Game development has been simplified with Apple's new programming language—Swift. It starts with the absolute basics such as the Xcode interface and takes you all the way to Swift programming.

You will take a walk through the creation of 2D and 3D games followed by an introduction to SpriteKit and SceneKit. The book also looks at how game objects are placed in 3D scenes, how to use the graphics pipeline, and how objects are displayed on mobile screens. You will also delve into essential game concepts such as collision detection, animation, particle systems, and scene transitions. Finally, you will learn how to publish and distribute games to the iTunes store.

Who this book is written for

This book is for novices as well as proficient game developers who want to learn more about game development using Swift. If you are from a 2D game development background and would like to learn the basics of 3D game development, this book is for you. Additionally, if you want to learn the basics of graphics programming and shaders, this book will be a good starting point.

What you will learn from this book

- Learn Apple's latest programming language—Swift
- Acquaint yourself with the Xcode interface
- Create stunning games with SpriteKit and SceneKit
- Understand the basics of graphics programming
- Intensify your gameplay experience by learning how to add customized sound
- Develop highly interactive games using animation, sound, collision detection, particle systems, and shaders
- Package and publish games on the Apple App Store

Create robust and spectacular 2D and 3D games from scratch using Swift—Apple's latest and easy-to-learn programming language.
In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 1 'Getting Started'
- A synopsis of the book’s content
- More information on Learning iOS 8 Game Development Using Swift

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.
Learning iOS 8 Game Development Using Swift

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 cast 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.
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
Getting Started

- 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.

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:

---

### My Apple ID

**Create an Apple ID.**

An Apple ID is your user name for everything you do with Apple: Shop the iTunes Store, enable iCloud on all your devices, buy from the Apple Online Store, make a reservation at an Apple Retail Store, access the Apple Support website, and more.

Read the Apple Customer Privacy Policy>

---

### Create an Apple ID.

**Name**

Please enter your full name.

- **First Name**
- **Middle Name** (Optional)
- **Last Name**

**Apple ID and Password**

Enter your primary email address as your Apple ID. This will be used as the contact email address for your account.

- **Apple ID**
- **Password**
- **Confirm Password**

**Security Questions**

Select three security questions below. These questions will help us verify your identity should you forget your password.

- **Security Question**
- **Answer**
- **Security Question**
- **Answer**
- **Security Question**
- **Answer**

---

---
• **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:

- **Date of Birth**
  Combined with your security question, this will help us verify your identity if you forget your password or need to reset it.

- **Rescue Email Address**
  Give us a rescue email address where we can send you a link to confirm your identity and let you reset your information should any security issues arise. This address is only for communicating information about your security details. We won’t send any other types of messages to this address.

• **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:

![CAPTCHA Image](image)

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:

![Apple Developer Screen](image)

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.
Getting Started

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.

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.
Getting Started

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:

![Image of registration options]
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:
Getting Started

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:

![Playground window](image)

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.
Getting Started

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:

![Hello, World!](image_url)

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
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: Set<UITouch>, withEvent event: UIEvent) {
        // Called when a touch begins */
        for touch in touches {
            let location = touch.locationInNode(self)
            let sprite = SKSpriteNode(imageNamed: "Spaceship")
            sprite.setScale(0.5)
            sprite.setScale(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`, `touchMoved`, `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 orAlembic 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:

![Diagram of 3D coordinate systems]

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 it is 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:

```swift
class GameScene: SKScene {

override func didMoveToView(view: SKView) {
    // Setup your scene here
    let myLabel = SKLabelNode(fontNamed: "Chalkduster")
    myLabel.text = "Hello, World!"
    myLabel.fontSize = 45;
    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.
Getting Started

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 GameViewController class has other functions such as shouldAutoRotate, prefersStatusBarHidden, supportInterfaceOrientation, and didReceiveMemoryWarning. 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 its 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:

![Diagram of Metal and its interactions with the GPU](image)

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.
Where to buy this book

You can buy Learning iOS 8 Game Development Using Swift from the Packt Publishing website.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.

Click here for ordering and shipping details.