said file can't be found, even though the file exits in the project hierarchy. In that case, click on the **+** button and add the files.



Adding the fireRocket sound effect

Okay! With all the files imported, let's call the `fireRocket` sound effect whenever the hero fires a rocket. At the end of the `addRockets` function in the `GamePlayScene.swift` file, add the following line of code:

```
let fireRocker =
SKAction.playSoundFileNamed("fireRocket.mp3",
waitForCompletion: false)

self.runAction(fireRocker)
```

To play sounds, you will also use the `SKAction` class. The `SKAction` class has a property for playing sounds called `playSoundFileNamed`, and it takes two parameters. The first parameter is the name of the sound file, and the second is a Boolean value that determines the length of the action. If it is `true`, then the duration of the action is equal to the length of the audio track. Otherwise, it is understood that the action terminated immediately.

Notice that we have to provide the full name along with the extension when playing the sound. So, if you are using a `.caf` file in the project and you want to call the file, you must replace the extension accordingly.

If you build and run the game, you will hear the sound effect whenever you click on the right of the screen. Yes, it is really that easy!

Although this is easy, you might have noticed that the first time you fired the rocket, there was a bit of a delay between you tapping the screen and the sound file actually playing. This is because when the sound effect is called for the first time, the system has to fetch the file from the memory. The system has to do this for each of the files we will be calling. Once it has loaded initially, from the next time onwards, the file is already stored in the memory, so it is called instantaneously.

Adding the enemy-kill sound effect

Next, we will add an effect that will occur whenever an enemy gets killed. So, in the `checkCollision` function, where we check for a collision between the player's rocket and the enemies, add the following code after the code by which we increase the score in the `GamePlayScene.swift` file:

```
let enemySound =
SKAction.playSoundFileNamed("enemyKill.mp3",
waitForCompletion: false)

self.runAction(enemySound)
```

And ta-da! The effect will start playing each time an enemy gets killed. You can run the action on the enemy or on the class itself by saying `self`; it doesn't really make any difference, as long you make sure you call the action on something, otherwise, the sound won't get played.

Adding a sound effect at GameOver

Next, we will add a sound for when the game is over. So, in the `GameOver` function, add the line highlighted in the following code snippet after we removed action on the self and the hero. This is important because if you add it before, then the action will be removed immediately and you will be sitting there scratching your head, thinking that you did add the sound, but yet no sound is playing:

```
//removing actions
self.removeAllActions()

for enemy in enemies{

     enemy._sprite.removeAllActions()
}

hero.removeAllActions()
```

```
//playing one last action
let gameOverSound =
SKAction.playSoundFileNamed("gameOver.mp3",
waitForCompletion: false)

self.runAction(gameOverSound)
```

Next, we will add a pop sound effect for whenever an on-screen button is pressed. So, in the `touchesbegan` function, where we check whether the `menuBtn` is pressed, we add the following code just before presenting the scene:

```
let popSound =
SKAction.playSoundFileNamed("pop.mp3",
waitForCompletion: false)
self.runAction(popSound)
```

Now, run the game and let it get over. When you press the home button, there is no sound. There is no sound!?!

Well, since the scene was changed as we moved from `GameplayScene` to `MainMenuScene`, the sound wasn't played.

A simple fix to this problem would be to call `MainMenuScene` after a small delay so that in the meantime, the sound will be played. To do this, create a function called `btnPressed` and copy and paste all of the code that calls `MainMenuScene` in it, as shown here:

```
func btnPressed(){

   let scene = MainMenuScene(size: self.size);
   self.view?.presentScene(scene)

}
```

Also, remove the following lines as they are already getting called in the `btnPressed` function:

```
let scene = MainMenuScene(size: self.size)
self.view?.presentScene(scene)
```

Then, right after where we call `popSound`, we add the following lines of code, in which we create `SKAction`. This will call the `btnPressed` function after a 1-second delay:

```
let  buttonPressAction = SKAction.sequence(
[SKAction.runBlock(btnPressed),
SKAction.waitForDuration(1.0)])
```

```
self.runAction(buttonPressAction)
```

If you press the home button now, the sound will be played, and after a second, the scene will change to `MainMenuScene`.

Similar to how you added `popSound` here, in `MainMenuScene`, you can add the sound effect when the reset and play buttons are clicked on. When the reset button is pressed, you can just call `popSound`, as it will play without any trouble—this is because the scene is not changing. You will need to use a `btnPressed` function when `playBtn` is pressed, as the scene will change instantaneously and the sound will be not be played. So, make sure you create a `btnPressed` function and call it after a delay when the play button is pressed to hear the sound effect play out completely. The code is as follows:

```
    override func touchesBegan(touches: NSSet,
withEvent event: UIEvent) {
        /* Called when a touch begins */

        for touch: AnyObject in touches {
            let location =
touch.locationInNode(self)

            let _node:SKNode =
self.nodeAtPoint(location);

            if(_node.name == "playBtn"){

                //println("[GameScene] play btn
clicked ");

                let popSound =
SKAction.playSoundFileNamed("pop.mp3",
waitForCompletion: true)
                self.runAction(popSound)

                let  buttonPressAction =
SKAction.repeatActionForever(SKAction.sequence([SKAction.runBlock(btnPres
                self.runAction(buttonPressAction)

            }

            else if (_node.name == "resetBtn"){

                let popSound =
SKAction.playSoundFileNamed("pop.mp3",
waitForCompletion: true)
                self.runAction(popSound)
```

```
 NSUserDefaults.standardUserDefaults().setInteger(0,
forKey: "tinyBazooka_highscore")

 NSUserDefaults.standardUserDefaults().synchronize()

                var currentHighScore =
NSUserDefaults.standardUserDefaults().integerForKey("tinyBazooka_highscor
                currentHighScoreLabel.text =
"Current High Score: \(currentHighScore)";

            }

        }
    }
```

Adding background music

Our game is now populated with sound effects, but it will be so much more awesome if we add background music. Since the player will need to keep switching between `MainMenuSceen` and `GameplayScene`, there is no point in using `SKAction` to play sounds, as it won't play correctly or will get truncated each time the scene is changed.

For this purpose, we will have to use Apple's audio visual class called `AVFoundation` to help us get around this issue. As you might have guessed, `AVFoundation` is a superclass that handles both the audio and video functionalities, so if you want any video to be played in your game (say, for a tutorial), then you can very well use this class—that is why it was created. But let's get back to including the background music in our game.

Adding audio loops

To create audio loops, we have to import `AVFoundation`, and since we have to play the audio track as soon as the game starts, let's add it in the `GameViewController` class. So, add the following at the top of the class, as highlighted:

```
import UIKit
import SpriteKit
import AVFoundation
```

Next, after we create the class, we include the following highlighted line:

```
class GameViewController: UIViewController {

let bgMusic = AVAudioPlayer(contentsOfURL:
NSBundle.mainBundle().URLForResource("bgMusic",
withExtension: "mp3"), error: nil)
```

Here, we create a new constant called `bgMusic`. To it, we assign the `AVAudioPlayer` function. It takes two parameters. The first is the location of the file, and the second is the error message to be displayed.

In the first parameter, we pass the name and extension so that the file can be retrieved from the resource location of main bundle. In the `URLForResource` function, we pass the name of the file, which is `bgMusic`, and provide the extension, which is of the MP3 format. This will get the `bgMusic.mp3` file, which was previously added in **Copy Bundle Resources** under the **Build Phases** section.

Also, this needs to be retrieved before the view is loaded. If we add this line to `viewDidLoad`, then it won't cause errors, but the file will not play properly, so it is called outside the `viewDidLoad` function.

Once the file is loaded in the `viewDidLoad` function, add the following line to tell the file to loop forever and start playing:

```
bgMusic.numberOfLoops = -1
bgMusic.play()
```

By assigning a value of `-1` to the number of loops, we are telling it to play the sound file in a loop continuously. If we assign `0`, it will play it once; if we assign `1`, it will play the sound file twice; and so on.

And that is all! Now, build and run the game to enjoy the awesome background music.

Now that we have added sound in our game, let's also add a scrolling background for fun.

Parallax background theory

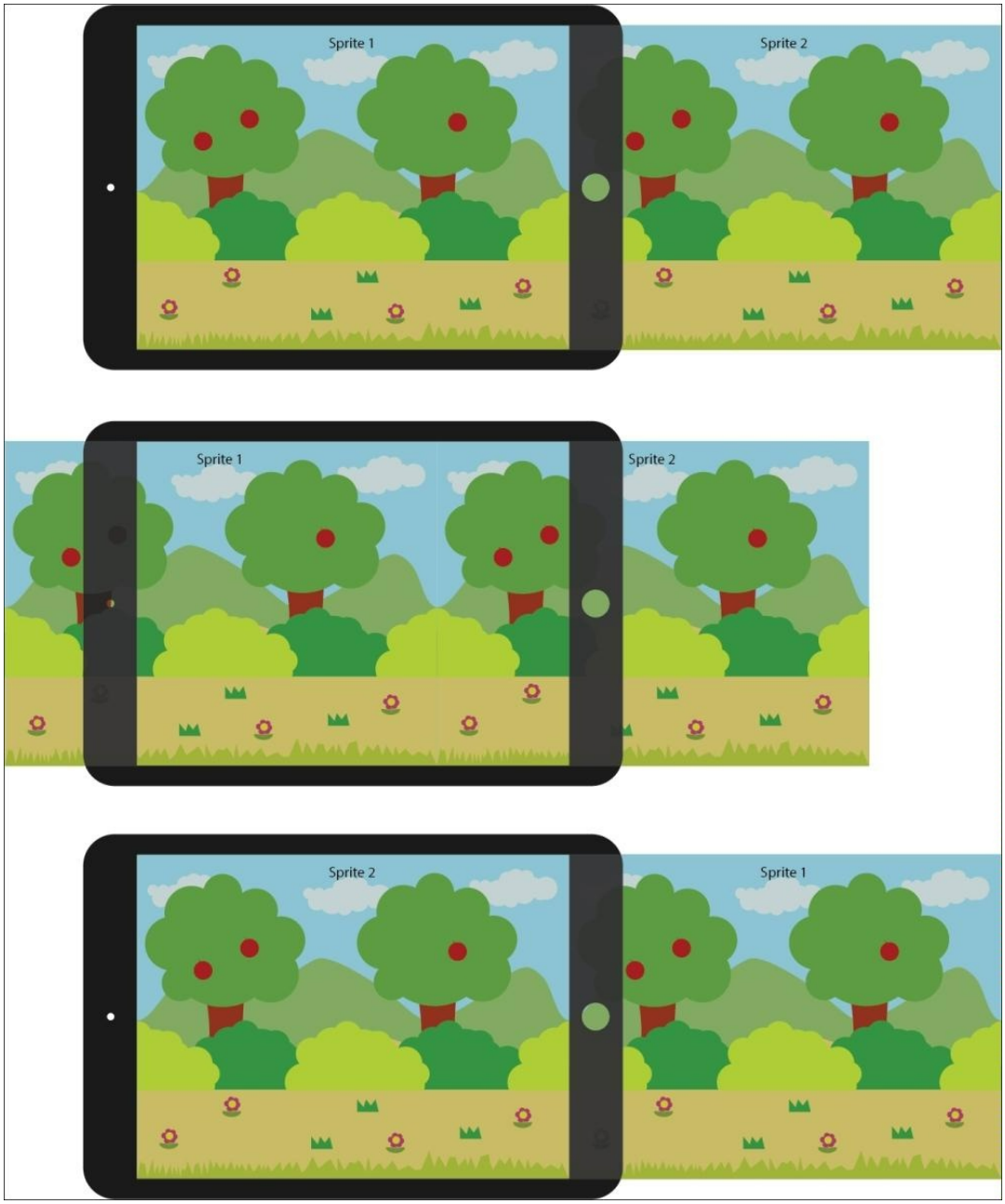
In this section, we will add a parallax or scrolling background. This is a very popular effect in games where the objects in the foreground will move faster than the objects in the background, which will move much slower and give the illusion of depth and motion. This is much similar to the movies of the yesteryears, where the hero or the subject will be stationary and act as if they are galloping on a horse, and the background will be looped to give the illusion that the hero is actually moving forward in the scene.

We will be implementing a very simple parallax effect, where all the objects in the background such as the trees, bushes, and grass will move at the same speed. So, we will just take the background image and make it move in a loop.

> **Note**
>
> This is how the parallax effect is achieved: instead of one sprite of the background image, we will use two sprites and place them adjacent to each other horizontally at the start of the game, as seen in the following image. The first sprite will be visible but the second sprite will be offscreen and won't be visible to the player initially.

When the game starts, both the sprites will be moved with a certain speed in the negative *x* direction, that is, towards the left of the screen. Both the sprites will be moving at the same speed. So, once the game starts, `sprite1` will slowly go offscreen bit by bit and the second sprite will start becoming visible.

Once the first sprite goes completely offscreen, it is quickly moved to the end of the second sprite, which is the same position the second sprite was at the start of the game.



Then process continues in a loop. Both the sprites always move towards the left of the screen. After each sprite goes offscreen on the left, it is placed at offscreen on the right of the screen and continues to move left.

There are a couple of things that need to be kept in mind when creating assets for parallax scrolling and coding a parallax effect. The first is that when creating assets for a parallax effect, the art needs to be continuous. For example, if you look at middle of the preceding image, you will see that the mountains look like a continuous mountain range. Even though `sprite1` and `sprite2` are two different images, when put together, they appear to be a single image. This can again be seen in the light-green bush below the mountain. The left part of the bush is in `sprite1` and the right is in `sprite2`,

yet when the two sprites are kept adjacent to each other, it gives a seamless illusion of being part of a single bush.

The second aspect to keep in mind is the image gaps. Even if you make the images seamless and make the sprites move at the same speed, you might sometimes encounter gaps between the sprites. This is not a very common problem, but in some frameworks, it might exist. To counter it, you can stretch the images just by a bit so that the sprites overlap each other and it is not very obvious to the player. The other method is to make sure you manually place the sprites at the end of the on-screen sprite and also make the necessary adjustments, if required, to bridge the gap between the sprites.

This is the main theory behind parallax scrolling. Let's look at it in practice in the upcoming code.

Implementing the parallax effect

To create a parallax effect for the background, we have to create a new class similar to how we created the `MovingSprite` class. So, go to **File** | **New** | **File** and create a new swift file called `ParallaxSprite`.

In the file, import SpriteKit at the top of the file and create some constants. In the class, we will just take the name of the file that we want for the parallax effect. Then we will create two sprites called `sprite1` and `sprite2` from it. We will take a value of speed at which we want to move the sprites. We will then take the instance of the `GameplayScene` class so that we can add the sprites to the gameplay class. We will also create a global constant to get the size of the view:

```
import Foundation
import SpriteKit

class ParallaxSprite{

    let _sprite1: SKSpriteNode!
    let _sprite2: SKSpriteNode!
    let _speed : CGFloat = 0.0
    let _viewSize:CGSize!

} //class end
```

Next, we will create the `init` function for the class in which we will take the *names* of sprite to be parallax, the *speed*, and the gameplay scene, and initialize the constants we created at the top of the class:

```
    init(sprite1: SKSpriteNode, sprite2:
SKSpriteNode, viewSize: CGSize, speed: CGFloat){

        _speed = speed
        _viewSize = viewSize

        _sprite1 = sprite1
        _sprite1.position = CGPoint(x:
_viewSize.width/2, y: _viewSize.height/2)

        _sprite2 = sprite2
        _sprite2.position = CGPoint(x:
_sprite1.position.x + _sprite2.size.width - 2, y:
_viewSize.height/2)

    }//init
```

In the `init` function, we initialize the speed variable with whatever value was passed in. We will also assign the size of the view that can be retrieved

from the global constant we created in the `GameplayScene` class. We also assign the two sprite names passed into the local sprite variables: `_sprite1` and `_sprite2`.

The `_sprite1` object is positioned at the center of the view, so the `x` and `y` position is obtained by diving the width and height of `viewSize` by half.

For the second sprite, `_sprite2`, the height is kept at half the height of the screen, but as for the position, it is placed so that the left edge of `sprite2` overlaps the right edge of `sprite1`. So, the second sprite is kept at the `x` position equal to the width of `sprite2`. The `-2` is small adjustment factor that is used to make sure that the two sprites overlap each other. This was added after some trial and error to arrive at the number.

> **Tip**
>
> You can increase or decrease this value to see what effect it has, and if you want, you can add more or less overlapping, depending on your preference.

Also, note that the sprites are not added to the current class but to the `GameplayScene` class. We can't add the sprites to the current class because we don't inherit from `SKNode` or `SKSpriteNode`, so the current class doesn't have the `addChild` property.

Next, we define the `update` function, as we will need to update the positions of `sprite1` and `sprite2` continuously. So, right after the `init` function, add the `update` function:

```
    func update(){

        _sprite1.position.x += _speed
        _sprite2.position.x += _speed

        if((_sprite1.position.x +
_sprite1.size.width/2) < 0){

            _sprite1.position = CGPoint(x:
_sprite2.position.x + _sprite1.size.width - 2, y:
_viewSize.height/2)

        }


        if((_sprite2.position.x +
_sprite2.size.width/2) < 0){
```

```
            _sprite2.position = CGPoint(x:
_sprite1.position.x + _sprite2.size.width - 2 ,
y: _viewSize.height/2)

        }

}//update
```

In the `update` function, we increment the position of the sprites with a speed, so that the sprites move. So, since we are incrementing and not decrementing the value, when we create an instance of this class, we will need to remember to provide a negative speed value so that sprite moves towards the left, or else sprite will start going in the positive *x* direction.

Next, we check whether the right edge of `sprite1` has gone beyond the left end of the screen. If the sprite has gone off-screen, we get the position of `sprite2`, place `sprite1` at the end of the sprite, and subtract it by the adjustment factor to avoid the gap. The similar process is done for `sprite2` also, but here we place it at the end of `sprite1`.

That is for the `ParallaxSprite` class.

To implement this class, go to the `GameplayScene` class and add a global constant after `var score:Int = 0` at the top of the class. Type `ParallaxSprite` with the name as `scrollingBg`, as shown in the following code. Don't forget the exclamation mark at the end:

```
let scrollingBg:ParallaxSprite!
```

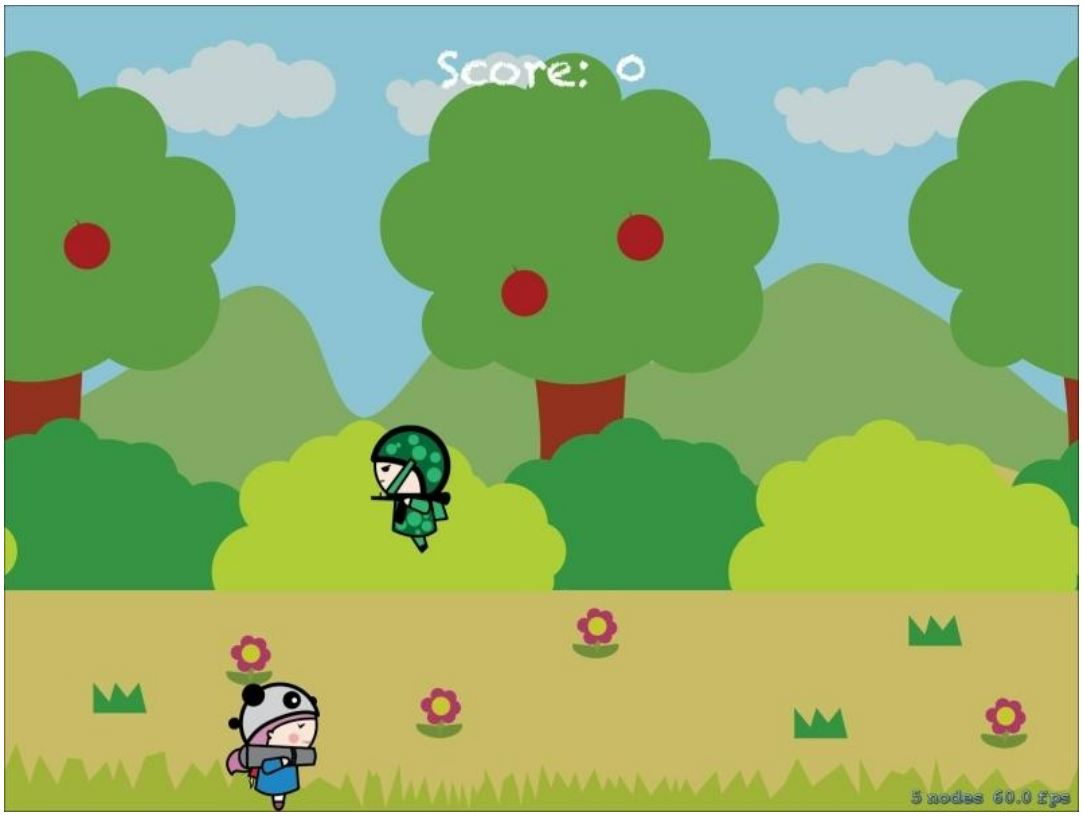
Next, we remove the code we added to include the BG sprite in the `init` function, and add these lines in its place:

```
            let BG1 = SKSpriteNode(imageNamed: "BG");
  self.addChild(BG1);
            let BG2 = SKSpriteNode(imageNamed: "BG");
  self.addChild(BG2);

            scrollingBg = ParallaxSprite(sprite1:
BG1, sprite2: BG2, viewSize: viewSize, speed:
-5.0)
```

We create two sprite constants called `BG1` and `BG2`, and pass the image `BG`. Next, we initiate the `scrollingBg` class. In it, we pass the names of the image that we want to create scrolling effect for, which is the `BG1` and `BG2` sprites. We give it a speed of `-5.0` and give the current instance of the gameplay scene to it, that is, `self`.

Next, we need to call the update function of the `ParallaxSprite` class so that the objects' positions get updated, and then we should have our scrolling background:

```
if(!gameOver){

        scrollingBg.update()

        updateGameObjects()
        updateHero()
        checkCollision()
}//gameOver check
```



Summary

So finally, in this chapter, we have added some music and effects to go along with the gameplay. We also looked at the difference in audio formats and how we can convert one audio format to other.

To top it all, we added a parallax effect to the game, making it look livelier. Like the background, you can also pass other objects to create an even better parallax effect. For free royalty effects and music, you can visit www.freesound.org or www.soundbible.com.

We will see how to add an opening splash screen, add icons, upload the game to the App Store, and add the final touches to the game in Chapter 10, *Publishing and Distribution*. In the next chapter, we will look at some of the more advanced features of SpriteKit such as physics, lighting, and so on. See you in the next chapter!

Chapter 7. Advanced SpriteKit

In this chapter, we will cover some advanced features that are present in SpriteKit, such as lighting and physics, that will make the process of game development easier and will make our game looking prettier. With lighting, we can create a light source and make certain objects in the scene cast shadows and get affected by the light. Using a physics engine, we can make the game objects automatically get affected by gravity and forces applied externally.

We will also look at how to import classes, which are written in Objective-C, so that if you have already written some classes for the SpriteKit Objective-C, we can easily bring them into Swift without the need to rewrite the code again.

Using this new knowledge of importing Objective-C classes, we will look at tools such as Glyph Designer and Spine. With Glyph Designer, we can have custom fonts that take up less space and processing power than regular labels, and with Spine, we can create skeletal-based animation, which is a better optimized way of creating animations.

The topics covered in this chapter are as follows:

- Lighting and shadows
- Sprite Illuminator
- Physics
- Objective-C in Swift
- Glyph Designer
- Skeletal animations

Lighting and shadows

We can create light sources in SpriteKit using a **LightNode**. The LightNode can be placed in the scene like a sprite node by adding it to the scene.

To create a light source, open the `MainMenuScene` class and add the following after we added the background to the scene:

```
let lightNode = SKLightNode()
lightNode.position = CGPoint(x: viewSize.width *
0.5, y: viewSize.height * 0.75)
lightNode.categoryBitMask = 1
lightNode.falloff = 0.25
lightNode.ambientColor = UIColor.whiteColor()
lightNode.lightColor = UIColor(red: 1.0, green:
1.0, blue: 0.0, alpha: 0.5)
```

```
lightNode.shadowColor = UIColor(red: 0.0, green:
0.0, blue: 0.0, alpha: 0.3)
addChild(lightNode)
```

Similar to how we create an SKSpriteNode, we create an SKLightNode to create lights in Swift. We position it by centering it along the width and placing it at three quarters the height of the screen.

If you build and run the game now, you will see that there is no change to the scene. This is because we have to tell the scene specifically which objects should be affected by the light source. This is done by assigning a category bitmask to the light source so that we can later go to the object and tell it to be affected by the light of a certain bitmask. Here, we assign the bitmask as `1`. Since the bitmask takes a UINT32, there can be 32 light sources in total in a scene at a time, as there are 32 bits or 4 bytes in an integer.

By setting the bitmask category of the LightNode to `1`, we are saying the first bit is switched ON for this light source. So, while assigning categories, you won't use general integer numbers to define a category. So, the category of a light should be 1, 2, 4, 8, 16, and so on. Here, `1` means that the first bit is ON, `2` means that the second bit is ON, and `4` means that the third bit is ON, and so on. Do not use general numbers such as 1, 2, 3, 4, and 5 and so on for defining categories, as it would lead to unexpected results.

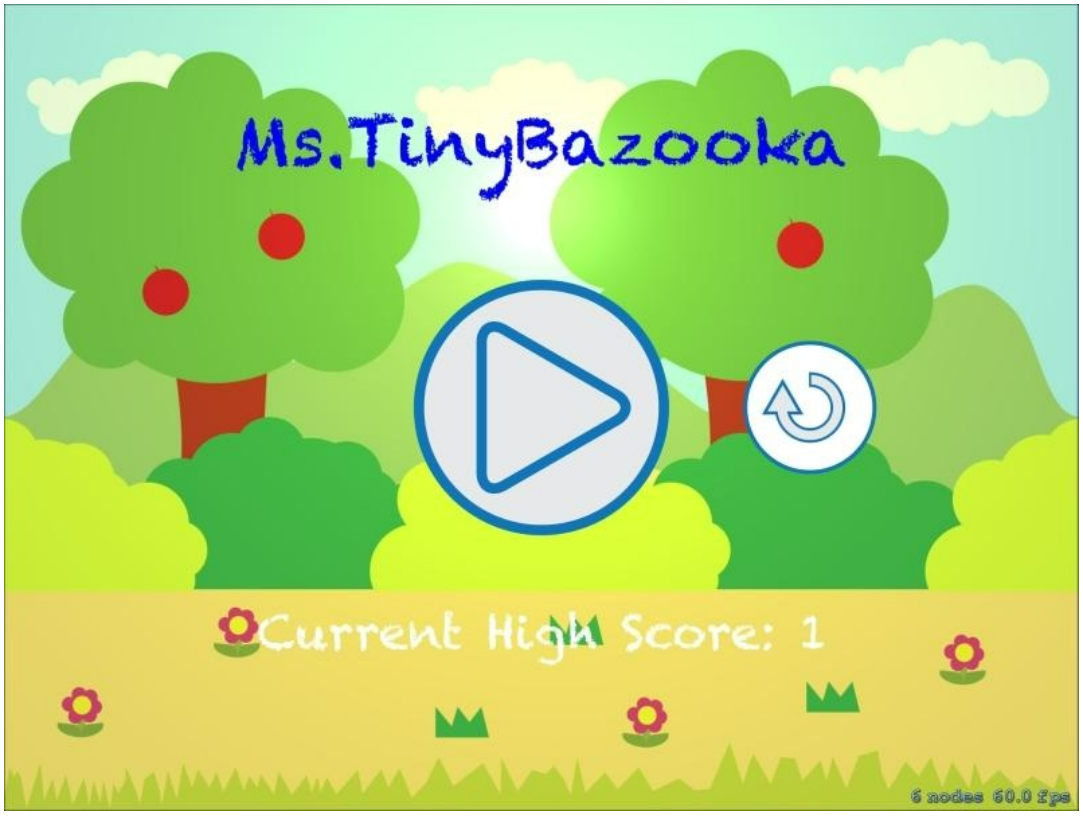
While creating light sources, we also need to provide other information regarding the light source, such as falloff. Like all light sources, it has maximum intensity at the source, and the more you move away from the source, the weaker and weaker the intensity becomes. Falloff determines how quickly the light source loses its intensity. A value of `1` means that it will never lose its value of intensity and `0` means that it will lose it immediately.

Along with bitmask and falloff, we also need to provide the color of the light source, the ambient light, and the color of the shadows. For the light and shadow color, we give white and black. Note that in the shadow color, we reduced the value of the opacity as otherwise the shadows will be completely black. For the ambient color, we reduced the blue variable to zero, as there is enough blue from the sky in the background. Then, we add the light to the scene.

To actually make an object get affected by the light source, we have to assign the `lightBitMask` property of that object to the category bitmask of the light source we assigned earlier.

So, to make the BG get affected by the light source, add the following in the code after where we added the BG to the scene and run the game to see the result, as shown in the following image:

```
BG.lightingBitMask = 1
```



Wow! Isn't that pretty? Here I changed the color of the font to blue so that the light source is easily visible. This can be done by adding the following highlighted line in `myLabel`:

```
let myLabel = SKLabelNode(fontNamed:"Chalkduster")
myLabel.text = "Ms.TinyBazooka"
myLabel.fontSize = 65
myLabel.position = CGPoint(x: viewSize.width/2,
y: viewSize.height * 0.8)
myLabel.fontColor = UIColor.blueColor()
self.addChild(myLabel)
```

Now to cast shadows, all we have to do is call `shadowCastBitMask` on the object that you want to cast shadows. Assign the category bitmask of the light source to it so that shadows can be cast.

We will ask the play button sprite image to cast a shadow, so after adding the play button to the scene, add the following code:

```
self.addChild(playBtn)
playBtn.name = "playBtn"

playBtn.shadowCastBitMask = 1
```

Now, the play button will cast a shadow depending upon where the light source is.

There are a few things to keep in mind while creating light sources and shadows in a scene. Only sprites can cast shadows in a SpriteKit scene. I tried applying it to the label and it wasn't able to create shadows from the text.

If you are making an object get affected by the light, make sure that it is never deleted and then added back into the scene. If the object is removed, the light source has to recalculate the lighting in the scene, and this process is done again once the object is added back into the scene. This will cause a flicker while recalculating the light and shadows in the scene.

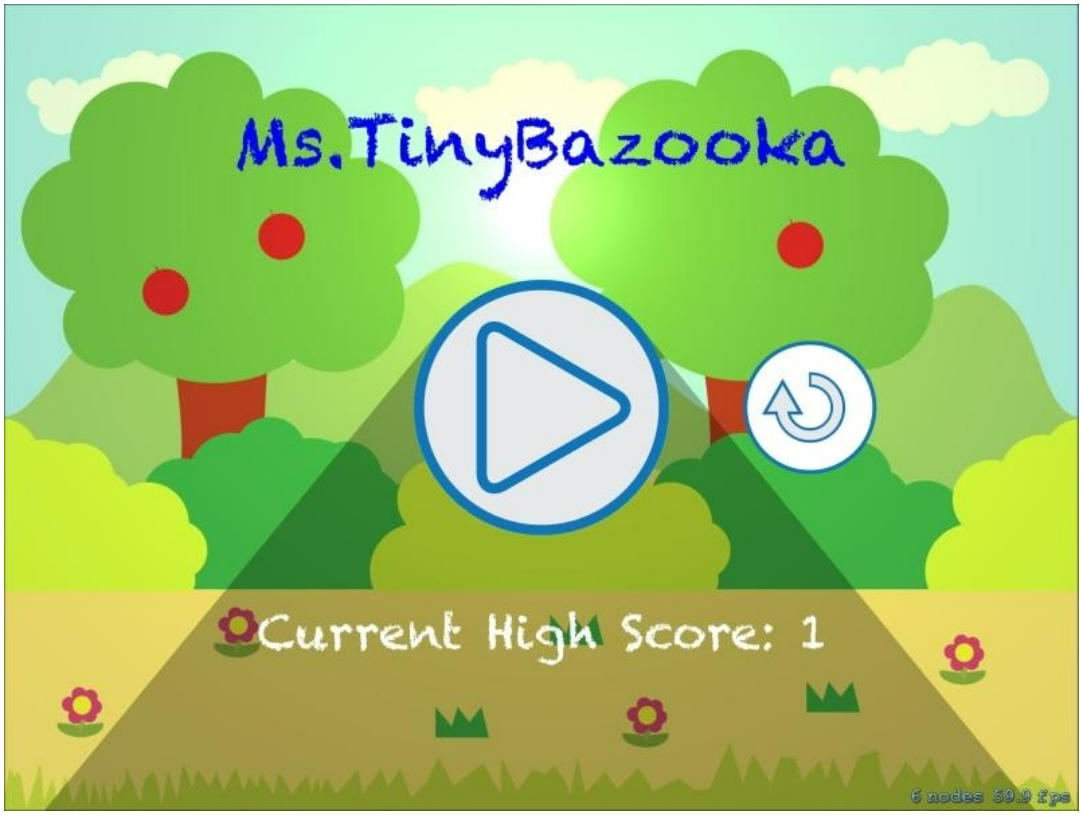
Note

It's is better to add shadows to objects that you know won't get deleted in the future.

Also, in `GameViewController`, change `ignoreSiblingOrder` to `false`:

```
skView.ignoresSiblingOrder = false
```

With that said, let us run the scene to see the shadow in action, as shown in the following image:



This is awesome. But wouldn't it be great if we could add some moving clouds that cast shadows in the scene? Also, since the cloud is moving, we can use the parallax sprite class to make the BG scroll like we did in the previous chapter. We can also add a sprite for the sun and make it spin.

To do this for this chapter, we will create a separate sprite for the sky and change the BG sprite to not include clouds, as we will be adding a new cloud sprite and moving it using the moving sprite class and making it loop around the scene. So, go to the resources folder of this chapter, go into the lighting directory, and get the sky, sun, cloud, and the new BG images. Create image assets called sky, sun, and cloud. Replace the old BG sprite with the new ones.

In the `MainMenuScene` class, add the sky sprite first as follows:

```
let sky = SKSpriteNode(imageNamed: "sky",
normalMapped: true)
sky.position = CGPoint(x: viewSize.width/2, y:
viewSize.height/2)
self.addChild(sky)
```

Then, add the sun sprite to the scene as follows:

```
let sun = SKSpriteNode(imageNamed: "sun")
sun.position = CGPoint(x: viewSize.width * 0.5,y:
viewSize.height * 0.75)
```

```
addChild(sun)
sun.lightingBitMask = 1

sun.runAction(SKAction.repeatActionForever(SKAction.rotateByAngle(1,
duration: 1)))
```

After adding the sun to the scene, make it get affected by the light. Then, we create a new `repeatActionForever` variable on it so that it will rotate the sun sprite by one degree every second forever.

Next, at the top of the `MainMenuScene` class, add the following global variables:

```
var scrollingBg:ParallaxSprite!
var cloud:MovingSprite!
var cloudNode:SKSpriteNode!
```

As we want to call the update methods on the `scrollingBg`, `cloudNode`, and cloud objects in the `update` function, we want them to be global variables.

Now, initiate the classes and variables in the `didMoveToView` method.

First create `cloudNode` as follows:

```
cloudNode = SKSpriteNode(imageNamed:"cloud");
cloudNode.position = CGPoint(x: viewSize.width/2,
y: viewSize.height * 0.9)
addChild(cloudNode)
```

Next, initiate the `shadowcaste` property on `cloudNode` and add it to the cloud moving sprite variable so that we can update its position later:

```
cloudNode.shadowCastBitMask = 1
cloud = MovingSprite(sprite: cloudNode, speed:
CGPointMake(-3.0, 0.0))
```

We make the cloud move slowly when compared to the background since the background is moving at speed of `-5.0`. If we make both of them move at the same speed, then it will look as if the background and cloud are one image and moving together.

Below this, add the light source.

After this, initiate the `scrollingBg` class by creating two sprite images called `BG1` and `BG2`. Add them to the scene and initiate `lightingBitMask` on both the background sprites.

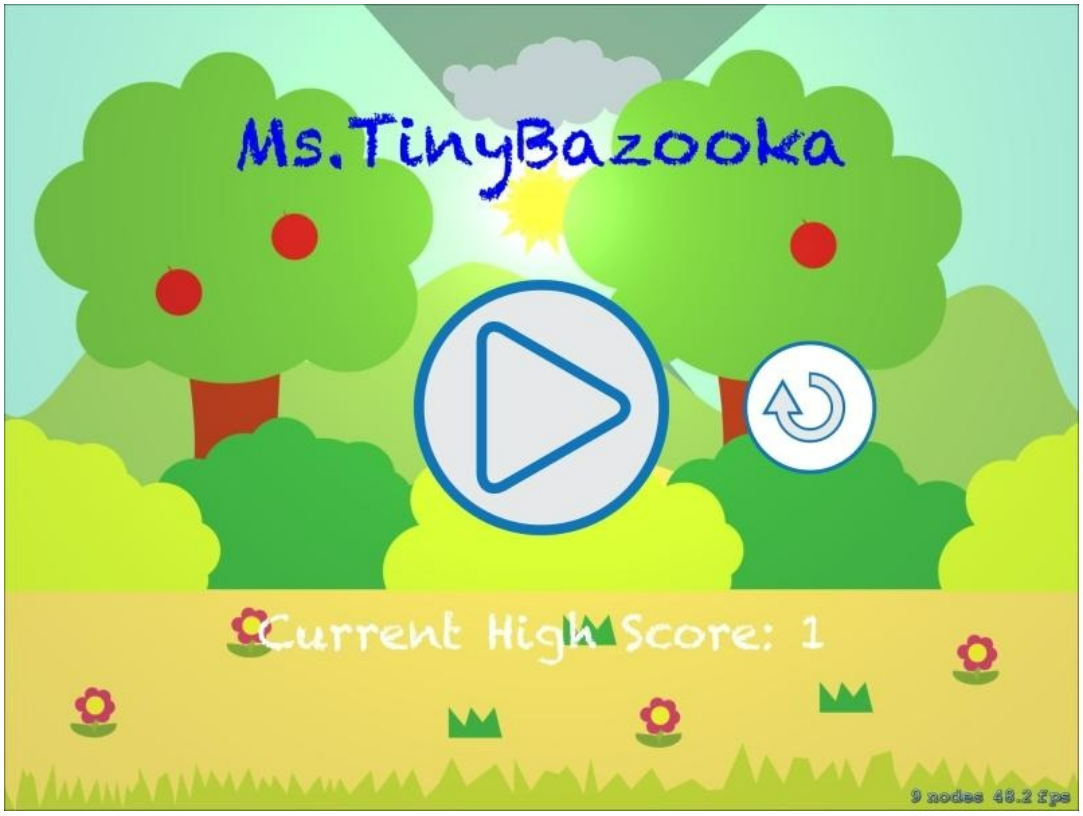
Next, initiate the `scrollingBg` class by passing in the background sprites, the `viewsize`, and the speed as follows:

```
let BG1 = SKSpriteNode(imageNamed: "BG")
self.addChild(BG1)

let BG2 = SKSpriteNode(imageNamed: "BG")
self.addChild(BG2)

BG1.lightingBitMask = 1
BG2.lightingBitMask = 1

scrollingBg = ParallaxSprite(sprite1: BG1,
sprite2: BG2, viewSize: viewSize, speed: -5.0)
```

Next, we have to update the position of `scrollingBg`, the cloud, and the cloud node objects. So, add an update function and add the following:

```
override func update(currentTime: CFTimeInterval)
{
   /* Called before each frame is rendered */

   scrollingBg.update()
   cloud.moveSprite()

   if((cloudNode.position.x +
cloudNode.size.width/2) < 0){

       cloudNode.position.x = viewSize.width +
cloudNode.size.width/2
   }
}
```

Here we call the `update` and `moveSprite` functions of `scrollingBg` and cloud objects. We also update the position of the `cloudNode` sprite by settings its `x` position to the right of the screen once the image has gone beyond the left, as shown in the following image:



We have to be careful with the order in which the objects were added to the scene to make sure the shadows are cast properly. The order is as follows:

- `sky`
- `sun`
- Cloud (`cloudNode`)
- `lightNode`
- `scrollingBg` (`BG1` and `BG2`)

This is important since the shadow is cast in the same depth as the light source and not at the depth of the object.

Tip

In the preceding image, though we ask the light to cast on the play button, we don't see a shadow as we saw in the earlier screenshot since the background layer is above that. If you bring `lightNode` to the top-most layer, you will see the play button cast a shadow, but then even though the clouds are behind the background layer, their shadows will be seen over the trees in the background layer, which is odd. So, it is important at which layer you add the light source.

One more thing you have to be careful about is the size of the image that you want the light to get affected by. You will see that that I didn't enable the `lightBitMask` property on the sky image, as it brought down

the FPS of the game to 45 and everything was running slowly on my iPadMini Retina. So I disabled lighting on the sky layer and just kept it on the run and background sprites. But this was still causing stutter and slowdown on iPad3, so I had to completely disable lighting on all the images to have a consistent 60 FPS.

Adding lighting and shadows is very hardware and processor intensive, so make sure you do enough testing on all the devices while implementing them in your game so that, irrespective of which device the game is running on, it runs at a smooth FPS.

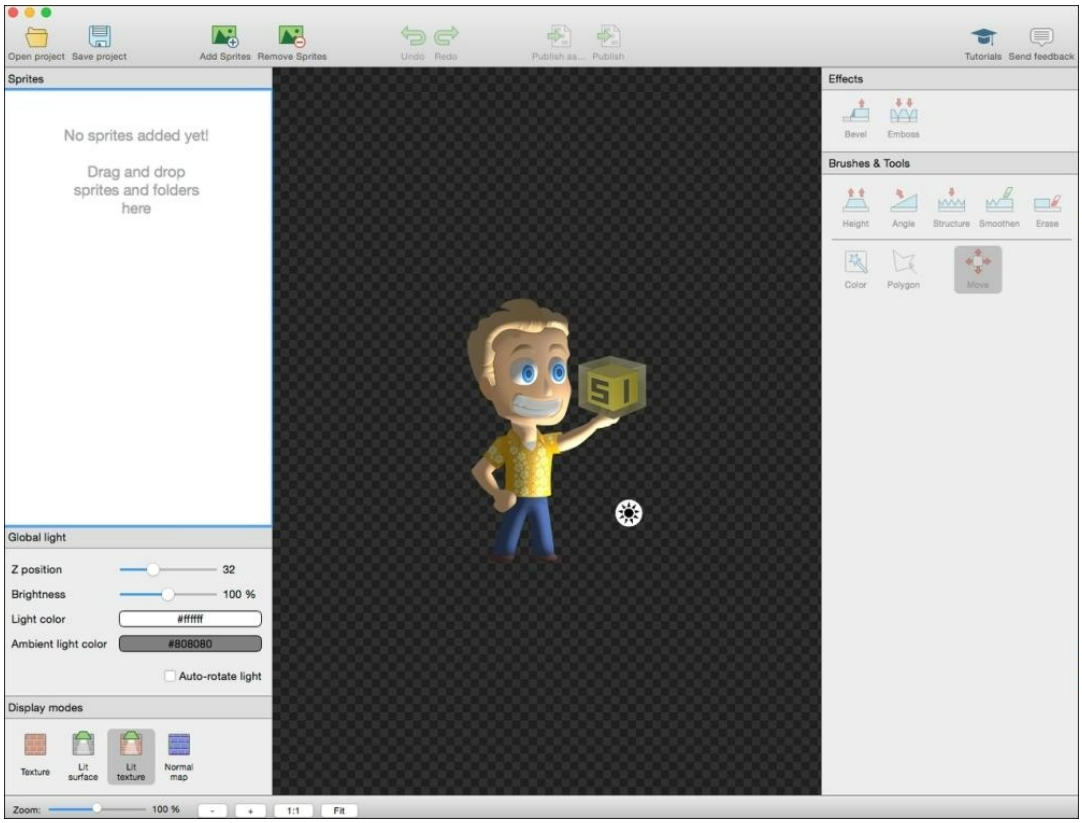
This is all for lighting; we will move to the `GameplayScene` class to look at physics and see how to implement it. We will return to `MainMenuScene` when we look at Glyph Designer and Spine later in this chapter.

In the next section, we will see how to add physics by replacing the small physics engine we created earlier, which was used to create gravity to pull the player down or bump the player up.

Sprite Illuminator

Although SpriteKit adds a normal map on its own, you can also specify your own normal maps for better results. To download Sprite Illuminator, go to https://www.codeandweb.com/spriteilluminator and download the DMG file.

Once Sprite Illuminator has been downloaded, double-click on the DMG file and copy the application to the **Applications** folder. Go to the Launchpad and start the application. Once the application launches, you should see the window as in the following screenshot:



Let us look at the interface in detail.

On the left, we have the **Sprites** panel. The middle panel is the **Preview** panel, in which you can see a preview of all the changes as you make them. The panel on the right is called the **Tools** panel. We will drill down further into each of these panels.

The Sprites panel

In the **Sprites** panel, you can modify the sprite, global light, and display mode as follows:

- **Sprite**: Here we can add or remove the sprites for which we want the normal map to be created.
- **Global light**: We can change the property of the global light source by affecting its `z` position, brightness, light color, and ambient color. This is just for visualization purposes, in the game, we will have to add a light source and change its properties in SpriteKit.
- **Display mode**: By default, the **Lit texture** mode is ON. This gives a preview of how the image will look with lighting and normal map enabled. Texture mode will just show the texture without lighting and normal map. Lit surface will show the image with the light source but without the normal map. The **Normal map** mode will show only the normal map that you have created for an image.

The Preview panel

The **Preview** panel will show the preview of the image depending upon the mode you select in the **Display mode** section.

The Tools panel

We will be spending most of the time in the tools panel. This is where we will be creating the height map for any image. The **Tools** panel includes **Effects** and **Brushes and Tools**:

Under **Effects**, using the **Bevel** and **Emboss** tool we can add height or depth to the map. This works exactly how the same tools work in Photoshop.

Under **Brushes,** five brushes are included. The brushes included are **Height**, **Angle**, **Structure**, **Smoothen**, and **Erase**:

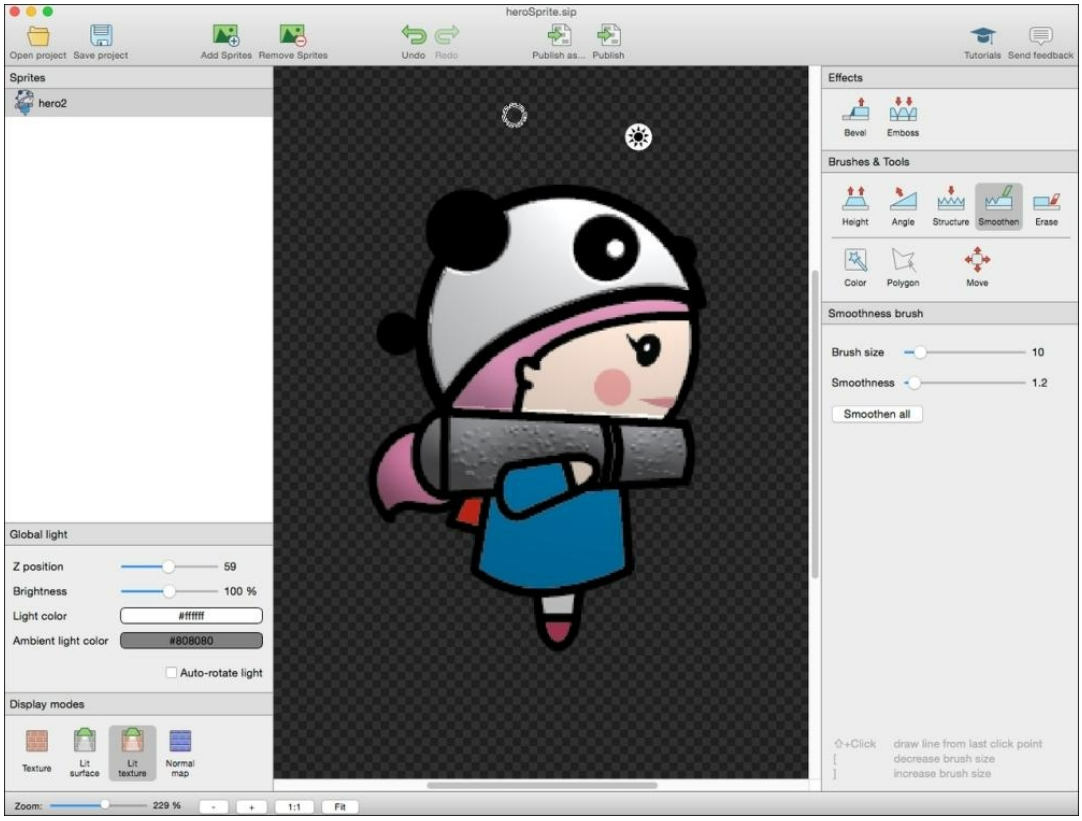
- **Height**: This can be added to an area that you wish to make appear to be protruding from the image. You can specify the values for **Brush Size**, **Height**, **Hardness**, **Contour**, and **Direction** for it.
- **Angle**: This can be added to an area which might be at a particular angle to the direction of the light. This is most useful while developing isometric tile maps. Apart from specifying the values for **Brush Size**, **Opacity**, and **Hardness**, we can specify the direction of the surface by selecting the area around the sphere.
- **Structure**: This brush is used to add some rough texture to an area. You can modify the brush size, density, height, and direction.
- **Smoothness**: This brush is used to smoothen an area. The brush size and smoothness can be controlled by the slider.
- **Erase**: If we wish to erase an effect or brush we can do so by using the erase brush

There are a couple of **selection tools** that are provided that can make the selection process a little simpler. So, if you want to apply a brush or an effect in a particular area, you can isolate that area using the selection tool and apply the effect in just that area:

- **Wand/Color**: This tool is like the wand tool in Photoshop. You can also adjust the tolerance level.
- **Polygon**: You can create a closed polygon loop around the area of an image with this tool.
- **Move**: You can move the image with this tool.

Once you are happy with your creation, you can click on the **Publish** button and a normal map will be created for your image. The normal map will be a PNG file with the same file name as your original file suffixed with `_n`.

Using Sprite Illuminator, I have created a normal map for the hero sprite (shown in the following screenshot). The file and resources are included in the `Resources` folder of this chapter.



In the `Sprite Illuminator` folder in the `Resources` directory, I have created the two versions of the regular image and normal map for both the sizes. I have also renamed the files to avoid any name conflicts. Drag these four files into the project.

To use these assets in the `GamePlayScene.swift` file, comment out the old code for adding and animating the hero sprite, and add the following highlighted code instead.

Here, instead of loading the hero file in the image set, we are assigning the `heroSI` and `heroSI_n` texture to the hero variable.

Also, we set the `lightBitmask` property of the hero to `1` so that the light source affects it.

```
//hero = SKSpriteNode(imageNamed: "hero");
//hero.position = CGPoint(x: viewSize.width/4, y:
viewSize.height/2)
//self.addChild(hero)


hero = SKSpriteNode(texture:
SKTexture(imageNamed: "heroSI"), normalMap:
SKTexture(imageNamed: "heroSI_n"));
hero.position = CGPoint(x: viewSize.width/4, y:
viewSize.height/2)
self.addChild(hero)

hero.lightingBitMask = 1

/*
let heroAtlas = heroAnim()
let heroIdleAnimArray = heroAtlas.hero_Idle_()
let animaiton =
SKAction.animateWithTextures(heroIdleAnimArray,
timePerFrame: 0.2)
let animate =
SKAction.repeatActionForever(animaiton)
hero.runAction(animate)
*/
```

Physics

In other frameworks, you will most probably have to import a physics engine library of your choice, such as Box2d, or chipmunk would configure it to make it work properly. You would also have to write custom code for making collision detection work. In SpriteKit, every scene has physics running in the background as soon as the scene is created. You are not required to do anything else to make it work. So, in the gameplay scene, we will disable the physics engine we created and replace it with SpriteKit's inbuilt physics engine.

Open up the `gameplayScene.swift` file and comment out the `updateHero` function in the `update` function. As you might remember, the `updateHero` function took care of making the hero get affected by gravity, making sure the hero was inside the screen at all times and also making sure thrust is applied when the player taps the left half of the screen. Using the inbuilt physics engine, we will see how we can make it do all the work for us.

As I said earlier, the physics is already active, meaning that there is already some gravity that is acting on the scene. So, let us make the hero get affected by gravity.

In the `init` function, right after we added the hero sprite to the scene, add the following line:

```
hero.physicsBody = SKPhysicsBody(rectangleOfSize:
hero.size)
```

This is all we have to do to tell the hero to get affected by physics. The `physicsBody` property of any sprite will assign a body to the sprite, making it behave like a solid object, meaning that now this sprite will have physics properties like any object in real life. It will have a density, respond to friction, be bouncy, and get affected by other bodies. Now we can apply force to it or make it move with a certain velocity. If some moving body hits this body, as Newton said, it would react to that hit and move.

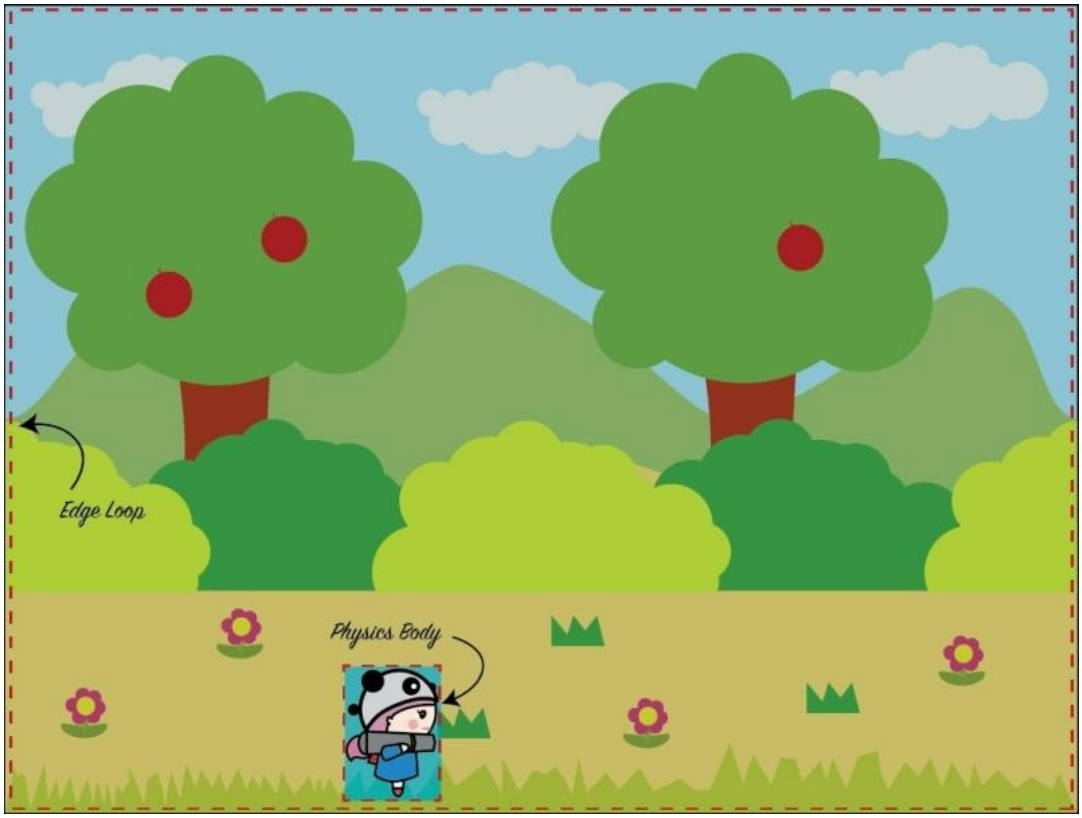
When we assign a body, we have the option of giving shape to the body. Here we are giving a rectangle with the same dimensions as the sprite. If we wish, we could have given the body the shape of a circle or we could also use a custom shape.

If we build and run the game now, we will see that the hero will just fall through the bottom of the screen. This is because nothing is preventing the hero from going through. To counter this downward motion, we have to create a body around the scene to prevent the hero from falling down.

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

```
self.physicsBody = SKPhysicsBody(
                                edgeLoopFromRect:
CGRect(x: 0,

  y: 0,
                                                width:
viewSize.width,
                                                height:
viewSize.height))
```

Previously, we attached a physics body to the hero, and now we are initiating the physics body property of the scene itself. The difference here is that it is an `edgeLoop` body type. An edge loop is different from a regular body type in that the former doesn't have volume. So, it doesn't have mass, density, friction, and so on, and you can't apply force or make it move with a velocity. But other bodies will get affected by it, meaning that if a regular body is moving and gets obstructed by an `edgeLoop`, then the regular body will stop moving.



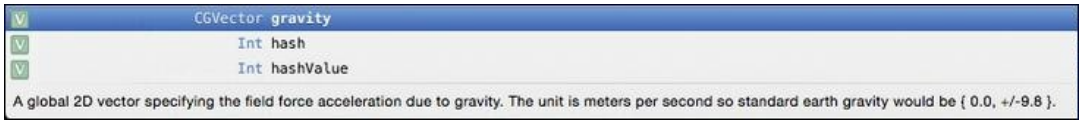
While creating an edge loop, we have to pass in the shape of the loop, so here we are giving the shape of a rectangle starting from the origin that is the bottom left of the screen, which is (0, 0), and the size of the view by passing in the width and height of the screen.

Now, if you build and run, you will see that the hero stops. In fact, the hero not only stops but also bounces a bit once she hits the bottom of the screen. You can set the mass, density, friction, and restitution values depending upon how heavy or bouncy you need your character to be by changing the values, as shown in the following. The values range from `0` to `1` in all cases: `0` being less bouncy, dense, affected by friction, and so on, and `1` being the opposite end of the spectrum.

```
hero.physicsBody?.restitution = 0
hero.physicsBody?.friction = 0
hero.physicsBody?.density = 0
hero.physicsBody?.mass = 0
```

You can also change the value of the gravity in the scene so if you wanted to make a level set on the moon, you can change the default gravity to 1/6th of its original value to give that effect.

One important thing to note about the physics engine is that it is not in pixels but in real-world values. For example, the default value of gravity is actually 9.8 meters/(second * second). All the values, which are actually pixels, are converted to meters, and SpriteKit does the conversion from pixels to meters internally.



Note

To have moon like gravity, access the gravity property of the physics world property of the scene and change gravity to 1/6th of 9.8 as follows:

self.physicsWorld.gravity = CGVector(dx: 0, dy: -1.64)

The `gravity` property expects a `CGVector` value since we want the gravity to exert a downward force, the value of `x` is zero and `y` is kept at `-1.64` from the default value of `-9.8` as this is what gravity would be on the moon. You can change it back to `-9.8` for a more Earthy feeling.

We can now add a force on the hero such that we can thrust her up in the air. So, in the `touchesBegan` function, we can remove the code that we previously added to push the hero up in the air and apply a physics force in the upward direction. But before we add the force, we have to set the velocity of the hero zero since the hero would have to overcome the

downward velocity to move upward. If the velocity in the negative *y* direction is too big, then, irrespective of how much force you apply upward, it will all be nullified by the downward gravitational force acting on the hero. First make the downward velocity of the hero zero and then apply the force upward. Take the following thrust code:

```
//thrust.y = 15.0
```

Replace this with the following code:

```
hero.physicsBody?.velocity = CGVectorMake(0, 0)
hero.physicsBody?.applyImpulse(CGVectorMake(0,
300))
```

Here we set the hero's velocity to `0`. Since there is no velocity acting in the *x* direction, it doesn't matter if we set just the `Y` value to `0` or both the `X` and `Y` values to `0`.

Next, we apply an **impulse** in the *y* direction with an `X` value of `0` and `Y` value equal to `300`. But wait!

Note

What is this impulse thing?

All this time we were talking about applying force and now we are actually applying an impulse. In the physics engine, there is a separate property called force and how it works is that once you apply a force to a body, the force will be constantly applied to the body. We just want the force to be applied once after we tap the left half of the screen. So, we apply an impulse and not force. If we wanted the player to keep moving up once we tap the screen, we should use the force property instead of impulse. So, be sure how you want your object to behave and apply either a force or an impulse on your physics object accordingly.

And that is all. Now you have the player behaving exactly as we did with our own homegrown physics engine previously. Run the game and test it.

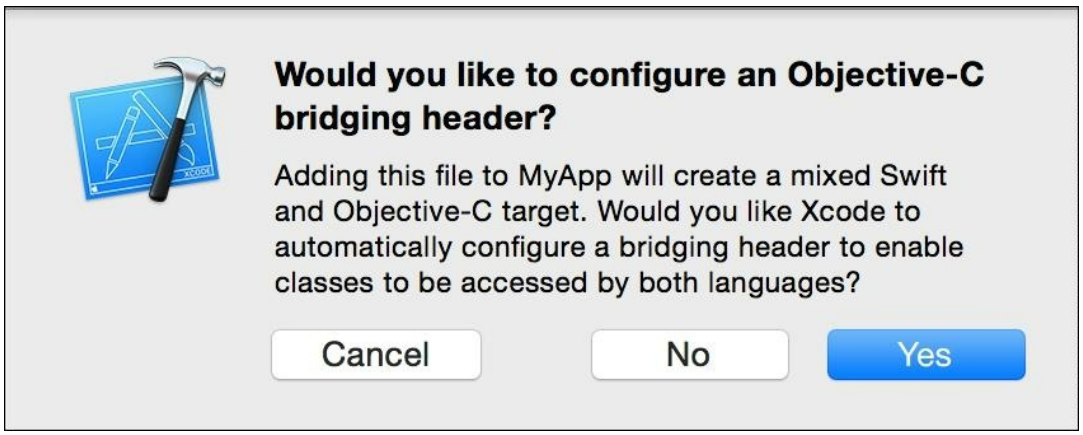
Objective-C in Swift

Let us now look at how to import Objective-C classes into Swift and use them. We will then use this to use tools such as Glyph Designer and Spine, which have implementations in Objective-C but don't have specific classes in Swift.

For making Objective-C classes work with Swift, you will need to create a bridging header file. The file is usually named with the convention `<ProjectName>-Bridging-Header.h`, and then you will need to add the file location into **Objective-C Bridging Header** under **Swift Compiler – Code Generation** in the project `Build Settings`.

> **Note**
>
> If you are using the code provided along with this book, you might need to modify the path to avoid compilation errors.

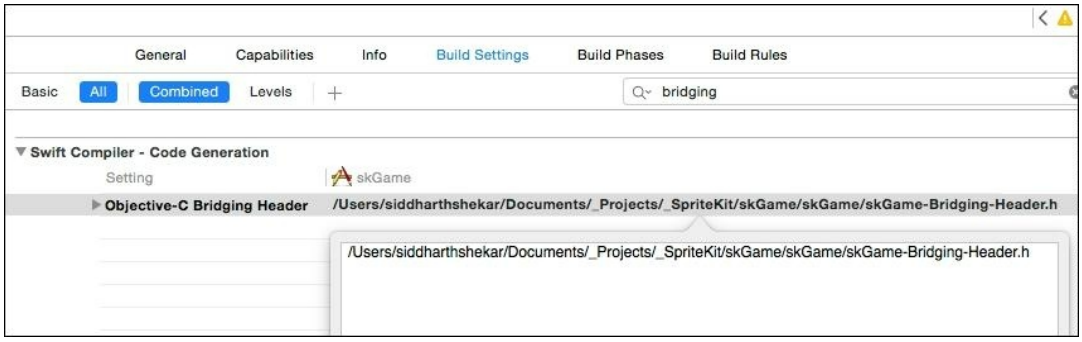
Usually, if you create a new header file in a swift project, you will be greeted with the following window asking to treat the current file created as the bridging header file, and Xcode will automatically add the location of the file in **Build Settings**. In case it doesn't pop up, we will have to walk through the steps to make sure Xcode knows where to look for the bridging header file manually.



For creating the bridging header file for the current project, go to **File** | **New** | **File** and then select **Source** under **iOS** and select the header file with an "H" on it. Name the file **skGame-Bridging-Header** and click on **Create**.

Now, go to **Build Settings** and in the search type `bridging`, as shown in the following screenshot. Double-click to the right of **Objective–C Bridging Header** and drag-and-drop the bridging header file that we just created from the project onto the box. Hit *Enter*. Now, the project knows the location

of the bridging header file. We can use this file to call the header files of Objective-C classes so that they can be shared with Swift.



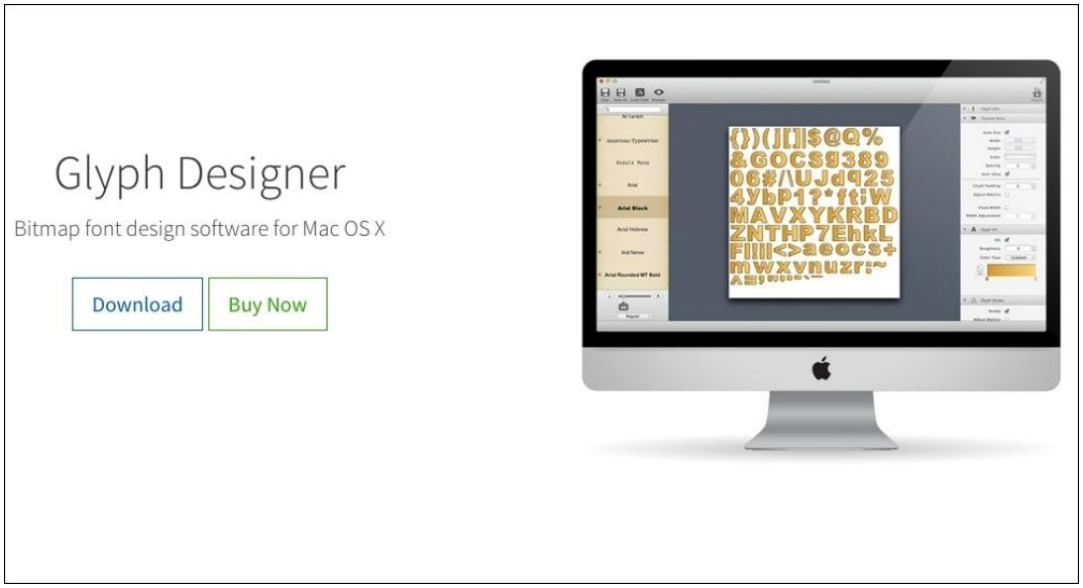
Glyph Designer

Glyph Designer is an application that can be used to make fonts that can be used in the game. But don't we already have an SKLabel for that? That is true. But SKLabel takes the font from the system, and converts the font files into an image at run time and then displays it onto the screen. So, whenever the score needs to be incremented, the system will need to convert the font into an image and then display that on the screen. This is very similar to the problem we faced with sprites earlier and used Texture Packer to get around it.

Although you can use system fonts for bigger games, it is better to use a Bitmap font, a font in which the letters and numbers are already converted to images rather than converting them every time. So, with Glyph Designer, we can create a Bitmap font and use it to better optimize the game.

Bitmap fonts are similar to a spritesheet and will have an image with all the letters, numbers, and symbols in it, and this image file will be accompanied with a data file that has the locations and size of the symbols and letters. Whenever a letter needs to be displayed on the screen, the data file will be checked for the location of the letter, which will be retrieved and then displayed on screen.

The trial version of the application can be downloaded from https://71squared.com/glyphdesigner.

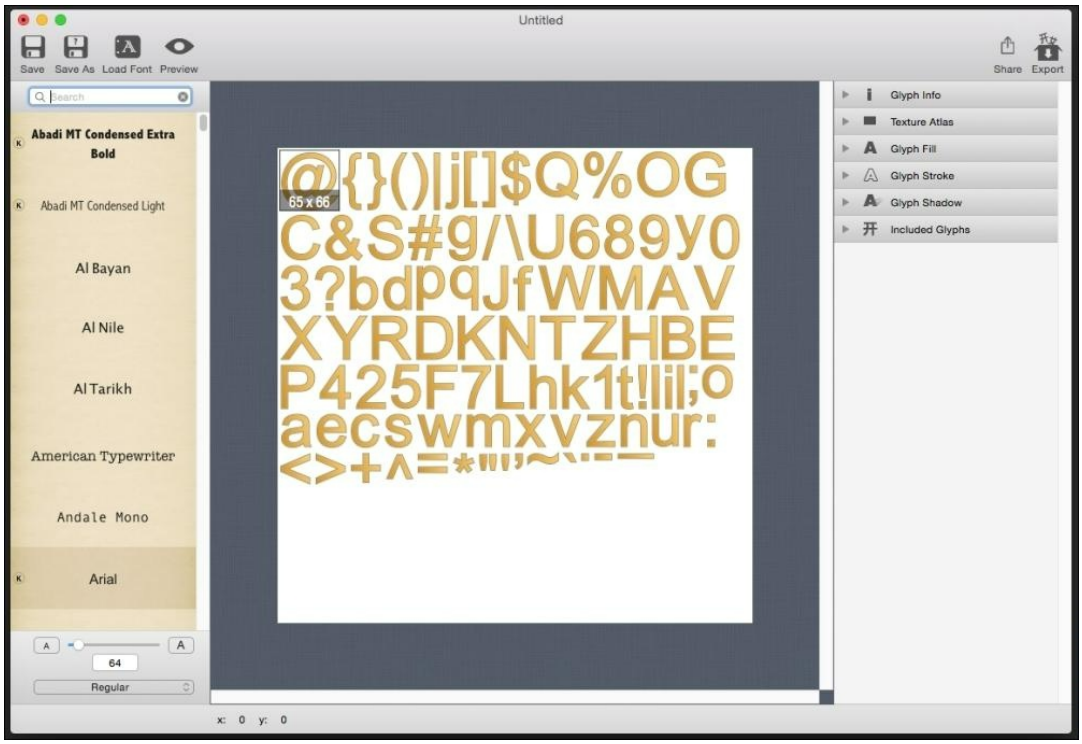


After downloading the application, you can open it and it will create a new untitled project. On the left panel, you will see a list of all the fonts that are present in the system. The center view shows the spritesheet of the file that

will be created. This is a preview window and will dynamically change according to the changes you make.

The right panel is where you will be making most of the changes after selecting the font that you want to modify on the left. On the right panel, you will find the following headings (as shown in the following screenshot):

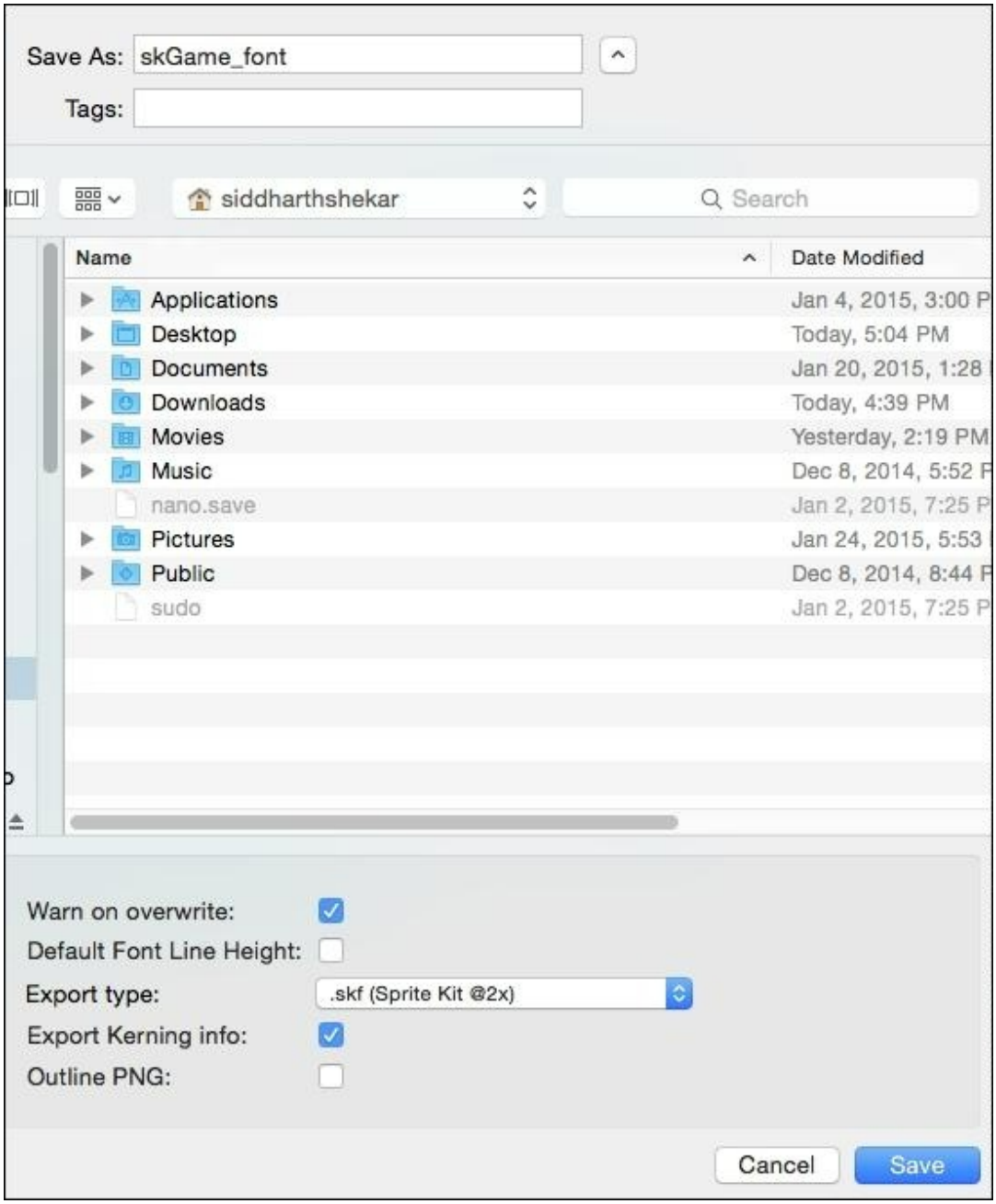
- **Glyph Info**
- **Texture Atlas**
- **Glyph Fill**
- **Glyph Stroke**
- **Glyph Shadow**
- **Included Glyphs**



You should be mostly concerned with **Glyph Fill**, **Stroke**, and **Shadow**:

- **Glyph Fill**: We can select the type of fill, that is **Solid**, **Gradient**, or **Image**. So, basically we can select the color of the font here.
- **Glyph Stroke**: This will create a new stroke effect around the letter. You can select the color and the size of the stroke here.
- **Glyph Shadow**: Here we can select the color and direction of the shadows. There are two types of shadows: inner and outer. This effect will give the letters a bit of depth.

Once you are satisfied with the changes, click on the **Export** button and select the format. You can select the `skf @2` version or the normal `skf` version for generating the files for **2x** and **1x** resolutions.



Implementing a Bitmap font

Make sure you select `.skf` in the export type. This will create a folder with `.atlas` at the end with all the characters and symbols in it and a `.skf` file that is the data file associated with the font.

Drag both the `.atlas` folder and the `.skf` file into the project.

Now we are ready to implement the Bitmap font in the game:

1. For making Glyph Designer work with SpriteKit, we will need the universal static library created by 71Squared. Go to https://71squared.zendesk.com/hc/en-us/articles/200037472-Using-

Glyph-Designer-1-8-with-Sprite-Kit and download the `libSSBitmapFont.zip` file from the bottom of the page.

2. After the file is downloaded, extract it and drag the `SSBitmapFont.h` and `SSBitmapFontLableNode.h` files into the project. Don't drag the folder containing the files, but just the individuals files themselves. Also, drag the `libSSBitmapFont.a` file into the project.
3. Go to the bridged header file we created earlier and import the two header `SSBitmapFont.h` and `SSBitmapFontLabelNode.h` files in the bridged header file as follows:

```
#ifndef skGame_skGame_Bridging_Header_h
#define skGame_skGame_Bridging_Header_h

#endif

#import "SSBitmapFont.h"
#import "SSBitmapFontlabelNode.h"
```

Now we can access the files anywhere in our Swift project.

4. To make sure everything works correctly, just build the project to make sure that there aren't any errors popping up.
5. In the `MainMenuScene` class, we will use the Bitmap font to display the name of the game instead of `SKLabelNode`. Open the `MainMenuScene` class. At the top of the class, add the following code. This is similar to what we did to get the `bgMusic.mp3` file in the `GameViewController` class. But here we will get the SKF file from the main bundle location of the project.

```
var bmFontFile = SSBitmapFont(file:
NSBundle.mainBundle().URLForResource("skGame_font",
withExtension: "skf"), error: nil)
```

6. Here we create a new variable called `bmFontFile`, and using the `SSBitmapFont` class we imported from Objective-C, we get the name of the `skGame_font` font file from the project location. Along with the name, we also need to provide the extension of the `skf` file.
7. Since we have the `bmFont` file saved, we can use this file to create new text or a label by passing in the text and assigning the position, and adding it to the scene. So, we replace the code of `SKLabelNode` that we had added previously with the following code to see the Bitmap font in action:

```
let bmFontText =
bmFontFile.nodeFromString("Ms.TinyBazooka")
bmFontText.position  = CGPoint(x:
viewSize.width/2,
   y: viewSize.height * 0.8)
addChild(bmFontText)
```

8. We create a new constant called `bmFontText` and use the `nodeFromString` property of `bmFontFile` to assign the `Ms.TinyBazooka` text to it. We set the position as usual and add the `bmFontText` to the scene:



You can immediately see that the Bitmap font is a lot sharper than the text created with `SKLableNode`. So, it is no surprise that these days, games made by both professional companies both independent developers use Bitmap fonts instead.

Skeletal animation

In an earlier chapter, we saw how to make animations in the game using frame-based animation, in which we imported a series of images and created an animation by cycling through the frames. Although frame-based animations are good, they can be tedious to make. The artist has to draw each frame and you can't have too many frames if you want to keep the bundle size low. As a result, the animations don't look very fluid. Moreover, if you want to make some changes to the character, then it is back to the drawing board for the artist, as he has to go through all the frames of the animations and redo them.

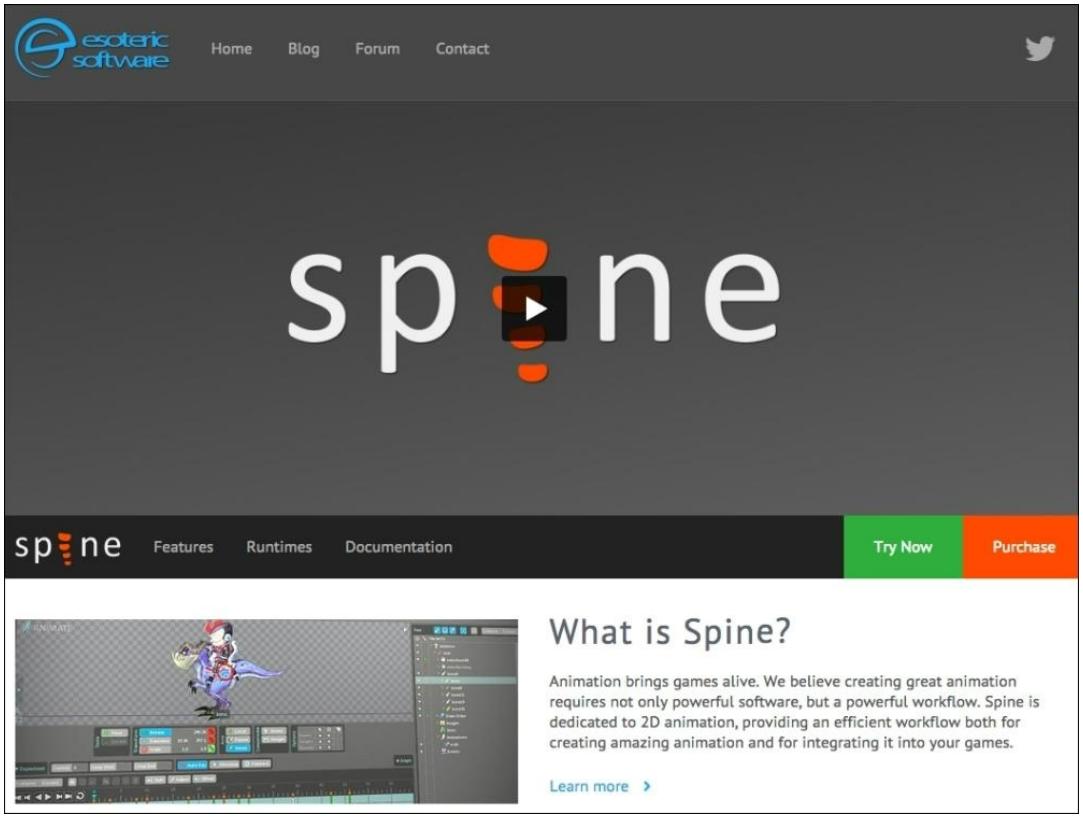
Using the skeletal animation technique instead of making the individual frames for animation, we import individual parts of the character and make a spritesheet, as shown in the following:



Using an application, we position the parts of the character and then create an animation. This way we can create different animations such as walking, running, jumping, attacking, and so on, from the body parts. All the animations are exported as a data file.

When the data file and character parts are brought into the game, the data file will be referred to place parts of the character to form the posture of the character. Later, when we call an animation to play on the character, the data file will be referred to again to create the movement dynamically. Lets us see how to create skeletal animations.

For creating the animations, we will be using an application called **Spine** by Esoteric Software. You can download the trial version from their website at http://esotericsoftware.com/.



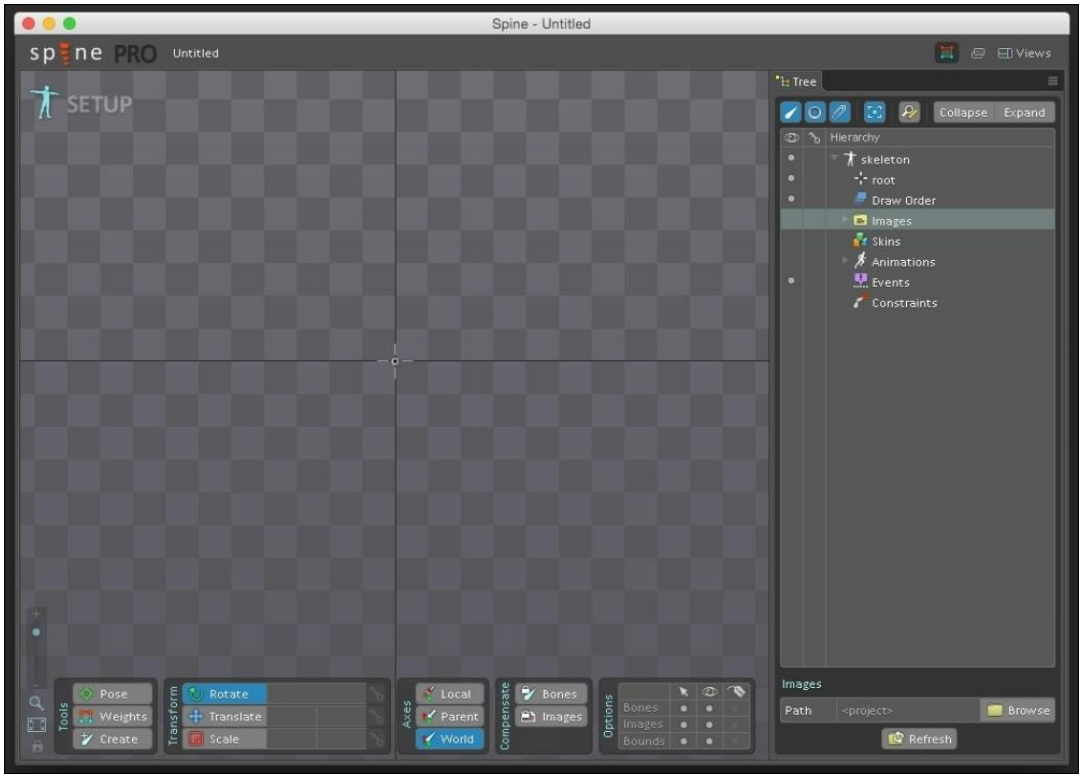
The trial version won't allow you to save the project but I have included the project file with the resources for this chapter so that you can open it in the trial version and play around with it.

After downloading the DMG file from the site, double-click on it to install it. Once the installation is complete, click on the spine logo on the top left side of the screen to create a new project, as shown in the following screenshot.

Next we have to bring in the parts of the character so that we can pose the character properly for animation. This phase is called the character setup.

In the resources for this chapter in the spine folder, you will find a folder called `heroParts`, copy this folder onto the desktop.

When you open Spine, the default project will be loaded. For creating a new project, click on the Spine icon on the top left and then select **New Project**.

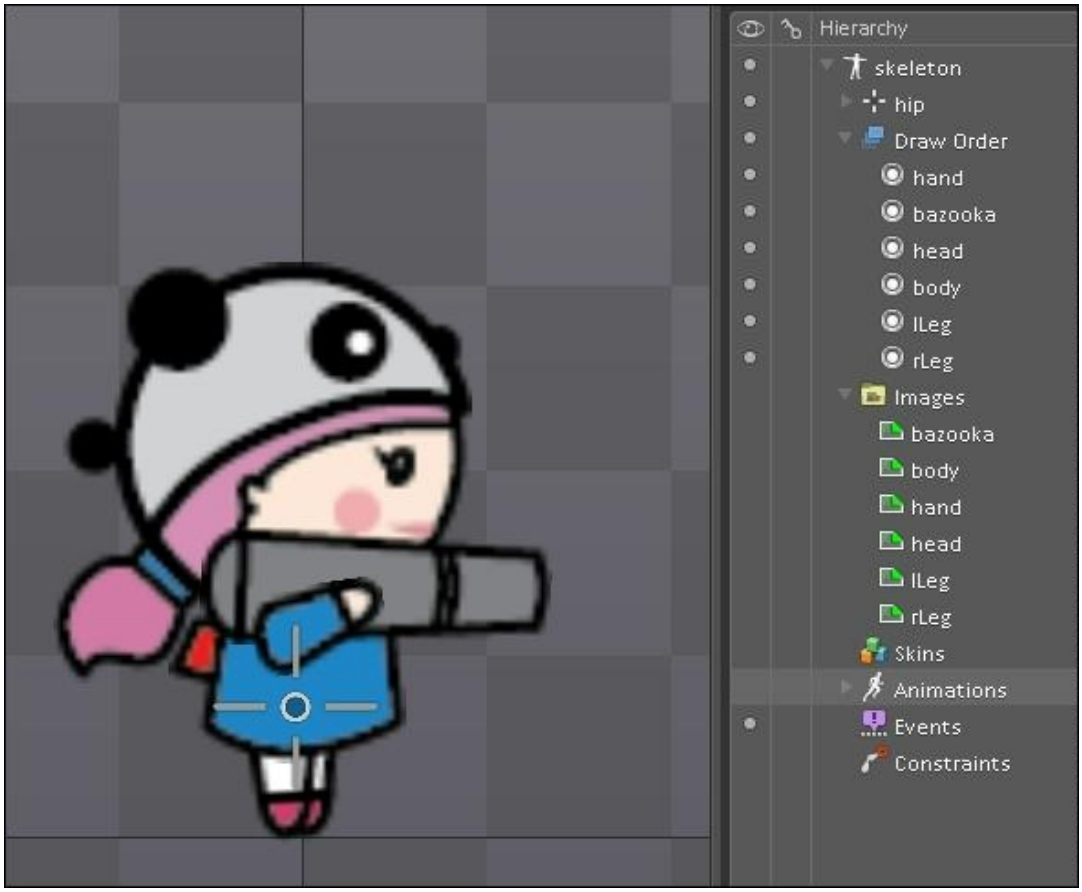


On the **Tree** panel on the right-hand side, in the **Hierarchy** panel, select the **Images** folder, as in the preceding screenshot. At the bottom you will now be able to select the path to the `heroParts` folder. Click on **Browse** and point it to the other `heroParts` folder on the desktop.

All the parts will appear under the **Images** tab. Next, drag all the parts onto the checked view panel. You can click on the individual parts and use the rotate and translate button in the **Transform** panel on the bottom to place the parts of the hero. If some parts need to be in front of others, then click on the triangle next to **Draw Order** in the **Hierarchy** panel and drag an image to make it appear above or below any other object.

The small plus at the center of the view panel is the root node. Move this node to the middle of the character by locking the images by clicking on the **Images** button in the **Compensate** panel on the bottom of the screen. You can rename the root node by clicking on it. Rename it to `hip` for the sake of convenience.

Once the character is set up properly, it should look like the following image. Refer to the following screenshot for checking the **Draw Order**:



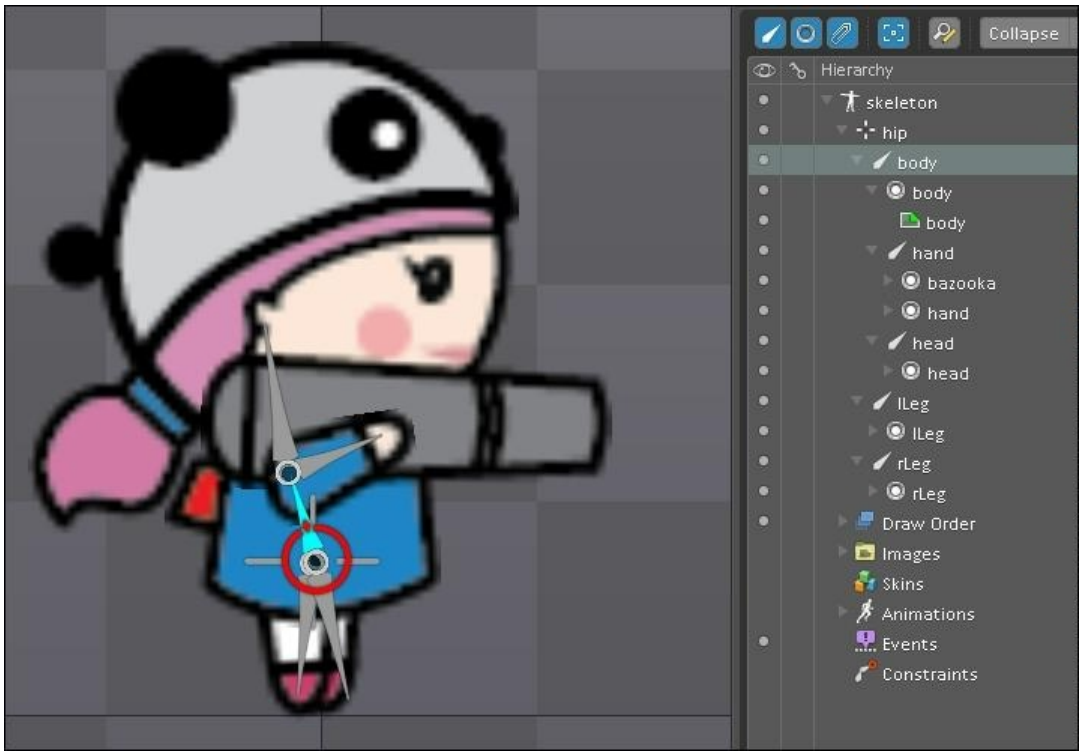
Next we will draw the bones. Bones work very similarly to human bones. You can attach one or more parts of the character to a bone, and then when you move or rotate the bone, the character part will move or rotate accordingly.

First, we will create bones for the legs. To create bones, click on the **Create** button with the bones icon on it on the **Tools** panel on the bottom of the screen. Now we will create a bone from the hip to the left foot. Left click on **hip** to start creating a bone. While still holding the left mouse button, move the mouse toward the left-foot image. Press the *Shift* key on the keyboard while over the left foot. Once the left foot is highlighted, release the mouse button and *Shift* key.

Now do the process again for the other foot. Start from the hip and press the *Shift* key, and when the other foot is highlighted, release the mouse and *Shift* key. It is OK if the bone is not perfectly aligned with the foot.

Again, from the hip, create a bone to the body so that it is closer to the start of the hand, but make sure the body is highlighted and not the hand. Name this bone `body` in the **Hierarchy** panel. Now, from the end of this newly created bone, create one bone for the hand and another for the head of the character.

You will notice that the bazooka is still attached to the hip or the root bone. In the **Hierarchy** panel, move the bazooka node under the hand bone. This way, the bazooka will move along with the hand bone. The following screenshot shows the bone hierarchy after completion:



Now you can click on the individual bones, rotate them, and when you rotate the bone, the body part also rotates along with it. Here is the character in different poses.

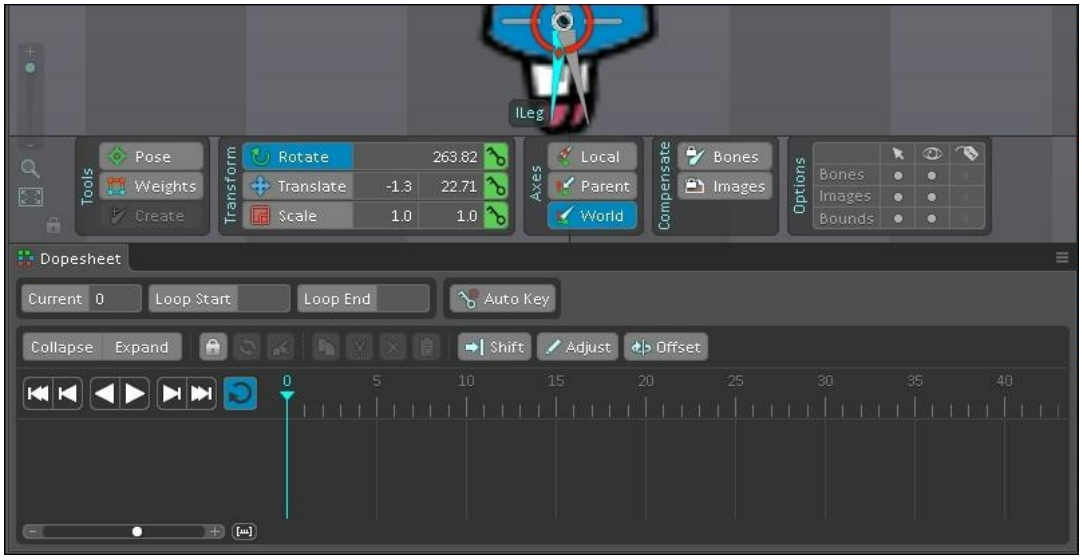
Note

Make sure the rotate button is clicked on in the **Transpose** panel and the images are not locked.



Now, the character is setup and ready to be animated. Click on the **Setup** button on the **View** panel. You are now in **Animation** mode. A new panel will open up below called **DopeSheet**. Click on the **Animations** heading under the **Hierarchy** panel and click on **NewAnimations** at the bottom of the screen. A new window will open, asking for a name, enter `walk` here and Click on **OK**.

Apart from the new **Dopesheet** panel, you will see that the **Transformation** panel has also changed a bit. If you click on any bone, a green key symbol appears next to the **Rotate**, **Translate**, and **Scale** buttons, as shown in the following screenshot:



These keys record the changes made in the **Rotation**, **Translation**, and **Scale** buttons of that bone. Green signifies that no changes have been recorded. Click on the three green buttons in the **Transform** panel at the 0th frame. Now, the values have been recorded as the keys have changed from green to red. Now, move the time line slider in blue from the 0th frame in the dopesheet to the 5th frame. Again the keys are green. Rotate the leg back a little bit and once again click on all three keys to make them red.

If you move the timeline slider between the 0th and 5th frames, you will see the leg rotating back and forth. Click on the play button to see the animation in action. Click on the loop button to the right end of the play button to loop the animation. This is how an animation is created.

We will create a simple walk cycle animation that we will later import in the game and play the animation. Undo all the actions to go back to the original pose.

For creating the walk cycle, we record the position, rotation, and scale of all the bones in the 0th, 6th, 12th, 18th, and 24th frames. The pose on the 0th and 24th frames is the same. The pose in the 12th frame is the opposite of

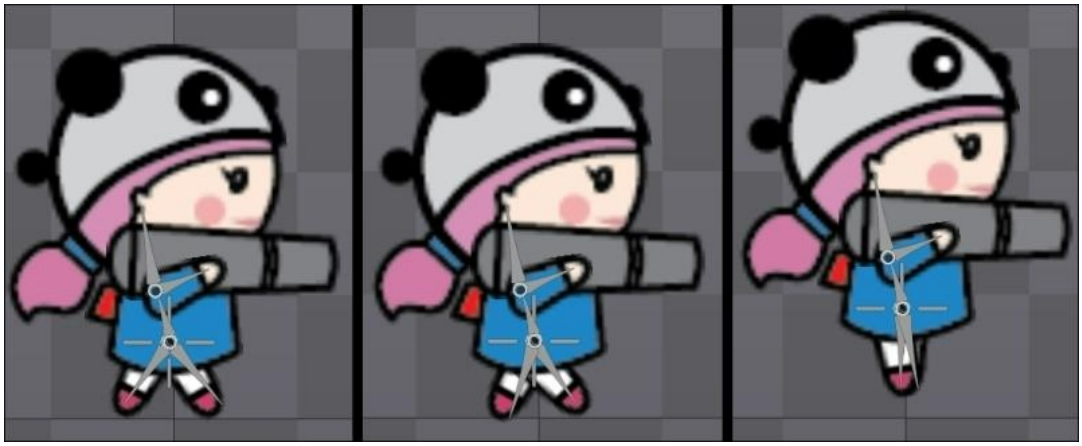
that in the 0th frame, as the position of the feet will be interchanged, meaning the foot that was previously at the back in the 0th frame will be at the front and vice versa. At the 6th and 18th frames, the feet will be brought back together and the character will be raised by moving the hip/root.

The following screenshot shows the poses at different frames. Starting from the left, the first pose is for the 0th and 24th frames. So, rotate the leg bones apart, and select all the bones and create a key by clicking on all three green key buttons. Keeping the same pose, move the time slider to the 24th frame and click on the key buttons again.

The middle image shows the pose for the 12th frame where the legs are switched. So, once again, rotate the legs and move the time slider to the 12th frame and click on the key buttons.

The image on the right shows the pose for the 6th and 18th frames. Here, move the legs closer and raise the character by moving the hip bone up. Move the time slider to the 6th frame and create a key frame. To create the key frame for the 18th frame, no changes are made to the pose. Just move the time slider to the 18th frame and create a new key frame. That's it; your walk cycle is ready. Click on the play button and enjoy!

Make sure all the bones were selected when clicking on the green button to record the frame.



Now, if you have the essential or pro version of Spine, you can export the data file by clicking on the Spine icon on the top left and then clicking on **Export**.

The exported data type will be of type `.json`. Select the location where you want the data file to be exported and leave the other values as default. Click on the **Export** button to export the JSON data file.

When you export the file, you will notice that the data file is named skeleton. Rename the file to `player` manually, as every time the file is named

skeleton by default and we don't want this file to be overwritten while creating JSON files for other characters.

To create the atlas for the images for the spine animation, create a folder called `player.atlas` and copy all the character parts from the `heroParts` folder into it.

Now drag the `player.atlas` and `player.json` files into the project.

For animating the character, we require the Spine runtime. Similar to Glyph Designer, it is written in Objective-C, but as we did earlier, we will import the header file in the bridging header file and make the Objective-C classes accessible in Swift.

To get the header files, go to https://github.com/mredig/SGG_SKSpineImport, and download the ZIP file and extract it. From the extracted folder, go to the `SpineImporter` folder and drag all the files in the folder to the Swift project.

In the `Bridging Header` file, add the `SpineImport.h` file as follows:

```
#import "SSBitmapFont.h"
#import "SSBitmapFontlabelNode.h"
#import "SpineImport.h"
```

In the `MainMenuScene` class, we will add the player animation. At the top of the class, create a global variable hero and assign the `SGG_Spine` class to it as follows:

```
  var hero = SGG_Spine()
```

Now, add the following code right after where we added `BG1` and `BG2` to the scene:

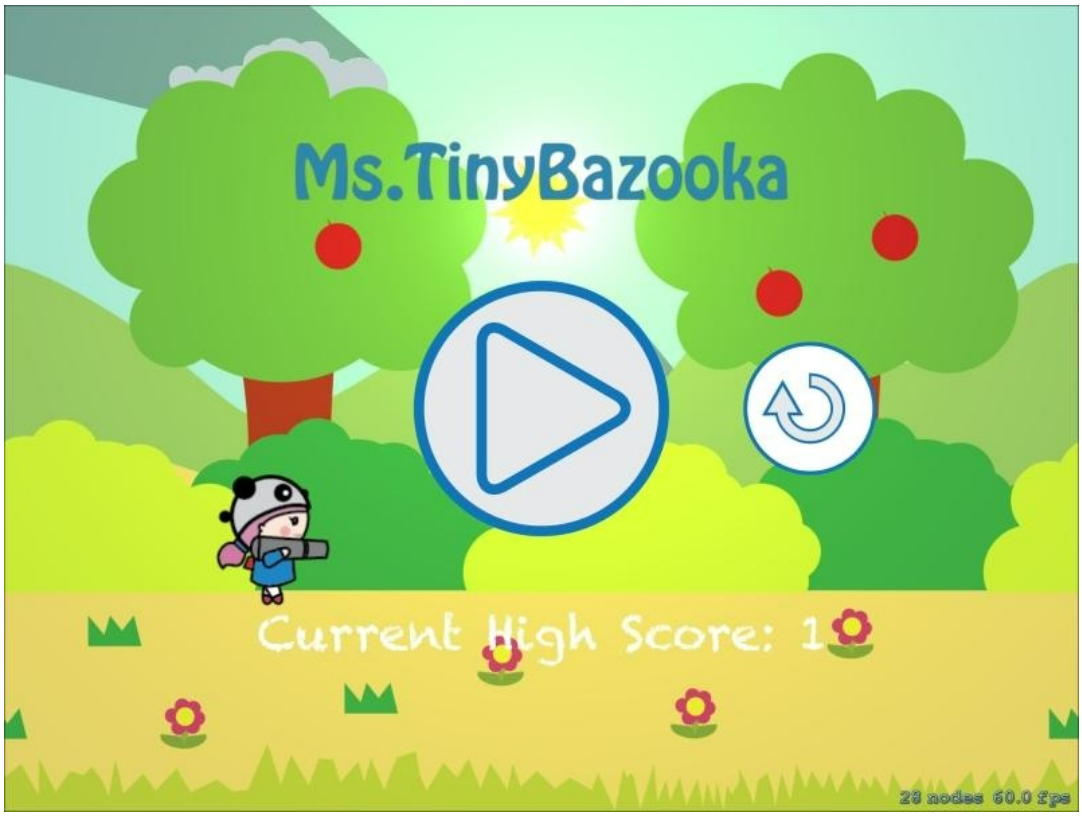
```
hero.skeletonFromFileNamed("player",
                           andAtlasNamed:
"player",
                           andUseSkinNamed: nil)

hero.position = CGPoint(x: viewSize.width/4, y:
viewSize.height * 0.25)
hero.xScale = 1.25; hero.yScale = 1.25;
hero.runAnimation("walk", andCount: -1)
addChild(hero)
```

We provide the name of the JSON data file and the name of the atlas in the `skeletonFromFile` property of the hero. Since we didn't use any skins in the game, the third parameter is kept `nill`.

We, then, position the hero variable and increase the scale a bit.

To tell which animation to start playing, we use the `runAnimation` property of the hero and assign the walk animation we created in spine.

Finally, we add the hero to the scene. Build and run the game to see the final result, as shown in the following screenshot:



Summary

In this chapter, you saw how to add lighting and shadows to our game without much effort. Since Apple included it with SpriteKit, you can be sure the code to create the effect is well optimized. In other frameworks, this effect has to be written by a developer, and the developer needs to have good experience to make an optimized lighting and shadow effect.

We also had a brief introduction to SpriteKit's physics engine and replaced our homemade physics engine with it. Here we have barely even scratched the surface of the possibilities with the physics engine. With good knowledge and experience, we can make our own Angry Birds clone.

Apart from SpriteKit's Lighting and Physics engines, we also saw how to bring Objective-C code into Swift and make use of it to implement tools such as Glyph Designer and Spine. Both Glyph Designer and Spine are professional tools that are an absolute must for game developers and designers. They really help in optimizing and simplifying the game development process by a lot.

It is time to say goodbye to Ms. TinyBazooka, for, in the next two chapters, we will be entering the world of 3D game development. But we will return to SpriteKit in Chapter 10, *Publishing and Distribution*, where we will see how to publish this game to the App Store.

Chapter 8. SceneKit

So, this is it! Finally, we move from the 2D world to 3D. With SceneKit, we can make 3D games quite easily, especially since the syntax for SceneKit is quite similar to SpriteKit.

When we say 3D games, we don't mean that you get to put on your 3D glasses to make the game. In 2D games, we mostly work in the x and y coordinates. In 3D games, we deal with all three axes *x*, *y*, and *z*.

Additionally, in 3D games, we have different types of lights that we can use. Also, SceneKit has an inbuilt physics engine that will take care of forces such as gravity and will also aid collision detection.

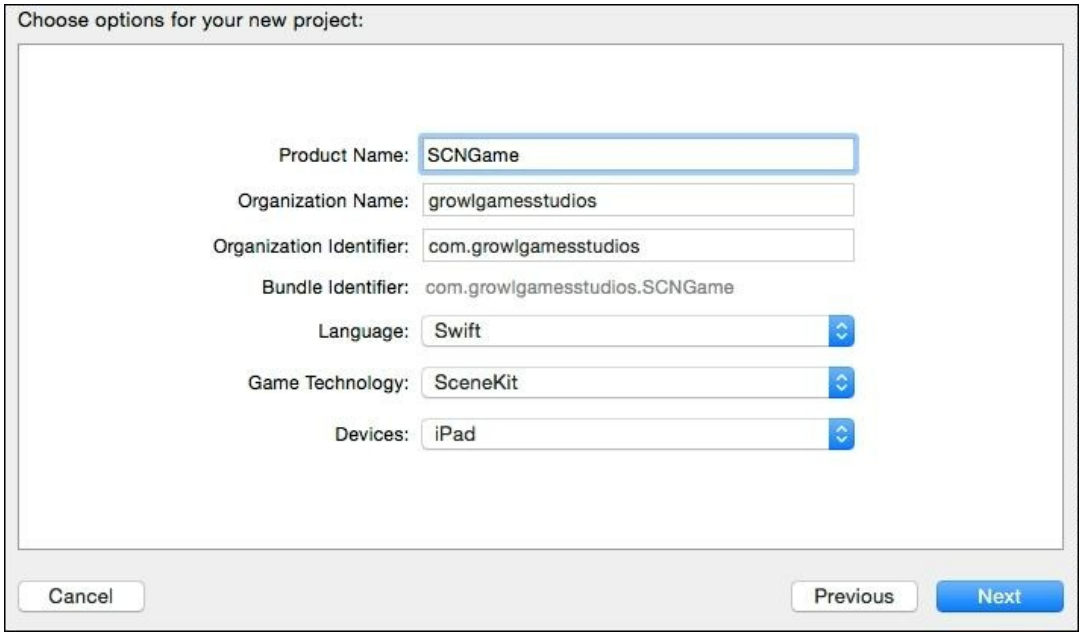
We can also use SpriteKit in SceneKit for GUI and buttons so that we can add scores and interactivity to the game. So, there is a lot to cover in this chapter. Let's get started.

The topics covered in this chapter are as follows:

- Creating a scene with SCNScene
- Adding objects to a scene
- Importing scenes from external 3D applications
- Adding physics to the scene
- Adding an enemy
- Checking collision detection
- Adding a SpriteKit overlay
- Adding touch interactivity
- Finishing the gameloop
- Adding wall and floor parallax
- Adding particles

Creating a scene with SCNScene

First, we create a new SceneKit project. It is very similar to creating other projects. Only this time, make sure you select SceneKit from the **Game Technology** drop-down list. Don't forget to select **Swift** for the language field. Choose **iPad** as the device and click on **Next** to create the project in the selected directory, as shown in the following screenshot:



Once the project is created, open it. Click on the `GameViewController` class, and delete all the contents in the `viewDidLoad` function, delete the `handleTap` function, as we will be creating a separate class, and add touch behavior.

Create a new class called `GameSCNScene` and import the following headers. Inherit from the `SCNScene` class and add an `init` function that takes in a parameter called view of type `SCNView`:

```
import Foundation
import UIKit
import SceneKit

class GameSCNScene: SCNScene{

    let scnView: SCNView!
    let _size:CGSize!
    var scene: SCNScene!

        required init(coder aDecoder: NSCoder) {
        fatalError("init(coder:) has not been
implemented")
    }

    init(currentview view: SCNView) {

        super.init()
    }
}
```

Also, create two new constants `scnView` and `_size` of type `SCNView` and `CGSize`, respectively. Also, add a variable called `scene` of type `SCNScene`.

Since we are making a SceneKit game, we have to get the current view, which is the type `SCNView`, similar to how we got the view in SpriteKit where we typecasted the current view in SpriteKit to `SKView`.

We create a `_size` constant to get the current size of the view. We then create a new variable `scene` of type `SCNScene`. `SCNScene` is the class used to make scenes in SceneKit, similar to how we would use `SKScene` to create scenes in SpriteKit.

> **Tip**
>
> Swift would automatically ask to create the `required init` function, so we might as well include it in the class.

Now, move to the `GameViewController` class and create a global variable called `gameSCNScene` of type `GameSCNScene` and assign it in the `viewDidLoad` function, as follows:

```
class GameViewController: UIViewController {

  var gameSCNScene:GameSCNScene!

    override func viewDidLoad() {
      super.viewDidLoad()
      let scnView = view as SCNView
      gameSCNScene = GameSCNScene(currentview:
scnView)
    }
}// UIViewController Class
```

Great! Now we can add objects in the `GameSCNScene` class. It is better to move all the code to a single class so that we can keep the `GameSceneController` class clean.

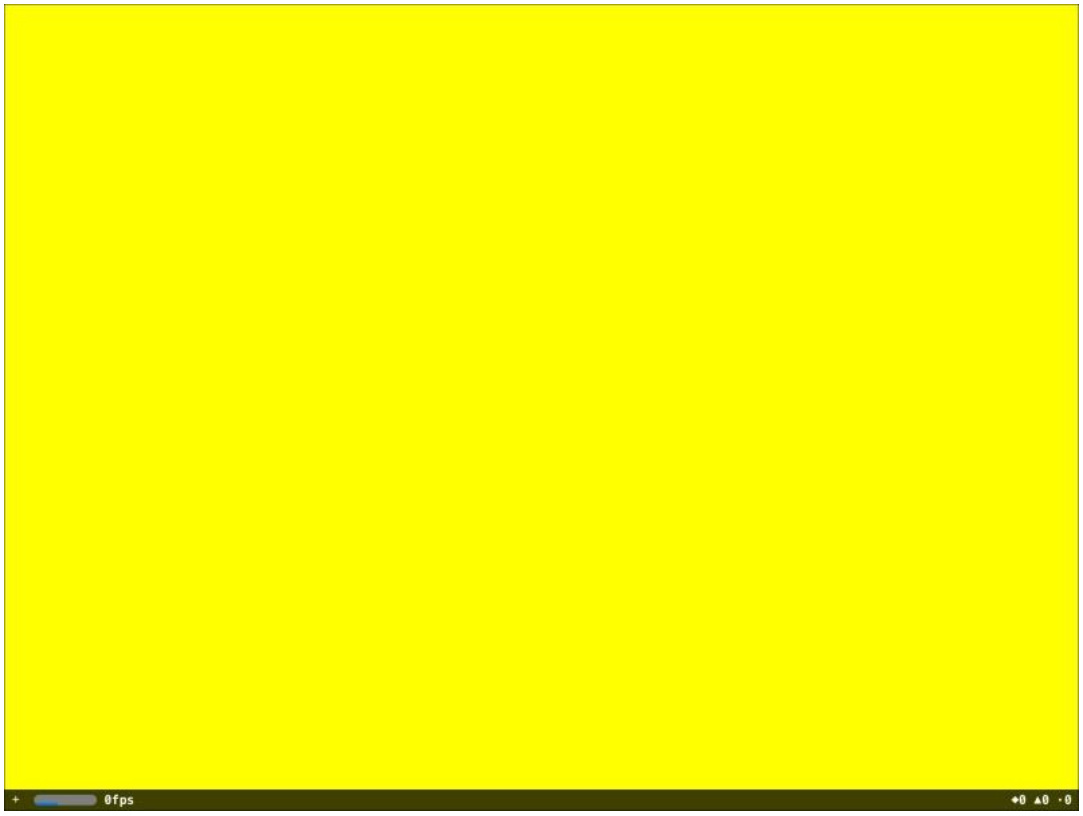
In the `init` function of `GameSCNScene`, add the following after the `super.init` function:

```
scnView = view
_size = scnView.bounds.size

// retrieve the SCNView
scene = SCNScene()
scnView.scene = scene
scnView.allowsCameraControl = true
scnView.showsStatistics = true
scnView.backgroundColor = UIColor.yellowColor()
```

Here, we first assign the current view to the `scnView` constant. Next, we set the `_size` constant to the dimensions of the current view.

Next we initialize the scene variable. Then, assign the scene to the scene of `scnView`. Next, enable `allowCameraControls` and `showStatistics`. This will enable us to control the camera and move it around to have a better look at the scene. Also, with statistics enabled, we will see the performance of the game to make sure that the FPS is maintained.

The `backgroundColor` property of `scnView` enables us to set the color of the view. I have set it to yellow so that objects are easily visible in the scene, as shown in the following screenshot. With all this set we can run the scene.



Well, it is not all that awesome yet. One thing to notice is that we have still not added a camera or a light, but we still see the yellow scene. This is because while we have not added anything to the scene yet, SceneKit automatically provides a default light and camera for the scene created.

Adding objects to the scene

Let us next add geometry to the scene. We can create some basic geometry such as spheres, boxes, cones, tori, and so on in SceneKit with ease. Let us create a sphere first and add it to the scene.

Adding a sphere to the scene

Create a function called `addGeometryNode` in the class and add the following code in it:

```
func addGeometryNode(){

   let sphereGeometry = SCNSphere(radius: 1.0)
   sphereGeometry.firstMaterial?.diffuse.contents
= UIColor.orangeColor()

   let sphereNode = SCNNode(geometry:
sphereGeometry)
   sphereNode.position = SCNVector3Make(0.0, 0.0,
0.0)
   scene.rootNode.addChildNode(sphereNode)
}
```

For creating geometry, we use the `SCNSphere` class to create a sphere shape. We can also call `SCNBox`, `SCNCone`, `SCNTorus`, and so on to create box, cone, or torus shapes respectively.

While creating the sphere, we have to provide the radius as a parameter, which will determine the size of the sphere. Although to place the shape, we have to attach it to a node so that we can place and add it to the scene.

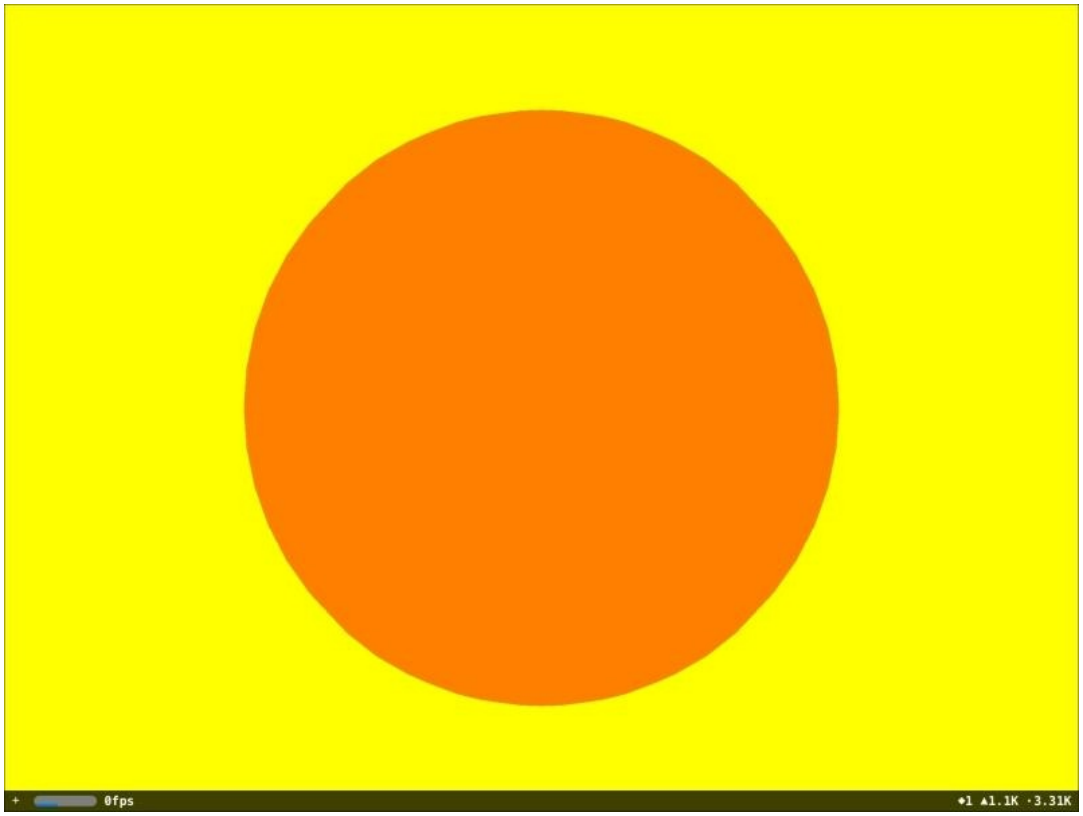
So, create a new constant called `sphereNode` of type `SCNNode` and pass in the sphere geometry as a parameter. For positioning the node, we have to use the `SCNvector3Make` property to place our object in 3D space by providing the values for `x`, `y`, and `z`.

Finally, to add the node to the scene, we have to call `scene.rootNode` to add the `sphereNode` to scene, unlike SpriteKit where we would simply use `addChild` to add objects to the scene.

With the sphere added, let us run the scene. Don't forget to add `self.addGeometryNode()` in the `init` function.

We did add a sphere, so why are we getting a circle (shown in the following screenshot)? Well, the basic light source used by SceneKit just enables to

us to see objects in the scene. If we want to see the actual sphere, we have to improve the light source of the scene.



Adding light sources

Let us create a new function called `addLightSourceNode` as follows so that we can add custom lights to our scene:

```
func addLightSourceNode(){

    let lightNode = SCNNode()
    lightNode.light = SCNLight()
    lightNode.light!.type = SCNLightTypeOmni
    lightNode.position = SCNVector3(x: 10, y: 10,
z: 10)
    scene.rootNode.addChildNode(lightNode)

    let ambientLightNode = SCNNode()
    ambientLightNode.light = SCNLight()
    ambientLightNode.light!.type =
SCNLightTypeAmbient
    ambientLightNode.light!.color =
UIColor.darkGrayColor()
    scene.rootNode.addChildNode(ambientLightNode)
}
```

We can add some light sources to see some depth in our sphere object. Here we add two types of light source. The first is an omni light. Omni lights start at a point and then the light is scattered equally in all directions. We also add an ambient light source. An ambient light is the light that is reflected by other objects, such as moonlight.

> **Note**
>
> There are two more types of light sources: directional and spotlight. Spotlight is easy to understand, and we usually use it if a certain object needs to be brought to attention like a singer on a stage. Directional lights are used if you want light to go in a single direction, such as sunlight. The Sun is so far from the Earth that the light rays are almost parallel to each other when we see them.

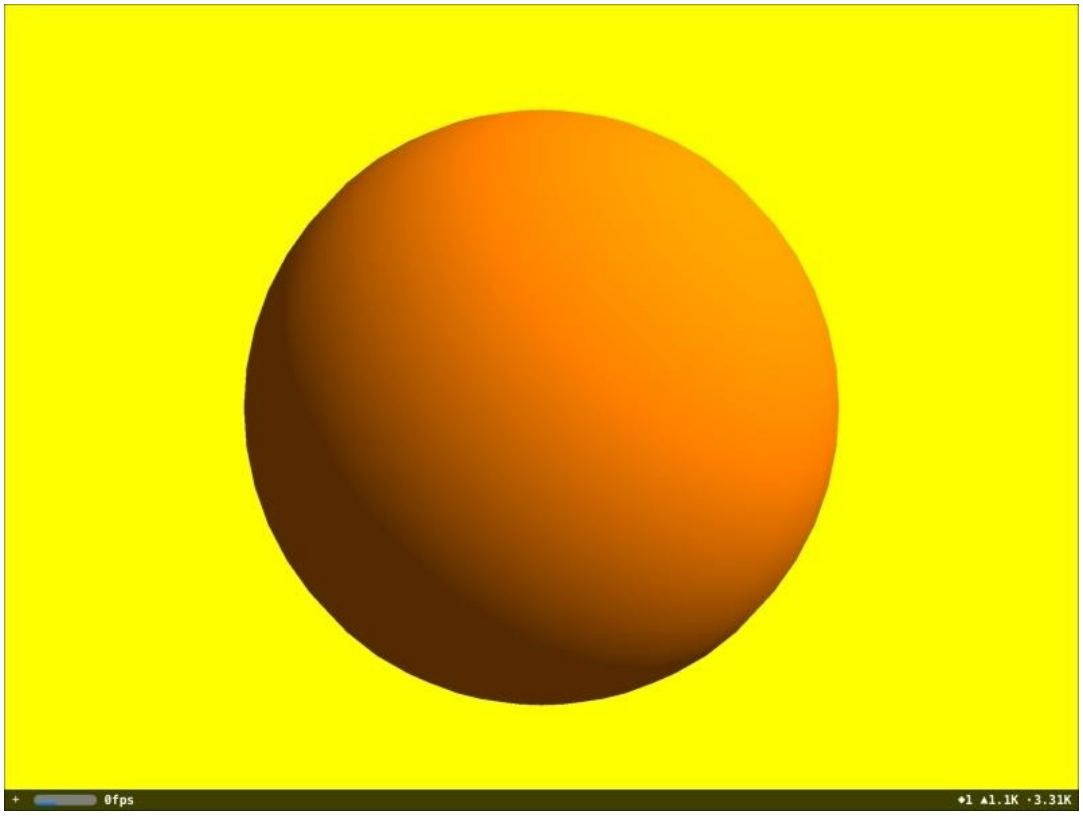
For creating a light source, we create a node called `lightNode` of type `SCNNode`. We then assign `SCNLight` to the light property of `lightNode`. We assign the omni light type to be the type of the light. We assign position of the light source to be at 10 in all three *x*, *y*, and *z* coordinates. Then, we add it to the rootnode of the scene.

Next we add an ambient light to the scene. The first two steps of the process are the same as for creating any light source:

1. For the type of light we have to assign `SCNLightTypeAmbient` to assign an ambient type light source. Since we don't want the light source to be very strong, as it is reflected, we assign a `darkGrayColor` to the color.
2. Finally, we add the light source to the scene.

There is no need to add the ambient light source to the scene but it will make the scene have softer shadows. You can remove the ambient light source to see the difference.

Call the `addLightSourceNode` function in the `init` function. Now, build and run the scene to see an actual sphere with proper lighting, as shown in the following screenshot:



You can place a finger on the screen and move it to rotate the cameras as we have enabled camera control. You can use two fingers to pan the camera and you can double tap to reset the camera to its original position and direction.

Adding a camera to the scene

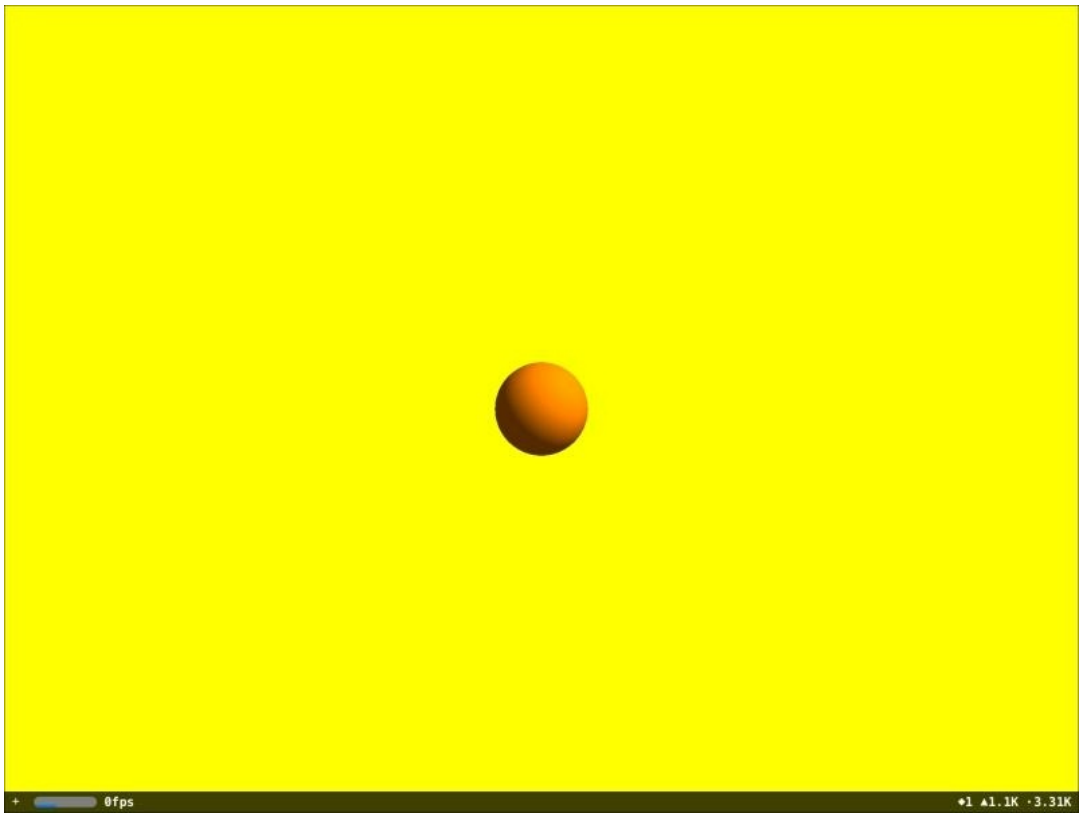
Next let us add a camera to the scene, as the default camera is very close. Create a new function called `addCameraNode` to the class and add the following code in it:

```
func addCameraNode(){

 let cameraNode = SCNNode()
    cameraNode.camera = SCNCamera()
    cameraNode.position = SCNVector3(x: 0, y: 0,
z: 15)
    scene.rootNode.addChildNode(cameraNode)

}
```

Here, again we create an empty node called `cameraNode`. We assign `SCNCamera` to the camera property of `cameraNode`. Next we position the camera such that we keep the `x` and `y` values at zero and move the camera

back in the `z` direction by 15 units. Then we add the camera to the rootnode of the scene. Call the `addCameraNode` at the bottom of the `init` function.

In this scene, the origin is at the center of the scene, unlike SpriteKit where the origin of a scene is always at bottom right of the scene. Here the positive *x* and *y* are to the right and up from the center. The positive *z* direction is toward you.



We didn't move the sphere back or reduce its size here. This is purely because we brought the camera backward in the scene.

Let us next create a floor so that we can have a better understanding of the depth in the scene. Also, in this way, we will learn how to create floors in the scene.

Adding a floor

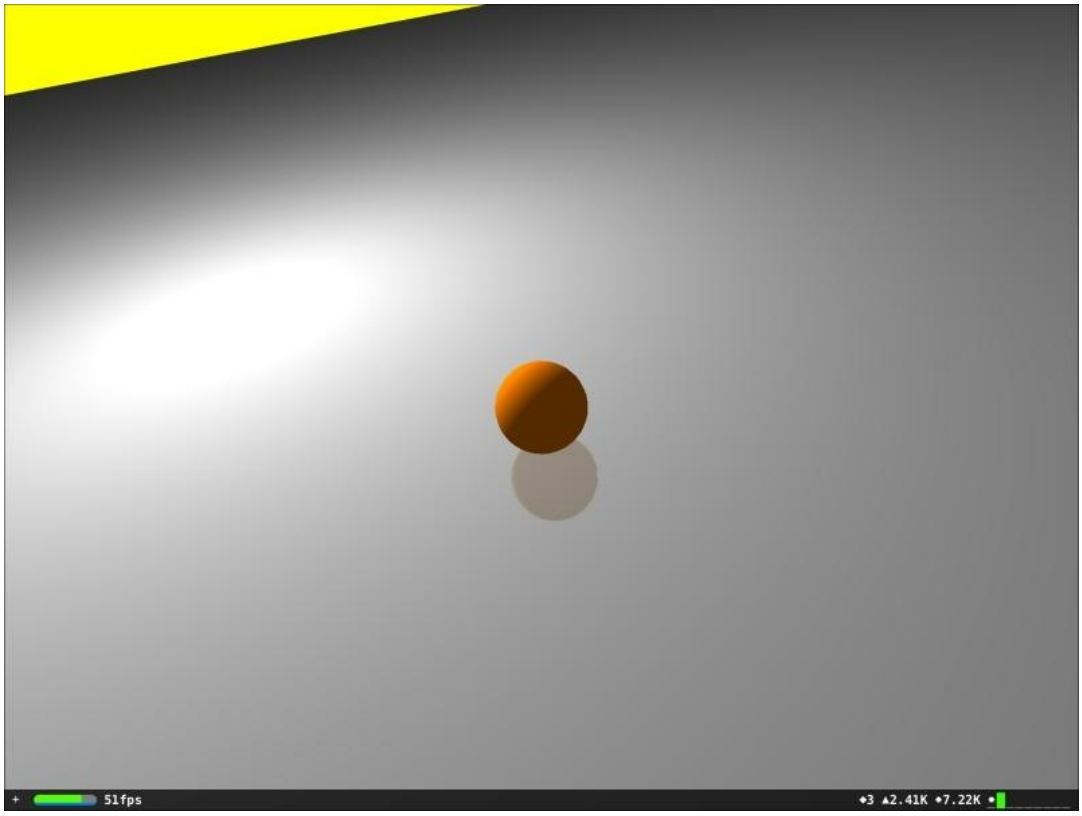
In the class, add a new function called `addFloorNode` and add the following code:

```
func addFloorNode(){

    var floorNode = SCNNode()
    floorNode.geometry = SCNFloor()
    floorNode.position.y = -1.0
    scene.rootNode.addChildNode(floorNode)
```

```
}
```

For creating a floor, we create a variable called `floorNode` of type `SCNNode`. We then assign `SCNFloor` to the geometry property of `floorNode`. For the position, we assign the `y` value to `-1` as we want the sphere to appear above the floor. At the end, as usual, we assign the `floorNode` to the root node of the scene.

In the following screenshot, I have rotated the camera to show the scene in full action. Here we can see the floor is gray in color and the sphere is casting its reflection on the floor, and we can also see the bright omni light at the top left of the sphere.

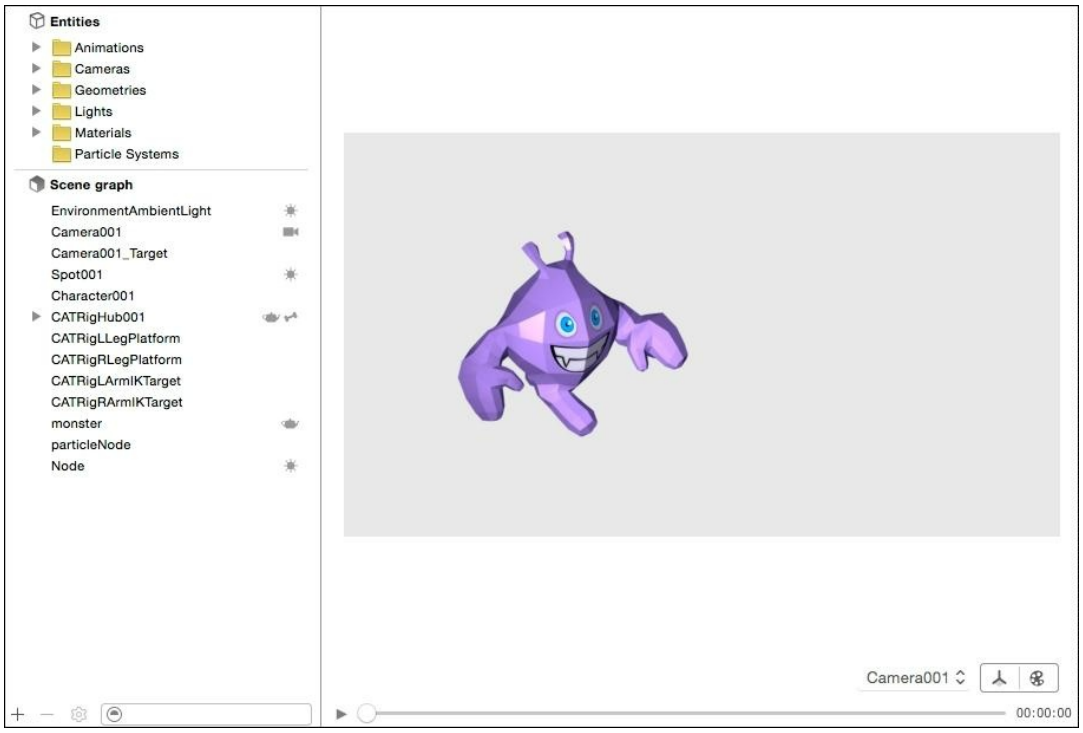


Importing scenes from external 3D applications

Although we can add objects, cameras, and lights through code, it will become very tedious and confusing when we have a lot of objects added to the scene. In SceneKit, this problem can be easily overcome by importing scenes prebuilt in other 3D applications.

All 3D applications such as 3D StudioMax, Maya, Cheetah 3D, and Blender have the ability to export scenes in Collada (`.dae`) and Alembic (`.abc`) format. We can import these scenes with lighting, camera, and textured objects into SceneKit directly, without the need for setting up the scene.

In this section, we will import a Collada file into the scene. In the resources folder for this chapter, you will find the `monsterScene.DAE` file. Drag this file into the current project.



Along with the DAE file, also add the `monster.png` file to the project, otherwise you will see only the untextured monster mesh in the scene.

Click on the `monsterScene.DAE` file. If the textured monster is not automatically loaded, drag the `monster.png` file from the project into the monster mesh in the preview window. Release the mouse button once you see a (**+**) sign while over the monster mesh. Now you will be able to see the monster properly textured.

The panel on the left shows the entities in the scene. Below the entities, the scene graph is shown and the view on the right is the preview pane.

Entities show all the objects in the scene and the scene graph shows the relation between these entities. If you have certain objects that are children to other objects, the scene graph will show them as a tree. For example, if you open the triangle next to **CATRigHub001**, you will see all the child objects under it.

You can use the scene graph to move and rotate objects in the scene to fine-tune your scene. You can also add nodes, which can be accessed by code. You can see that we already have a camera and a spotlight in the scene. You can select each object and move it around using the arrow at the pivot point of the object.

You can also rotate the scene to get a better view by clicking and dragging the left mouse button on the preview scene. For zooming, scroll your mouse wheel up and down. To pan, hold the *Alt* button on the keyboard and left-click and drag on the preview pane.

Note

One thing to note is that rotating, zooming, and panning in the preview pane won't actually move your camera. The camera is still at the same position and angle. To view from the camera, again select the **Camera001** option from the drop-down list in the preview pane and the view will reset to the camera view.

At the bottom of the preview window, we can either choose to see the view through the camera or spotlight, or click-and-drag to rotate the free camera. If you have more than one camera in your scene, then you will have **Camera002**, **Camera003**, and so on in the drop-down list.

Below the view selection dropdown in the preview panel you also have a play button. If you click on the play button, you can look at the default animation of the monster getting played in the preview window.

The preview panel is just that; it is just to aid you in having a better understanding of the objects in the scene. In no way is it a replacement for a regular 3D package such as 3DSMax, Maya, or Blender.

You can create cameras, lights, and empty nodes in the scene graph, *but you can't add geometry such as boxes and spheres*. You can add an empty node and position it in the scene graph, and then add geometry in code and attach it to the node.

Now that we have an understanding of the scene graph, let us see how we can run this scene in SceneKit.

In the `init` function, delete the line where we initialized the scene and add the following line instead. Also delete the objects, light, and camera we added earlier.

```
 init(currentview view:SCNView){

super.init()
    scnView = view
    _size = scnView.bounds.size

    //retrieve the SCNView
    //scene = SCNScene()

 scene = SCNScene(named: "monsterScene.DAE")

    scnView.scene = scene
    scnView.allowsCameraControl  = true
    scnView.showsStatistics = true
    scnView.backgroundColor =
UIColor.yellowColor()

//    self.addGeometryNode()
//    self.addLightSourceNode()
//    self.addCameraNode()
//    self.addFloorNode()
//

}
```

Build and run the game to see the following screenshot:



You will see the monster running and the yellow background that we initially assigned to the scene. While exporting the scene, if you export the animations as well, once the scene loads in SceneKit the animation starts playing automatically.

Also, you will notice that we have deleted the camera and light in the scene. So, how come the default camera and the light aren't loaded in the scene?

What is happening here is that while I exported the file, I inserted a camera in the scene and also added a spotlight. So, when we imported the file into the scene, SceneKit automatically understood that there is a camera already present, so it will use the camera as its default camera. Similarly, a spotlight is already added in the scene, which is taken as the default light source, and lighting is calculated accordingly.

Adding objects and physics to the scene

Let us now see how we can access each of the objects in the scene graph and add gravity to the monster. Later in this chapter, we will see how we can add a touch interface by which we will be able to make the hero character jump by applying an upward force.

Accessing the hero object and adding a physics body

So, create a new function called `addColladaObjects` and call an `addHero` function in it. Create a global variable called `heroNode` of type `SCNNode`. We will use this node to access the hero object in the scene. In the `addHero` function, add the following code:

```
init(currentview view:SCNView){
    super.init()
    scnView = view
    _size = scnView.bounds.size

    //retrieve the SCNView
    //scene = SCNScene()
    scene = SCNScene(named:
"monster.scnassets/monsterScene.DAE")

    scnView.scene = scene
    scnView.allowsCameraControl  = true
    scnView.showsStatistics = true
    scnView.backgroundColor =
UIColor.yellowColor()

    self.addColladaObjects()

//    self.addGeometryNode()
//    self.addLightSourceNode()
//    self.addCameraNode()
//    self.addFloorNode()

  }

func addHero(){

    heroNode = SCNNode()

 var monsterNode =
scene.rootNode.childNodeWithName(
 "CATRigHub001", recursively: false)
```

```
        heroNode.addChildNode(monsterNode!)
     heroNode.position = SCNVector3Make(0, 0, 0)


        let collisionBox = SCNBox(width: 10.0,
    height: 10.0,
                length: 10.0, chamferRadius: 0)

        heroNode.physicsBody?.physicsShape =
    SCNPhysicsShape(geometry: collisionBox, options:
    nil)

     heroNode.physicsBody =
    SCNPhysicsBody.dynamicBody()
        heroNode.physicsBody?.mass = 20
        heroNode.physicsBody?.angularVelocityFactor =
    SCNVector3Zero
     heroNode.name = "hero"

        scene.rootNode.addChildNode(heroNode)
    }
```

First, we call the `addColladaObjects` function in the `init` function, as highlighted. Then we create the `addHero` function. In it we initiate the `heroNode`. Then, to actually move the monster, we need access to the `CatRibHub001` node to move the monster. We gain access to it through the `ChildWithName` property of `scene.rootNode`. For each object that we wish to gain access to through code, we will have to use the `ChildWithName` property of the `rootNode` of the scene and pass in the name of the object.

If recursively is set to `true`, to get said object, SceneKit will go through all the child nodes to get access to the specific node. Since the node that we are looking for is right on top, we said `false` to save processing time.

We create a temporary variable called `monsterNode`. In the next step, we add the `monsterNode` variable to `heroNode`. We then set the position of the hero node to the origin.

For `heroNode` to interact with other physics bodies in the scene, we have to assign a shape to the physics body of `heroNode`. We could use the mesh of the monster, but the shape might not be calculated properly and a box is a much simpler shape than the mesh of the monster. For creating a box collider, we create a new box geometry roughly the width, height, and depth of the monster.

Then, using the `physicsBody.physicsShape` property of the `heroNode`, we assign the shape of the `collisionBox` we created for it. Since we want the body to be affected by gravity, we assign the physics body type to be dynamic. Later we will see other body types.

Since we want the body to be highly responsive to gravity, we assign a value of `20` to the `mass` of the body. In the next step, we set the `angularVelocityFactor` to `0` in all three directions, as we want the body to move straight up and down when a vertical force is applied. If we don't do this, the body will flip-flop around.

We also assign the name `hero` to the monster to check if the collided object is the hero or not. This will come in handy when we check for collision with other objects.

Finally, we add `heroNode` to the scene.

Add the `addColladaObjects` to the `init` function and comment or delete the `self.addGeometryNode`, `self.addLightSourceNode`, `self.addCameraNode`, and `self.addFloorNode` functions if you haven't already. Then, run the game to see the monster slowly falling through.

We will create a small patch of ground right underneath the monster so that it doesn't fall down.

Adding the ground

Create a new function called `addGround` and add the following:

```
func addGround(){

    let groundBox = SCNBox(width: 10, height: 2,
                           length: 10,
chamferRadius: 0)

    let groundNode = SCNNode(geometry: groundBox)

    groundNode.position = SCNVector3Make(0,
-1.01, 0)
    groundNode.physicsBody =
SCNPhysicsBody.staticBody()
    groundNode.physicsBody?.restitution = 0.0

    scene.rootNode.addChildNode(groundNode)
}
```

We create a new constant called `groundBox` of type `SCNBox`, with a width and length of `10`, and height of `2`. Chamfer is the rounding of the edges of the box. Since we didn't want any rounding of the corners, it is set to `0`.

Next we create a `SCNNode` called `groundNode` and assign `groundBox` to it. We place it slightly below the origin. Since the height of the box is `2`, we

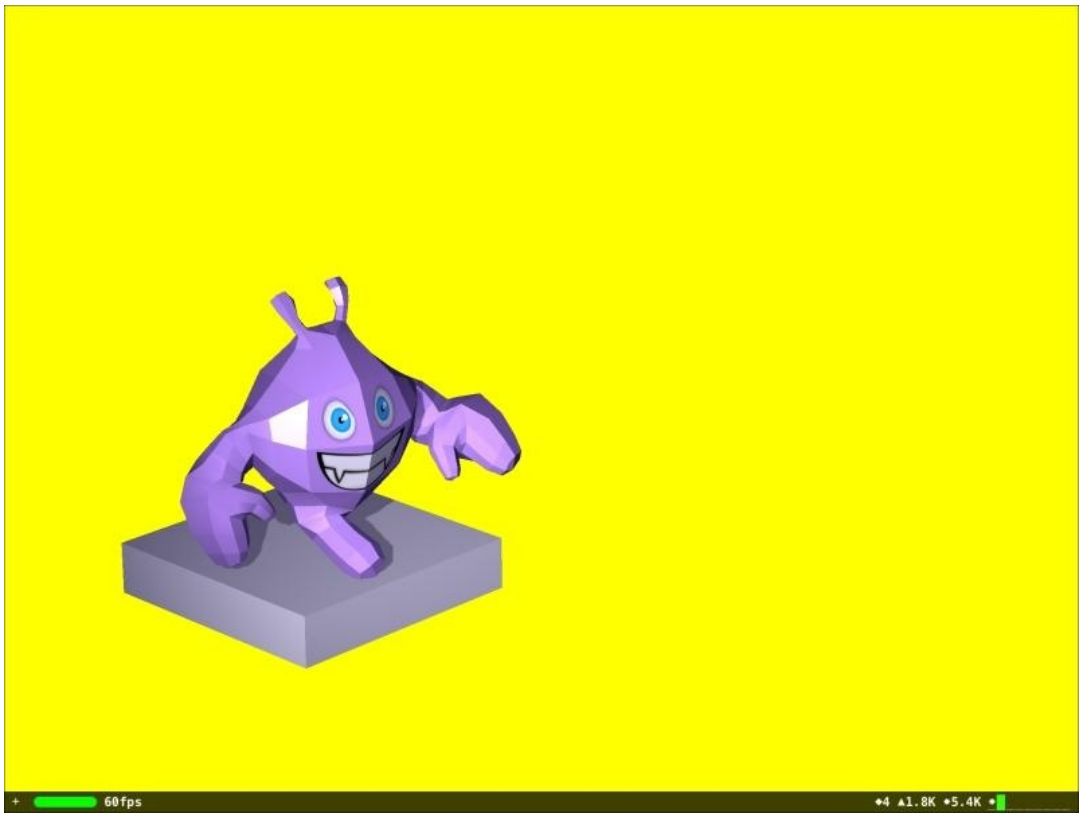
place it at `-1.01` so that `heroNode` will be `(0, 0, 0)` when the monster rests on the ground.

Next we assign the physics body of type static body. Also, since we don't want the hero to bounce off the ground when he falls on it, we set the restitution to `0`. Finally, we then add the ground to the scene's rootnode.

The reason we made this body static instead of dynamic is because a dynamic body gets affected by gravity and other forces but a static one doesn't. So, in this scene, even though gravity is acting downward, the hero will fall but `groundBox` won't as it is a static body.

You will see that the physics syntax is very similar to SpriteKit with static bodies and dynamic bodies, gravity, and so on. And once again, similar to SpriteKit, the physics simulation is automatically turned on when we run the scene.

Add the `addGround` function in the `addColladaObjects` functions and run the game to see the monster getting affected by gravity and stopping after coming in touch with the ground.



Adding an enemy node

To check collision in SceneKit, we can check for collision between the hero and the ground. But let us make it a little more interesting and also learn a new kind of body type: the kinematic body.

For this, we will create a new box called `enemy` and make it move and collide with the hero. Create a new global `SCNNode` called `enemyNode` as follows:

```
let scnView: SCNView!
let _size:CGSize!
var scene: SCNScene!
var heroNode:SCNNode!
var enemyNode:SCNNode!
```

Also, create a new function called `addEnemy` to the class and add the following in it:

```
func addEnemy(){

    let geo = SCNBox(width: 4.0,
  height: 4.0,
  length: 4.0,
  chamferRadius: 0.0)

    geo.firstMaterial?.diffuse.contents =
UIColor.yellowColor()

    enemyNode = SCNNode(geometry: geo)
    enemyNode.position = SCNVector3Make(0, 20.0 ,
60.0)
    enemyNode.physicsBody =
SCNPhysicsBody.kinematicBody()
    scene.rootNode.addChildNode(enemyNode)

    enemyNode.name = "enemy"
}
```

Nothing too fancy here! Just as when adding the `groundNode`, we have created a cube with all its sides four units long. We have also added a yellow color to its material. We then initialize `enemyNode` in the function. We position the node along the *x*, *y*, and **z** axes. Assign the body type as kinematic instead of static or dynamic. Then we add the body to the scene and finally name the `enemyNode` as `enemy`, which we will be needing while checking for collision. Before we forget, call the `addEnemy` function in the `addColladaObjects` function after where we called the `addHero` function.

Note

The difference between the kinematic body and other body types is that, like static, external forces cannot act on the body, but we can apply a force to a kinematic body to move it.

In the case of a static body, we saw that it is not affected by gravity and even if we apply a force to it, the body just won't move.

Here we won't be applying any force to move the enemy block but will simply move the object like we moved the enemy in the SpriteKit game. So, it is like making the same game, but in 3D instead of 2D, so that you can see that although we have a third dimension, the same principles of game development can be applied to both.

For moving the enemy, we need an `update` function for the enemy. So, let us add it to the scene by creating an `updateEnemy` function and adding the following to it:

```
func updateEnemy(){

     enemyNode.position.z += -0.9

     if((enemyNode.position.z - 5.0) < -40){

        var factor = arc4random_uniform(2) + 1

        if( factor == 1 ){
            enemyNode.position =
SCNVector3Make(0, 2.0 , 60.0)
        }else{
            enemyNode.position =
SCNVector3Make(0, 15.0 , 60.0)
        }
    }
}
```

In the `update` function, similar to how we moved the enemy in the SpriteKit game, we increment the `z` position of the enemy node by 0.9. The difference being that we are moving the *z* direction.

Once the enemy has gone beyond `-40` in the *z* direction, we reset the position of the enemy. To create an additional challenge to the player, when the enemy resets, a random number is chosen between `1` and `2`. If it is `1`, then the enemy is placed closer to the ground, otherwise it is placed at 15 units from the ground.

Later, we will add a jump mechanic to the hero. So, when the enemy is closer to the ground, the hero has to jump over the enemy box, but when the enemy is spawned at a height, the hero shouldn't jump. If he jumps and hits the enemy box, then it is game over. Later we will also add a scoring mechanism to keep score.

For updating the enemy, we actually need an update function to add the `enemyUpdate` function to so that the enemy moves and his position resets. So, create a function called update in the class and call the `updateEnemy` function in it as follows:

```
func update(){

    updateEnemy()
}
```

Updating objects in the scene

In SceneKit, there is an update function that gets called after the scene is rendered. We will use this function to call the `update` function of our game. In the `GameViewController` class, add the following function:

```
func renderer(aRenderer: SCNSceneRenderer,
updateAtTime time: NSTimeInterval) {

  gameSCNScene.update()
}
```

To call the `rendererUpdateAtTime` function, the `GameViewController` class needs to inherit from `SCNSceneRendererDelegate`. So, where the class is created, add the following:

```
class GameViewController:
UIViewController, SCNSceneRendererDelegate {
```

Next, in the `viewDidLoad` function, set the current delegate to self as follows:

```
let scnView = view as SCNView
scnView.delegate = self
```

Note

Delegates are a part of code design patterns. Using delegates, a class can let a different class gain access and perform some of its

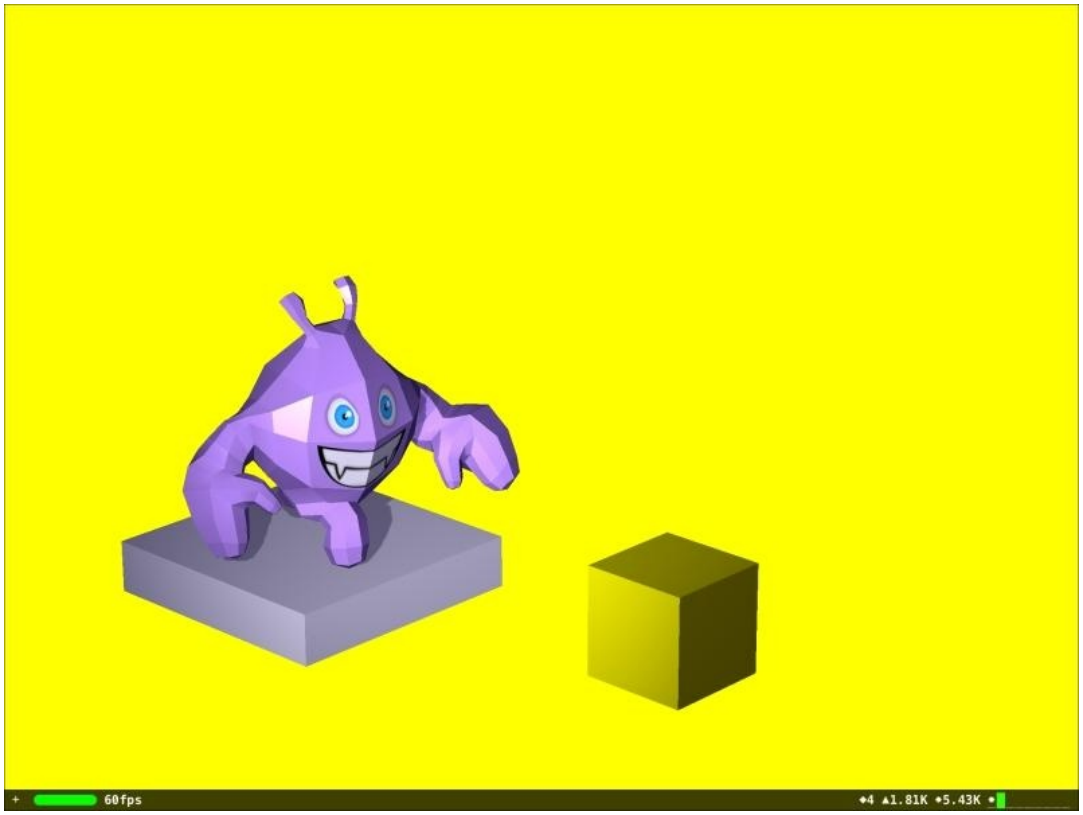
responsibilities. Here `SceneRenderer` is delegating to `scnView` by assigning the delegate as self.

The `rendererUpdateAtTime` function is a system function that gets called after all the objects are rendered in the scene. So, once the scene is rendered, the objects in the scene can be updated, otherwise it might result in artifacting.

Now, if we build and run the game, we see `enemyBox` getting updated.

But there is a problem, when the box hits the hero, the hero gets knocked off his pedestal and goes flying. This is because, firstly, the hero is a dynamic body, so external forces will affect him. Secondly, though we are moving the box manually without applying any force, e we are still moving the box and there is some inertial force calculated by SceneKit, so once the box hits the hero, the energy is transferred to the hero, and it acts like an external force applied on the hero so the hero starts moving.

Since we constrained the rotation of the hero using `heroNode.physicsBody?.angularVelocityFactor = SCNVector3Zero`, when the box hits him, he is not rotating. If we comment or delete the line, the hero will spin because of the box hitting him.



We will fix this issue when we check for collision. When the collision occurs, we will reset the position of the hero and box to their initial positions. So, let us next look at how to check for collision.

Checking for contact between objects

The physics engine of SceneKit has inbuilt functions that check for contact between objects when physics is enabled. A contact is triggered when two objects are just about to touch each other.

For checking contacts, we have to use the `didBeginContact` function of `physicsWorld`; so add the following code to the class. Also, we have to inherit from `SCNPhysicsContactDelegate` and set the `GameSCNScene` class as the contact delegate.

> **Note**
>
> Similar to calling `RenderDelegate` in `GameViewController`, to be able to receive "contact" events in the provided `GameSCNScene` by the physics engine and be able to manipulate them, the `GameSCNScene` class inherits from `SCNPhysicsContactDelegate` and the scene's `contactDelegate` is set to `self`.

So, at the top of the class, inherit from `SCNPhysicsContactDelegate`:

```
class GameSCNScene:
SCNScene,SCNPhysicsContactDelegate{
```

And, in the `init` function, add the following to set the current class as contact delegate:

```
scene.physicsWorld.contactDelegate = self
```

Next, add the `didBeginContact` function to the class as follows:

```
func physicsWorld(world: SCNPhysicsWorld,
didBeginContact contact: SCNPhysicsContact) {


    if( (contact.nodeA.name == "hero" &&
 contact.nodeB.name == "enemy") )
    {
        contact.nodeA.physicsBody?.velocity =
SCNVector3Zero
  println("contact")
```

```
        }
    }
```

When two physics objects collide, the nodes are stored in the contact variable. Since we already named our physics body objects, we check if the first body is of the hero's and the other body that the hero collided with is of the enemy. If true, we print out contact and also set the velocity of the first body to zero.

Since ours is a small game, we can guess that body *A* would be the hero and body *B* will be the enemy. In bigger games with many collisions happening in a second, it might be hard to determine which is body *A* and which is body *B*. In such cases, we will have to check for both cases, that is, is body *A* of enemy or hero and vice versa for body *B*, then make the necessary conclusion.

So, now we have added a hero and an enemy and we have our collision set up. Next we will see how to implement a 2D overlay over our 3D scene so that we can display buttons and score and add a game over condition for our simple game.

Adding a SpriteKit overlay

For showing scores and buttons for the game, we will add 2D SpriteKit layer. For adding the overlay, create a class called `OverlaySKscene`. In this class, add the following:

```
import SpriteKit

class OverlaySKScene: SKScene {

    let _gameScene: GameSCNScene!
    let myLabel: SKLabelNode!
    var gameOverLabel: SKLabelNode!
    var jumpBtn: SKSpriteNode!
    var playBtn: SKSpriteNode!

    required init?(coder aDecoder: NSCoder) {
        fatalError("init(coder:) has not been
implemented")
    }

    init(size: CGSize, gameScene: GameSCNScene){

    super.init(size: size)

    }
}
```

To import SpriteKit we will have to create a subclass of SpriteKit. Create global variables of type `GameSCNScene`, `SKLabelNodes`, and `SpriteNodes`. Here we create two `LabelNodes`: one for displaying score and the other to show "game over" text. We also create two `spriteNodes`: one for the play button and the other for the jump button.

We add the required `init` function and the default init function. The default `init` will take in the size of the scene and reference to the `GameSCNScene` class as parameters.

In the `init` function, we initialize the super class.

Adding labels and buttons

Next, in the `init` function, add the following code:

```
        _gameScene = gameScene

        myLabel =
SKLabelNode(fontNamed:"Chalkduster")
        myLabel.text = "Score: 0";
```

```
        myLabel.fontColor = UIColor.whiteColor()
        myLabel.fontSize = 65;
        myLabel.setScale(1.0)
        myLabel.position = CGPointMake(size.width
* 0.5, size.height * 0.9)
        self.addChild(myLabel)

  gameOverLabel =
SKLabelNode(fontNamed:"Chalkduster")
        gameOverLabel.text = "GAMEOVER";
        gameOverLabel.fontSize = 100;
        gameOverLabel.setScale(1.0)
        gameOverLabel.position =
CGPointMake(size.width * 0.5, size.height * 0.5)
        gameOverLabel.fontColor =
UIColor.whiteColor()
        self.addChild(gameOverLabel)
        gameOverLabel.hidden = true


        playBtn = SKSpriteNode(imageNamed:
"playBtn")
        playBtn.position = CGPoint(x: size.width
* 0.15, y: size.height * 0.2)
        self.addChild(playBtn)
        playBtn.name = "playBtn"

        jumpBtn = SKSpriteNode(imageNamed:
"jumpBtn")
        jumpBtn.position = CGPoint(x: size.width
* 0.9, y: size.height * 0.15)
        self.addChild(jumpBtn)
        jumpBtn.name = "jumpBtn"
        jumpBtn.hidden = true
```

In the `init` function, first, we set the `gameScene` passed to the `_gameScene` property of the scene.

Next we initialize both the `scoreLabel` and `gameOverLabel`. We set the values for `text`, `color`, `textsize`, and `position` and add it to the scene. In `gameOverLabel`, we set hidden to `true`, as we only want the text to display once the game is over.

We then initialize the play and jump buttons that we made in SpriteKit. We set the jump button sprite to be hidden, as we want it to display only when the play button is clicked and the game starts. The images for the jump and play button are provided in the resources folder of the chapter.

Adding touch interactivity

For adding touch interactivity, we will use the `touchesBegan` function similar to how we have always used it in SpriteKit. So, here we get the location of the touch and name of the sprite under the touch location. If the sprite name is `jumpBtn` and the `gameOver` Boolean is `false`, then we call the `heroJump` function in the `gameScene` class. If `gameOver` is `true` and if the play button is clicked, then we call the `startGame` function in the SceneKit class.

So, add the function as follows to detect touches:

```
    override func touchesBegan(touches: NSSet,
withEvent event: UIEvent) {
        /* Called when a touch begins */

        for touch: AnyObject in touches {
            let location =
touch.locationInNode(self)

            let _node:SKNode =
self.nodeAtPoint(location);

            if(_gameScene.gameOver == false){

                if(_node.name == "jumpBtn"){

                    _gameScene.heroJump()
                }

                }else{

                if(_node.name == "playBtn"){

                    _gameScene.startGame()
                }
            }
        }
    }
```

That's all for the SpriteKit class. We will be adding the `gameOver` Boolean, and `heroJump` and `startGame` functions in the SceneKit class. The code will show some errors until we create it, so ignore the errors for now.

Finishing the game loop

Move back to the SceneKit class and import SpriteKit at the top of the class as follows:

```
import UIKit
import SceneKit
import SpriteKit
```

Also, create a global variable called `skScene` of type `OverlaySKScene`. Add a new function called `addSpriteKitOverlay` and add the following code in the SceneKit class:

```
func addSpriteKitOverlay(){

    skScene = OverlaySKScene(size: _size,
gameScene: self)
    scnView.overlaySKScene = skScene
    skScene.scaleMode =
SKSceneScaleMode.ResizeFill

}
```

Here we initialize the `skScene` global variable we created earlier and pass in the size of the current scene and the current SceneKit class. Next we assign the `skScene` class to the `overlaySKScene` property of `scnView`. Finally, we set the `scaleMode` of the `skScene` variable to type `SKSceneScaleMode.ResizeFill`.

Finally, call the `addSpriteKitOverlay` function in the `addColladaObjects` function as follows:

```
func addColladaObjects(){

        addHero()
        addGround()
        addEnemy()
        addSpriteKitOverlay()
}
```

Making the hero jump

We still need to add the Boolean and functions to the class to make our game work. So, at the top of the class, create a global variable called `gameOver` and set it to `true`.

Next, create a new function called `heroJump` as follows:

```
func heroJump(){

heroNode.physicsBody?.applyForce(SCNVector3Make(0,
1400, 0), impulse: true)

}
```

Here we apply an upward force of `1400` units in the *y* direction to `heroNode`.

Next, create the `gameStart` function as follows and add it into the class:

```
func startGame(){

    gameOver = false
    skScene.jumpBtn.hidden = false
    skScene.myLabel.hidden = false
    skScene.playBtn.hidden = true
    skScene.gameOverLabel.hidden = true

    score = 0
    skScene.myLabel.text = "Score: \(score)"
}
```

The `gameOver` Boolean is set to `false`. We set the jump button and `scoreLabel` to be visible and hide the play button and `gameOverlabel`.

For keeping track and displaying the score, we need a score variable, so create a global variable called `score` of type `int` and initialize it to `0` at the top of the class. Again, in the `startGame` function, set the value to `0` so that every time the function is called the value is reset. Also, we set the `scoreLabel` text to reflect the current score at the start of the game.

For scoring in the game, we will increment the score every time the enemy block goes beyond the screen and gets reset. If the block hits the hero, then it will be game over.

So, in the `enemyUpdate` function, add the following highlighted line after we check if the enemy's `Z` position is less than `-40` to update the score and `scoreLabel` text:

```
func updateEnemy(){

    enemyNode.position.z += -0.9

    if((enemyNode.position.z - 5.0) < -40){

        var factor = arc4random_uniform(2) + 1

        if( factor == 1 ){
```

```
                enemyNode.position =
SCNVector3Make(0, 2.0 , 60.0)
            }else{
                enemyNode.position =
SCNVector3Make(0, 15.0 , 60.0)
            }
             score++
            skScene.myLabel.text = "Score: \(score)"
        }
    }
```

Setting a game over condition

In the `didBeginContact` function, add the following right after we reset the velocity of the player to zero:

```
gameOver = true
GameOver()
```

Here we set `gameOver` to `true` and call a `GameOver` function where we will set the visibility of the labels and buttons. So, add a new function called `GameOver` to the SceneKit class as follows:

```
func GameOver(){

    skScene.jumpBtn.hidden = true
    skScene.playBtn.hidden = false
    skScene.gameOverLabel.hidden = false

    enemyNode.position = SCNVector3Make(0, 2.0 ,
60.0)
    heroNode.position = SCNVector3Make(0, 0, 0)
}
```

Here, once the game is over, we hide the `jumpButton` and unhide the `playButton` and `gameOverLabel`. We then reset the position of the enemy and hero to their initial state.

Next we have to make sure that the `enemyUpdate` function is only called when `gameOver` is false. In the `update` function, enclose the `enemyUpdate` function in an `if` statement as follows:

```
if(!gameOver){

   updateEnemy()

}
```

Finally, we have to adjust the gravity in the scene, otherwise the hero will get tossed into the air since the gravity is currently so low. In the

`addColladaObjects` function, add the following line at the end of the function:

```
scene.physicsWorld.gravity = SCNVector3Make(0,
-300, 0)
```

Now our game loop is ready. If you press the play button, the game will start and the jump button will be visible, and the hero will jump when it is pressed. The score will increase each time the hero successfully avoids the enemy block, and the game will be over if he hits the enemy block. Once the game is over, the play button will be visible and jump button will be hidden. Tapping the play button again will reset everything and the game will start again.

Fixing the jump

There is still one problem though. You can keep on tapping the jump button and the hero will keep on going up. We don't want that. We want the hero to jump only when he is grounded. For this, we will add a small counter and disable the jump when the hero is in the air.

For this, add a new global variable called `jumpCounter` of type `int` and initialize it to `0`. In the `update` function of the class, add the following:

```
jumpCounter--

if(jumpCounter < 0 ){

        jumpCounter = 0
}
```

Here we decrement the value of `jumpCounter` and once it is less than zero, we set the value equal to `0`.

Next, in the `heroJump` function, enclose where we apply force to the hero to jump in an if condition as follows:

```
if(jumpCounter == 0){

 heroNode.physicsBody?.applyForce(SCNVector3Make(0,
                    1400,
                    0),
                    impulse: true)

    jumpCounter = 25
}
```

Now, the hero will only jump when `jumpCounter` is equal to 0. If it is equal to `0`, then the force is applied and the counter is set to `25`.

> **Note**
>
> The number was arrived at after trial and error to ensure that the jump button won't be pressed when the monster is in the air.

In the `update` function, we decrement this value, until then the force cannot be applied. Once `jumpCounter` is set to `0` again, the hero can jump again.

So, finally, we can run and test the game. Make sure the `addSpriteKitOverlay` function is called in the `addColladaObjects` function.



Adding wall and floor parallax

What is a game without a parallax effect? In SpriteKit, we added parallax using sprites, while in SceneKit, we will use planes to add it to the scene. Apart from adding parallax, we will also see how to add diffuse, normal, and specular maps to the plane. Also, we will learn what those terms even mean.

So, as usual, we create a new function in which we will add all these planes. Add four global `SCNNodes` as follows:

```
var parallaxWallNode1: SCNNode!
var parallaxWallNode2: SCNNode!
var parallaxFloorNode1: SCNNode!
var parallaxFloorNode2: SCNNode!
```

Also, add a function to the scene called `addWallandFloorParallax` as follows:

```
    func addWallandFloorParallax(){

       //Preparing Wall geometry
        let wallGeometry = SCNPlane(width: 250,
height: 120)

 wallGeometry.firstMaterial?.diffuse.contents =
"monster.scnassets/wall.png"
        wallGeometry.firstMaterial?.diffuse.wrapS
= SCNWrapMode.Repeat
        wallGeometry.firstMaterial?.diffuse.wrapT
= SCNWrapMode.Repeat

 wallGeometry.firstMaterial?.diffuse.mipFilter =
SCNFilterMode.Linear

 wallGeometry.firstMaterial?.diffuse.contentsTransform
= SCNMatrix4MakeScale(6.25, 3.0, 1.0)


 wallGeometry.firstMaterial?.normal.contents =
"monster.scnassets/wall_NRM.png"
        wallGeometry.firstMaterial?.normal.wrapS
= SCNWrapMode.Repeat
        wallGeometry.firstMaterial?.normal.wrapT
= SCNWrapMode.Repeat

 wallGeometry.firstMaterial?.normal.mipFilter =
SCNFilterMode.Linear
```

```
 wallGeometry.firstMaterial?.normal.contentsTransform
= SCNMatrix4MakeScale(6.25, 3.0, 1.0)


 wallGeometry.firstMaterial?.specular.contents =
"monster.scnassets/wall_SPEC.png"

 wallGeometry.firstMaterial?.specular.wrapS =
SCNWrapMode.Repeat

 wallGeometry.firstMaterial?.specular.wrapT =
SCNWrapMode.Repeat

 wallGeometry.firstMaterial?.specular.mipFilter =
SCNFilterMode.Linear

 wallGeometry.firstMaterial?.specular.contentsTransform
= SCNMatrix4MakeScale(6.25, 3.0, 1.0)


 wallGeometry.firstMaterial?.locksAmbientWithDiffuse
= true

      //Preparing floor geometry
        let floorGeometry = SCNPlane(width: 120,
height: 250)

 floorGeometry.firstMaterial?.diffuse.contents =
"monster.scnassets/floor.png"

 floorGeometry.firstMaterial?.diffuse.wrapS =
SCNWrapMode.Repeat

 floorGeometry.firstMaterial?.diffuse.wrapT =
SCNWrapMode.Repeat

 floorGeometry.firstMaterial?.diffuse.mipFilter =
SCNFilterMode.Linear

 floorGeometry.firstMaterial?.diffuse.contentsTransform
= SCNMatrix4MakeScale(12.0, 25, 1.0)


 floorGeometry.firstMaterial?.normal.contents =
"monster.scnassets/floor_NRM.png"
        floorGeometry.firstMaterial?.normal.wrapS
= SCNWrapMode.Repeat
        floorGeometry.firstMaterial?.normal.wrapT
= SCNWrapMode.Repeat

 floorGeometry.firstMaterial?.normal.mipFilter =
SCNFilterMode.Linear
```

```
 floorGeometry.firstMaterial?.normal.contentsTransform
= SCNMatrix4MakeScale(24.0, 50, 1.0)

 floorGeometry.firstMaterial?.specular.contents =
"monster.scnassets/floor_SPEC.png"

 floorGeometry.firstMaterial?.specular.wrapS =
SCNWrapMode.Repeat

 floorGeometry.firstMaterial?.specular.wrapT =
SCNWrapMode.Repeat

 floorGeometry.firstMaterial?.specular.mipFilter =
SCNFilterMode.Linear

 floorGeometry.firstMaterial?.specular.contentsTransform
= SCNMatrix4MakeScale(24.0, 50, 1.0)


 floorGeometry.firstMaterial?.locksAmbientWithDiffuse
= true

        //assign wall geometry to wall nodes
  parallaxWallNode1 = SCNNode(geometry:
wallGeometry)
        parallaxWallNode1.rotation =
SCNVector4Make(0, 1, 0, Float(-M_PI / 2))
        parallaxWallNode1.position =
SCNVector3Make(15, 0, 0)

 scene.rootNode.addChildNode(parallaxWallNode1)

        parallaxWallNode2 = SCNNode(geometry:
wallGeometry)
        parallaxWallNode2.rotation =
SCNVector4Make(0, 1, 0, Float(-M_PI / 2))
        parallaxWallNode2.position =
SCNVector3Make(15, 0, 250)

 scene.rootNode.addChildNode(parallaxWallNode2)

        //assign floor geometry to floor nodes

        parallaxFloorNode1 = SCNNode(geometry:
floorGeometry)
        parallaxFloorNode1.rotation =
SCNVector4Make(0, 1, 0, Float(-M_PI / 2))
        parallaxFloorNode1.rotation =
SCNVector4Make(1, 0, 0, Float(-M_PI / 2))

        parallaxFloorNode1.position =
SCNVector3Make(15, 0, 0)

 scene.rootNode.addChildNode(parallaxFloorNode1)
```

```
        parallaxFloorNode2 = SCNNode(geometry:
floorGeometry)
        parallaxFloorNode2.rotation =
SCNVector4Make(0, 1, 0, Float(-M_PI / 2))
        parallaxFloorNode2.rotation =
SCNVector4Make(1, 0, 0, Float(-M_PI / 2))
        parallaxFloorNode2.position =
SCNVector3Make(15, 0, 250)

 scene.rootNode.addChildNode(parallaxFloorNode2)
    }
```

OMG!! This is a whole lot of code. But don't panic. We will go through it systematically. Just look at the code where it says `preparing wall geometry`. First, we will see how the wall geometry is set up, and then once you have an understanding of it, we will see how to set up the floor geometry.

We create a new constant called `wallGeometry` and assign a `SCNPlane` to it. The difference between an `SCNPlane` and `SCNFloor` is that here we can set the dimensions of the plane. So, very simply we set the width and height of the plane to be `250` by `120` units.

Next, we assign a material to the plane. Until now, we have only seen how to assign a color to an object in SceneKit. Here we assign three kinds of maps to the plane. The first is a diffuse.

> **Note**
>
> A diffuse material is an image or texture that you want to paste on to a plane.

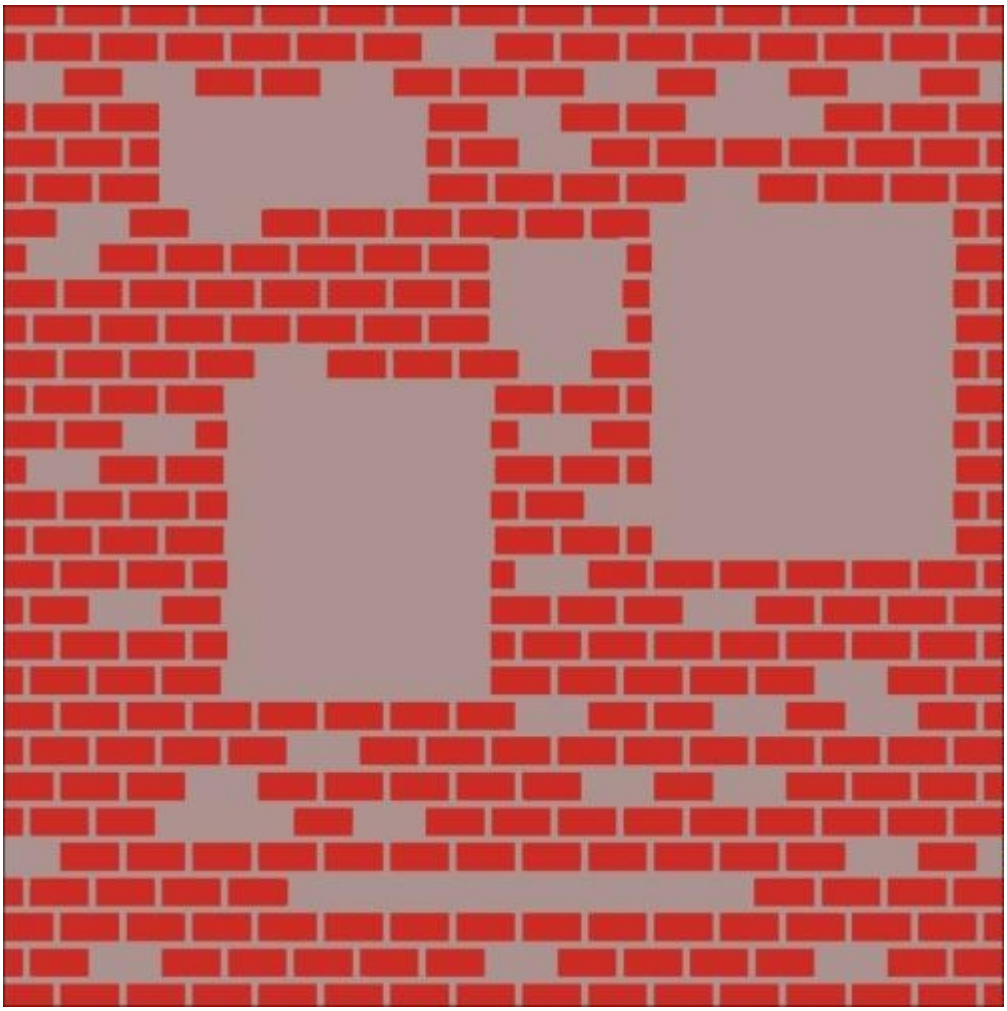
Textures are like wallpapers on walls. Imagine an unpainted wall; now you can either color the wall or apply wallpaper. Adding paint is done in the digital world by applying a color, like we did for `enemyblock` where we assigned a yellow diffuse color to it. To apply a wallpaper in the digital world, we apply using textures or images. Here we apply the `wall.png` image to the wall plane geometry.

Notice that the wall plane is pretty big in terms of width and height. If we let it be, the wall.png image will have to stretch to fit the dimension of the wall. So we use the `wrapt` and `wraps` functions so that the wall texture is repeated in both the *x* and *y* direction of the plane without stretching the wall texture. So this is what happens in the next two lines. We are just repeating the wall texture over in both the directions until it fills the whole plane.

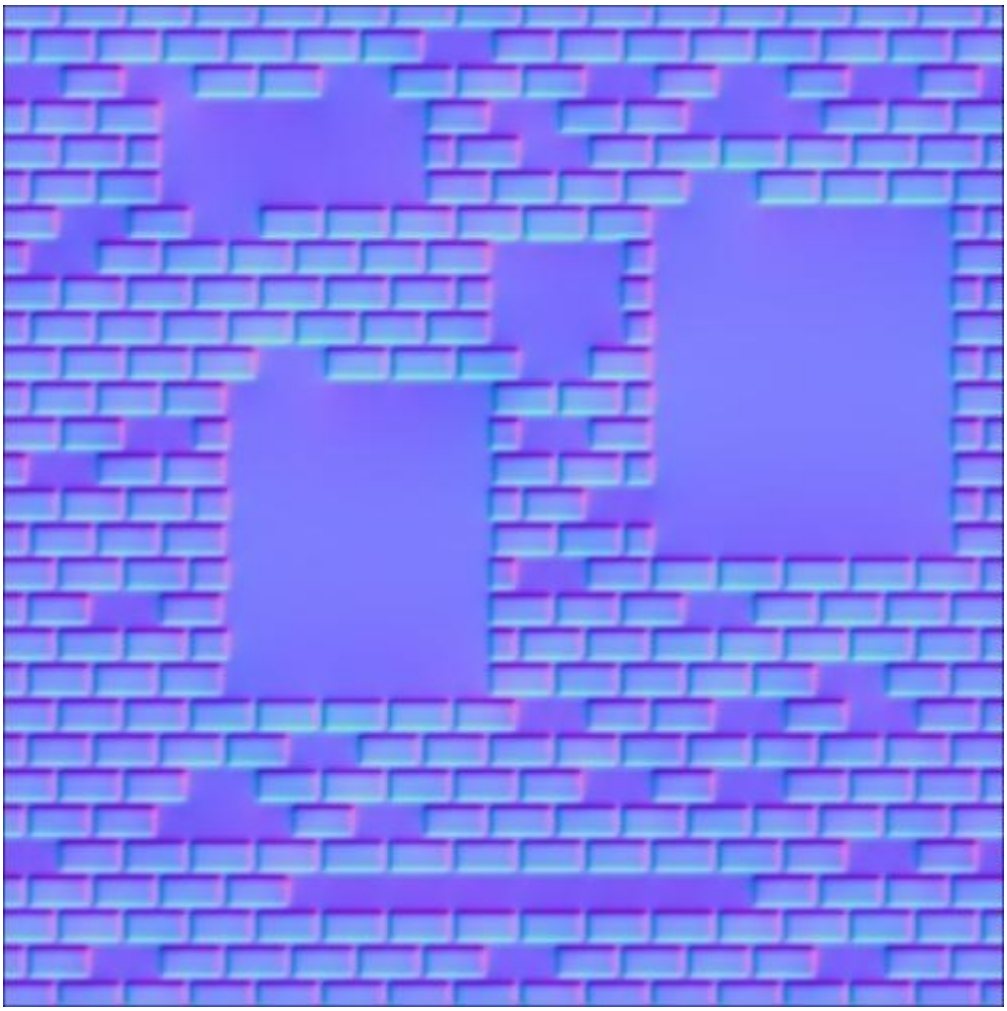
In the next step, we assign the `mipFiler` to `linear`. The filter will decide how much detail needs to be added to the texture. If the camera is far enough away, it will generate a lower resolution texture to reduce the burden on the CPU. If the camera comes closer, then a higher resolution image will be created so that all the details of the texture will be visible. This is purely used for optimization purposes. Linear filter mode is the most basic type of mode of filters. There are other modes called bilinear and trilinear, and so on that will give even better results but are computationally expensive. For our purposes, linear filtering will suffice. You can see the difference by changing the code and running the game on a device.

For the diffuse at the end, we scale down the texture of the image itself depending upon how big or small we want the texture to appear on the plane. While scaling, we have scaled it to the same proportion as the size of the geometry. So here in the *x* and *y* plane, we scale it down by a factor of 1/40 times the values of width and height of the geometry. Since we are not scaling in the *z* direction, we keep it at `1`.

The following is the image of the diffuse map of the wall:



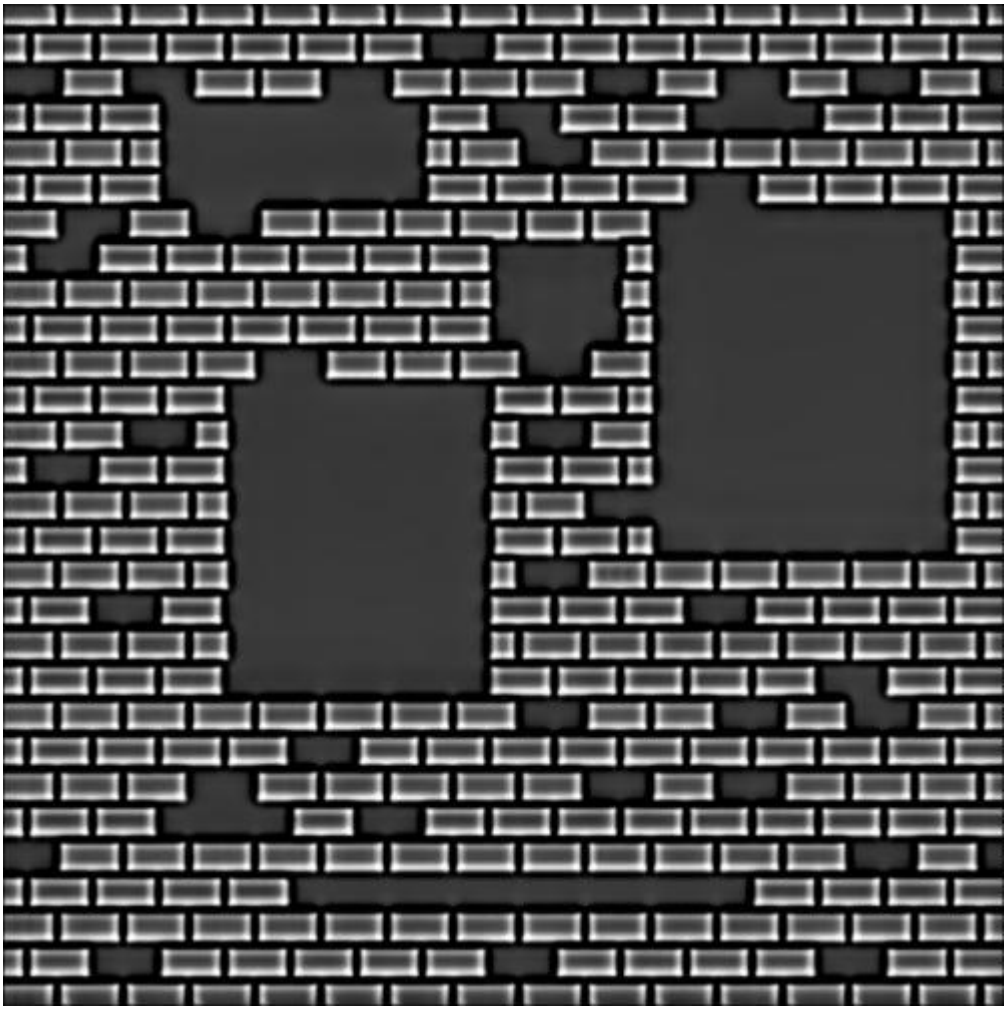
Now, the same five steps are repeated for the normal and the specular map. We saw that to add wallpaper to the plane we have to use a diffuse map. Now, what if this wallpaper has some bumps and holes in it? Not all walls are so smooth. So, to add this roughness to the wallpaper, we use what is called a normal map. The following is the image of the normal map:



It basically calculates how light should behave once it hits the normal surface. Depending on the direction of the light and if there is a dump or hole in the normal map, the lighting will be calculated automatically. All this is done using a normal map. A normal is nothing but an image. The code will take information from this image to create the desired effect. So, here we assign a normal map called `wall_NRM.png` and assign it as content. The next four steps are very similar to the ones we performed for diffuse maps.

Next, let us look at specular maps. This map will decide which parts of the texture are shiny and which parts are not. Imagine your wallpaper was made of stainless steel. To bring this effect, we assign the specular map, and to add this, we use the `wall_SPEC.png` file. The next four steps are the

same, only now we do it separately for the specular map. The image of the specular map is as follows:



The same steps are repeated for the floor geometry as well, only this time we have flipped the width and height values.

Once the floor and wall geometries are ready, we assign them to the wall and floor nodes.

For the wall nodes, we assign the geometry to the node. Then we rotate the node so that the wall is vertical. We rotate the node by –90 degrees in the *y* direction. Then we place the first wall at `(15, 0, 0)`, and then add it to the scene. We place the second wall node at `(15, 0, 250)`, which will make it adjacent to the first plane.

For the floor nodes, we follow a similar process, but here we have to rotate it twice to be horizontal to the ground.

With our wall and floor nodes ready, we can update the positions of the planes to create the parallax effect. So, in the `update` function, add the following code:

```
parallaxWallNode1.position.z += -0.5
parallaxWallNode2.position.z += -0.5
parallaxFloorNode1.position.z += -0.5
parallaxFloorNode2.position.z += -0.5

if((parallaxWallNode1.position.z + 250) <= 0){
     self.parallaxWallNode1.position =
SCNVector3Make(15, 0, 250)
}

if((parallaxWallNode2.position.z + 250) <= 0){
     self.parallaxWallNode2.position =
SCNVector3Make(15, 0, 250)
}

if((parallaxFloorNode1.position.z + 250) <= 0){
     self.parallaxFloorNode1.position =
SCNVector3Make(15, 0, 250)
}

if((parallaxFloorNode2.position.z + 250) <= 0){
     self.parallaxFloorNode2.position =
SCNVector3Make(15, 0, 250)
}
```

This will look very familiar to you now. Like we updated and reset the background sprites in the SpriteKit game, we are going update the positions of the four nodes and then reset the position of all of them if they have gone beyond 250 units in the *z* direction.

Add the `addWallandFloorParallax` function to the `addColladaObjects` function.

For this scene, I have also added an ambient light node using the scene graph, as otherwise the scene was looking very dark. Go to the `monsterScene.DAE` file, and under the scene graph, click on the plus sign and add a new node. Next, right-click on the node and select **addLight**. In the panel on the right, select the **Attributes** inspector, and under the **Type**, select **Ambient**. Below **Type**, select **Color** and choose a dark blue or purple color. Run the game now and you should see the result, as shown in the following screenshot:


Score: 0
60fps

Adding particles

As the icing on the cake, we will include a rain particle effect. For creating a particle effect in SceneKit, go to **File** | **New**, and under **Resource**, select **SceneKit Particle system**. Click on **Next**, and on the next screen, select **Rain** from the **Particle System Template** drop-down list. Click on **Next** and give the file a name. I called it `rain`. So now you will have `rain.scnp` and `spark.png` files in the project.

To position the particle better, go to the scene graph and create a node called `particleNode`, and translate and rotate the node such that it is pointing toward the hero.

Create a new function called `addRainParticle` to the class and add the following code:

```
func addRainParticle(){

let rain = SCNParticleSystem(named: "rain",
inDirectory: nil)

    var particleEmitterNode = SCNNode()
    particleEmitterNode =
scene.rootNode.childNodeWithName("particleNode",
recursively: true)!

    particleEmitterNode.addParticleSystem(rain)

 scene.rootNode.addChildNode(particleEmitterNode)

rain.warmupDuration = 10

}
```

We create a new constant called `rain` and assign `SCNParticleSystem` to it, and provide the `rain` particle system we created in it.

A new `SCNNode` called `particleEmitterNode` is created and we assign the `particleNode` we created in the scene graph to it. Then we assign the rain particle system to it. Then we add the `particleEmitterNode` to the scene.

We use the `warmupDuration` of the particle system and assign a value of `10` to it. This is done so that when the game starts, the rain particle effect is fast-forwarded to look as if it was already raining.

You can select the `rain.scnp` file and change the parameters to better suit your needs. Build and run to see our finished SceneKit game. Call the `addRainParticle` function at the end of `addColladaObjects` function.



Tip

Adding audio to the game is exactly like how we added the main theme into the SpriteKit game. So, I will leave you guys to experiment with it to figure out how to add audio to the scene.

Also, I didn't want to want to repeat again how to import the assets to the game as we have seen how to do that for over four chapters now. But make sure that while calling the files, you provide the correct folder locations. If not, then the assets won't be retrieved properly, causing build errors.

Summary

In this chapter, we saw how to make a 3D game in SceneKit. From making simple geometries to floors, we created a full-fledged game will a complete game loop. We added a scene already created in a 3D software package with animation and imported it into SceneKit. We didn't have to add a camera or a light source as it was already part of the 3D scene.

We imported the COLLADA object into the scene and saw how to access the objects through code. We added an enemy and physics to the scene. We used SceneKit's physics engine to calculate collision and also applied force to the hero object.

Additionally, you also saw how to integrate SpriteKit into SceneKit to display the score and buttons on the scene. We also used SpriteKit's `touchBegan` function to detect touches on the screen and created the play and jump buttons.

Parallax scrolling was also added to the scene using planes. Also, you saw different types of maps such as diffuse, normal, and specular maps, and the functionality of each. Finally, we added a rain particle system to the scene.

In the next chapter, you will dive deeper into graphics programming and see how objects are actually displayed on to the screen using the Metal Graphics library.

Chapter 9. Metal

Before we start creating a game, you should understand how to display things on the screen. This is always taken for granted, as all frameworks have a class called `sprite`, in which we just give a `.png` or `.jpg` file and say `addChild` and `tada`; we then have an image appearing on the screen. Moreover, with just a few simple functions such as move, scale, and rotate, we can even transform the sprite's position, size, and rotation. In reality, this `sprite` class does a whole lot of work just to display the image on the screen.

In this chapter, we will look at Metal—a new graphics library from the people at Apple. This graphics library will help us to display objects on the screen. It is a communication tool that talks to the processor, the memory, the **graphics processing unit** (**GPU**), and the screen.

If you are coming from a DirectX or OpenGL background, you will see that the process to display stuff on the screen, otherwise known as a graphics pipeline, is very similar to that in Metal. Metal's graphics pipeline is programmable with the use of a shader language, which uses C++11 as the base. We will go through it in detail in this chapter. Let's get rocking with Metal!

We'll cover the following topics in this chapter:

- Overview
- Graphics pipeline and shaders
- The basic Metal project
- The colored quad project
- The textured quad project

Overview

Metal is a graphics API that is used to display anything on the screen. Metal is very specific to iOS 8 and above; moreover, it will only work on a device with an A7 chip and above. It is said that it will run ten times faster than OpenGLES. OpenGLES is another graphics library. Unlike Metal, OpenGLES is open source and works cross-platform. The trade-off here is that you can make it ten times faster, meaning that you can add mode particles and objects in the screen, however, you can't run your games on other operating systems such as Android and Windows Phone. Games developed with OpenGLES can be run on other devices with minor changes to the code. In fact, SpriteKit and SceneKit are developed by using OpenGLES. So, do you want to make a game with more objects on the

screen or would you rather make your game available on other platforms? The choice is up to you.

In Metal, for rendering anything on the screen, you have to do it in two stages. The first stage is the preparation or initialization stage, and the next one is the drawing stage. In the preparation stage, we first get access to the GPU, ready the resources such as vertices and buffers, and prepare the render pipeline and the view in which you want to the object to be rendered into. After the preparation stage, we can get into actually drawing the image in the "draw" stage.

The graphics pipeline and shaders

Let's look at these stages in detail. We'll look at the preparation stage first.

The preparation/initialization stage

The following steps are included in this stage:

1. Get device.
2. Command queue.
3. Resources.
4. Render pipeline.
5. View.

We'll look at each step, one by one.

Get device

First, we have to get the device that will be responsible for rendering our object. This will let us know the capabilities of the GPU in terms on how powerful it is and what its features are. In Metal, it will basically tell us what device we are running the game on, that is, whether it is running on an iPhone, iPad, or OS X. It will also tell us which version of the device it is and whether it is an iPhone 6, 5, 4, or any other device.

Command queue

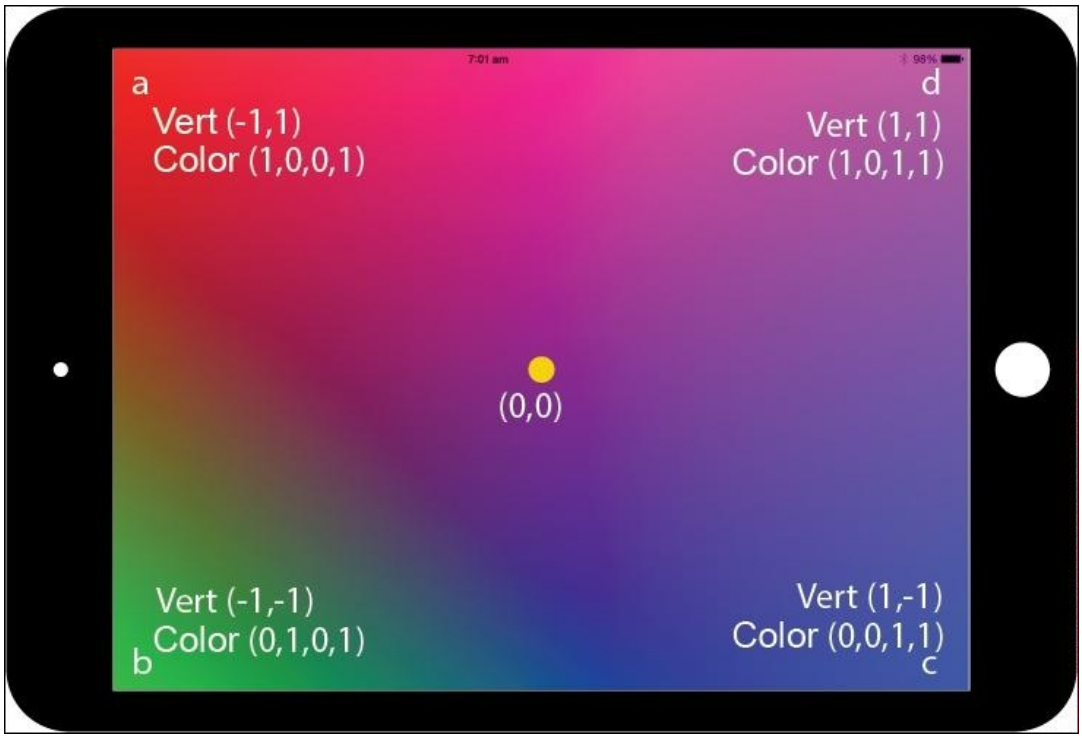
Once we know which device we are working on, we get the next command in the queue. Information is sent from the CPU to the GPU in an asynchronous manner. This means that both the CPU and the GPU don't work on the same items at a particular time. When the CPU is done with the calculations, the information is fed to the GPU. If the GPU is busy, the information has to wait while the CPU works on the next process. For this, we need a queue. So basically, the command to the GPU waits in a queue before it can be executed. Therefore, in this step, we get the next free slot in the queue so that information can be fed from the CPU to the GPU.

Resources

In this step, we ready the vertex, which is the information that we want to pass to the GPU. We will declare the properties that each vertex has (for example, the coordinates of each vertex at a basic level). We can also provide other information, such as color for each of the coordinates. We also need to store this data in the memory so that when the command executes, the GPU will retrieve the information. Information is stored in

buffers. For each property of the vertex, a buffer is created. A buffer, as we saw in the first chapter, is nothing but a location in the memory.

In the following screenshot, we see that there are four coordinates specified **a**, **b**, **c**, and **d**, each having their own vertex and color property:



Render pipeline

This step is broken down into two steps. In the first step, you'll have to create a descriptor. A descriptor is where you initialize the pipeline. We will tell the pixel which format to use while rasterizing the image. We will also be passing in our vertex shaders and pixel shader function here. Once the descriptor is ready, we can pass it to the *render pipeline* state. The state now contains all the information for the pipeline to be easily passed around. We will look at shaders in a later section, as it is better understood in action.

Note

Recap the following from Chapter 1, *Getting Started*:

- **Vertex/Geometry Shader**: The information is then passed into the vertex shader. Vertex shaders are programmable by using shader language. The language is similar to C. Using this language, we can change the position, causing the object to move, scale, or rotate like how you can do in your `update` function within the game loop.
- **Pixel/Fragment Shader**: As in the vertex shader, in which you were able to do vertex modification, pixel shaders will enable you to make

pixel-based operations. As this a shader, you know that this is also programmable. Using pixel shaders, you can create effects such as changing the color and transparency of the texture provided.

View

The last step is to set up and ready the view. In the view stage, we will actually need access to the layer attached to the view on which the object will be drawn. The layer is like a blank canvas that is ready to be drawn on. We get the layer of the view so that we can keep drawing and erasing on it.

We are done with the initialization stage; next, we will see the steps to actually start drawing something on the screen.

The draw stage

Here, we will finally draw the vertices that we sent in the first stage. This stage also has a few steps that need to be followed:

1. Start render pass.
2. Get command buffer.
3. Draw.
4. Commit the command buffer.

Start render pass

We prepare the layer for drawing the object. We assign the layer to be drawn onto and clear the surface with the default color.

Get command buffer

The Command buffer is the place where the command for rendering is stored. We need to get access to the command buffer in order to execute the commands.

Draw

Finally, the drawing takes place on the layer. This is done by a render command encoder, which takes the code from the command buffer and encodes it into machine language in order to render the image. We pass the pipeline state and the vertex buffer and then draw the image. This whole step is done off screen.

Commit the command buffer

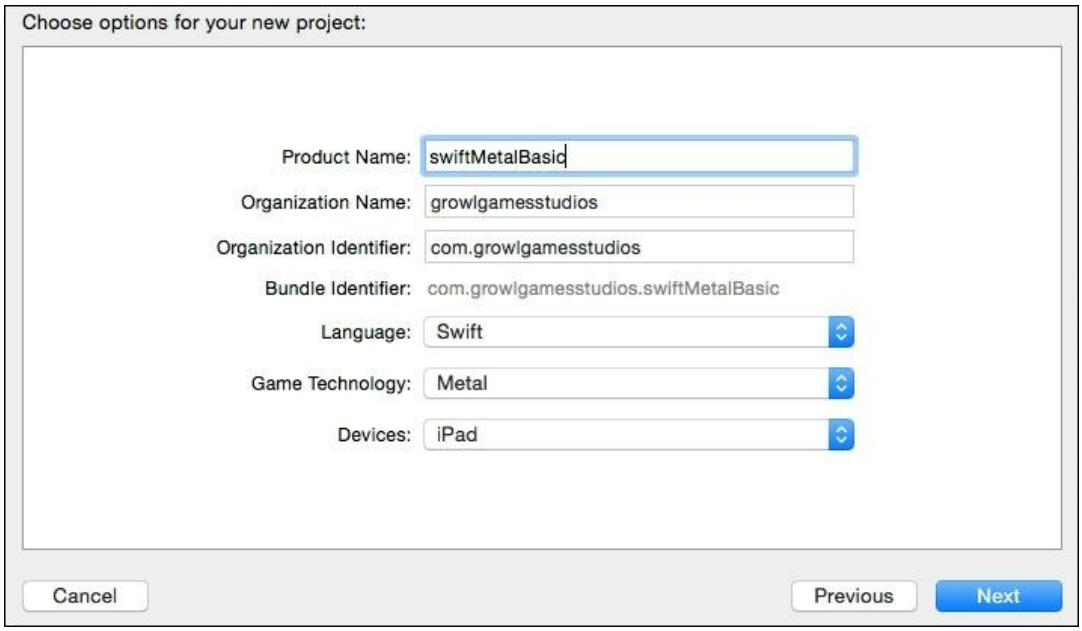
This is the final stage, in which image rendering has been done, and the image is ready to be displayed onto the screen.

With all of this theoretical knowledge with us now, let's put it all to practice to see how we can display something to look at on the screen.

The basic Metal project

In this first project, we will create a basic triangle and display it on the device.

Create a new Xcode project. Select **Metal** as the technology and **Swift** as the language for the project:



Select a location to save the project in. In the `GameViewController.swift` file, we delete all the contents of the `viewDidLoad` functions, so that we can start from the absolute basics.

As discussed in the overview, the first thing that we have to do is get the device that the application will work on. Add the following line:

```
//get device
let device: MTLDevice =
MTLCreateSystemDefaultDevice()
```

We create a new constant called `device` of the `MTLDevice` type and assign `MTLCreateSytemDefaultDevice` to it. So now, we have direct access to the device.

Next, we have to create `commandQueue` for the device:

```
//Create Command Queue
var commandQueue: MTLCommandQueue =
device.newCommandQueue()
```

We get the command from the device and assign it to a new variable called `commandQueue`.

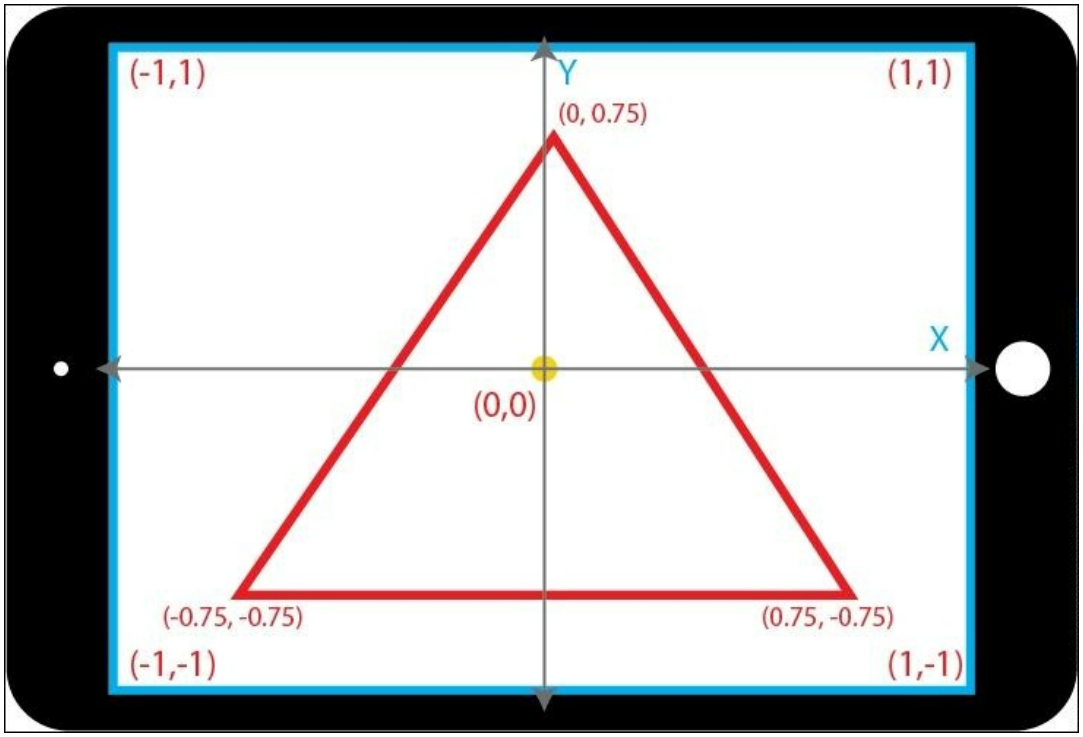
In the next step, we prepare resources such as the vertex information and vertex buffers. To create vertex definitions, create an array at the top of the file called `vertexArray`, as shown in the following lines of code:

```
let vertexArray:[Float] = [

    0.0, 0.75,

    -0.75, -0.75,

    0.75, -0.75]
```

Imagine the shape of a screen to be a rectangle, with its width and height being 2 pixels each and the center of the screen being the origin. So, in the preceding array, the 0th, 2nd, and 4th values are *x* coordinates, and the 1st, 3rd, and 5th items are the corresponding *y* coordinates.

Here, we are passing in three pairs of `x` and `y` values to draw a triangle. For the first value, `x` is at the origin and `y` is at `.75` in the *y* direction from the center of the screen. The next two coordinates are to the bottom left and bottom right of the origin.

So now, we have our vertices ready. Next, we have to create a vertex buffer so that we can store these vertices in it. Create a new variable called `vertexBuffer` of type `MTLBuffer`, as shown in the following diagram, and assign the `vertexArray` variable we created along with the size of the array and `nil` for `options`.




```
var vertexBuffer: MTLBuffer! =
device.newBufferWithBytes(vertexArray,
length: vertexArray.count *
sizeofValue(vertexArray[0]),
                          options: nil)
```

Next, we have to create our vertex and fragment shader.

Shaders are small pieces of code that are compiled at runtime. There are two types of shaders: **vertex** and **fragment**:

- **Vertex shaders**: This lets us perform vertex manipulation if we want an outside the game code. By vertex manipulation, we mean moving, rotating, and translating the each of the vertices, and therefore, the whole object. This is at the basic level; we can perform more complex operations by using vertex shaders. The vertex shader gets called as many times as the number of coordinates we pass in. So in this case, it will be called three times.
- **Fragment shaders**: These can be used to perform manipulations at the pixel level. They can be used to make various effects such as blur, pixelate, cel shading, and so on. Unlike Vertex shaders, pixel or fragment shaders can be called as many times as required to fill the space within the triangle with a color or texture.

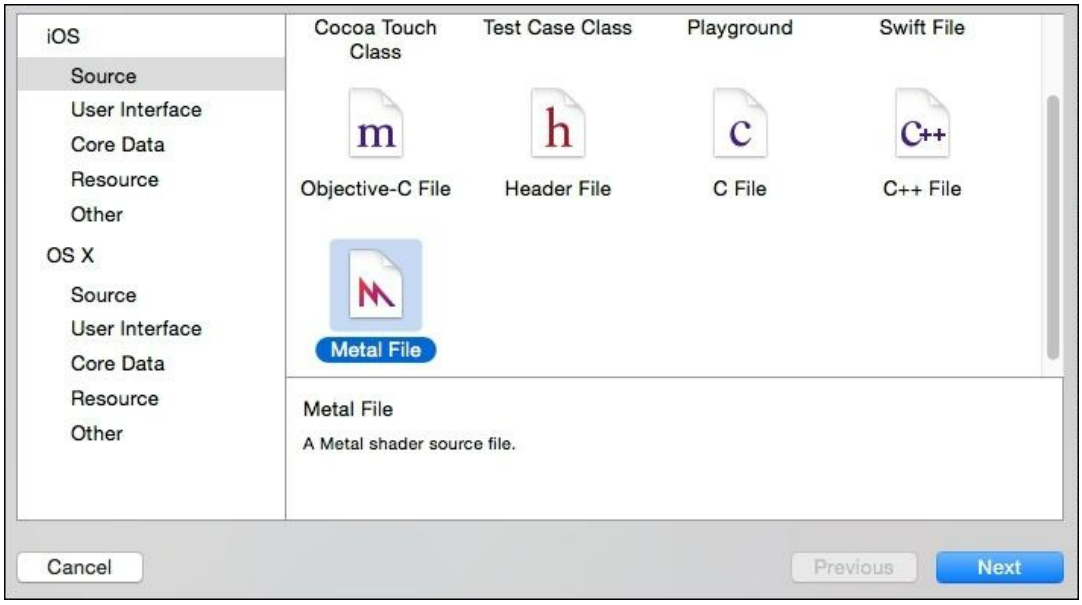
Let's see how to write our first shader files. First, we will write a vertex shader. In the project file, you will already have a new file called

`Shaders.metal`. This is the shader file for Metal, in which you will write both your vertex shader and pixel shader.

People with an OpenGL or DirectX background may be wondering where the other file is. That is one of the differences between Metal and other shader languages. Metal uses just one file, and in this file, you can write both the shaders. Each shader is not a file but a function. So later, when we pass the shader in the pipeline descriptor, we won't be giving the file name of the shader but the name of the function.

You can create more than one shader file and write your vertex shader in one file and pixel shader in the other, or vice versa. Metal really doesn't care which file you put what in, as long as you make sure that you are calling the right functions.

To create additional metal shader files, you can go to **File** | **New** | **File** | **Source** and select **Metal File**:



In the `Shaders.metal` file, remove everything and add the following code, as we will start from the basics:

```
#include <metal_stdlib>
using namespace metal;

vertex float4 myVertexShader(const device float2
* vertex_array [[ buffer(0) ]],
                             uint vid
[[ vertex_id ]]) {


    return float4(vertex_array[vid],0,1);
}
```

```
fragment float4 myFragmentShader() {

    return float4(1.0, 0.0, 1.0, 1.0);
}
```

At the top, we include the metal standard library and use namespace metal. People with C++ will feel at home as the metal shader language is written in a modified version of C++11.

The first function right after that is the vertex shader function. Shader functions start with the keyword vertex or fragment to denote whether the function is a vertex shader or a fragment shader.

So, in the vertex shader, the function has the keyword `vertex` and it returns a `float4`. A float 4 is like a struct with four float values: `x`, `y`, `z`, and `w`; or `r`, `g`, `b`, and `a`.

> **Note**
>
> Shaders also have their own data types, such as `float`, `float2`, `float3`, and `float4` or `int`, `int2`, `int3`, and `int4`. Since shaders usually deal with vertices or colors which are a `float3` with `x`, `y`, and `z` values and `float4` with `r`, `g`, `b`, and `a` values.
>
> You can also perform mathematical operations on these values. For example, if you have two `float3` variables called `vert1` and `vert2` and you multiply `vert1` and `vert2`, then the resultant `vert3` will be created with the `x` values multiplied to create a new `x` value. And similarly, `y` and `z` values will be multiplied with `vert1` and `vert2` to create `vert3` with the new `x`, `y`, and `z` values.

After the return type, we specify the name of the function.

The function takes in two attributes. The double rectangle bracket signifies that it is an attribute. Attributes are like properties. Here, we pass the `vertexArray` through the buffer we created. In the attribute, we pass the vertex array through the buffer at index `0`. Later, you will see that we assign our `vertexBuffer` an index value, which refers to the `0` position here, so that the shaders knows which buffer is the vertex buffer.

The next attribute that the function takes in is the vertex ID. This is generated automatically, depending upon how many vertices we pass in. We pass in three pairs of *x* and *y* coordinates, so three vertex IDs will be generated for this.

Next, in the function it returns a `float4` vertex for each of the vertex IDs. As we have to return a float 4, we add the extra `0` and `1` at the end. You might be wondering how we are returning four values when there are only returning three values: `vertex_array[vid]`,`0`, and `1`. In shader languages, you can club `x` and `y` in a single variable. Here `vertex_array[vid]` is one variable, but it actually holds two objects in it, that is, the `x` and `y` values for that coordinate.

We then create the function for the fragment shader. In a similar way to the vertex shader, we start with the shader type followed by return type and then provide the name of the function. We are not passing anything into the function yet. It does return a `float4` value. As it is a fragment shader and fragment shaders are used for making pixel manipulation, the four values here are the RGBA values of color. So here, the triangle that we will be drawing will be purple in color. If you want all four colors to be of the same value, we can perform the following operation:

```
return float4(0.56);
```

This will return all RGBA values as equal to `0.56`. So the triangle will be gray in color and will be transparent as the value of alpha `a` is also at `0.56`. This way of writing values looks very odd as we don't follow this practice in regular mathematics, but with constant use, you will get used to it and, in fact, appreciate it, as it is more convenient for programming shaders.

> **Note**
>
> Fragment shaders can also be called pixel shaders as they are the same for the most part, but make sure that while creating a pixel shader function, you use the `fragment` keyword, otherwise Metal won't understand what you are talking about.

So, we are done with our shader file; let's now continue with our regular code in `GameViewController.swift` file.

We have added the shader function to our shader library. Once the shader is compiled, it is added to the `shader` library so that it can be retrieved later to save effort in compiling the shader again.

Get the library from the device and add the shader functions to it. We also create new constants for getting the vertex and shader functions from the device, which will be required to pass in the render descriptor, as shown in the following code:

```
//library - collection of functions that can be
retrieved by name
let defaultLibrary = device.newDefaultLibrary()
let newVertexFunction =
defaultLibrary!.newFunctionWithName("myVertexShader")
let newFragmentFunction =
defaultLibrary!.newFunctionWithName("myFragmentShader")
```

Next, we create the render pipeline descriptor. First, we have to create a descriptor to assign to a state, later on. So, let's create a new pipeline descriptor, as shown in the following code:

```
//Render Pipeline
let pipelineStateDescriptor =
MTLRenderPipelineDescriptor()
```

In the descriptor, we provide the vertex and shader function and the pixel format to be used, as shown in the following code:

```
pipelineStateDescriptor.vertexFunction =
newVertexFunction
pipelineStateDescriptor.fragmentFunction =
newFragmentFunction
pipelineStateDescriptor.colorAttachments[0].pixelFormat
= .BGRA8Unorm
```

Pixel formats specify the order of the color components, bit depth per component, and data type. There are more than two dozen formats. To know more about the different types of pixel formats, you can visit Apple's documentation at https://developer.apple.com/library/ios/documentation/Metal/Reference/MetalConstants_Ref/#

We then create a `RenderPipeline` state from the descriptor, as shown in the following code:

```
//Render pipeline state from descriptor
var pipelineState: MTLRenderPipelineState!
pipelineState =
device.newRenderPipelineStateWithDescriptor(
pipelineStateDescriptor,
error: nil)
```

A new variable, `pipelineState`, of the `MTLRenderPipeLineState` type is created, and the `pipeLibeStateDescriptor` constant is passed to it.

Next, we create a layer of type `CAMetalLayer` and add it to the current view, so that we can draw the object on it. So, add the following code to prepare the view and add the layer to it:

```
//prepare view with layer
let  metalLayer = CAMetalLayer()
metalLayer.device = device //set the device
metalLayer.pixelFormat = .BGRA8Unorm
metalLayer.frame = view.layer.frame
view.layer.addSublayer(metalLayer)
```

To the `metalLayer` constant, we assign the device, pixel format of layer (the same as what we assigned in the pipeline descriptor), and the frame size, which is equal to the size of the frame of the view (frame size is the same as the screen size). Finally, add `metalLayer` as a sublayer to the current view layer.

This is all that is required for setting up everything. We can move on to the next stage, that is, actually drawing the triangle.

In the next step, we create a render pass descriptor. Before we can create it, however, we need to get a reference to the next drawable texture from the layer as it will be required to pass it to the render descriptor:

```
//get next drawable texture
var drawable = metalLayer.nextDrawable()
```

Next, we create the render descriptor:

```
//create a render descriptor
let renderPassDescriptor =
MTLRenderPassDescriptor()

renderPassDescriptor.colorAttachments[0].texture
= drawable.texture //assign drawable texture

renderPassDescriptor.colorAttachments[0].loadAction
= .Clear //clear with color on load

renderPassDescriptor.colorAttachments[0].clearColor
= MTLClearColor(red: 1.0,
            green: 1.0,
            blue: 0.0,
            alpha: 1.0) // specify color to clear
it with
```

Now, we create a new constant called `renderPassDescriptor`, of type `MTLRenderPassDescriptor`. First, we assign the texture of the drawable layer so that whatever it renders gets drawn in the texture. So, the texture of the drawable layer is passed in.

Next, the load action is called. Once loaded, the layer is first cleared with a color. Then, we pass a color with which the layer will be cleared. Here, we pass a purple color.

With that, our descriptor is ready. Next, we have to render the layer and the triangle. So first, we get the command buffer from the command queue. These are the commands that are stored in the memory:

```
//Command Buffer - get next available command
buffer
let commandBuffer = commandQueue.commandBuffer()
```

All these commands need to be encoded in the machine language through `MTLRenderCommandEncoder`. We pass the `renderPassDescriptor` variable here to encode the render code:

```
//create Encoder - converts code to machine
language
let renderEncoder:MTLRenderCommandEncoder =
commandBuffer.renderCommandEncoderWithDescriptor(renderPassDescriptor)!
```

Next, we have to set the pipeline and vertex buffer state in the encoder. While passing in `vertexBuffer`, we have to pass in the offset and index buffer values. As we created a new buffer, the offset value is `0` and for index, we pass `0`. This index value is what the vertex shader referred and passed in `[[buffer(0)]]` to the value:

```
//provide pipelineState and vertexBuffer
renderEncoder.setRenderPipelineState(pipelineState)
renderEncoder.setVertexBuffer(vertexBuffer,
offset: 0, atIndex: 0)
```

Finally, we can draw the triangle by creating a primitive type. Eventually, all shapes are made of triangles, as we saw in the example of the ship in the first chapter. The number of vertices and their positions define the shape of the object. Here, we are creating a single triangle shape, so we are passing three vertices to create a triangle. This same triangle primitive is used to make squares, cubes, teapots, spheres, and so on:

```
//drawing begin
renderEncoder.drawPrimitives(.Triangle,
vertexStart: 0, vertexCount: 3, instanceCount:
1) //drawin
```

We have finally drawn, so we can end the encoding:

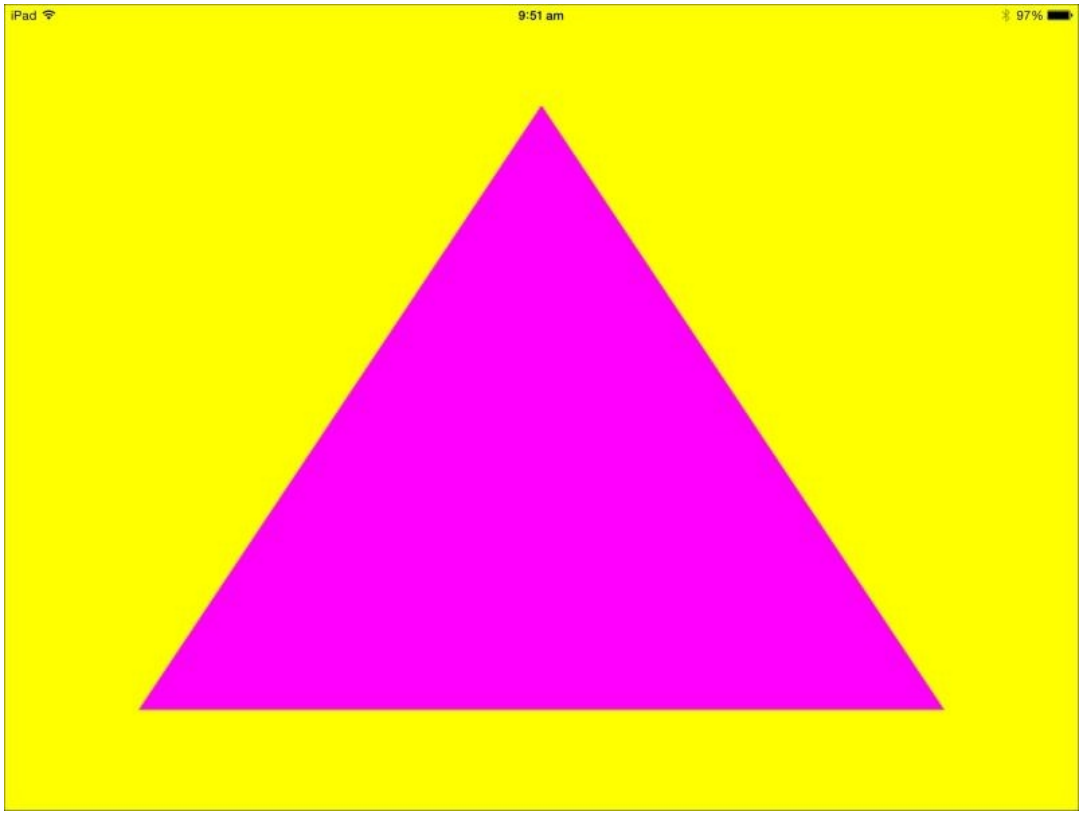
```
//End drawing
```

```
renderEncoder.endEncoding()
```

Although we have drawn the triangle, we still need to present it on the screen. In the next step, we will provide the texture and commit it to the view:

```
//commit to view
commandBuffer.presentDrawable(drawable)
commandBuffer.commit()
```

That is all. Finally, you will be able to see the purple triangle with the yellow background on the screen, as shown in the following screenshot:



Congrats! You've successfully completed it.

All of this is just for drawing the triangle, but I hope you are now able to appreciate the effort. The problem is that if you do any of the steps incorrectly, the chances are that the triangle won't get displayed.

The code is also kept very basic. We can definitely optimize the code by adding a renderer class and creating the vertex array as a separate class.

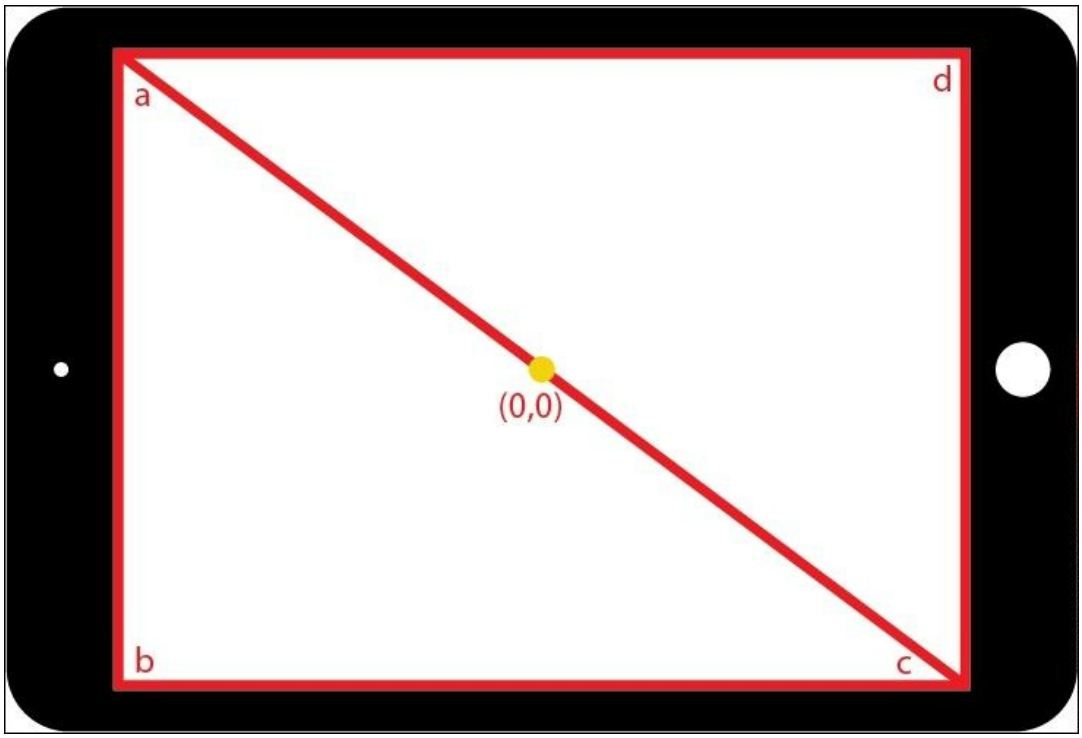
In the next example, we will create a square. This time, we will pass colors for each of the coordinates instead of typing the color value in the fragment shader function.

The colored quad project

To create a colored square, we need to make some changes to `vertexArray`, as we will need to pass six vertices instead of three. We have to pass six vertices because, as you saw earlier, we can draw only in triangles. So, we need three points for the triangle to form the top part of the square and three more points to form its bottom part:

```
let vertexArray:[Float] = [

    -1.0, 1.0, 0, 1,  //a
    -1.0, -1.0, 0, 1, //b
    1.0, -1.0, 0, 1,  //c
    -1.0, 1.0, 0, 1,  //a
    1.0, -1.0, 0, 1,  //c
    1.0, 1.0, 0, 1,   //d
]
```

You will see that the `a` and `c` points are repeated to form the second triangle, because the diagonal of the square are the same points for the first triangle.



Notice that in the vertex array, we are now passing four values per vertex instead of two, as we did in the case of the triangle. This will simplify matters while modifying the shader function.

The values of the coordinates provided are the `x`, `y`, and `z` values, and an additional fourth parameter `w` is also passed in. Like the `z` value in two-dimensional space, it doesn't have much significance yet. Later, when you create a three-dimensional object, the `w` parameter plays a major role. But, as of now, let the value remain `1`.

Additionally, as we saw while creating the triangle, the view is a 2 x 2 rectangle with the origin in the center. As we are passing in coordinates between `1` and `-1` in the *x-y* direction, the rectangle will actually cover the whole screen. If you still want to see the yellow background, change the value `1` to a smaller value, as we did in the case of the triangle. Do not change the values corresponding to `0`.

Similar to passing vertices, we will also pass color values for each of these coordinates as a buffer in the code to create a new array called `colorArray`, as shown in the following lines:

```
let colorArray:[Float] = [

    1, 0, 0, 1, //a
    0, 1, 0, 1, //b
    0, 0, 1, 1, //c

    1, 0, 0, 1, //a
    0, 0, 1, 1, //c
    1, 0, 1, 1, //d
]
```

These are simple RGBA values for the respective coordinates. Here, a means that *red* = `1`, *green* = `0`, *blue* = `0`, and *alpha* = `1`. The values of each of these lie between `0` and `1`. So here, `a` will be all red in color. We can create custom colors by keeping red as `1` and adding more green or blue to the mix, as in the case of `c` and `d`.

Next, in a similar way to how we created a `vertexBuffer`, we have to create a `colorBuffer` of type `MTLBuffer`:

```
let colorBuffer =
device.newBufferWithBytes(colorArray,
                length: colorArray.count *
sizeofValue(colorArray[0]), //sizeof(colorArray)
                options: nil)
```

We pass in the `colorArray` that we created and the size of the whole array.

We don't have to make any changes to the device, layer, or the render pipeline in the code. But we do need to change the vertex and shader functions, as we are going to be passing information on the color.

In any shader language, we can also create our own data types. A new data type called `VertexInOut` is created by using a struct in the shader file. So type the following code in the `Shader.metal` file:

```
struct VertexInOut{
    float4  position [[position]];
    float4  color;
};
```

We create a struct with two `float4` values, of which one is for position and the other is for the color. The position with double square brackets is used to indicate that we will be passing and retrieving the position attribute through the position property.

The `vertex` function is changed, as shown in the following lines of code:

> **Note**
>
> A packed variable means that you cannot access each of the components individually, unlike a regular `float4`. For example, in a regular `float4` that has position data, we can access the `x`, `y`, `z`, and `w` values, but in a packed `float`, we cannot do that.

```
vertex VertexInOut vertexShader(uint vid
[[ vertex_id ]],
                constant packed_float4* position
[[ buffer(0) ]],
                constant packed_float4* color
[[ buffer(1) ]]){

    VertexInOut outVertex;

    outVertex.position = position[vid];
    outVertex.color    = color[vid];

    return outVertex;
};
```

Here, we first tell the `vertex` function type with the `vertex` keyword; we return a type called `VertexInOut` that we created earlier and provide the name of the function name. We provide the vertex ID, the position buffer with index `0`, and the color buffer with index `1` to the function.

In the function, we create a new variable called `outVertex` of type `VertexInOut`, assign the position and color values for each of the vertex IDs, and then return the `outVertex` variable.

We also need to make changes to the fragment shader function, as the color that is passed in through the buffer needs to be applied to the cube:

```
fragment half4 fragmentShader(VertexInOut inFrag
[[stage_in]]){

    return half4(inFrag.color);
};
```

In the fragment shader, we use a `fragment` keyword to specify that it is a fragment shader, to return a `half4` (which is like a `float4` but consumes less memory), and to provide a name for the `shader` function. The function takes in the `VertexInOut` variable. The `[[stage_in]]` part is used to signify that the operation will have to be done on a per pixel basis. In the function, we ask it to return the typecast color value.

Note

More information about `stage_in` can be found at https://developer.apple.com/library/ios/documentation/Metal/Reference/MetalShadingLangua var-qual/func-var-qual.html#//apple_ref/doc/uid/TP40014364-CH4-SW13.

As we added a new color buffer and increased the number of vertices, we have to make changes to the `renderEncoder`, as shown in the following code:

```
//draw - prep drawing

renderEncoder.setRenderPipelineState(pipelineState)

renderEncoder.setVertexBuffer(vertexBuffer,
offset: 0, atIndex: 0)

renderEncoder.setVertexBuffer(colorBuffer,
offset: 0, atIndex: 1)

renderEncoder.drawPrimitives(.Triangle,
vertexStart: 0, vertexCount: 6, instanceCount: 1)

renderEncoder.endEncoding()
```

We set a new vertex buffer and pass the color buffer and the index value of `1`. In `drawPrimitives`, we are still creating a primitive of type triangle, but this time it's with six vertices.

Finally, as we have changed the shader functions, we need to update the names in the pipeline, as highlighted in the following code:

```
pipelineStateDescriptor.vertexFunction =
defaultLibrary!.newFunctionWithName("vertexShader")

pipelineStateDescriptor.fragmentFunction =
defaultLibrary!.newFunctionWithName("fragmentShader")

pipelineStateDescriptor.colorAttachments[0].pixelFormat
= .BGRA8Unorm
```

Now that it is all built, we can see the colored square, or quad, as it is generally known:



You might be wondering that when you passed in four colors for only the four vertices, how come the whole screen is colored and the colors are also merging.

The vertex shader function gets called depending upon the number of vertices passed in. So the vertex shader function got called six times. On the other hand, the fragment shader function is actually called as many

times as is required to fill the area generated by each triangle primitive. The colors are interpolated between the points, depending upon the distance from a coordinate. For this example, the bottom left coordinate was passed the color value of green. As it moves away from the bottom-left corner, the color slowly merges with other colors, thus reducing the value of green. So at the center, we have a mix of all the colors. This is a general feature of any fragment shader.

The texture quad project

For the next project, we will create a textured quad object. This is the basic building block of a sprite class. Here, we will be taking an image and pasting it on top of the quad or square that we created in the earlier project.

The reason why I said that it is a basic building block for a sprite class is that we won't be able to move, rotate, or scale the sprite; we will just be displaying the sprite on the screen.

Going back to referring to the process of adding a sprite to any quad in a similar way to adding wallpaper to wall, we can add the wallpaper right-side up or upside down on the wall. Similarly, while adding images to quads, we have to specify which way is up, otherwise the sprite will be pasted upside down or sideways on the quad.

For this, we have to pass in one more array of coordinates, which are called texture coordinates. The texture coordinates are different than vertex coordinates such that the vertex coordinates are with reference to the screen coordinates system, with the center of the screen being the origin. Refer to the image provided.

> **Note**
>
> We are working in 2D, so we can call it the screen coordinates system, which is the same as the world coordinate system for convenience for now, but in 3D it is actually the world coordinate system.

The texture coordinate system is with respect to each quad or rectangle. Moreover, the top left of the quad is the origin for the texture coordinate system. So, for moving the quad around the screen, you will change the values in the vertex array. The texture coordinates will be changed to actually move the image around within the quad.

So, with all that theory out of the way, let's create a new array called `textureCoordsArray`, as shown here. But before that, change `vertexArray` and reduce the size of the quad that we created earlier, so that we can have a better understanding of what is happening with the texture coordinate:

```
let vertexArray:[Float] = [

    -0.75, 0.75, 0, 1,  //a

    -0.75, -0.75, 0, 1, //b
```

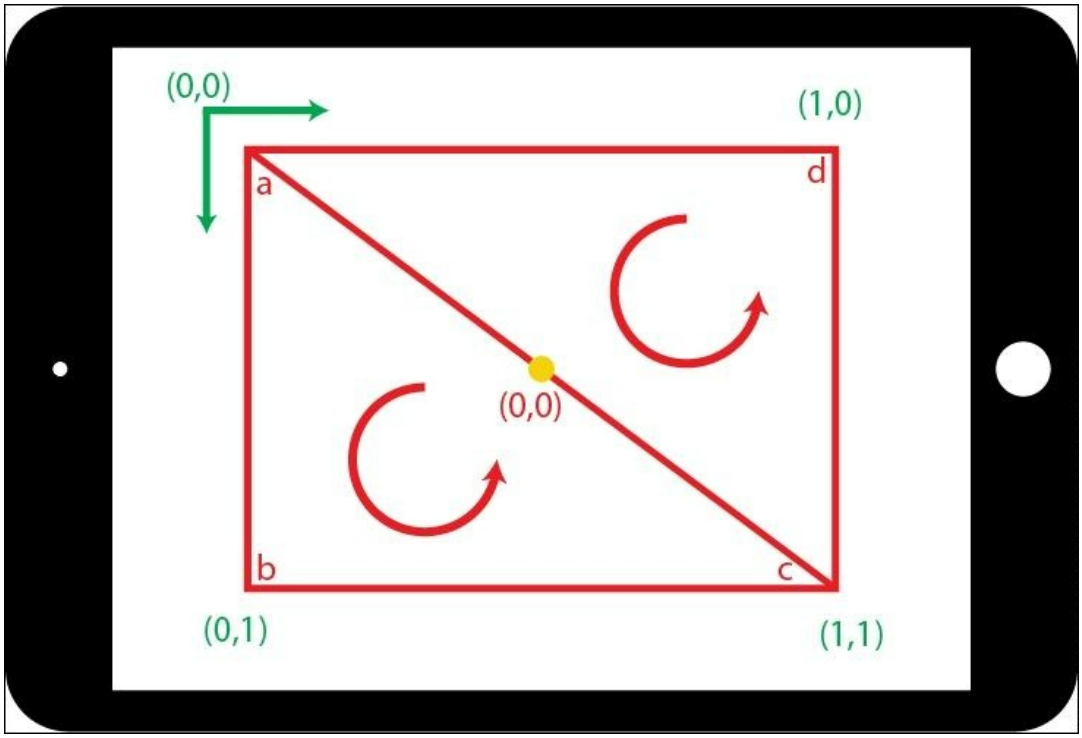
```
        0.75, -0.75, 0, 1,  //c

        -0.75, 0.75, 0, 1,  //a

        0.75, -0.75, 0, 1,  //c

        0.75, 0.75, 0, 1,   //d
    ]
```

Next, add the texture coordinate array:

```
    let textureCoordsArray:[Float] = [

        0.0, 0.0, //a

        0.0, 1.0, //b

        1.0, 1.0, //c

        0.0, 0.0, //a

        1.0, 1.0, //c

        1.0, 0.0  //d
    ]
```

While creating a textured quad, there are two things that we need to pay close attention to:

- The `vertex` array provided for each triangle set needs to be in the anti-clockwise direction. So, for the first three position vertices in the vertex array, the order needs to be `abc` and `acd` respectively for both triangles, otherwise the texture will not be displayed properly.
- The order of the `vertex` array and texture coordinate needs to be the same; if the vertex array order is `abc` and `acd`, then the texture coordinate order needs to be specified in the same order.



In the preceding image, the red letters **a**, **b**, **c**, and **d** denote the vertex coordinates, and the green coordinates denote the texture coordinates. The red origin is the origin of the vertex coordinate, and the green origin is the origin of the texture coordinate.

With the texture coordinates ready, we have to create a texture buffer of the type `MTLBuffer` and pass in the texture coordinate array into it, along with the size of the array. This is similar to how we created buffers earlier for the respective arrays:

```
//initialize textureCoordBuffer
let textureCoordBuffer: MTLBuffer =
device.newBufferWithBytes(textureCoordsArray,
length: textureCoordsArray.count *
sizeofValue(textureCoordsArray[0]),
options: nil)
```

Next, we have to load the texture that we want to paste onto the quad. For this, we import the `Bg2.png` files that we used in the SpriteKit project. Add the following lines of code at the end of the initialization stage.

First, we get the file from the local bundle location. We get the path to the file by passing the name of the file along with the extension. The data has to be then retrieved using the `NSData` method, in which we pass the path, as shown here:

```
//get texture
```

```
        let path =
NSBundle.mainBundle().URLForResource("Bg2",
withExtension: "png")
        let data = NSData(contentsOfURL: path!)
```

Next, we get the image from the data and store it in a constant of type `UIImage`:

```
        let image = UIImage(data: data!)
```

Then, we get the width and height and specify the color space for the image. A color space decides how the colors are to be interpreted. There are other ways in which colors can be stored, apart from the RGBA values; for example, we can provide colors in the CMYK format. As the color values in the image are specified in RGB, we have to specify it here.

The width and height are obtained using the `CGImage` class, so we convert the image from the `UIImage` type to the `CGImage` type in this step:

```
        let width =
CGImageGetWidth(image?.CGImage)
        let height =
CGImageGetHeight(image?.CGImage)
        let colorSpace =
CGColorSpaceCreateDeviceRGB();
```

To store each pixel of the image, we need to specify the size of the data of the whole image, so memory is created for the data to be written into. Each pixel takes up 4 bytes of memory space. So to get the memory occupied by the whole image, we have multiply the width with the height and then multiply these values by four, which will give the data value for the bitmap image. This value is stored in a `bitmapData` constant as shown here:

```
        let bitmapData = calloc(height * width *
4, UInt(sizeof(UInt8)))
```

As mentioned earlier, we assign the number of bytes per pixel in a constant called `bytesPerPixel`. Also, we create a `bytesPerRow` constant to get the number of bytes per row:

```
        let bytesPerPixel: UInt = 4
        let bytesPerRow: UInt = bytesPerPixel *
width
```

We also need to specify how many bits does each component in a pixel take. A pixel is made up of `R`, `G`, `B`, and `A` values. For storing each value, we require 8 bits each. So for storing each RGBA value, we need 32 bits in

total. As we'll be requiring the value of bits per pixel component later on, we store the value in a constant called `bitsPerComponent` here:

```
let bitsPerComponent: UInt = 8
```

Next, we create a context that creates the environment by storing all the required data of the image. To the context we have to provide `bitmapData`, width and height of the image, bits per component, bytes per row, and `colorSpace` and `bitmapInfo` at the end.

In the bitmap information, we specify if there is an alpha channel in the image, the location of the alpha channel, and if the values are integers or float values:

```
let context =
CGBitmapContextCreate(bitmapData,
                      width,
                      height,
                      bitsPerComponent,
                      bytesPerRow,
                      colorSpace,

 CGBitmapInfo(CGImageAlphaInfo.PremultipliedLast.rawValue))
```

Next, we get the rectangular size of the image. We create a `rect` variable of type `CGRect` and pass in the origin and the width and height of the image, as shown in the following lines of code:

```
let rect = CGRectMake(0.0,
0.0,
                      CGFloat(width),
                      CGFloat(height));
```

Next, we get the image and store the RGBA data into `bitmapData` through the context. We first clear the context and then pass in the context, the rectangle, and the image:

```
CGContextClearRect(context, rect);
CGContextDrawImage(context, rect,
image?.CGImage);
```

For loading the texture into Metal, we need a texture descriptor, which stores all the relevant information. We create a new constant called `textureDescriptor` and a texture descriptor with the pixel format of RGBA8 uniform normal and pass in the width, height, and the data if we want the image to be **mipmapped**.

Mipmap, as we saw in SceneKit, will create a lower resolution of the image and display it if the camera is far away from the texture to reduce the workload on the system, as shown here:

```
        let textureDescriptor  =
MTLTextureDescriptor.texture2DDescriptorWithPixelFormat(.RGBA8Unorm,
            width: Int(width),
            height: Int(height),
            mipmapped: false)
```

We create a texture of type `MTLTexture` and pass in the texture descriptor:

```
        let texture: MTLTexture =
device.newTextureWithDescriptor(textureDescriptor)
```

Finally, we use the replace region function of type `MTLTexture` to replace the pixels with the image data that is stored in `bitmapData`. We pass in the region that is basically a rect and then we specify `mipmapLevel`, which is kept at `0` and sliced to determine which surface of the quad we will paste the image onto, as we just have one quad we specified as `0`. If we had more quads, such as a cube, we would have to specify values other than `0` for other faces. Next, we pass the `bitmapData`, which has the RGBA of the image stored in it; next, we pass in the `bytesPerRow` and `bytesPerImage` values in, as shown in the following lines of code:

```
        let region = MTLRegionMake2D(0, 0,
Int(width), Int(height))

        texture.replaceRegion(region,
            mipmapLevel: 0,
            slice: 0,
            withBytes: bitmapData,
            bytesPerRow: Int(bytesPerRow),
            bytesPerImage: Int(bytesPerRow *
height))
```

As we are passing in a texture coordinate buffer and we need a texture to be drawn, we have to make some changes to the shader file. So, go to the shader file.

First, change the `VertexInOut` struct, as shown here:

```
struct VertexInOut
{
    float4  position [[position]];
    float4  color;
    float2 m_TexCoord [[user(texturecoord)]];
```

```
};
```

In the highlighted code, the user keyword is used in shaders while specifying attributes that are user-defined:

```
vertex VertexInOut vertexShader(uint vid
[[ vertex_id ]],
                        constant float4* position
[[ buffer(0) ]],
                   constant packed_float4* color
[[ buffer(1) ]],
          constant packed_float2* pTexCoords
[[ buffer(2) ]])

{
    VertexInOut outVertex;

    outVertex.position =  position[vid];
    outVertex.color    =  color[vid];
    outVertex.m_TexCoord = pTexCoords[vid];

    return outVertex;
};
```

As the `textureCoordinate` buffer needs to be passed in, along with the position and color information, we pass it with the buffer index `2`. Also, while returning `outVertex`, we will assign the texture coordinate for that vertex ID in the function.

Next, we also need to make changes to the `fragment` shader function, as we will be passing the texture into the fragment shader. So draw it on the quad, as shown in the following lines of code:

```
fragment half4 texturedQuadFragmentShader(
  VertexInOut inFrag [[ stage_in ]],
                       texture2d<half>  tex2D
[[ texture(0) ]])
{
    constexpr sampler quad_sampler;

    half4 color = tex2D.sample(quad_sampler,
inFrag.m_TexCoord);

    return color;
}
```

Along with the `stage_in` parameter, which we passed last time, we are also going to be passing the texture into the fragment shader with an index value of `0`.

In the function to actually pick the color, a sampler is used. The sampler will decide how to pick the color from the texture that is passed in. The sampler is set as a `constexpr`, or a constant expression that is similar to a `const` type.

The sampler picks the colors, depending on the texture coordinates from the texture 2D that were provided using the `text2D.sample` function. The resultant color is stored and returned by the fragment shader.

Next, in `renderEncoder`, add the following highlighted lines:

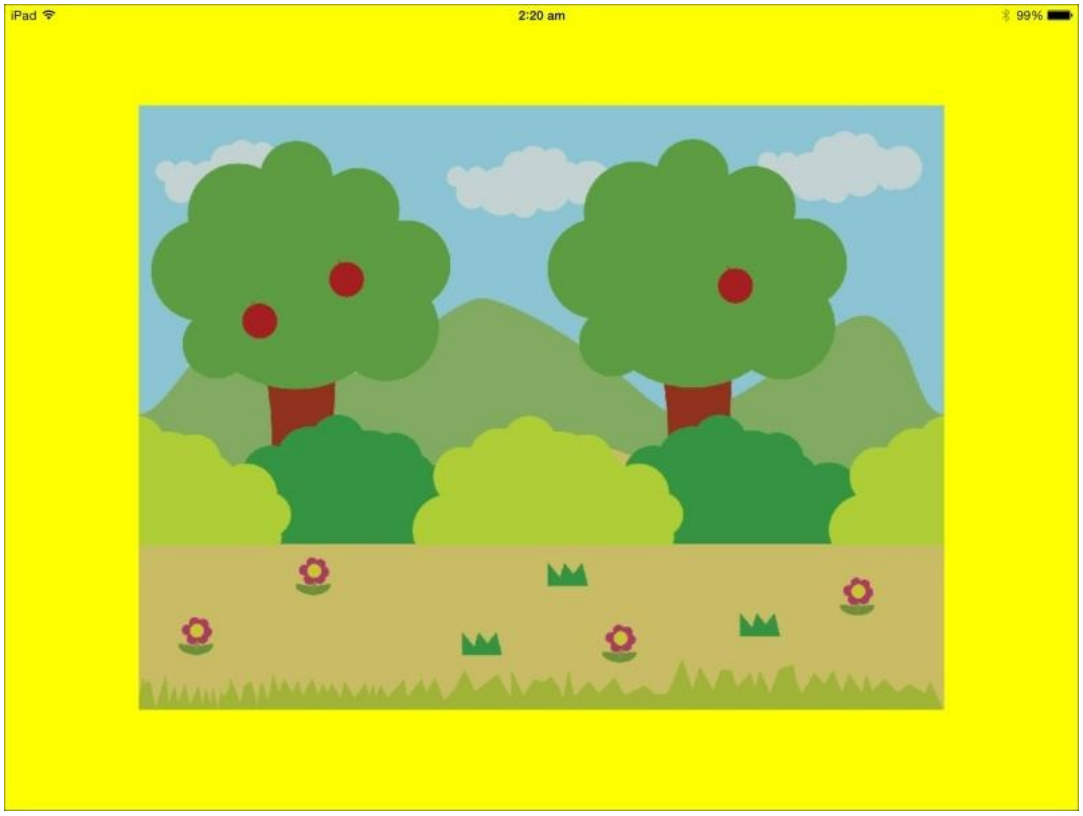
```
renderEncoder.setVertexBuffer(colorBuffer,
offset: 0, atIndex: 1)

renderEncoder.setVertexBuffer(textureCoordBuffer,
offset: 0, atIndex: 2)
renderEncoder.setFragmentTexture(texture,
atIndex: 0)

renderEncoder.drawPrimitives(.Triangle,
vertexStart: 0, vertexCount: 6, instanceCount: 1)
```

Notice that although `vertexBuffer` and texture have the same index, they are treated differently, as `vertexBuffer` is one type of buffer that is separate from a texture. As the `textureCoordinate` buffer is a type buffer, we have to pass in an index value of `2` as we are already passing `0` and `1` for the vertex and the color buffer respectively.

Finally, make changes to the pipeline descriptor to use the newly created fragment shader, as shown here:

```
pipelineStateDescriptor.vertexFunction =
defaultLibrary!.newFunctionWithName("vertexShader")

pipelineStateDescriptor.fragmentFunction =
defaultLibrary!.newFunctionWithName("texturedQuadFragmentShader")

pipelineStateDescriptor.colorAttachments[0].pixelFormat
= .BGRA8Unorm
```

And finally, there is one more thing to do, which is to build it and run!



With this, we have gone full circle in this book. The first thing that you learned in 2D SpriteKit game development was how to add an image to the scene. We typed in three lines of code and the image appeared on the screen.

In reality, we have to do all this to get a simple image onto the screen. But, as I said earlier, this is just the beginning; we still haven't seen how to move, rotate, or scale the quad.

We haven't yet looked at the depth buffer, which decides whether a portion of the image needs to be drawn on screen or not, depending on any other object that is in front of the current object.

Also, we have only looked at 2D. We haven't yet created a cube, for which additional vertices will be required. Deeper knowledge of algebra, trigonometry, and matrices will be required. Additional understanding of projections and model space, world space, view space, and screen space will be required, all of which are integral parts of graphics programming.

It's needless to say that these topics are way beyond the scope of this book. In fact, a whole book could be dedicated to graphics programming using Metal, while another book could be dedicated to using Metal Shader language to create cool effects.

For learning Metal, I recommend learning OpenGLES first as it has been there for so many years. Once you have a good understanding of that, you can put the knowledge into experimentation with Metal.

For learning OpenGLES, I recommend *Building Android Games with OpenGLES*. Although it teaches development for Android as OpenGLES is cross-platform, you can use the same concepts for iOS game development. The link to the video can be found at https://www.packtpub.com/game-development/building-android-games-opengl-es-video.

Summary

In this chapter, you saw how to create a simple triangle, quad, and a texture quad, and how to display these to the screen. You scratched the surface of graphics programming and learned the meaning of terms such as vertices, buffers, textures, and shaders.

This is merely the start of the learning process; there is still a lot to learn as graphics programming is a vast and deep subject, with courses specifically designed for it.

I hope this chapter has generated some interest for the subject in you; if nothing more, I think you will at least have some appreciation for the people who sit for hours and develop frameworks and engines for you so that you can make the game of your dreams, without having any knowledge of the subject.

Talking about dreams, in the next chapter, you'll see how to finally publish a game on the iOS App Store.

Chapter 10. Publishing and Distribution

So, the moment is finally here. We have put in our time and effort and made this awesome game. Now, we want the world to know about our creation.

The road to publishing a game is actually quite straightforward, provided you follow the steps correctly. Just as we created a developer profile so that we can run the game on the device, we also need to create a publisher profile in order to publish the game on the App Store or run it on any device other than our own.

The next step will be to create the app in the iTunesConnect portal. This is where you will give the app a name, a description, an icon, and screenshots. Finally, you will upload the app file from the system to the App Store.

You will also have the opportunity to decide whether you want to give the game away for free or charge a price.

Although achievements and leaderboards are not covered in this book, you can definitely add them to the game, along with ads, so that you can monetize the game.

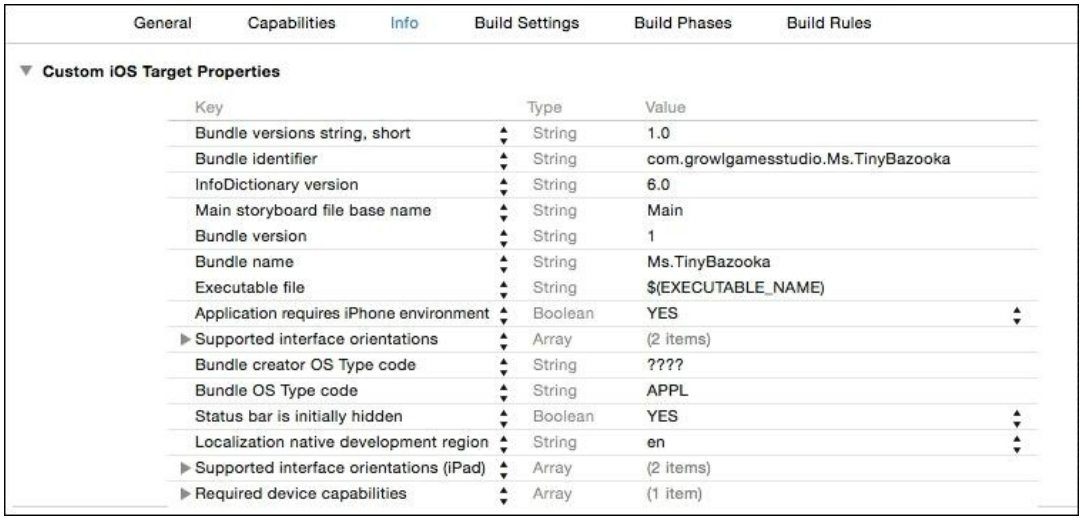
So, let's get started with getting the game ready for being published. Here's what you will learn in this chapter:

- Getting the app ready
- Distribution certificate
- The iTunesConnect portal
- Creating the app
- Ad hoc installation
- References
- Alternatives
- Final remarks

Getting Ms. tinyBazooka ready

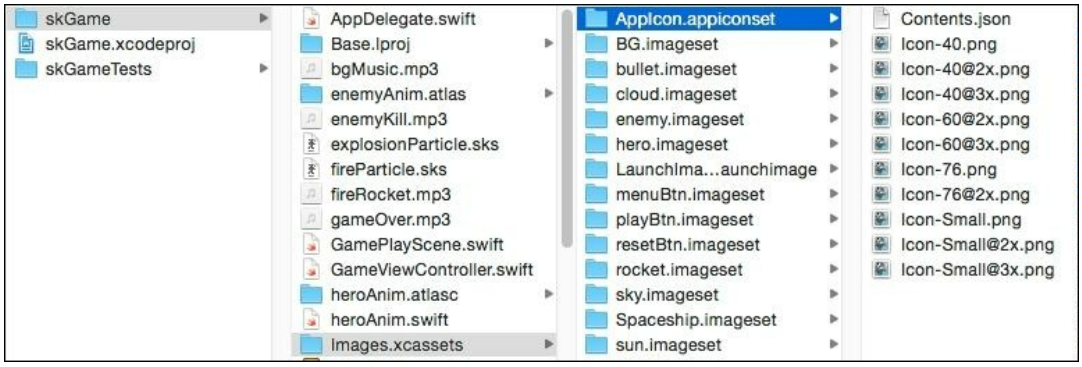
First, we will make some changes to the app. Click on the project root and then on the information screen of the project. Here, change the bundle identifier by double-clicking on the area adjacent to it. For example, I changed the bundle name to `com.growlgamesstudio.Ms.TinyBazooka`. Also, change the bundle name to reflect the name of the game; otherwise, the name of the project will be displayed, which we don't want.

Make sure that the bundle version is set to `1.0`. This is the release version. Later, when you make updates to the game, you will need to change the number for every new update you make. The rest of the items can remain as default.



Next, we will change the icons so that instead of the default icon, we can use our own icon for the game. In the project directory in the hard disk, navigate to the `images.xcassets` folder and replace the `AppIcon.appiconset` folder, which is the one currently present, with the one provided in the resources folder of the chapter.

Now, if you build the game, the name will reflect properly, along with the new icon that you have provided.

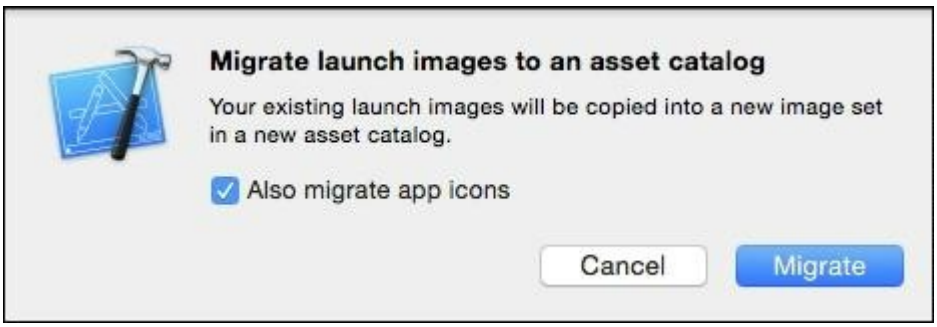


This will also be reflected in your project if you click on **AppIcon** in the `images.xcassets` folder in your project.



This has the icons specified for all iOS devices. So later, when I want to make a universal app, I don't have to specify icons for the different iPhone and iPod devices.

Finally, we will also change the launch image, which shows a white screen as of now, with the name of the game and some copyright information. Click on the project root. In the **General** tab, which is under the **App Icons and Launch Images** section, click on the **Use Assets Catalog** button next to **Launch Image source**.



Next, click on **Migrate**. Then, click on the small arrow pointing to the right to the launch image dropdown:



This will take you to the launch image in the assets catalog. Since we are developing only for iPad landscape mode, we can drag-and-drop the background images into the **1x** and **2x** slots of the landscape section. Make sure that the **Launch Screen File** field is empty in **App Icons Source** and **Launch Images Source**.

Finally, we will remove the debug information that shows the number of nodes added and the FPS. So, go to the `GameViewController` class and comment or delete these two lines:

```
// Configure the view.
let skView = self.view as SKView
//skView.showsFPS = true // comment this line
//skView.showsNodeCount = true //Comment this line
```

Now, when you launch the game, you will see the background image instead of the other white background. The app will show the proper icon, and the debug information will be removed.

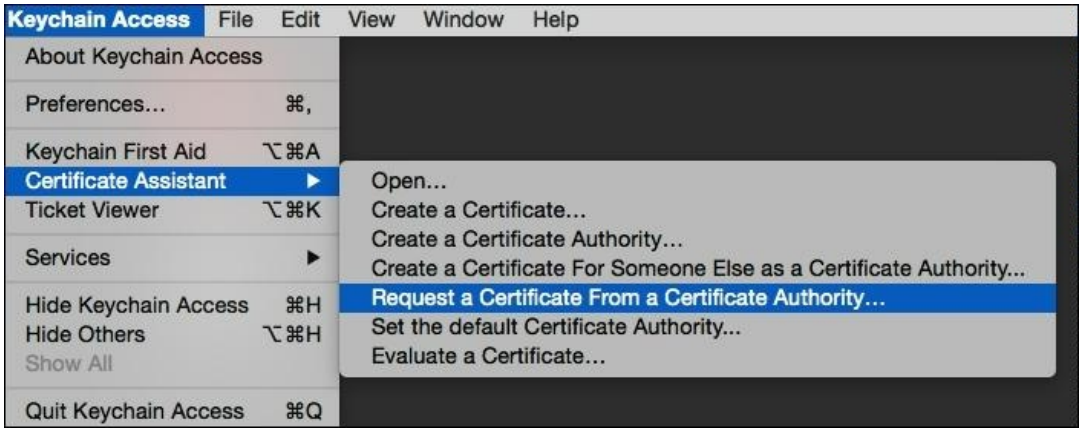
Generating the distribution certificate

To distribute the app, we need the distribution certificate. This is included with all the apps that are released on the App Store because without it, an app cannot be installed on any device.

For getting the distribution certificate, the steps are similar to setting up the developer certificate. Go to the Apple developer portal and sign in with the developer account you created in the first chapter at https://developer.apple.com/devcenter/ios/index.action.

On the right-hand side of the screen, click on the **Certificates, identifiers and profiles** link under **iOS Developer Program**. Then, click on **Certificates**. Next, click on the **+** sign in the top-right corner of the screen.

In the next screen, select **App Store and AdHoc** and click on **Continue**. In the next step, create a **Certificate Signing Request** (**CSR**) file, as the portal requires it to be uploaded to it. Go to Launchpad, search for `KeyChain`, and open it.



Then, navigate to **KeyChainAccess** | **Certificate Assistant** | **Request a Certificate Authority**.



Enter your e-mail address and name, select **Saved to disk**, and click on **Continue**. Save the file on the desktop in the next step.

On the developer website, click on **Continue**. In the next step, click on **Choose file** and **Generate**. Next, download the certificate created. Double-click on the file after the download to install the certificate.



On the website, click on **Done**. Now we are ready distribute the game.

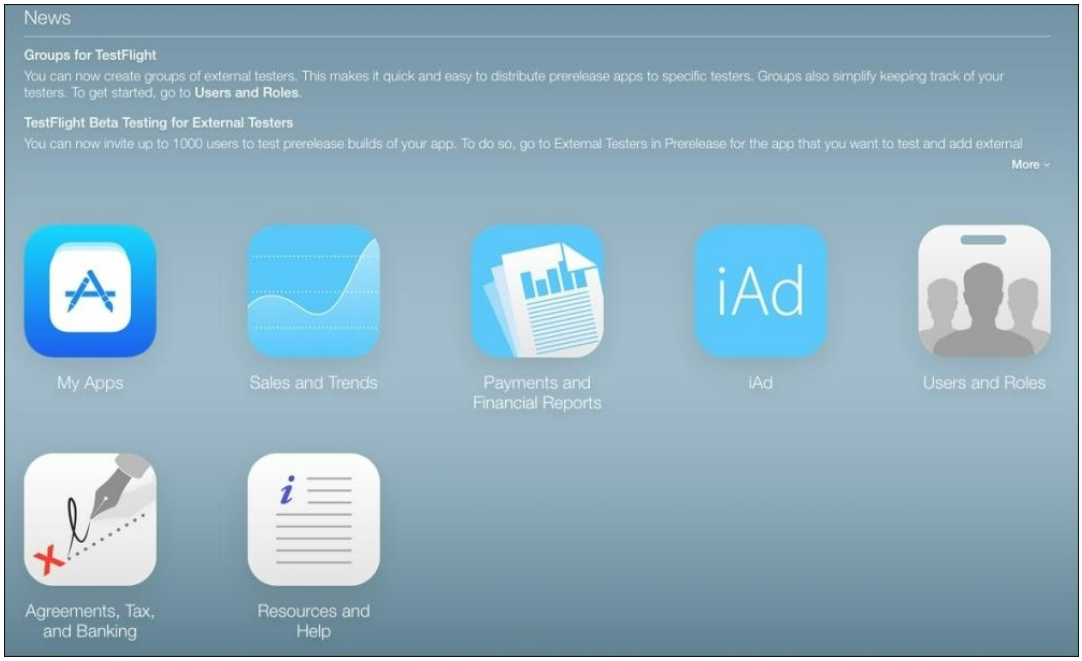
The iTunesConnect portal

To create the app, we will need to go to the iTunesConnect portal. In the developer portal, click on the iTunesConnect link, or go to https://itunesconnect.apple.com.

You might be asked to sign in, so do it using the developer ID and password.

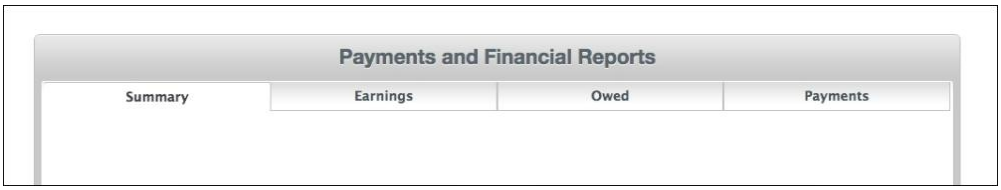


Once in, you should see a page like this:



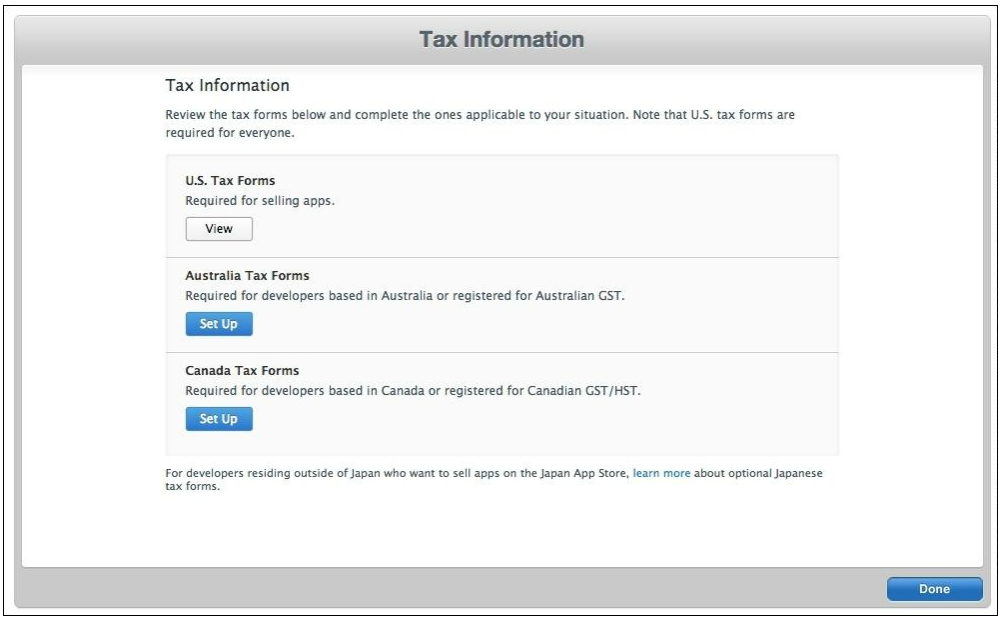
The various parts of this page are as follows:

- **My Apps**: This is where we will create new apps and update existing apps
- **Sales and Trends**: This will show the number of units of each app you have sold over a period of time. You can check the sales per territory, platform, category, content type, transaction type, and so on.
- **Payment and Financial Reports**: Here, you will get a report of the summary of the items sold, earnings, amount owed by Apple, and payments made by Apple to you:



- **iAds**: Apple has its own ad network from which you can pool and show ads in your game. You have to use the iAds workbench to configure iAds.
- **Users and Roles**: Here, information regarding the roles of individuals involved with the account can be updated. The person who created the Apple ID needs to input their details here. Later, if you add testers for the game, you'll need to include their details here, as well.
- **Agreements, Tax and Banking**: As the name suggests, this is where you will need to sit with an attorney, go through Apple's agreements, and digitally agree and sign all of them. You will also be providing your banking information here so that when you create a paid app, you can be paid for the sale of the app. The **Tax** section will have tax forms that will need to be filled in. Primarily, there are forms for US,

Canada, and Australia, depending on your location. If you are not from any of these three countries, then you will need to fill in the US tax forms.



- **Resources and Help**: Here, you have links to resources such as forums and FAQs that you can refer to if you need answers to any questions. You can also click on **Contact Us** and send an e-mail with your query. Apple's personnel will reply to you.

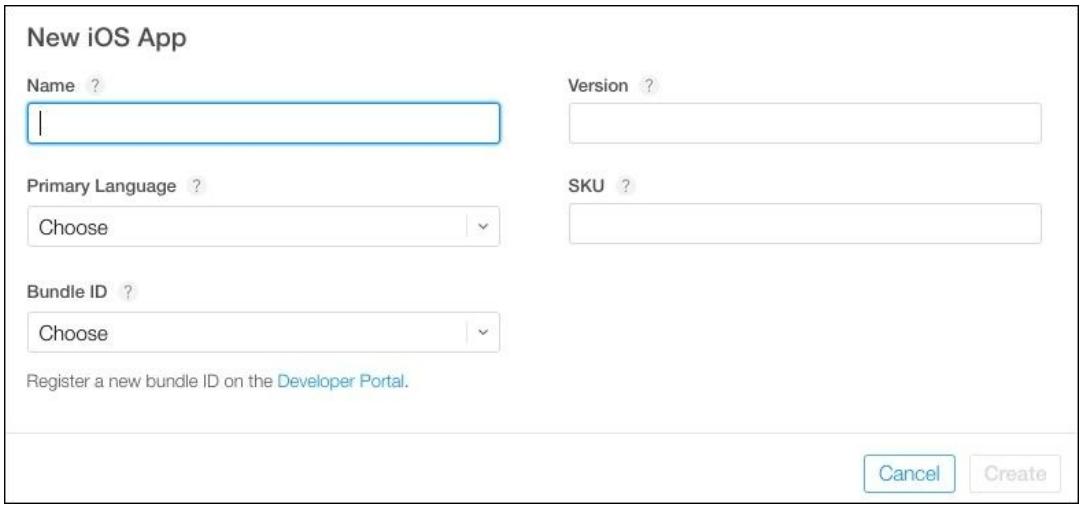
With the basic introduction out of the way, we can click on **My Apps** to publish and distribute the Ms. TinyBazooka game.

Creating the app

To create a new app, in the **My Apps** section, click on the **+** sign in the top-left corner of the page. Then, click on **New iOS App**, as shown in this screenshot:



The following window will open up:

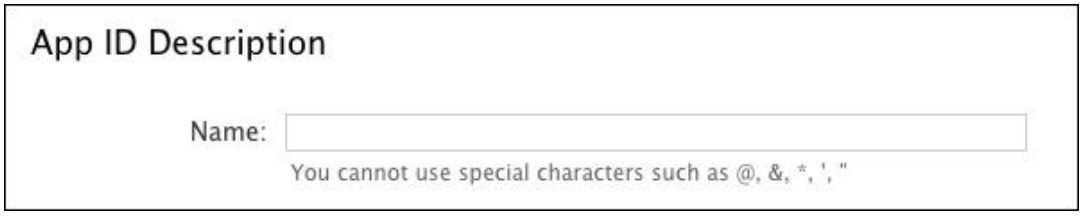


The various parts of this page are as follows:

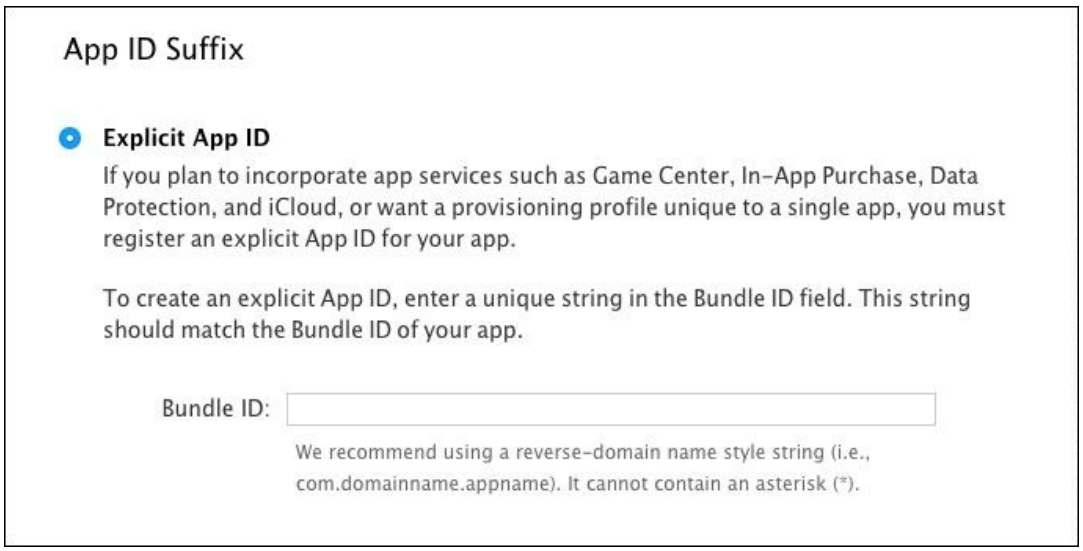
- **Name**: This is the name of the app as it will appear in the App Store. If the name you are suggesting has already been taken, then you will need to give an alternate name and also change it in the info file in Xcode.
- **Version**: This needs to be the same as the version number you provided in the information in the project. Since our game is Version 1.0, we type in this value.

- **Primary Language**: Choose the language used in the game. You can choose English.
- **SKU**: This is the stock keeping unit number you need to provide. You can enter `001`.
- **Bundle ID**: This is what the app will be recognized by on the app store. We need to register the `com.growlgamesstudio.Ms.TinyBazook` bundle ID in the developer portal so that when we upload the app to iTunesConnect, it will link up the app to the bundle ID.

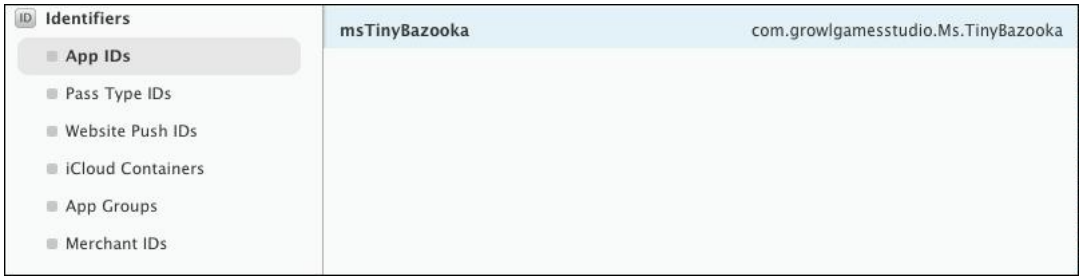
Click on the developer portal link under the Bundle ID drop-down menu.



In **App Id Description**, type a name for the app; this is just for reference. Next, in **App ID Suffix**, type the app bundle ID exactly as you typed in the info. So, in this case, I have to type `com.growlgamesstudio.Ms.TinyBazooka`. Once again, this is unique for each app.



Click on **Continue** to proceed. On the next page, after confirming your app ID, click on **Submit**. The registration is now complete. Click on **Done** at the bottom of the window.

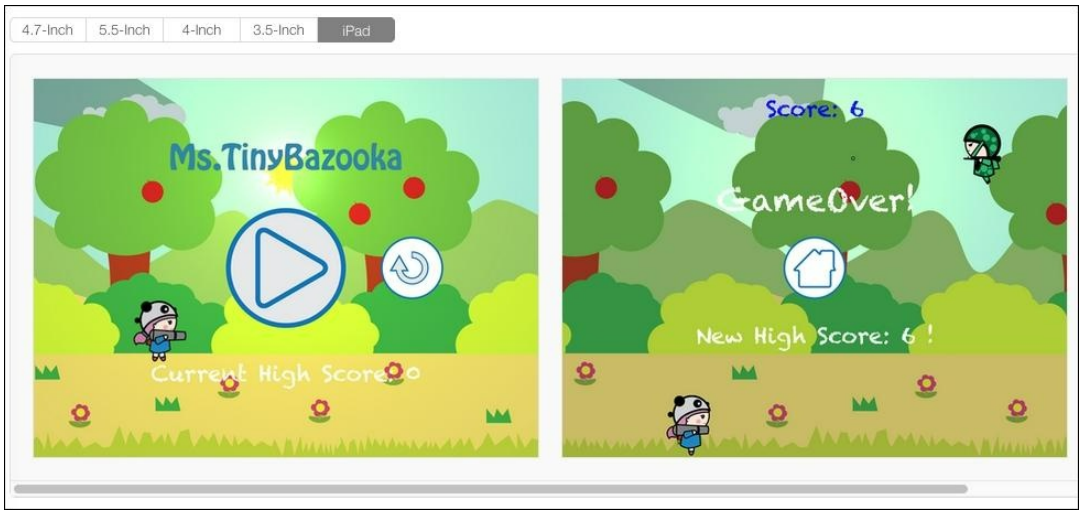


You will now see that the bundle ID has been added to **App IDs** under **Identifiers** in the developer portal.

Go back to the **App Creation** window and in the **BundleID** drop-down list, you will be able to select the tinyBazooka bundle ID. Select it and click on **Submit**. This might take a couple of minutes to update and reflect in the drop-down menu.

You will now be greeted with the app page. Here, we will add the description and screenshots, and upload the binary file. Next to the **Submit for Review** button, there is a **Save** button, which will light up whenever you make a change. The **Save** button is your friend, so use it as often as possible so that you don't lose any changes that you make.

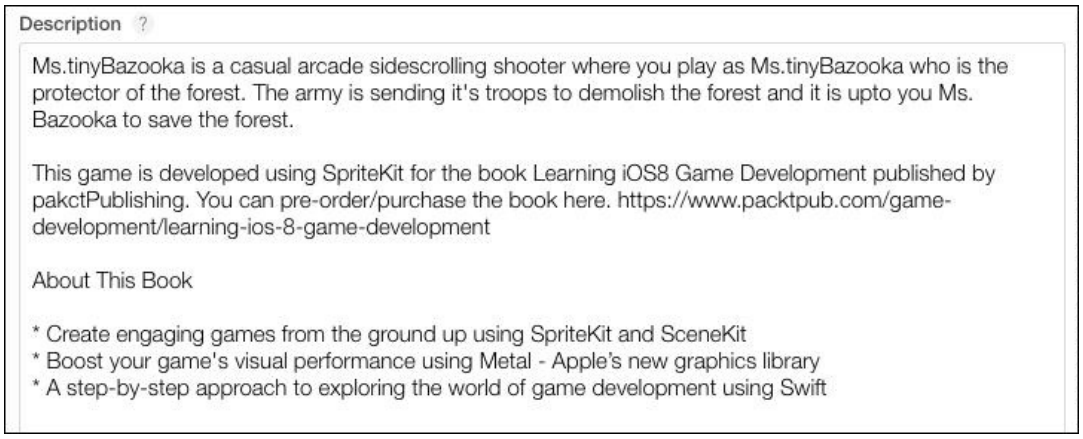
First, we will add some screenshots. You can create a screenshot from your device by clicking on the power and home buttons at the same time on the device. Take five screenshots, plug the device into the Mac, and transfer the images to the desktop. Then you can drag the files from the desktop to the space on the iTunes app page. You can drag-and-drop each file separately or all files at the same time. Click on the **iPad** tab at the top, as we are developing the game only for the iPad, for now.



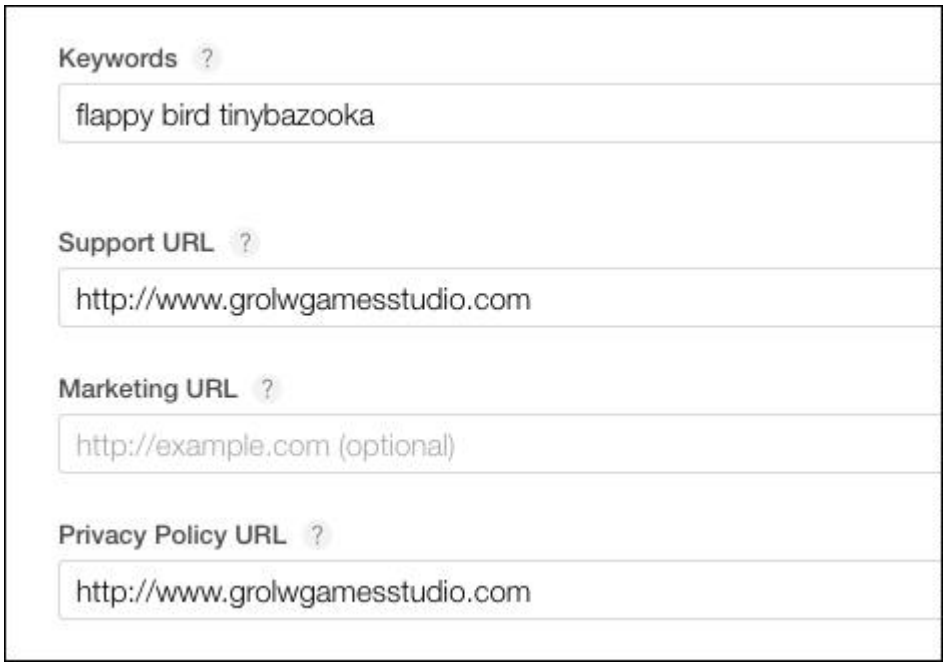
Next, we will have to provide the name of the app as it will appear on the App Store. So, in the **Name** section, type the name of the app:



We will have to provide a small description of the game. We can highlight the genre, story, and pretty much anything we want to tell users about the app here. You can keep it as long or as short as you want.



To the right of the description, we can provide keywords, support, marketing, and privacy policy URLs. Keywords, the description, and the title are parts of the ranking algorithm, so do good research to make sure that your game gets ranked higher.



Next, we need to upload the icon for the app. The icon shouldn't have rounded corners and alpha. It would also better to have higher resolution;

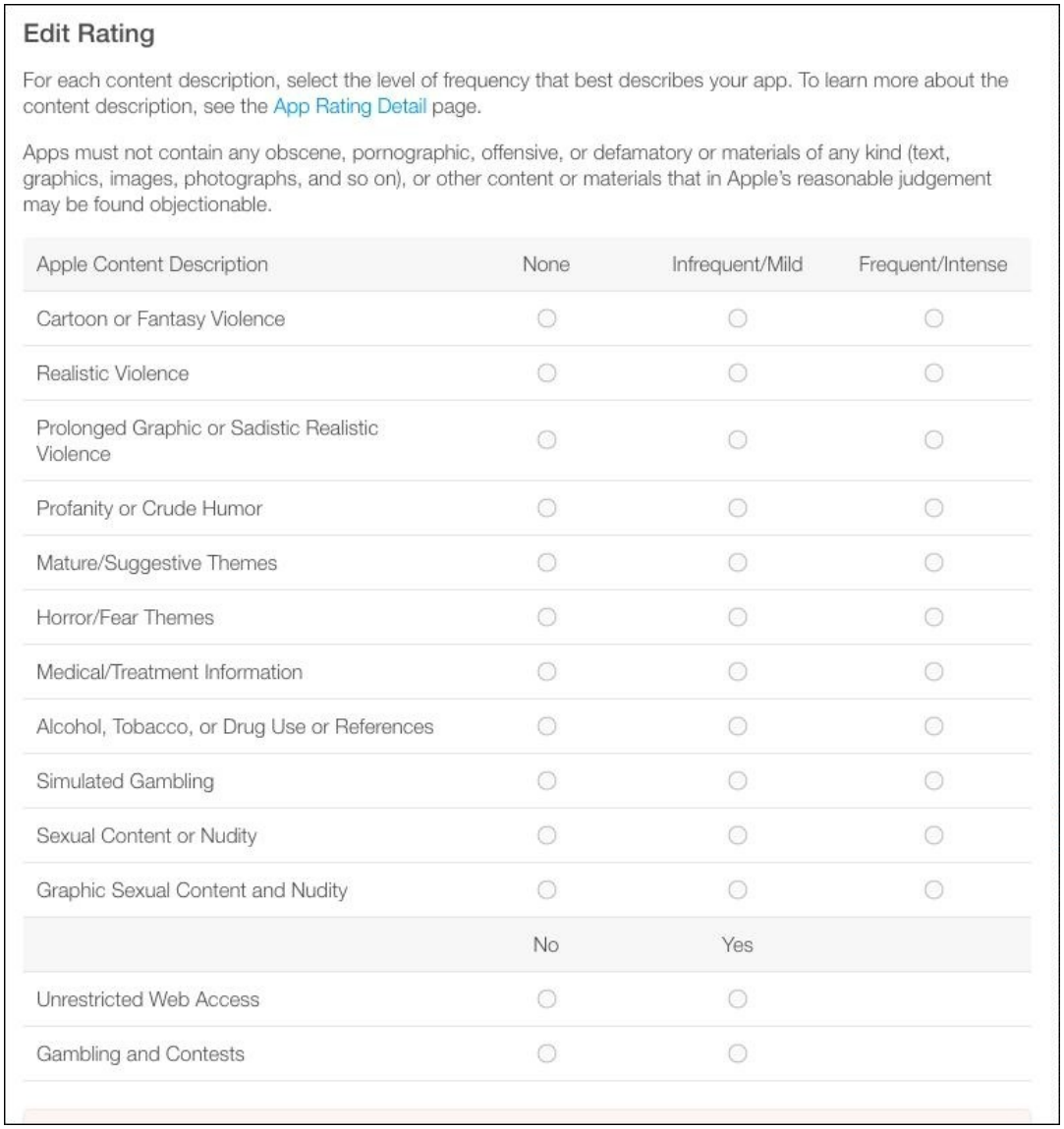
1024 x 1024 would be ideal. A sample icon has been provided in the `Resources` folder for you to test. Drag the icon into the **App Icon** slot.



Now, we have to select the main the category for the app. You can select up to two subcategories. For the main category, choose **Games**. The subcategory mainly depends on the genre of the game. Here, I have chosen **Action and Arcade** as subcategories.



Next, we have to specify the rating for the game. In the **Ratings** section, click on the **Edit** link:

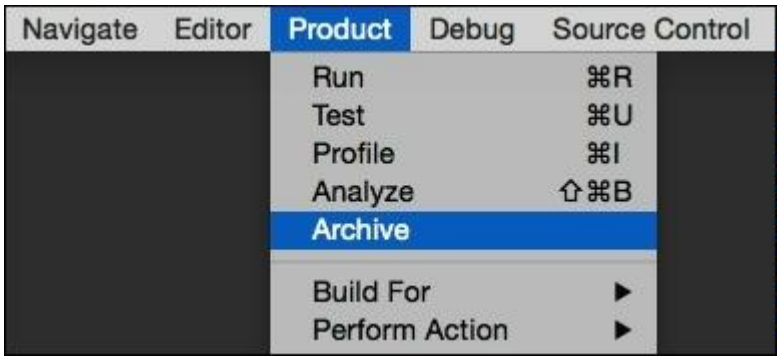


Rate each of the items as **None** if the content described doesn't appear at all in the game. If it does, then select either infrequent or very frequent. Depending on this, the game will be rated for a suitable audience. So, if your target audience is kids, then it should be **None** for the first section and **No** for the last two items ideally. This will make the game rated for ages 4 and above. Otherwise, you are targeting either a teen or an adult audience, so choose carefully.

Next, under **Edit License Agreement**, the standard end user license agreement is selected by default. Otherwise, you can include a custom license:



Now, we have to upload the build of the game. So, open the game in Xcode. Once Xcode opens, on the top menu, select **Product** and then **Archive**.

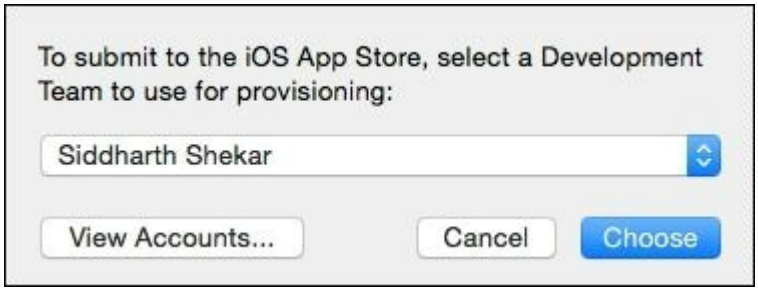


Once the app is archived, the following window will open up. If it doesn't, you can also access it by going to **Window** | **Organizer** | **Archives**:

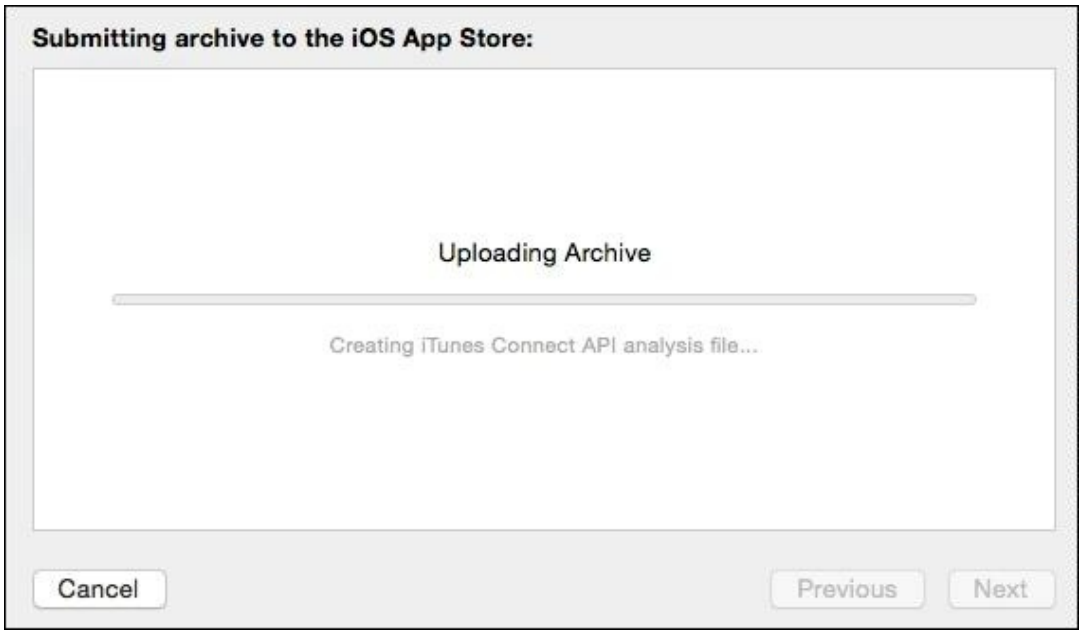


Notice that the identifiers and version number match what we input when we created the app in iTunesConnect. This is important because the bundle cannot be uploaded otherwise. To upload the build, click on the **Submit** button to the right.

Next, Xcode will check for the team for provisioning profile. If you are an individual, your account name will show up as shown in the following screenshot. Select it and click on **Choose**:



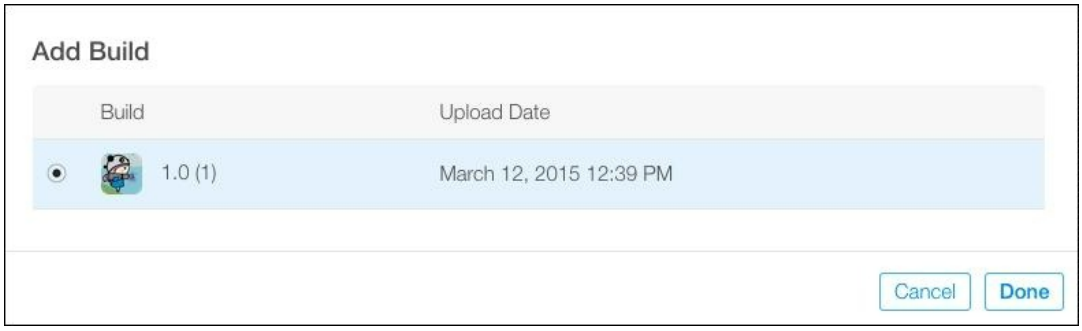
Next, the **Send App to Apple** window will pop up. Click on **Submit**. Now the archive bundle will be built and it will be uploaded to Apple.



This might take 5-25 minutes, depending on the time of the year and your Internet connection's upload speed. During Christmas and other holidays, it might take even longer because a lot of developers will be uploading their apps, so it is better to think ahead and upload in advance.

Once the binary is uploaded, click on **Done**. If you get some errors, don't be alarmed. Fix the errors that are mentioned, create a new archive, and upload a new binary.

Now, go back to the app page on iTunesConnect. Under the **Add Build** section, click on the **+** sign on the left, select the build that you have just uploaded, and select **Done** in the pop-up window:

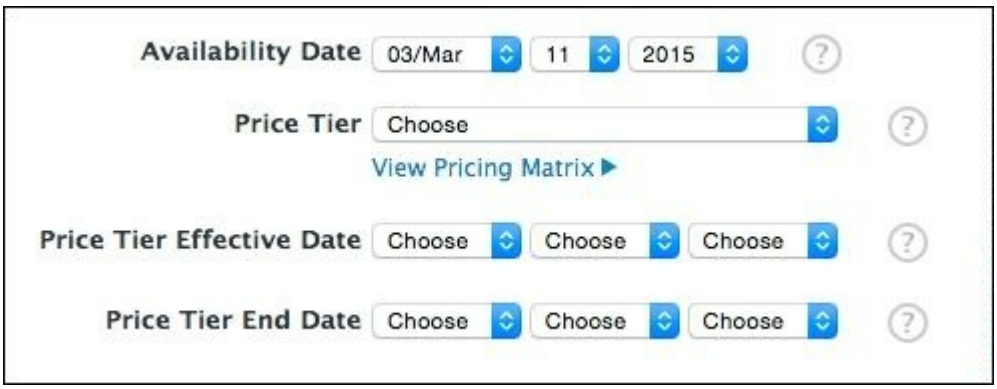


Next, in the **App Review Information** heading, add your contact information. Include your name, e-mail address, and contact information.

Under **Version Release**, you can select either **Automatically** releasing the version or **Manually Releasing** the game.

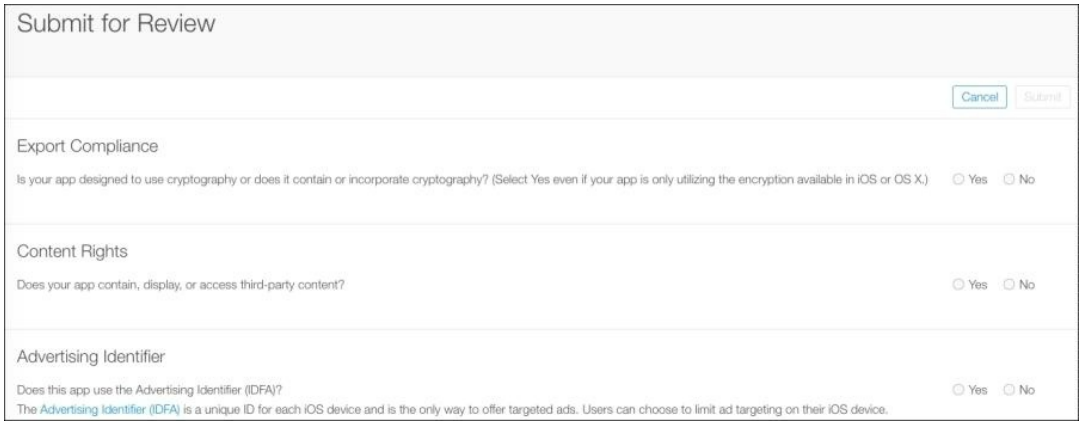
If you select **Automatically** releasing, the app will be released as soon it passes the review process. In the manual release, you can provide a date on which the app will be released. So, even if the app passes the review, it will still not be released until the release date.

Next, we have to specify the pricing for the game. At the top, select **Pricing**. Here, you can select the date you want to release your app on and the pricing tier for the app.



If you have selected automatic release, then you don't have to specify the date. For pricing, you can select from any of the tiers provided or you can select free. To create a paid app, you have to update the banking information in the **Banking and Taxation** heading.

After selecting the pricing, click on **Save** at the bottom of the screen. Next, click on **Submit for Review**:



If your app includes an encryption, a third-party application or advertisements, click on **Yes**, or else select **No**. Then, click on **Submit**.

Congratulations! You have successfully submitted your app for review. The status of the app will change to **Waiting for Review**. It usually takes 5-7 days for an app to be reviewed by Apple's app review team, so sit back and hold tight. You will receive an e-mail saying whether your app is accepted or rejected.



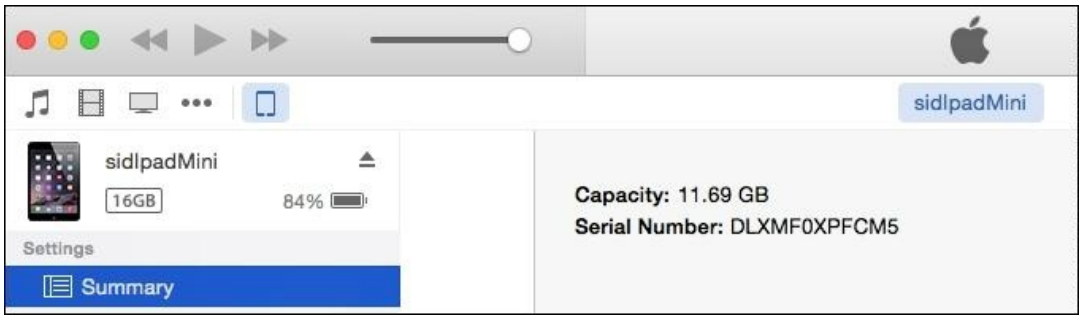
If the app is accepted, it will appear in the App Store shortly. If the app is rejected, don't worry! You will be able to rectify the errors and submit the app again for review. It will again take about a week for it to be reviewed. So, when making apps, make sure that you avoid making errors in the upload. Also prepare more in advance when you want to release your app on the App Store.

Creating an ad hoc app

You can build the game to a device by connecting it to the machine, but what if a friend or a client would like to run the app on their device and they live somewhere else? For that, you can create an ad hoc app. The steps are quite similar to creating and publishing on the App Store; it's just that this time, you will need to provide the device IDs on which the app can run.

So first, we need to add the device ID of the device on which we want to run the app. The device ID is a unique number, which is specific to a particular device. So, if your friend has an iPad and an iPhone and he would like to run it on both the devices, then you will need to ask for both the device IDs from him. The device ID is also a **user device ID** (**UDID**).

To get the UDID, connect the device to the Mac and open iTunes. In the summary under **Settings**, click on **Serial Number** to reveal the UDID of the device:



You can also go to **Devices** under **Window** in Xcode and select the device to show the UDID of the device:



So, once you get the UDID from your friend, open the Apple Developer Portal. Go to the **Certificates, Identifiers and Profiles** section. Click on **Devices** in the panel on the left side.

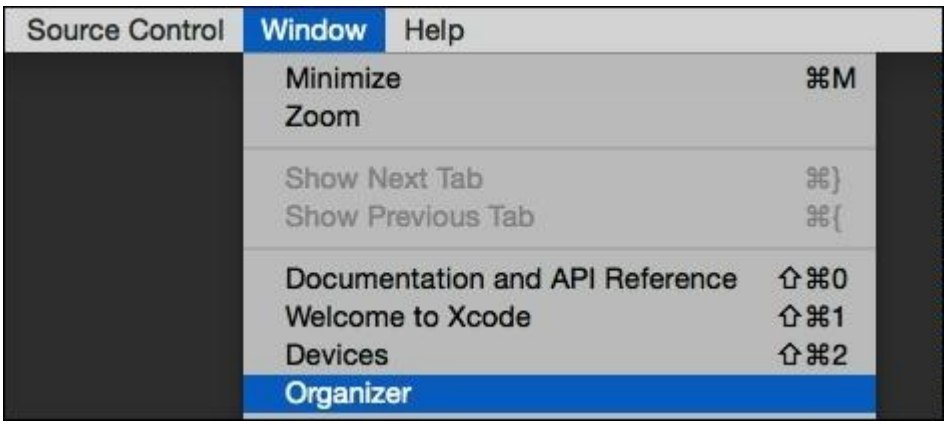
Once it opens, click on the **+** sign in the top-right corner of the page.



In the **Name** section, type the name of the friend and their device, enter the UDID directly below it, and click on **Continue**. Repeat the process to add the UDID for the other device.

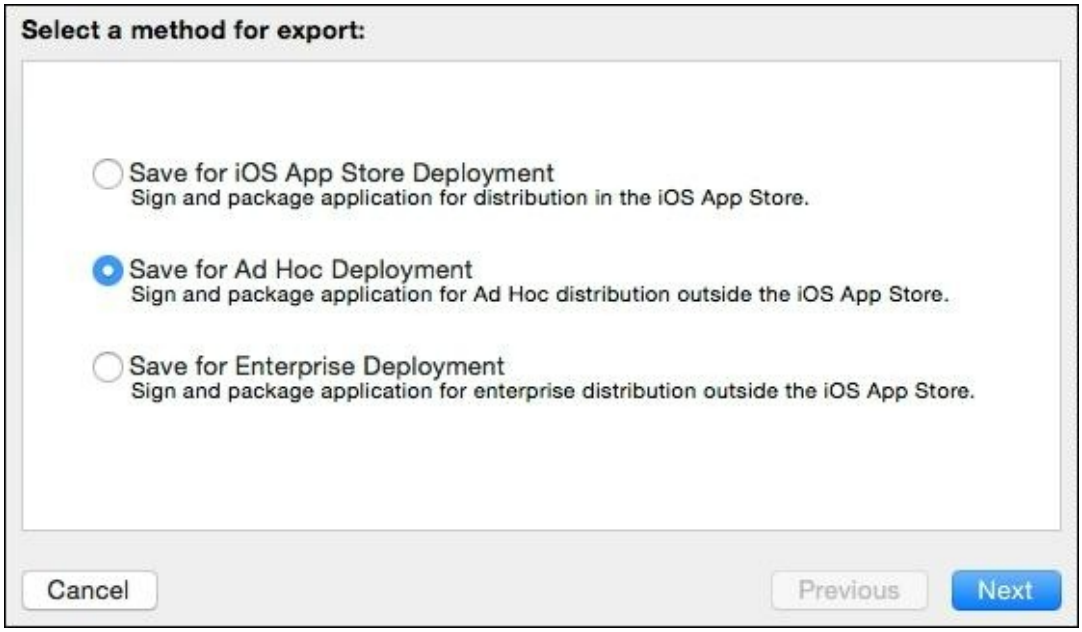
You can also upload a file with the UDID and register all devices at once by selecting **Register Multiple Devices** and uploading the file. In any case, note that Apple will let you register up to 100 devices. So, it is more suitable for testing your game with testers.

Now, open up the project on Xcode. In the top bar, go to **Window** and select **Organizer** from the drop-down list.



This will open the same window that opened when you archived your file for publishing. Instead of clicking on the Submit button, click on the **Export**

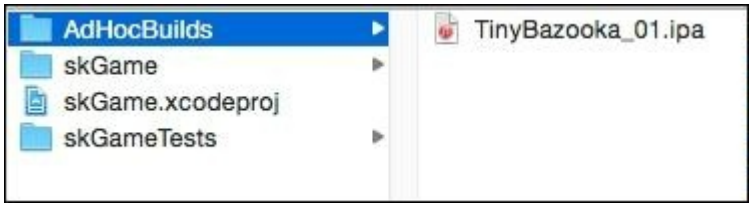
button this time. Under **Select the Method for Export**, select **Save for AdHoc deployment** and click on **Next**.



Next, select **Development Team to Use for Provisioning**. Select your name from the drop-down list and click on **Choose**.

Now, an archive will be created. In the next screen, a summary will be provided with the app and the files that will be part of the bundle. Click on **Export**.

Next, you will be asked to choose a location where you want the exported file to be saved. So, create a directory in the project folder and name it builds. Name and save the file in it.



This `.ipa` file is what needs to be sent to your friend now, so that they can run on it on their devices.

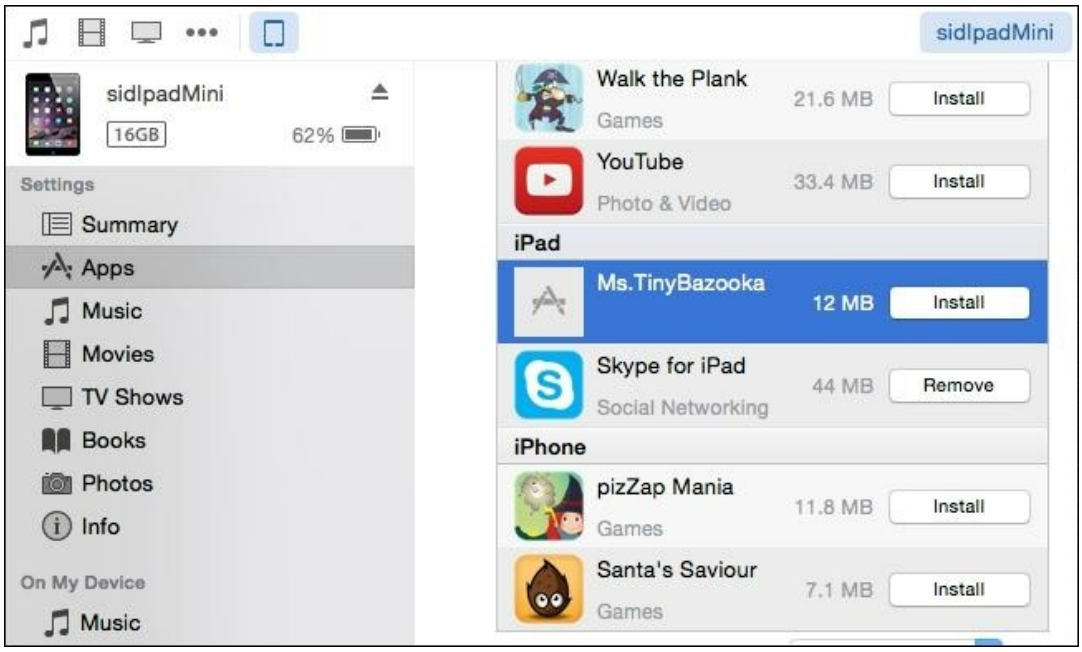
Next, let's see how to run this on a device. Connect the device to your Mac or PC. You will need the latest version of iTunes to install the app. So, if you haven't updated iTunes, then now is a good time to do so. Once the device is connected, open iTunes; in most cases, iTunes should open automatically.

Click on the three dots next to the movies section on the top and select **Apps**.



This will show all the apps. Next, drag the `.ipa` file you just created into this area. The app will be added to the list of apps that are currently present. Then, select the device icon right next to the three dots you selected earlier. On the panel to the left under **Settings**, select **Apps**. This will show all the apps ready to be installed.

Scroll down the list to the see the apps that are designed for iPads and iPhones. Select the Ms.TinyBazooka app and click on **Install**.



Once you click on **Install**, it will change to **Will Install**. At the bottom of the windows, select **Apply**. Now, it will start installing the app onto the device.

Once installed, the **Will Install** text will change to **Remove**. This means that the installation is complete.

Now, you can run the app, the same as you would run any other app.

References

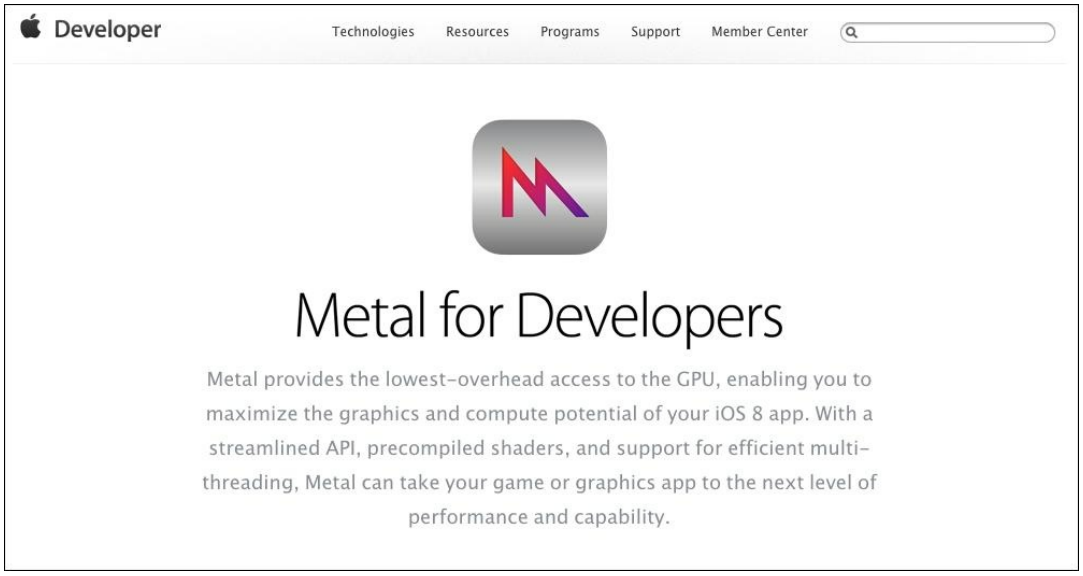
For Swift, SpriteKit, SceneKit, and Metal, Apple has provided very good documentation. All of these can be accessed from the Apple Developer Portal for free.

All the API and functions are very cleanly explained, and if you want a deeper understanding of the functions and variables used, then links are also provided for each. You can click on them and read through them in order to understand their implementation.

You can go to https://developer.apple.com/library/mac/navigation/ and search for SpriteKit or SceneKit for the respective documentation.



To learn more about Metal, you can visit the site at https://developer.apple.com/metal/:



For more information about Swift, you can visit the Apple site at https://developer.apple.com/swift/:

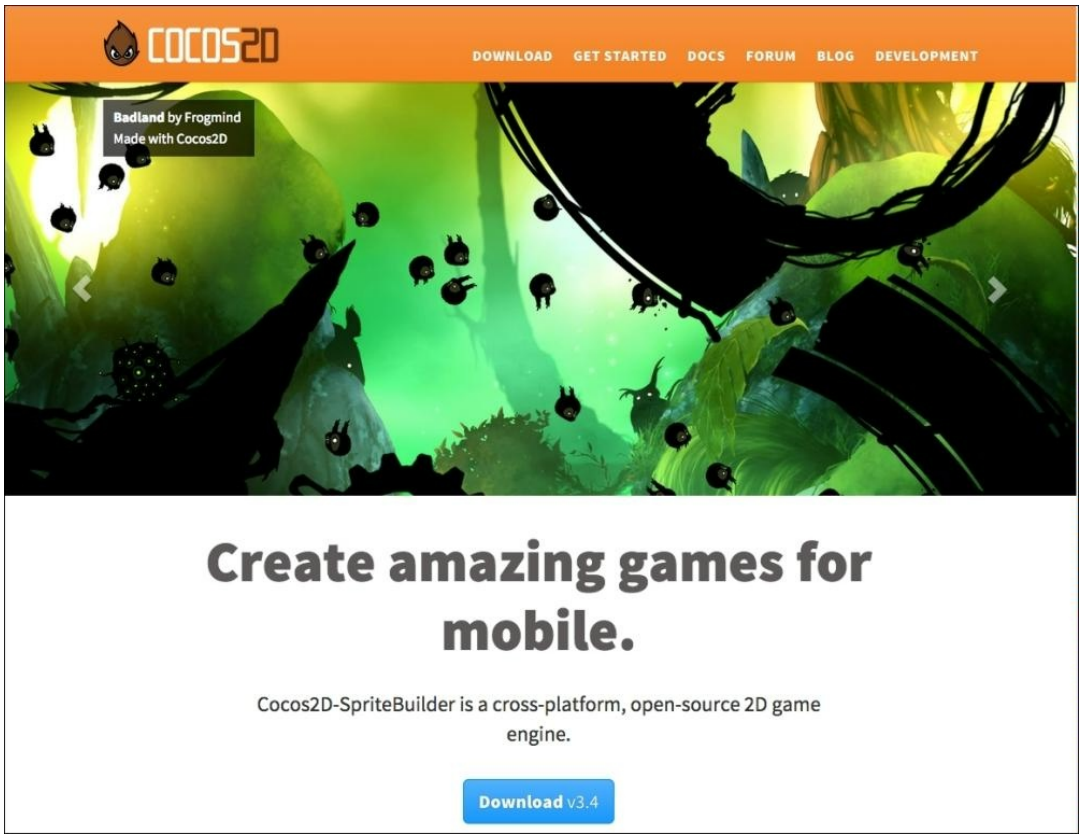


Alternative frameworks/engines

For SpriteKit and 2D game development, there are a couple of alternatives. Once you create a game using SpriteKit and if it becomes a hit, you would obviously want to bring it to other platforms as well, such as Android and Windows Phone.

For cross-platform game development, you have Cocos2d and Coco2d-x. With Cocos2d, you can develop games for iOS and Android, simultaneously. You can develop your game in either Objective-C or Swift. Once you've understood development using SpriteKit, you will notice that the syntax is pretty similar, so you will feel right at home.

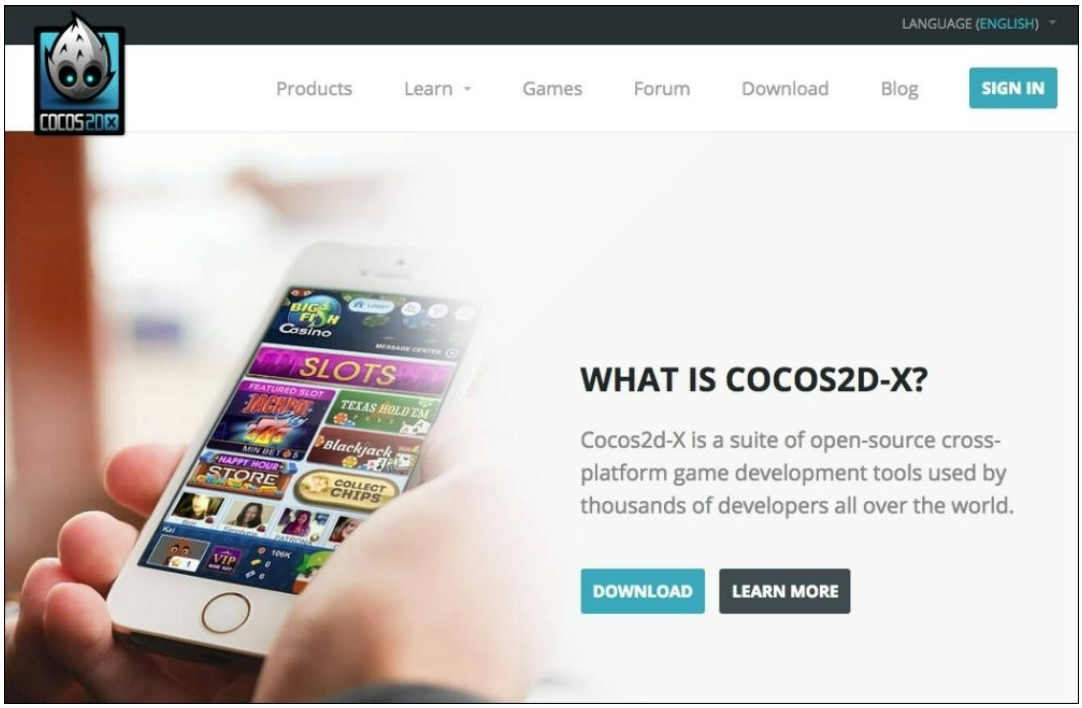
Industry professionals have used Cocos2d for a very long time to develop stellar games. It is open source and completely free. It has a good community, where you can ask questions to help find answers for your problems, and it is regularly updated. If you are interested, you can access it at http://cocos2d.spritebuilder.com/:



Like Cocos2d, Cocos2d-x is also completely open source and free and supports cross-platform game development. With Cocos2d-x, you can develop games for iOS, Android, Windows Phone, PC, Mac, Linux, and more.

You can create two-dimensional as well as three-dimensional games, using Cocos2d-x.

In order to develop games with Cocos2d-x, you are required to know C++. But there is also Cocos2d-js, which uses JavaScript as its development language. By using it, you can even develop a game for the Web. It can be downloaded from http://www.cocos2d-x.org/:



For three-dimensional game development engines, you can take a look at Unity and Unreal Engine. You can develop games for 2D as well using the engine.

Unity uses JavaScript, C#, and Boo as its languages for development, and Unreal uses C++. At GDC, both of these frameworks are completely free to use, so I really recommend checking them out at http://unity3d.com/5 and https://www.unrealengine.com/.

Final remarks

By now, I am hoping that you have got a good foundation for developing 2D and 3D games with Swift, SpriteKit and SceneKit. I also hope that you have gained some basic understanding of graphics programming with Metal.

With these tools, you should be able to develop and publish your games on the iOS store. This is the first step; there is more to learn about each of these frameworks, if you want to make better looking games.

With my experience in Swift, SpriteKit and SceneKit, I can say that both Swift and SceneKit were only introduced iOS 8 and Xcode 6. This is definitely a good start for Apple, and I hope that with the next version, these will be improved upon. As and when more people start accepting and using Swift and SceneKit, there will be a much bigger developer community that you will be able to take help from and be part of. This is just like SpriteKit, which is widely used by hobbyists and enthusiasts.

If you have any questions, would like an expert opinion, or would simply like to share your creations, you can mail me by visiting the **Contact Me** section of my company website. You can also follow me at `@sidshekar`, and I will reply to your query as soon as possible.

Summary

So, in this final chapter, we saw how to prepare our app for publishing and create the publishing license. You learned about the iTunesConnect portal, created an app in it, and finally published the app on the iOS store. At the end, we saw how to create an ad hoc build of the game that can be run on a specific device.

I had a wonderful time writing this book. Along the way, I myself learned a lot about these technologies and game development in general. I hope that you will enjoy this book as much as I enjoyed bringing it to you.

Happy game developing!

Index

A

- access specifiers / Access specifiers
- ad hoc app
    - creating / Creating an ad hoc app
- app
    - running, on device / Running the app on the device
    - creating / Creating the app
- Apple developer
    - URL, for registering / Creating an iOS developer account
- Apple developer portal
    - URL / Generating the distribution certificate
- Apple Developer Portal
    - references / References
- applicationDidBecomeActive function / The project folder
- applicationDidEnterBackground function / The project folder
- applicationdidFinishinLaunchingWithOptions function / The project folder
- application types, Xcode
    - about / Xcode application types
    - Master-Detail Application / Master-Detail Application
    - Page-Based Application / Page-Based Application
    - Tabbed Application / Tabbed Application
    - Single View Application / Single View Application
- applicationWillEnterForeground function / The project folder
- applicationWillResignActive function / The project folder
- applicationWillTerminate function / The project folder
- arithmetic increment/decrement
    - about / Arithmetic increment/decrement
- arithmetic operators
    - about / Arithmetic operators

- array
    - functions / Important array functions
- arrays
    - about / Arrays
    - looping through / Looping through arrays
    - objects, adding to / Adding, removing, and inserting objects into arrays
    - objects, inserting into / Adding, removing, and inserting objects into arrays
    - objects, removing from / Adding, removing, and inserting objects into arrays
- Audacity
    - about / Downloading and installing Audacity
    - URL, for downloading latest version / Downloading and installing Audacity
    - installing / Downloading and installing Audacity
    - downloading / Downloading and installing Audacity
- audio file formats
    - about / Audio file formats
- audio format
    - converting / Converting the audio format
- audio loops
    - creating / Adding audio loops

B

- background
    - adding / Adding a background and a hero
- background music
    - adding / Adding background music
- basic Metal project
    - overview / The basic Metal project
- basic SpriteKit animation

    - about / Basic SpriteKit animation

- Bitmap font
    - implementing, in game / Implementing a Bitmap font
- Breakpoint Navigator tab, Xcode Interface / The Breakpoint Navigator tab
- brushes, Sprite Illuminator
    - Height / The Tools panel
    - Angle / The Tools panel
    - Structure / The Tools panel
    - Smoothen / The Tools panel
    - Erase / The Tools panel

C

- camera
    - adding, to scene / Adding a camera to the scene
- CAPTCHA
    - about / Creating an iOS developer account
- Certificate Signing Request (CSR) / Generating the distribution certificate
- classes
    - about / Classes
    - initializers / Properties and initializers
    - properties / Properties and initializers
    - custom methods, creating in / Custom methods
    - inheritance / Inheritance
    - access specifiers / Access specifiers
- classes, SpriteKit
    - SKScene / Looking at the default SpriteKit project
    - SKSpriteNode / Looking at the default SpriteKit project
    - SKLabelNode / Looking at the default SpriteKit project
    - SKAction / Looking at the default SpriteKit project
- clipping
    - about / 2D and 3D coordinate systems

- Cocos2d
  - URL / Alternative frameworks/engines
- Cocos2d-x
  - about / Alternative frameworks/engines
  - URL / Alternative frameworks/engines
- collision detection
  - about / Collision detection
- colored quad project
  - overview / The colored quad project
- comparison operators
  - about / Comparison operators
- composite operations
  - about / Composite operations
- conditional expressions / Conditional expressions
- custom methods
  - creating, in classes / Custom methods

D

- 2D coordinate system / 2D and 3D coordinate systems
- 2D game development
  - alternatives / Alternative frameworks/engines
- 3D coordinate system / 2D and 3D coordinate systems
- 3D objects
  - about / Understanding 3D objects
- Debug Navigator tab, Xcode Interface / The Debug Navigator tab
- Debug panel, Single View Project
  - about / The Debug panel
- decision making statements
  - about / Decision-making statements

    - if statement / The if statement
    - if else statement / The if else statement
    - else if statement / The else if statement
    - conditional expressions / Conditional expressions
    - switch statement / The switch statement

- default SceneKit project / Looking at the default SceneKit project
- default SpriteKit project / Looking at the default SpriteKit project
- device

    - app, running on / Running the app on the device

- dictionary

    - about / Dictionary
    - objects, adding to / Adding and removing objects from the dictionary
    - objects, removing from / Adding and removing objects from the dictionary
    - items, looping through / Looping through items in the dictionary
    - functions / Dictionary functions

- distribution certificate

    - generating / Generating the distribution certificate

- do while loop / The do while loop
- draw stage, graphics pipeline and shaders

    - about / The draw stage
    - render pass, starting / Start render pass
    - command buffer, obtaining / Get command buffer
    - drawing / Draw
    - command buffer, committing / Commit the command buffer

E

- else if statement / The else if statement
- emitter

    - about / Particle systems

- enemies

    - adding, to scene / Adding enemies

- enemy-kill sound effect

    - adding / Adding the enemy-kill sound effect

- enemy bullets
    - adding, to scene / Adding enemy bullets
- enemy node
    - adding / Adding an enemy node
- external 3D applications
    - scene, importing from / Importing scenes from external 3D applications

F

- Find Navigator tab, Xcode Interface / The Find Navigator tab
- fireRocket sound effect
    - adding / Adding the fireRocket sound effect
- floor
    - adding, to scene / Adding a floor
- floor parallax
    - adding, to scene / Adding wall and floor parallax
- for in loop / The for in loop
- for loop / The for loop
- fragment shaders / The basic Metal project
- functions
    - about / Functions
    - simple functions / Simple functions
    - parameter, passing to / Passing a parameter
    - multiple parameters, passing to / Passing more than one parameter
    - value, returning / Returning a value
    - default value, assigning to parameter / Default and named parameters
    - multiple values, returning / Returning more than one value
- functions, GameViewControllerclass class
    - shouldAutoRotate / The basics of SceneKit

    - prefersStatusBarHidden / The basics of SceneKit
    - suppoertedInterfaceOrientation / The basics of SceneKit
    - didReceiveMemoryWarning / The basics of SceneKit

- functions, SpriteKit

    - touchesBegan / Looking at the default SpriteKit project
    - update / Looking at the default SpriteKit project

G

- game

    - particle system, adding to / Adding a particle system to the game
    - Bitmap font, implementing in / Implementing a Bitmap font

- gameloop, finishing

    - about / Finishing the game loop
    - hero, jumping / Making the hero jump
    - gameover condition, setting / Setting a game over condition
    - hero jump, fixing / Fixing the jump

- GameOver

    - sound effects, adding at / Adding a sound effect at GameOver

- game over

    - displaying / Displaying the game over screen

- gameplay scene

    - adding / Adding a gameplay scene

- GameScene.sks / Introduction to SpriteKit and SKScene
- GameScene.swift / Introduction to SpriteKit and SKScene
- GameViewController.swift / Introduction to SpriteKit and SKScene
- Glyph Designer

    - about / Glyph Designer
    - URL, for downloading trial version / Glyph Designer

- graphics pipeline

    - about / The graphics pipeline
    - vertices / The graphics pipeline
    - primitives, generating / The graphics pipeline

    - Vertex/Geometry Shader / The graphics pipeline
    - rasterization / The graphics pipeline
    - Pixel/Fragment Shader / The graphics pipeline
    - testing and mixing / The graphics pipeline
    - Frame Buffer / The graphics pipeline

- graphics pipeline and shaders

    - about / The graphics pipeline and shaders
    - preparation/initialization stage / The preparation/initialization stage
    - draw stage / The draw stage

- ground

    - adding, to scene / Adding the ground

H

- hero

    - animating / Animating the hero

- hero object

    - accessing / Accessing the hero object and adding a physics body

- hero sprite

    - adding / Adding a background and a hero
    - position, updating / Updating the position of the hero

- hero spritesheet

    - creating / Creating the hero spritesheet

- high score

    - saving / Saving the high score

- high score count

    - resetting / Resetting the high score count

I

- if else statement / The if else statement

- if statement / The if statement
- impulse
    - about / Physics
- inheritance / Inheritance
- inspector tabs, Utility panel (Redux)
    - about / The inspector tabs
    - File Inspector / The File Inspector tab
    - Quick Help Inspector / The Quick Help Inspector tab
    - Identity Inspector / The Identity Inspector tab
    - Attributes Inspector / The Attributes Inspector tab
    - Size Inspector / The Size Inspector tab
    - Connections Inspector / The Connections Inspector tab
- installing
    - Xcode / Downloading and installing Xcode
- Integrated Development Environment (IDE) / Downloading and installing Xcode
- iOS developer account
    - creating / Creating an iOS developer account
- iOS developer portal
    - reference link / Running the app on the device
- Issue Navigator tab, Xcode Interface / The Issue Navigator tab
- items
    - looping through, in dictionary / Looping through items in the dictionary
- iTunesConnect Portal
    - about / The iTunesConnect portal
    - URL / The iTunesConnect portal

L

- lame library
    - reference link / Converting the audio format
- library, Utility panel (Redux)

    - File Template / The File Template tab
    - Code Snippet Library / The Code Snippet Library tab
    - Object Library / The Object Library tab
    - Media Library / The Media Library tab

- lighting
    - overview / Lighting and shadows
- LightNode
    - about / Lighting and shadows
- light source
    - creating / Lighting and shadows
- light sources
    - adding, to scene / Adding light sources
- logical operators
    - about / Logical operators
- looping
    - through arrays / Looping through arrays
- looping statements
    - about / Looping statements
    - while loop / The while loop
    - do while loop / The do while loop
    - for loop / The for loop
    - for in loop / The for in loop

M

- main menu button
    - adding / Adding the main menu button
- main menu scene
    - adding / Adding a main menu scene
- Master-Detail Application, Xcode / Master-Detail Application
- Metal

    - about / Introducing Metal
    - overview / Overview
    - URL / References

- mipmapped / The texture quad project
- modes, scene

    - AspectFill / Introduction to SpriteKit and SKScene
    - Fill / Introduction to SpriteKit and SKScene
    - AspectFit / Introduction to SpriteKit and SKScene
    - ResizeFill / Introduction to SpriteKit and SKScene

- Ms. tinyBazooka

    - obtaining / Getting Ms. tinyBazooka ready

- multiple parameters

    - passing, to function / Passing more than one parameter

N

- Navigation panel, Xcode Interface / The Navigation panel

O

- Objective-C, in Swift

    - about / Objective-C in Swift

- objects

    - adding, to arrays / Adding, removing, and inserting objects into arrays
    - removing, from arrays / Adding, removing, and inserting objects into arrays
    - inserting, into arrays / Adding, removing, and inserting objects into arrays
    - removing, from dictionary / Adding and removing objects from the dictionary
    - adding, to dictionary / Adding and removing objects from the dictionary
    - adding, to scene / Adding objects to the scene, Adding objects and physics to the scene
    - updating, in scene / Updating objects in the scene

    - contacts, checking between / Checking for contact between objects

- OpenGLES
    - about / Overview

- operators
    - about / Operators
    - arithmetic operators / Arithmetic operators
    - comparison operators / Comparison operators
    - logical operators / Logical operators
    - arithmetic increment/decrement / Arithmetic increment/decrement
    - composite operations / Composite operations

- optionals
    - about / Optionals

P

- Page-Based Application, Xcode / Page-Based Application
- parallax background theory / Parallax background theory
- parallax effect
    - implementing / Implementing the parallax effect

- parameter
    - passing, to function / Passing a parameter

- parameters, SpriteKit Particle Emitter
    - Name / Name
    - Background / Background
    - Particle texture / Particle Texture
    - Particles / Particles
    - Lifetime / Lifetime
    - Position Range / Position Range
    - Angle / Angle
    - Speed / Speed
    - Acceleration / Acceleration
    - Alpha / Alpha
    - Scale / Scale
    - Rotation / Rotation
    - Color Blend / Color Blend

    - Color Ramp / Color Ramp

- particle effects
    - creating / Creating particle effects

- particles
    - designing / Designing particles
    - adding / Adding particles

- particle system
    - about / Particle systems
    - adding, to game / Adding a particle system to the game

- physics
    - about / Physics
    - adding, to scene / Adding objects and physics to the scene

- physics body
    - adding, to scene / Accessing the hero object and adding a physics body

- Pixel/Fragment Shader / Render pipeline
- pixel formats
    - reference link / The basic Metal project

- pizZapMania game / The project folder
- player controls
    - adding, to scene / Adding player controls

- Playground
    - about / Introducing Playground

- preparation/initialization stage, graphics pipeline and shaders
    - about / The preparation/initialization stage
    - device, obtaining / Get device
    - command queue / Command queue
    - resources / Resources
    - pipeline, rendering / Render pipeline
    - view, setting up / View

- Preview panel, Sprite Illuminator

    - about / The Preview panel

- project folder, Single View Project
    - about / The project folder

- Project Navigation tab, Xcode Interface / The Project Navigation tab
- project root, Single View Project
    - about / The project root
    - General tab / General
    - Capabilities tab / The Capabilities tab
    - Info tab / The Info tab
    - Build Settings tab / The Build Settings tab
    - Build Phases tab / The Build Phases tab
    - Build Rules tab / The Build Rules tab

R

- Rain particle system
    - about / Particle systems

- Report Navigator tab, Xcode Interface / The Report Navigator tab

S

- scene
    - player controls, adding to / Adding player controls
    - enemies, adding to / Adding enemies
    - enemy bullets, adding to / Adding enemy bullets
    - creating, with SCNScene / Creating a scene with SCNScene
    - objects, adding to / Adding objects to the scene, Adding objects and physics to the scene
    - sphere, adding to / Adding a sphere to the scene
    - light sources, adding to / Adding light sources
    - camera, adding to / Adding a camera to the scene
    - floor, adding to / Adding a floor
    - importing, from external 3D applications / Importing scenes from external 3D applications
    - physics, adding to / Adding objects and physics to the scene
    - physics body, adding to / Accessing the hero object and adding a physics body
    - ground, adding to / Adding the ground
    - objects, updating in / Updating objects in the scene

• SceneKit

  ◦ exploring / Exploring SceneKit
  ◦ default SceneKit project / Looking at the default SceneKit project
  ◦ basics / The basics of SceneKit

• SCNScene

  ◦ scene, creating with / Creating a scene with SCNScene

• score

  ◦ keeping / Keeping score
  ◦ displaying / Displaying the score
  ◦ high score, saving / Saving the high score
  ◦ high score count, resetting / Resetting the high score count

• selections tools, Sprite Illuminator

  ◦ Wand/Color / The Tools panel
  ◦ Polygon / The Tools panel
  ◦ Move / The Tools panel

• settings panel, Texture Packer

  ◦ Data / Data
  ◦ Texture / Texture
  ◦ Layout / Layout
  ◦ Sprites / Sprites

• shadows

  ◦ overview / Lighting and shadows

• simple functions

  ◦ about / Simple functions

• Single View Application, Xcode / Single View Application
• Single View Project

  ◦ about / The Single View Project
  ◦ project root / The project root, General, The Info tab, The Build Settings tab, The Build Phases tab, The Build Rules tab
  ◦ project folder / The project folder
  ◦ Utility panel (Redux) / The Utility panel (Redux)
  ◦ Debug panel / The Debug panel

• skeletal animation

    - about / Skeletal animation

- SKScene
    - basics / Introduction to SpriteKit and SKScene

- sound effects
    - adding / Adding sound effects
    - fireRocket sound effect, adding / Adding the fireRocket sound effect
    - enemy-kill sound effect, adding / Adding the enemy-kill sound effect
    - adding, at GameOver / Adding a sound effect at GameOver

- source code manager (SCM) / Introducing Playground
- sphere
    - adding, to scene / Adding a sphere to the scene

- Spine
    - about / Skeletal animation
    - URL, for downloading trial version / Skeletal animation

- Sprite Illuminator
    - about / Sprite Illuminator
    - URL, for downloading / Sprite Illuminator
    - Sprites panel / The Sprites panel
    - Preview panel / The Preview panel
    - Tools panel / The Tools panel

- SpriteKit
    - exploring / Exploring SpriteKit
    - new features / New features in SpriteKit
    - default SpriteKit project / Looking at the default SpriteKit project
    - basics / Introduction to SpriteKit and SKScene
    - alternatives / Alternative frameworks/engines

- SpriteKit overlay
    - adding / Adding a SpriteKit overlay
    - labels, adding / Adding labels and buttons
    - buttons, adding / Adding labels and buttons

- SpriteKit particle file (.sks)

    - about / Particle systems

- sprite sheet animation
    - about / Sprite sheet animation
- Sprites panel, Sprite Illuminator
    - sprite / The Sprites panel
    - global light / The Sprites panel
    - display mode / The Sprites panel
- stage_in
    - reference link / The colored quad project
- statements
    - about / Statements
    - decision making statements / Decision-making statements
    - looping statements / Looping statements
- storyboard / The project folder
- Swift
    - about / Introducing Swift
    - URL / References
- switch statement / The switch statement
- Symbol Navigator tab, Xcode Interface / The Symbol Navigator tab

T

- Tabbed Application, Xcode / Tabbed Application
- Test Navigator tab, Xcode Interface / The Test Navigator tab
- texture atlas
    - about / Basic SpriteKit animation
- Texture Packer
    - overview / Exploring Texture Packer
    - URL, for downloading / Exploring Texture Packer
    - settings panel / Exploring Texture Packer, Texture, Layout
- texture quad project
    - overview / The texture quad project
- toolbar, Xcode Interface / The toolbar

- Tools panel, Sprite Illuminator
    - about / The Tools panel
    - Effects / The Tools panel
    - Bevel and Emboss tool / The Tools panel
    - selections tools / The Tools panel
- touch interactivity
    - adding / Adding touch interactivity

U

- UIApplicationDelegate subclass / The project folder
- unit testing
    - reference link / The Test Navigator tab
- Unity
    - URL / Alternative frameworks/engines
- universal static library, 71Squared
    - URL / Implementing a Bitmap font
- Unreal Engine
    - URL / Alternative frameworks/engines
- user device ID (UDID) / Creating an ad hoc app
- Utility panel
    - about / The Utility panel
- Utility panel (Redux), Single View Project
    - about / The Utility panel (Redux)
    - inspector tabs / The inspector tabs
    - library / The library

V

- variables
    - about / Variables
- Vertex/Geometry Shader / Render pipeline

- vertex shaders / The basic Metal project

W

- wall
    - adding, to scene / Adding wall and floor parallax
- while loop / The while loop

X

- Xcode
    - downloading / Downloading and installing Xcode
    - installing / Downloading and installing Xcode
    - application types / Xcode application types
- Xcode Interface
    - about / The Xcode interface
    - toolbar / The toolbar
    - Navigation panel / The Navigation panel
    - Project Navigation tab / The Project Navigation tab
    - Symbol Navigator tab / The Symbol Navigator tab
    - Find Navigator tab / The Find Navigator tab
    - Issue Navigator tab / The Issue Navigator tab
    - Test Navigator tab / The Test Navigator tab
    - Debug Navigator tab / The Debug Navigator tab
    - Breakpoint Navigator tab / The Breakpoint Navigator tab
    - Report Navigator tab / The Report Navigator tab
- XIB file / The project folder