Includes iOS 8 SDK

Beginning Swift Games Development for iOS

James Goodwill | Wesley Matlock

Apress®

For your convenience Apress has placed some of the front matter material after the index. Please use the Bookmarks and Contents at a Glance links to access them.

Apress®

Contents at a Glance

Introduction

Which Version of Swift Is Covered in This book?

This book covers version 1.1 of Swift. Swift 1.2 is currently in beta and under a nondisclosure agreement (NDA). When 1.2 is released, we will update the source in this book at both the Apress.com web site and James Goodwill's blog at www.jgoodwill.org. Please be aware that you will need to update your source when 1.2 is released.

What This Book Is

Game apps are one of the most popular categories in the Apple iTunes App Store. Well, the introduction of the new Swift programming language will make game development even more appealing and easier to existing and future iOS app developers. In response, James Goodwill, Wesley Matlock, and Apress introduce you to this book, *Beginning Swift Games Development for iOS*.

In this book, you'll learn the fundamental elements of the new Swift language as applied to game development for iOS in 2D and 3D worlds using both Sprite Kit and Scene Kit, respectively.

What You Need to Know

This book assumes you have a basic understanding of how to create applications for the iPhone using Xcode. You will also need a basic understanding of Apple's new programming language, Swift 1.1. We assume that you can download, install, and use the latest version of Xcode to create an application and run it on the iPhone simulator.

What You Need to Have

In terms of hardware, you need an Intel-based Macintosh running Mountain Lion (OS X 10.8) or later. Regarding software, you need Xcode 6.2 since that is the current version to include Swift 1.1. You can download Xcode from the App Store or Apple's developer web site at http://developer.apple.com.

What's in This Book?

In Chapter 1, you'll learn about what Sprite Kit is and how you create a new Sprite Kit game using Xcode. You will then dive in and create the beginnings of a Sprite Kit game starting from scratch. You will learn about SKNodes and their subclasses, and you'll use an SKSpriteNode to add both a background node and a player node.

In Chapter 2, we will step back a bit and give you a deeper look at Sprite Kit scenes, including how scenes are built and why the order they are built in can change your game. The chapter will close with a discussion of Sprite Kit coordinate systems and anchor points as they relate to SKNodes.

In Chapter 3, you'll work with Sprite Kit's physics engine and collision detection. The chapter will begin with a discussion of SKPhysicsBody—the class used to simulate collision detection. You will then turn on gravity in the game world and see how that affects the nodes. After that, you will add a touch responder to propel the playerNode up into space, and finally you will learn how to handle node collisions.

In Chapter 4, you'll start adding some real functionality to your game. You'll begin by making some small changes to the current GameScene. After that, you will add additional orb nodes to collide into. You will then add scrolling to your scene, allowing you to make it look like the player is flying through space collecting orbs. Finally, you will start using the phone's accelerometer to move the player along the x-axis.

In Chapter 5, you'll refactor the orb node's layout one last time with the goal of enhancing playability. After that, you will learn how you can use SKActions to move an SKSpriteNode back and forth across the scene and then make that same node rotate forever. The chapter will close with a look at how you can add colorizing effects to an SKSpriteNode using a colorize action.

In Chapter 6, you'll see how to define particle emitters and how to leverage them in Sprite Kit games. After that, you will learn how you can use them to add engine exhaust to the playerNode whenever an impulse is applied to the physicsBody.

In Chapter 7, you'll see how you can use SKLabelNodes to add text to your Sprite Kit games. Specifically, you'll see how you how to add a label that keeps up with the number of impulses remaining for the spaceman to use, and then you'll see how you can add scoring to the game to keep up with the number of orbs the spaceman has collected.

In Chapter 8, you'll learn how to implement scene transitions using Sprite Kit's SKTransition class. You will look at some of the different types of built-in transitions Sprite Kit makes available to you. You will also see how you can control each scene during a transition. At the end of the chapter, you will take your newfound knowledge and add a menu scene to your SuperSpaceMan game.

In Chapter 9, you'll learn some Sprite Kit best practices; specifically, you will see how you can create your own subclasses of `SKSpriteNode` so that you can better reuse your nodes. You will then move on to changing your game to load all the sprites into a single texture atlas that you can reference when creating all future sprites. After that, you will move on to externalizing some of your game data so that designers and testers can change the game play. Finally, you will close out the chapter when you prune your node tree of all nodes that have fallen off the bottom of the screen.

In Chapter 10, you'll learn about what Scene Kit is and how to create a new Scene Kit game using Xcode. You will then dive in and create the beginnings of a Scene Kit game starting from scratch. You will learn to about `SCNScene` and `SCNodes` with a Scene Kit primer.

In Chapter 11, you'll learn more about the scene graph and some of the basics of Scene Kit. You will start to create your game by loading the spaceman from his Collada file. You will also learn about the Scene Kit primitive geometries by adding these as objects for the spaceman to avoid.

In Chapter 12, you'll learn how Scene Kit uses lighting and the type of lighting that is available to you in Scene Kit. You will also examine how materials are added onto the `SCNNode`, as well as how the camera is used within the scene.

In Chapter 13, you'll learn about the basics of animating the objects in your game. You will see a couple of different ways to animate the nodes to give you more than one way to do your animations. Once you have completed this chapter, all of your objects will move within the scene.

In Chapter 14, you'll learn about collision detection within the scene. You will learn how to move the spaceman around the scene. Once you have the spaceman moving, you will learn how to detect when the spaceman runs into an obstacle.

In Chapter 15, you'll learn how to use a Sprite Kit scene within the Scene Kit scene. The chapter will show you how to create a screen to show you a timer that you will start when the user starts the game. The chapter will also show you how to display a "game over" screen and then restart the game.

In Chapter 16, you'll get some tips and tricks on using the Xcode editor. We will also explain some of the vectors and matrix methods in some detail to give you a better understanding of what those methods are doing behind the scenes.

In Appendix A, you'll take a quick (you might say a "swift") look at each of the features in the Swift programming language. We'll start by describing each feature and then cement your knowledge through consecutive examples.

Part

Swift and Sprite Kit

In this part of this book, we will cover the basics of Sprite Kit including how you render and animate sprites, add physics and collision detection, and control your game play with the accelerometer. You will also look at how you add particle emitters to enhance the appearance of your game. We will cover everything you need to know to create your own Sprite Kit game.

Chapter 1

Setting Up Your Game Scene and Adding Your First Sprites

Sprite Kit is Apple's exciting 2D game framework released in September 2013 with iOS 7. It is a graphics rendering and animation framework that gives you the power to easily animate textured images, play video, render text, and add particle effects. It also includes an integrated physics library. Sprite Kit is the first-ever game engine formally built into the iOS SDK.

In this chapter you will learn what Sprite Kit is and how you create a new Sprite Kit game using Xcode. You will then move on and create the beginnings of a Sprite Kit game starting from scratch. You'll learn about SKNodes and their subclasses, and you'll also use an SKSpriteNode to add both a background node and a player node to your game.

What You Need to Know

This section of this book assumes you have a basic understanding of how to build iPhone applications using Xcode and the Xcode Simulator. It also assumes you have a basic knowledge of the iOS/Mac programming language Swift. If you are not familiar with Swift, there is a brief introduction in the appendix at the back of this book.

This book will not cover how to program. It will focus only on Sprite Kit game programming.

What You Need to Have

To complete all of the examples in the book, you will need to have an Intel-based Macintosh running OS X 10.8 (Mountain Lion) or newer. You will also need Xcode 6+ installed. You can find both of these in the Apple App Store.

SuperSpaceMan

I feel the best way to learn anything is to do it. Therefore, in this book you are going to dive right in and create your own game. You will start off with the basic code for a 2D game, and you will add new features to the game as I introduce new topics with each chapter. At the end of the book, you will have a complete game.

The game you are going to create is inspired by Sega's popular Sonic Jump Fever (https://itunes.apple.com/us/app/sonic-jump-fever/id794528112?mt=8). It is a vertical scroller that accelerates the main character through obstacles and collectables, increasing your score as you collect rings.

This game will be similar, in that it is a vertical scroller, but your main character is going to be a space man who hurtles through space collecting power orbs while trying to avoid black holes that will destroy him.

Creating a Swift Sprite Kit Project

Before you can get started, you will need to create a Swift Sprite Kit project. So, go ahead and open Xcode and complete the following steps:

1. Select the menu File ➤ New ➤ Project.
2. Select Application from the iOS group.
3. Then select the Game icon. The choose template dialog should now look like Figure 1-1.

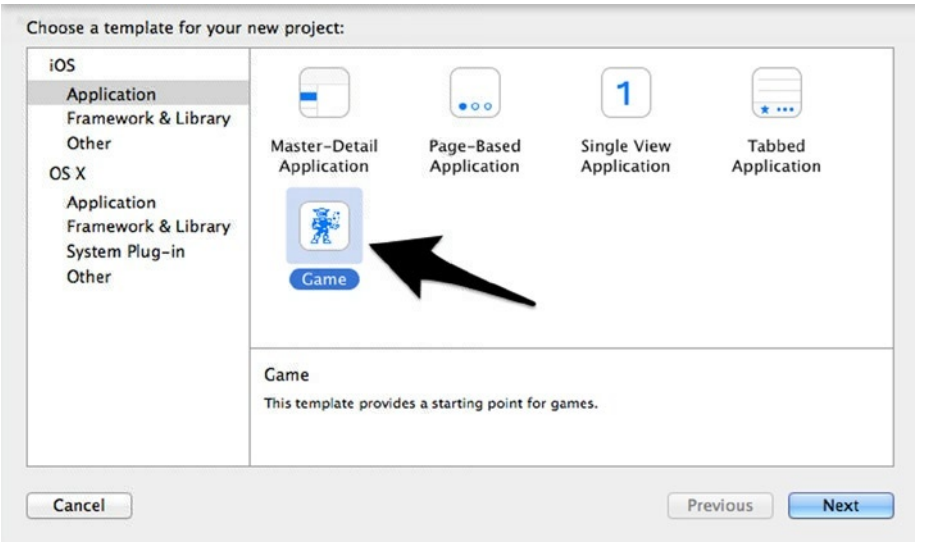

Figure 1-1. *The choose template dialog*

4. To move on, click the Next button.
5. Enter **SuperSpaceMan** for Product Name, **Apress** for Organization Name, and **com.apress** for Organization Identifier.
6. Make sure Swift is the selected language, Sprite Kit is the selected game technology, and iPhone is the selected device.
7. Before you click the Next button, take a look at Figure 1-2. If everything looks like this image, click the Next button and select a good place to store your project files and click the Create button.

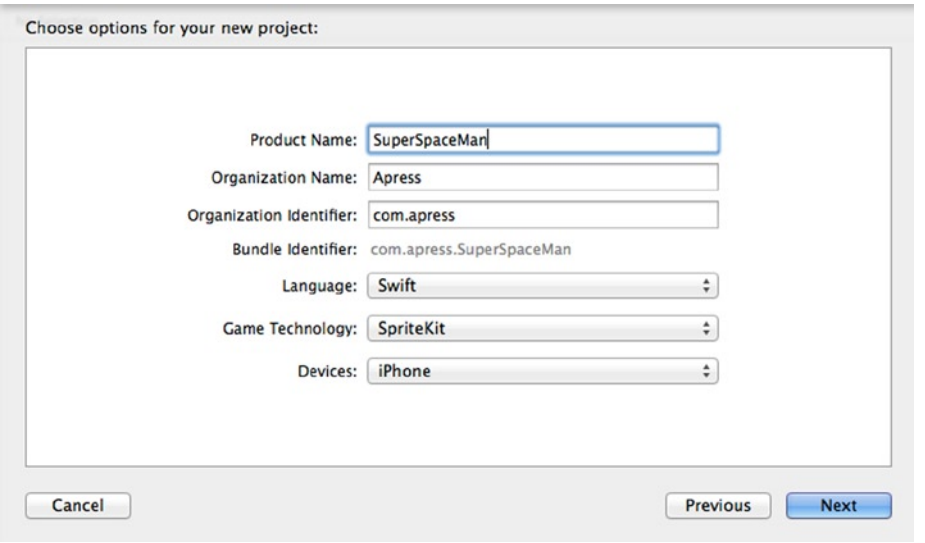

Figure 1-2. *The choose project options dialog*

> **Note** You will notice you are creating an iPhone-only game. This is only because the game you are creating lends itself better to the iPhone. Everything I will cover in this book translates to the iPad just as well.

You now have a working Sprite Kit project. Go ahead and click the Play button to see what you have created. If everything went OK, you will see your new app running in the simulator.

> **Note** The Xcode simulator may take awhile to start on some slower machines. Simulating Sprite Kit can be very taxing on your processors.

It does not do a whole lot yet, but there is more to it than the displaying of the universal "Hello, World!" Tap the simulator screen a few times. You will see rotating space ships displayed wherever you tap. Depending on where you tapped, you should see something similar to Figure 1-3.

Figure 1-3. *The Sprite Kit sample application*

Starting from Scratch

While the standard Sprite Kit template works great, you are going to be starting from scratch. Starting from nothing will allow you to see all the working parts in a Sprite Kit game and give you a much better understanding of what you are creating.

The first thing you need to do is make sure your game runs only in portrait mode. To do this, follow these steps:

1. Select the SuperSpaceMan project in the Project Explorer.
2. Then select SuperSpaceMan from Targets.
3. Deselect Landscape Left and Landscape Right.

At this point, your target settings should look like Figure 1-4.

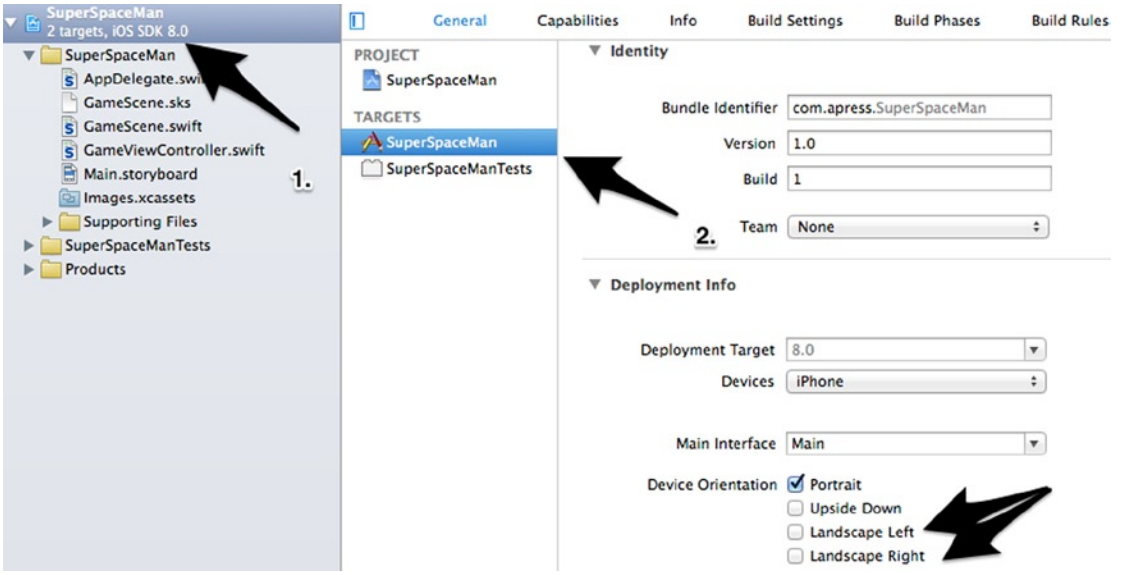

***Figure 1-4.** The SuperSpaceMan target settings*

The next thing you need to do is delete the file `GameScene.sks`. You will not be using the level editor in this book. You can find this file in the SuperSpaceMan group. Delete this file and then open `GameScene.swift` and replace its contents with the class in Listing 1-1.

Listing 1-1. GameScene.swift: The SuperSpaceMan Main GameScene

```
import SpriteKit

class GameScene: SKScene {

    required init?(coder aDecoder: NSCoder) {

        super.init(coder: aDecoder)
    }

    override init(size: CGSize) {

        super.init(size: size)

        backgroundColor = SKColor(red: 0.0, green: 0.0, blue: 0.0, alpha: 1.0)
    }
}
```

There is one more change you need to make before examining your baseline project. Open `GameViewController.swift` and replace its contents with the Listing 1-2 version of the same class.

Listing 1-2. GameViewController.swift: The SuperSpaceMan Main UIViewController

```
import SpriteKit

class GameViewController: UIViewController {

    var scene: GameScene!

    override func prefersStatusBarHidden() -> Bool {
        return true
    }

    override func viewDidLoad() {

        super.viewDidLoad()

        // 1. Configure the main view
        let skView = view as SKView
        skView.showsFPS = true

        // 2. Create and configure our game scene
        scene = GameScene(size: skView.bounds.size)
        scene.scaleMode = .AspectFill

        // 3. Show the scene.
        skView.presentScene(scene)
    }
}
```

Save all your changes and click the Play button once more. Wow, um. That was not very exciting. If you made all the changes, you should now be staring at a totally black screen with only the current frame rate displayed. This was the intent. You truly are starting from nothing.

Let's take a moment and examine each component of your new game. First, open `Main.storyboard`. Everything here should look pretty normal. You should see a single storyboard with a single `UIViewController`. Expand Game View Controller Scene in the Story Board Explorer and select Game View Controller, as shown in Figure 1-5.

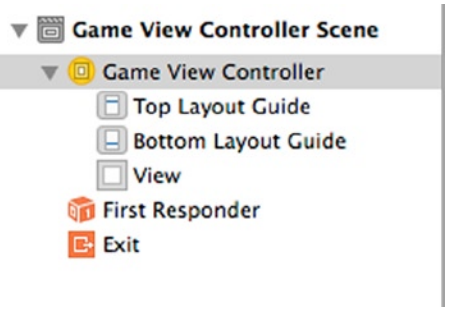

Figure 1-5. *Game View Controller Scene*

Now expand the Utilities view on the right side of Xcode and click Show the Identity inspector button. You will see the custom class of this `UIViewController` is your `GameViewController.swift`. Figure 1-6 shows you this connection.

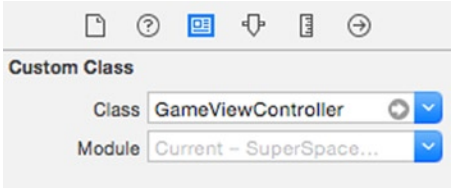

***Figure 1-6.** The custom class GameViewController*

There is one last thing to look at before you get back to the code portion of this tour. Go back to the Utilities view and select the Connections inspector. Notice the View outlet is connected to your `GameViewController.view`. Figure 1-7 shows this connection.

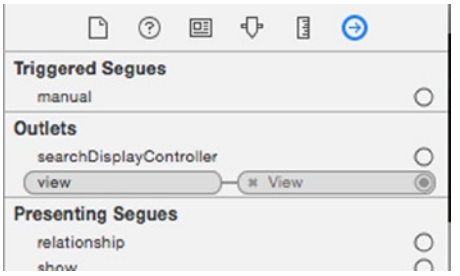

***Figure 1-7.** The View outlet*

The point of going through this examination of the storyboard is to show that while Sprite Kit is used to create games, the technology used to create games is just like what you would use to create any modern iOS app.

The GameViewController Class

Let's get back to the code. You can ignore `AppDelegate.swift`—it is the same boilerplate code you use to start all iOS Swift applications. `GameViewController.swift` is the best starting point. I included it earlier, but for the sake of convenience it is listed here again:

```
import SpriteKit

class GameViewController: UIViewController {

    var scene: GameScene!

    override func prefersStatusBarHidden() -> Bool {

        return true
    }
```

```
    override func viewDidLoad() {

        super.viewDidLoad()

        // 1. Configure the main view
        let skView = view as SKView
        skView.showsFPS = true

        // 2. Create and configure our game scene
        scene = GameScene(size: skView.bounds.size)
        scene.scaleMode = .AspectFill

        // 3. Show the scene.
        skView.presentScene(scene)
    }
}
```

Starting with the first line of this controller, you see a simple import including the Sprite Kit framework. This line makes all the Sprite Kit related classes available to your `GameViewController`. After that, you have a standard class definition—the `GameViewController` extends a `UIViewController`.

After the class definition, you see the declaration of the optional variable scene, which is declared as the type `GameScene`. `GameScene` is the class that will be doing most of your work. It is where you will be adding the game logic. You will look at this class in the next section.

Notice one thing about the `scene` variable. It is an optional. You know it is an optional because an exclamation point (!) follows its declaration. You declared this variable as an optional because you are not going to initialize it until the `viewDidLoad()` method fires and Swift requires you to initialize all properties in a class either at their declaration or in the `init()`. If you don't initialize a property in either of these locations, then you must declare the property as optional. You will see the use of optionals throughout all of the examples in this book.

After the scene declaration, you see an override of the `UIViewController`'s `viewDidLoad()` method. Here you return `true` because you don't want a status bar displayed in the game.

The next thing to check out is the `viewDidLoad()` method. This is where you really start to see your first active Sprite Kit code. The first thing you do, after calling `super.viewDidLoad()`, is to configure your main view. In the first step, you downcast your standard `UIView` to an `SKView`. The `SKView` is the view that will host your game scene mentioned earlier. For the most part, the `SKView` acts much like any `UIView`, with the exception that it has a collection of game-related properties and utility methods like the line following the downcast.

```
skView.showsFPS = true
```

This property of the `SKView` is used to show or hide the frames per second the application is rendering—the higher, the better.

After configuring the main view, you create and configure the `GameScene`.

```
scene = GameScene(size: skView.bounds.size)
scene.scaleMode = .AspectFill
```

The first line creates a new instance of the GameScene initializing the size to match the size of the view that will host the scene. After that, you set scaleMode to AspectFill. The scaleMode (implemented by the enum SKSceneScaleMode) is used to determine how the scene will be scaled to match the view that will contain it. Table 1-1 describes each of the available scaleMode properties.

***Table 1-1.** The SKSceneScaleModes*

SKSceneScaleMode	Definition
SKSceneScaleMode.Fill	The Fill scaleMode will fill the entire SKView without consideration to the ratio of width to height.
SKSceneScaleMode.AspectFill	The AspectFill mode will scale the scene to fill the hosting SKView while maintaining the aspect ratio of the scene, but there may be some cropping if the hosting SKView's aspect ratio is different. This is the mode you are using in this game.
SKSceneScaleMode.AspectFit	The AspectFit mode will scale the scene to fill the hosting SKView while maintaining the aspect ratio of the scene, but there may be some letterboxing if the hosting SKView's aspect ratio is different.
SKSceneScaleMode.ResizeFill	The ResizeFill mode will modify the size of the scene to fit the hosting view exactly.

> **Note** When setting the scaleMode property of the scene, you are using a shortened syntax to represent the mode you are setting, specifically, the .AspectFill mode. You can use this dot syntax because you know the type of the scaleMode property is an SKSceneScaleMode, which is an enum containing all of the scale modes.

Once you have the view and the scene configured, there is only one last thing to do—present the scene. This is done with the last line in viewDidLoad().

```
skView.presentScene(scene)
```

The GameScene Class

Now that I have walked you through each line of the GameViewController class, it is time to talk about the GameScene class. Again, for convenience's sake, I am including the source for the GameScene.swift file a second time:

```
import SpriteKit

class GameScene: SKScene {

    required init?(coder aDecoder: NSCoder) {

        super.init(coder: aDecoder)
    }
```

```
    override init(size: CGSize) {

        super.init(size: size)

        backgroundColor = SKColor(red: 0.0, green: 0.0, blue: 0.0, alpha: 1.0)
    }
}
```

As you look over GameScene, you will notice there is really not much to it. It extends SKScene and implements two init() methods; the first init() that takes an NSCoder can be ignored. You had to add this only because a Swift class does not inherit its parent's constructors. The init() you are interested in is the second method, which takes a CGSize parameter that represents the size you want the scene to be (in this case, the size you passed in from the GameViewController). After that, you pass the size to your superclass and then set the background color to black.

While there is not a whole lot to your current GameScene, this is where you will be doing almost all of your Sprite Kit work. SKScenes and the classes that extend them are the root nodes of all Sprite Kit content, and your GameScene will grow considerably as you move along in this book.

Adding a Background and Player Sprite

I have talked enough for one chapter. Let's get back to the game itself. In this, the last section of this chapter, you are going to just jump in and add a game background and a player sprite to your scene and see how they look.

Before you can do this, you need some image files. You can find all of the necessary assets for this book in the file assets.zip found at Apress.com. Go ahead and download and unzip this file.

Inside the unzipped folder you will find two folders one named Images and another named sprites.atlas. Now copy the entire spites.atlas folder directly into the SuperSpaceMan folder of the same project.

Next, open the Image.xcassets folder in Xcode. You will see a figure similar to Figure 1-8.

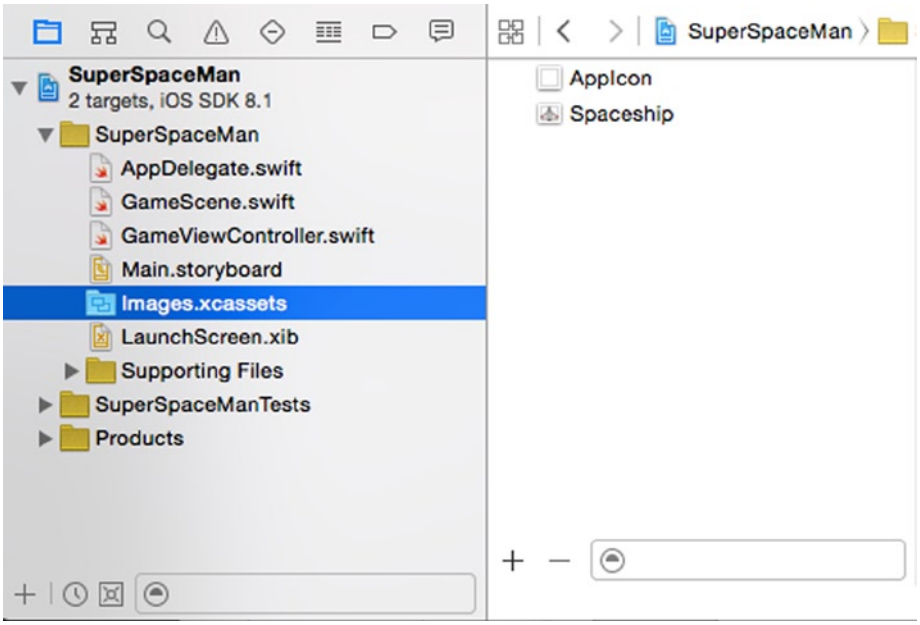

Figure 1-8. *Adding Image Assets*

Next, using Finder, browse to the Images folder in the previously downloaded zip file and finally select the three folders inside the Images directory and drag them onto the xcassets palette directly below the SpaceShip asset. When the files have been added, your xcassets palette will look like Figure 1-9.

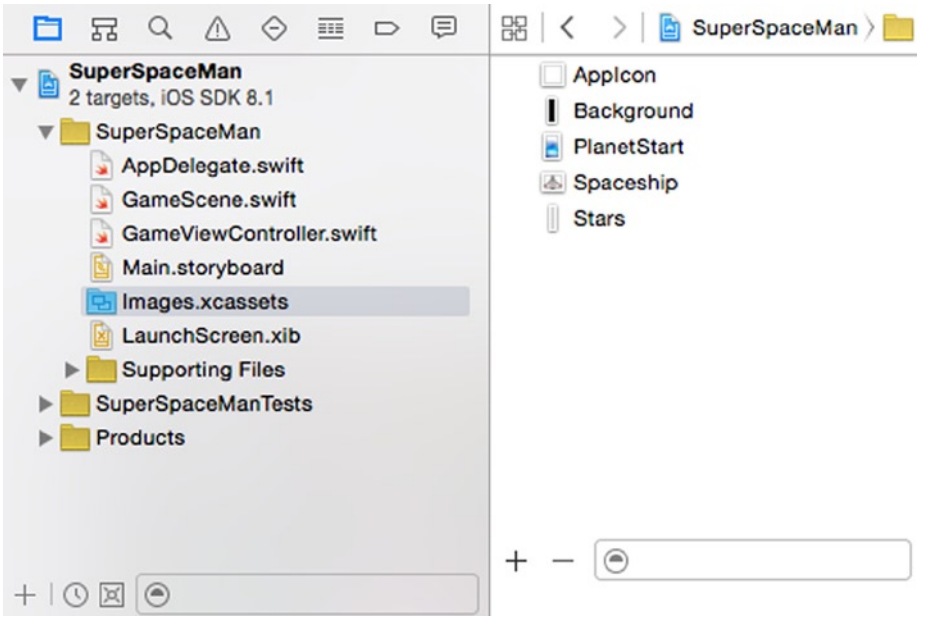

Figure 1-9. *The Added Image Assets*

Now that you have all of the images added to your project, let's put some of them to good use. Go back to `GameScene.swift` and add the following two lines to the beginning of the `GameScene` class:

```
let backgroundNode : SKSpriteNode?
var playerNode : SKSpriteNode?
```

Here you are adding two optionals, `backgroundNode` and `playerNode`, both of which are `SKSpriteNodes`. An `SKSpriteNode` is a descendent of an `SKNode`, which is the primary building block of almost all Sprite Kit content. `SKNode` itself does not draw any visual elements, but all visual elements in Sprite Kit based applications are drawn using `SKNode` subclasses. Table 1-2 defines the main descendants of `SKNode` that render visual elements.

Table 1-2. *The Descendants of SKNode That Render Visual Elements*

Class	Description
`SKSpriteNode`	A node that is used to draw textured sprites
`SKVideoNode`	A node that presents video content
`SKLabelNode`	A node that is used to draw text strings
`SKShapeNode`	A node that is used to draw a shape based upon a Core Graphics path
`SKEmitterNode`	A node that is used to create and render particles
`SKCropNode`	A node that is used to crop child nodes using a mask
`SKEffectNode`	A node that is used to apply Core Image filters to its child nodes

In this book you will be using three subclasses of SKNode: SKSpriteNode, SKLabelNode, and SKEmitterNode.

After adding the two SKSpriteNodes, add the following lines to the bottom of the GameScene.init(size: CGSize) method:

```
// adding the background
backgroundNode = SKSpriteNode(imageNamed: "Background")
backgroundNode!.anchorPoint = CGPoint(x: 0.5, y: 0.0)
backgroundNode!.position = CGPoint(x: size.width / 2.0, y: 0.0)
addChild(backgroundNode!)
```

The first line of this snippet creates an SKSpriteNode with an image named Background, which you just added to your Images.xcassets folder.

In this case you are drawing the background image. The next line of code determines where the new node will be anchored in your scene. Don't worry too much about this at the moment. I will be discussing anchor points in great detail in the next chapter. Just know that the anchor point of (0.5, 0.0) sets the anchor point of the background node to the bottom center of the node.

Next, you set the position of the backgroundNode. Here you are setting the node's position to an x-coordinate half the width of the scene, which is in the middle of the scene, and setting the y-coordinate to 0.0, which is the bottom of the scene.

The final line in the snippet adds the backgroundNode to your scene. To see what you have just accomplished, save your work and run the application again. You should now see your background displayed, as shown in Figure 1-10.

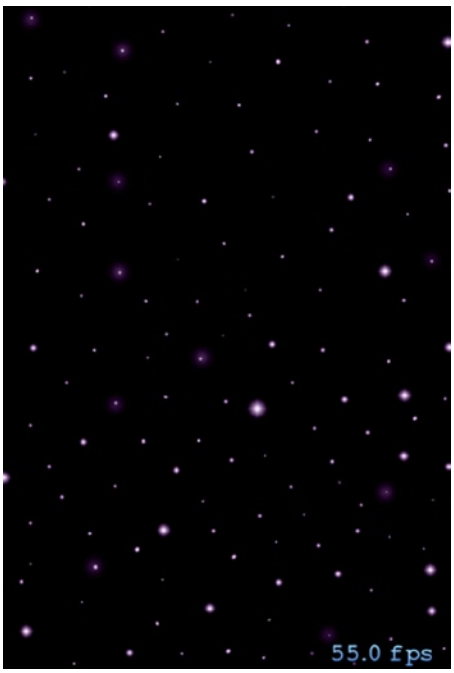

Figure 1-10. *The backgroundNode added to the GameScene*

That was easy enough. Now, let's add your player to your scene. Adding the player node is just as easy as adding the background. Take a look at the following snippet:

```
// add the player
playerNode = SKSpriteNode(imageNamed: "Player")
playerNode!.position = CGPoint(x: size.width / 2.0, y: 80.0)
addChild(playerNode!)
```

As you can see, you are creating a new `SKSpriteNode` using the image named `Player`. You then set the position of the `playerNode` and add it to the scene. Notice one difference here. You did not set the anchor point of the `playerNode`. This is because the default anchor point of all `SKNodes` is (0.5, 0.5), which is the center of the node. Again, don't worry about the positions or `anchorPoints` for now. I will be discussing them in the next chapter.

Go ahead and add this snippet to the bottom of the `GameScene.init()` method and save your changes. Now run the application one more time. You will now see the SuperSpaceMan positioned in front of the previously added `backgroundNode`, as shown in Figure 1-11.

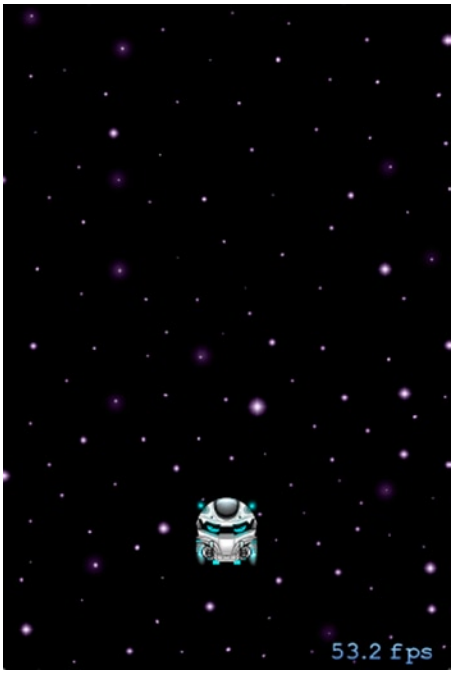

Figure 1-11. *The playerNode added to the GameScene*

After making these changes, your new GameScene.swift file should look like Listing 1-3.

Listing 1-3. GameScene.swift: The modified GameScene.swift

```
import SpriteKit

class GameScene: SKScene {

    let backgroundNode : SKSpriteNode?
    var playerNode : SKSpriteNode?

    required init?(coder aDecoder: NSCoder) {

        super.init(coder: aDecoder)
    }

    override init(size: CGSize) {

        super.init(size: size)

        backgroundColor = SKColor(red: 0.0, green: 0.0, blue: 0.0, alpha: 1.0)

        // adding the background
        backgroundNode = SKSpriteNode(imageNamed: "Background")
        backgroundNode!.anchorPoint = CGPoint(x: 0.5, y: 0.0)
        backgroundNode!.position = CGPoint(x: size.width / 2.0, y: 0.0)
        addChild(backgroundNode!)

        // add the player
        playerNode = SKSpriteNode(imageNamed: "Player")
        playerNode!.position = CGPoint(x: size.width / 2.0, y: 80.0)
        addChild(playerNode!)

    }
}
```

Summary

In this chapter you learned what Sprite Kit is and how you create a new Sprite Kit game using Xcode. You then dove in and created the beginnings of a Sprite Kit game starting from scratch. You learned about SKNodes and their subclasses, and you used an SKSpriteNode to add both a background node and a player node.

In the next chapter, you will dig a little deeper into Sprite Kit and discuss the details of the SKScene, including the coordinate system and anchor points. You will also look at how a scene's node tree is constructed.

Chapter 2

Sprite Kit Scenes and SKNode Positioning

In the previous chapter I talked about what Sprite Kit was and how you can use it to create 2D games. I then jumped right in and showed how to start working with the `SKSpriteNode` to create a background and player sprite and then showed how to add them to a game scene.

In this chapter, I am going to step back a bit and give you a deeper look at Sprite Kit scenes, including how scenes are built and why the order they are built in can change your game. I will close the chapter with a discussion of Sprite Kit coordinate systems and anchor points as they relate to `SKNode`s.

What Is an SKScene?

I used the `SKScene` object in the previous chapter to host the background and player nodes, but I really did not explain the scene I was using. I just used it to add the sprites and called it a day. It is now time to dig in and see how `SKScene` really works. I'll start by defining an `SKScene` object.

An `SKScene` object represents a scene of content in a Sprite Kit game. An `SKScene` object inherits from `SKEffectNode`, `SKNode`, `UIResponder`, and, of course, `NSObject`. It is constructed first by creating the scene and then by adding *n* number of other `SKNode`s to it. The scene plus all of its child nodes are called the *node tree*, and the scene is the *root* of the node tree. The nodes contained in a scene provide the content the scene will animate and render for display.

The following are the steps you performed in the previous chapter to create the node tree. They are the basic steps you will complete whenever setting up a game scene.

1. Create the `GameViewController`.
2. Have the `GameViewController` create its `UIView`.
3. Inside the `GameViewController.viewDidLoad()`, downcast the `UIView` to an `SKView` and set the `showFPS` property to `true`.

```
let skView = view as SKView
skView.showsFPS = true
```

4. Create an instance of the `SKScene` named `GameScene`, passing it its size in the constructor and setting the `scaleMode` property.

```
scene = GameScene(size: skView.bounds.size)
scene.scaleMode = .AspectFill
```

5. Inside the `init()` of the `GameScene`, add the `SKSpriteNode` objects to the scene.

```
backgroundNode = SKSpriteNode(imageNamed: "Background")
backgroundNode!.anchorPoint = CGPoint(x: 0.5, y: 0.0)
backgroundNode!.position = CGPoint(x: size.width / 2.0, y: 0.0)
addChild(backgroundNode!)

playerNode = SKSpriteNode(imageNamed: "Player")
playerNode!.position = CGPoint(x: size.width / 2.0, y: 80.0)
addChild(playerNode!)
```

6. Present the complete scene in the `GameViewController.viewDidLoad()` method.

```
skView.presentScene(scene)
```

At this point, you have a complete scene with a complete node tree. You can always add more nodes as the game progresses, but these are the basic steps that you will complete whenever you create a new `SKScene`.

The SKScene Rendering Loop

In this section, I will describe what happens after the `SKScene` is presented by the `SKView`. In a more traditional iOS application, you would render the view's content only once, and it would stay static until the model that the view is presenting changes. This is fine for a business app, but a game has the potential to constantly change.

Because of this dynamic characteristic, Sprite Kit is constantly updating the scene and its contents. This constant updating is called the *rendering loop* (see Figure 2-1).

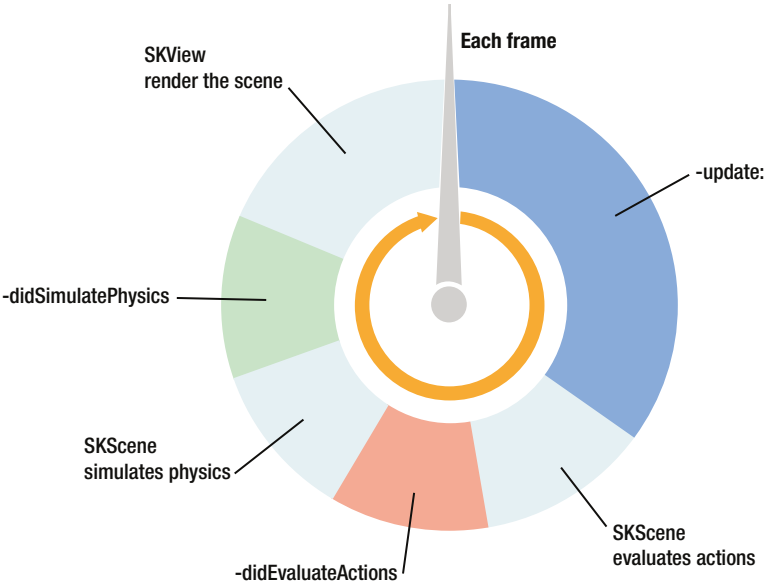

Figure 2-1. *The Sprite Kit rendering loop*

Each iteration of this loop generates the next frame in the scene. The steps involved in generating the next frame of a scene are as follows:

1. The scene calls its `update()` method. This is where you will have most of your game logic. More often than not, you will be moving nodes around, adding new actions to existing nodes, and handling user input. (I will talk about the `update()` method in Chapter 4.)
2. The scene next performs all programmed actions on its children. In this step, the scene will execute any actions you may have set up in step 1. (I will talk about actions in Chapter 5.)
3. The scene then calls the `didEvaluateActions()` method. This is where you would put any post-action game logic. An example would be testing the position of a node, after the actions were performed, and responding accordingly.
4. Next the scene executes any physics simulations on physics bodies in the scene. (I will discuss physics in Chapter 3.)
5. The scene calls the `didSimulatePhysics()` method. This is much like the `didEvaluateActions()` method in that this is where you would add any game logic to be performed after all the physics simulations are completed. This is your last chance to perform any game logic before the scene is rendered.
6. The scene is rendered.

You will see examples of each step in the rendering loop as you progress through each chapter of this book.