Getting Started with SpriteKit

Throughout this book, you will develop a complete game. The beautiful designs implemented in the game in this book will allow you to learn the basics of 2D game development, including creating and moving sprites and adding them to a game scene. You will also discover how to apply advanced techniques such as collision detection, action execution, playing music, or running animations to give a more professional aspect to the game. You will finish your first game by learning how to add a main menu and a tutorial, as well as saving and loading data to and from the player's device.

Finally, you will find out how to apply some mobile games techniques, such as accelerometer use and touch detection.

Who this book is written for

Getting Started with SpriteKit is for beginner-level iOS developers who want to add an extra edge to their apps and create amazing games using SpriteKit. It doesn't matter whether you have experience in iOS development or not as this book will show you the swift tricks you can use to create games.

What you will learn from this book

- Create and configure a SpriteKit project from scratch
- Load and manage the basic elements of games such as sprites, labels, and geometric primitives
- Handle touch events, detect collisions, and play audio files
- Create complex elements, animate sprites, and run the parallax effect
- Complete your games with key components such as a main menu, transitions between scenes, a tutorial, and the ability to load and save data
- Increase the efficiency of your device using the accelerometer or by adding shaders, lights, and shadows
- Learn complementary techniques such as creating or finding audio resources, applying SpriteKit to apps, and using third-party tools

In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 4 'From Basic to Professional Games'
- A synopsis of the book’s content
- More information on Getting Started with SpriteKit
Jorge Jordán is an iOS indie developer who's passionate about how things work since his childhood. This is the reason why he graduated in computer science and became a Java developer. After buying his first iPhone, he became deeply interested in its technology and spent his spare time learning how to develop apps for Apple's smartphones.

Over time, he founded www.insaneplatypusgames.com, where he tries to make his dreams and games come true. He is also a member of the tutorial team at www.raywenderlich.com.

Also, he has worked on a book titled Cocos2d Game Development Blueprints, Packt Publishing.

In his free time, he loves to play video games, play bass guitar, and watch TV series.

You can follow him on Twitter; his Twitter handle is @jjordanarenas.
I think that I'm not wrong if I say that we as developers have the healthy (or unhealthy) habit of trying to decompose mentally (or even physically) everything that comes to our hands into smaller pieces to understand how it works.

In my case, I have to acknowledge that I am a video game lover and a restless developer, and every time I get a game or an app, I can't stop myself from thinking about how an animation or a particular effect is achieved.

This concern helped me learn how to develop apps and games. Thanks to this, I discovered that Apple provides a powerful set of tools that can be combined to create amazing games and stunning apps.

One of these tools is SpriteKit, a 2D game engine that is at the forefront of the frameworks that are available on the market and which can be included in any application to provide its dynamic features.

If you are reading these lines, you probably are either a game lover with a curiosity to learn how video games are developed, or you are an app developer who wants to know how the animations that you have seen in other apps or games have been created.

At this point starts a journey full of new experiences and knowledge that will satisfy all your curiosity. So take a seat, get comfortable, and start reading and enjoying.
What this book covers

This book contains six chapters that will guide you through the process of creating a video game with the tools provided by SpriteKit. At the same time, you will learn how to deal with the common difficulties that you may come across when developing games and how to apply the techniques that you learn here to give a stunning look to an app.

Chapter 1, *The First Steps toward SpriteKit*, covers the basics of developing a game with SpriteKit. In this chapter, you will learn what a game engine is and what a new SpriteKit project looks like. In addition to this, you will learn the purpose of the SKNode and SKScene classes and how to utilize them to add a background and a sprite into the game.

Chapter 2, *What Makes a Game a Game?*, shows the main techniques used in game development, such as moving sprites on the scene, detecting touches, and handling collisions. In addition to this, you will learn how to create and update labels and play music and sound effects.

Chapter 3, *Taking Games One Step Further*, teaches advanced techniques such as how to create complex nodes by extending the SKNode class or implement the parallax effect. You will also learn how to draw geometrical primitives and animate sprites.

Chapter 4, *From Basic to Professional Games*, helps us provide our game with the needed components to consider it as a finished product. You will learn how to create a Game Over condition for the game and a main menu scene from where you will learn how to transition to a tutorial that you will create for players so that they know the mechanism of the game. You will also learn how to save and load data from internal and external sources.

Chapter 5, *Utilizing the Hardware and Graphics Processor*, teaches you how to get the most advantage from the hardware of physical devices using the accelerometer or creating stunning visual effects by adding shaders, lights, and shadows into a game.

Chapter 6, *Auxiliary Techniques*, covers the development of special effects, which are also known as particle systems, and ways to combine SpriteKit with UIKit in order to create robust games and dynamic apps. You will also learn how to use third-party tools to create custom audio and fonts and find resources that can be used in games.
In the last chapter, you had a look at how to include some advanced techniques to create games with some complexity. In this chapter, you will incorporate some components in your project, which will complete the game. You will create a condition to end a game and learn how to stop the game and create a game over scene. Also, you will add an initial menu to the game and learn how to create transitions between the different scenes of a game. You will also learn how to take advantage of this feature to include a tutorial in the game to teach users how to play it. Finally, you will have an understanding of how to store information, such as the best scores, and load data from external files.

The topics that you will learn in this chapter are as follows:

- How to end a game
- How to add a main menu to the game
- How to create transitions between scenes
- How to develop a tutorial
- How to load and save data

**Ending the game**

Before going any further, we will need to open the initial project for this chapter, which is similar to the one that we had at the end of the previous chapter. Therefore, unzip 7338_04_Resources.zip, where you will find InsideTheHat_init.zip. Unzip this and open the project with Xcode.
We had to look at the code needed to decrease the number of life points as well as the red life bar length. In this section, we will take advantage of this code to end the game when the number of life points is 0.

Usually, when a game is over, everything stops moving and some kind of text alert for the player pops on the screen. So, let’s see how we can do this in a SpriteKit project.

For this step, we will need a new label variable. So, let’s declare it by adding the following line at the top of GameScene:

```swift
private var labelGameOver: SKLabelNode!
```

Then, add the following block of code in the resetPositionAction action of the initializeEnemyActions method just after self.updateLifeBar() function:

```swift
// If we have lost all the life points
if self.lifePoints == 0 {
    self.gameOver()
}
```

When a collision with an enemy happens, we update the life points. If this value equals to 0, we call a new method. So, let’s implement it using the following lines:

```swift
func gameOver() {
    // Initialize the label with a font name
    labelGameOver = SKLabelNode(fontNamed:"MarkerFelt-Thin")
    // Set color, size and position
    labelGameOver.fontColor = UIColor.redColor()
    labelGameOver.fontSize = 60
    labelGameOver.position = CGPoint(x:view!.bounds.size.width/2, y:view!.bounds.size.height)
    // Specifying zPosition
    labelGameOver.zPosition = 5
    // Set text
    labelGameOver.text = "GAME OVER"
    // Add the label to the scene
    addChild(labelGameOver)
}
```

In this method, we initialized the label variable with the same font as the one that we used for the score label. Then, we set red as its font color and specified a font size that’s big enough to cover the screen.
Chapter 4

We are going to make the label appear at the top of the screen and in the center using a sequence of actions in order to achieve a dynamic result. This is the reason why we set the label's initial position at the top of the view in the center in a horizontal fashion.

Finally, we specified the zPosition value to ensure that the text is shown over the rest of the elements. We set a piece of text to indicate that the game is over, and we add the label to the scene.

Now that we have the label initialized, it's time to create the movement action that will make it appear dynamically. Add the following block of code at the end of the gameOver method:

```swift
// Creating movement action
let actionMoveDown = SKAction.moveTo(CGPoint(x:view!.bounds.size.width/2, y:view!.bounds.size.height/2), duration: 0.25)

// Creating movement action
let actionMoveUp = SKAction.moveTo(CGPoint(x:view!.bounds.size.width/2, y:view!.bounds.size.height/2 + 60), duration: 0.25)

// Creating block action
let stopGame = SKAction.runBlock {
    // Stop game
    self.view?.paused = true
}

// Creating block action
let stopMusic = SKAction.runBlock {
    // Stop background music
    self.backgroundMusic.stop()
}
```

We created a move action that will bring the label to the center of the screen, and then we created another one that will move it a little upwards.

We also created a runBlock action, where we execute the following command:

```swift
self.view?.paused = true
```

The paused attribute will set all the animations on the scene to a standby mode and it's a property that we can take advantage of if we want. For example, you can use it to stop the game when the Configuration or Pause menu is opened.

Finally, we want to stop the background music. Therefore, we created another block action to stop the audio player.
Let's add the last lines at the end of the `gameOver` method:

```swift
// Creating sequence of actions
let sequence = SKAction.sequence([actionMoveDown, actionMoveUp, actionMoveDown, stopGame, stopMusic])
// Run sequence
labelGameOver.runAction(sequence)
```

Using the preceding lines, we created a sequence of actions that will combine everything that moves the label from the top of the view to the center: the action that will move it a little upwards and the other action that will recover its position at the center of the view as well as the action that will pause the game and the music.

Finally, on executing this sequence, you will see something that's similar to the following screenshot if you run the game now:

![Game Over Label](image)

You will realize that the game doesn't stop instantly, because it needs to wait for the `Game Over` label's `move` action to finish before the scene is paused. We can fix it instantly by removing all the actions. But I wanted to show you this `paused` attribute as it can be used in other circumstances.
3-star challenge: restarting a game

We have seen how we can end our game, but what happens if the user wants to play it again? Should they kill the app and restart it just to play a new game? This is not a kind way of treating players. So let’s provide them with a way to replay the game easily.

I recommend that you take advantage of all the knowledge that you have acquired so far in order to create a label when the game is over, which will restart the game. With what we have learned in the previous chapters, you will be capable of finding a solution for this challenge.

Try to develop your own solution and then compare it with the following one.

Solution

First of all, we are going to need a label. So let’s declare label by adding the following line at the top of GameScene:

```swift
private var labelResetGame: SKLabelNode!
```

We will initialize labelResetGame when the game is over so that labelResetGame will happen in the gameOver method. For this purpose, replace the following line in the aforementioned method:

```swift
let sequence = SKAction.sequence([actionMoveDown, actionMoveUp, actionMoveDown, stopGame, stopMusic])
```

Replace this line with the following lines:

```swift
// Creating block action
let showLabelResetAction = SKAction.runBlock {
    // Show reset game label
    self.showLabelReset()
}
let sequence = SKAction.sequence([actionMoveDown, actionMoveUp, actionMoveDown, stopGame, stopMusic, showLabelResetAction])
```
We have created a new runBlock action. Therefore, we can make a call to a new method when the game is over. That's the reason why we have added this new action at the end of the sequence. Thus, the new label will be shown just when the game is stopped.

Let's implement this new method by adding the following block of code:

```swift
func showLabelReset() {
    // Initialize the label with a font name
    labelResetGame = SKLabelNode(fontNamed: "MarkerFelt-Thin")
    // Set color, size and position
    labelResetGame.fontColor = UIColor.greenColor()
    labelResetGame.fontSize = 30
    labelResetGame.position = CGPoint(x: view!.bounds.size.width/2, y: view!.bounds.size.height/2 - 60)
    // Specifying zPosition
    labelResetGame.zPosition = 5
    // Set text
    labelResetGame.text = "Reset Game"
    // Set node's name
    labelResetGame.name = "reset_label"
    // Add the label to the scene
    addChild(labelResetGame)
}
```

This method will initialize the previously declared label with the same font name that we have been using so far. Then, we specified its font color, which will be green, and we set its font size and position. Note how we set this label at the center but a little below the initial position.

We want the label to be visible. Therefore, we need to specify the zPosition value so that it's big enough for the text to be over the rest of the elements in the scene.

Finally, we set its text and specified a key value for its name attribute so that we can take advantage of it when identifying whether a user has touched this node. Once the label is created, we add it to the scene.
Run the game now, and you will see how this step affects it:

Now, we have to provide this label with the functionality of restarting the game. I thought that it would be a good idea to check the touchesBegan method to find out whether the user's interactions correspond to the label. Therefore, add the following lines at the end of the aforementioned method:

```swift
// Check if label touched
if self.nodeAtPoint(touch.locationInNode(self)).name == "reset_label"
{
    self.restartGame()
}
```
From Basic to Professional Games

This means that if the touch location in the scene corresponds to the node named reset_label, we will call a new method. Implement this method using the following block of code:

```swift
func restartGame() {
    // Reset values
    score = 0
    lastFrameTime = 0
    deltaTime = 0
    // Set doors to nil
    leftDoor = nil
    centerDoor = nil
    rightDoor = nil
    // Remove all children on scene
    self.removeAllChildren()
    // Remove all actions
    self.removeAllActions() // Restart the game
    self.view?.presentScene(self)
}
```

When restarting the game, we need to set the initial values of the variables that we use. That's the reason why we set `score`, `lastFrameTime`, and `deltaTime` to 0.

We set the doors variables to `nil` just to ensure that they are correctly reinitialized. Then, we removed all the children from the scene. So, when the game restarts, everything works fine and there are no errors.

Finally, we removed all the actions in the scene to start the game from scratch and then we called the following method:

```swift
    self.view?.presentScene(self)
```

This will replace the current scene with the specified one, which means that the current scene will replace the specified one with a new instance of the game's scene.
Run the game now, and you will see how we can easily play the game any number of times that we want:

![Game screenshot]

**Creating a main menu**

When playing mobile games, you will usually find an initial menu screen when you run it. These screens are usually shown to provide players with a way to modify some configurations, such as turning on or turning off the game sound, the selection of a game's difficulty, or just a way to hide a big load of components.

We are going to create a menu that will be shown when the game is run, where we will show a background and a button to start playing the game.

For this purpose, we will need to create a new scene file to manage this new screen behavior. To do so, perform the following steps:

1. In Xcode, select the File menu at the top of the screen.
2. Select New | File, and you will see the Files dialog box.
3. Select the Swift File option under iOS | Source and click on Next.
4. Name it MenuScene and choose the folder where you want it to be saved before clicking on Create button.
As you will see, the new file is almost empty. So let's fix this by adding the following lines to the MenuScene.swift file:

```swift
import SpriteKit

class MenuScene: SKScene {
}
```

We are creating the new file as a subclass of SKScene. Therefore, it will have the potential that the subclass provides. Now, we need to add a background. To do so, add the following variable declaration inside the implementation section of the new class:

```swift
private var background: SKSpriteNode!
```

This line doesn't need an in-depth explanation because we are just declaring a sprite variable as we have done before. Now, let's initialize it by calling a new method. Add the following block of code after the sprite variable:

```swift
override func didMoveToView(view: SKView) {
    self.initializeMenu()
}
```

We have added the needed didMoveToView method, and it is mandatory for it to be overridden as a subclass of SKScene. We also called a method, where we will initialize the menu. So let's implement it using the following lines:

```swift
func initializeMenu() {
    // Initialize menu background
    background = SKSpriteNode(imageNamed: "menu")
    background.zPosition = -1
    background.position = CGPoint(x: view!.bounds.size.width/2, y: view!.bounds.size.height/2)
    // Add the background
    addChild(background)
}
```

With these lines of code, we created a new sprite background, specifying a zPosition value, so that it has a position that's lower than that of the view. We set its position so that it is placed at the center of the view and then we added it to the scene.
Before running the game, we need to include some new images for the background. So perform the following steps to achieve this:

1. Right-click on **Art** and select **Add Files to InsideTheHat...**
2. You’ll find menu.png, menu@2x.png, menu@2x-ipad.png, and menu@3x.png in the 7338_04_Resources folder that you unzipped previously. Select these four files and click on **Add**.

At this point, the game will not show the new scene when it is initialized. We first need to make a little change to GameViewController. Replace the following line in the **viewDidLoad** method:

```swift
let scene = GameScene(size: view.bounds.size)
```

Replace this line with the following one:

```swift
let scene = MenuScene(size: view.bounds.size)
```

In this way, we will initialize the brand new menu as soon as we run the game. So now that everything is ready, execute the project. You will see the following screen:

![Image of the initial menu](image)

Now that we have the initial menu, we have no way to play the game. For this purpose, we will need some way to show the game’s scene. So keep reading and you will learn how to achieve it.
Transitions and scenes

Usually, games comprise more than one scene. For example, games use a different scene to show a **Configuration** menu, to show the classification table once the game is over, to show a screen with in-app purchases, and so on.

Due to this, we need a way to move between the scenes. The solution that's provided by most game engines is transitions, which are also known as the **SKTransition** instances in SpriteKit.

The SKTransition class

The SKTransition class inherits from **NSObject** and provides us with the ability of creating animated transitions between the scenes of the game.

We have several methods in this class that will allow us to create different types of transitions. For example, we can choose one of the following methods:

- **crossFadeWithDuration**: This method will create a typical cross-faded transition that will take the specified duration to end the transition.
- **doorsCloseHorizontalWithDuration**: This will create the effect of a door closing horizontally, taking the specified duration to end the door closing event.
- **doorsCloseVerticalWithDuration**: This will create the effect of a door closing vertically, taking the specified duration to end the door closing event.
- **doorsOpenHorizontalWithDuration**: This will create the effect of a door opening horizontally, taking the specified duration to end the door opening event.
- **doorsOpenVerticalWithDuration**: This is similar to the preceding method, but the only difference is that it creates an effect of a door opening vertically.
- **doorwayWithDuration**: In this case, the outgoing scene disappears as a pair of doors open, while the new scene moves from the background to the foreground.
- **fadeWithColor**: This transition will first fade to the specified color and then it will fade to the new scene, taking the specified time to finish the process of fading.
- **fadeWithDuration**: This is similar to the previous one, but in this case, the chosen color is black, and it will take the specified time to finish the fading process but according to the specified time.
• flipHorizontalWithDuration: The previous and the next scenes are interchanged by flipping across a horizontal line at the center of the view.

• flipVerticalWithDuration: This is similar to the preceding method, but in this case, the previous and the next scenes are interchanged by flipping across a vertical line at the center of the view.

• moveInWithDirection: The new scene will appear after following the specified direction (up, down, left, or right), and taking the specified time to finish the process.

• pushWithDirection: The new scene appears after following the specified direction (up, down, left, or right), and pushing the previous scene out of view and taking the specified duration to end the scene process.

• revealWithDirection: In this case, the scene that moves after following the specified direction is the old one and, while it moves, it will reveal the new scene. This method also allows us to specify a duration value.

• init(CIFilter:duration): This method allows us to create a transition by using a Core Image filter and specifying a duration value.

By default, both the outgoing and incoming scenes are paused. Therefore, no actions will happen on any of them, but we can choose to pause just one of them by calling the pausesIncomingScene or pausesOutgoingScene properties.

Now that we know the different options that we have to make transitions between scenes, let’s create one to move from the main menu scene to the game scene.

First of all, we will add a text label in the initial scene that will create the transition when the player touches it. So, let’s declare a new variable in MenuScene:

```swift
private var labelInitGame: SKLabelNode!
```

Let’s initialize it by adding the following block of code at the end of the `initializeMenu` method:

```swift
// Initialize the label with a font name
labelInitGame = SKLabelNode(fontNamed:"Arial Bold")
// Set color, size and position
labelInitGame.fontColor = UIColor(red: 0.929, green: 0.129, blue: 0.486, alpha: 1.0)
labelInitGame.fontSize = 60
labelInitGame.position = CGPoint(x:view!.width/2, y:view!.height/2)
// Set text
labelInitGame.text = "Init Game"
```
// Set node's name
labelInitGame.name = "init_game_label"

// Add the label to the scene
addChild(labelInitGame)

We initialized the label in a way that's similar to how we have initialized labels previously. We first created the label by specifying a font name. Then, we set its position. This time, we wanted to set the label with the same color as that of the rabbit's snout. Therefore, we created the UIColor object with the red, green, and blue attributes.

In order to specify the red, green, and blue attributes values, you need to choose the CGFloat values between 0.0 and 1.0.

We also set the alpha property to 1.0 value because we want the label to be fully visible. Then, we specified the font size and set label at the center of the view.

As we want the label to show its purpose, we set its text and specified the node's name property so that it can be used later to check whether screen has been touched.

Finally, we added it to the scene. Now, run the game and have a look at the results:
Now that we have the label on the screen, we just need to add its functionality. You can detect when it has been touched by adding the following lines of code:

```swift
override func touchesBegan(touches: Set<UITouch>, withEvent event: UIEvent?) {
    if let touch = touches.first {
        let location = touch.locationInNode(self)
        // Check if label touched
        if self.nodeAtPoint(location).name == "init_game_label" {
            self.initGame()
        }
    }
}
```

We have overridden the `touchesBegan` method so that we can detect and handle the touches on this scene. In this method, we take the location of the first touch and check whether the touch coincides with the *init game label*.

If it does, we call a new method, which can be implemented with the following block of code:

```swift
func initGame() {
    // Create scene transition
    let sceneTransition = SKTransition.
doorsOpenVerticalWithDuration(1.25)
    // Create next scene
    let gameScene = GameScene(size: view!.bounds.size)
    // Present next scene with transition
    self.view?.presentScene(gameScene, transition: sceneTransition)
}
```

We first created a scene transition that will show the next scene while the first menu disappears vertically, taking 1.25 seconds for the transition.

Then, we created the next scene that we want to show, which was created using the view's size as usual. Finally, we presented the new scene using the desired transition.
Run the game now and have a look at the new transition, but you can choose the one that you prefer:

![Init Game]

**Creating a tutorial**

When we first play a game, we usually go through a guided path called a tutorial that will show us how to play the game. This path will just let us perform some specific actions so that the players learn the mechanics of the game and the operations that are allowed in the game.

The best way of developing a tutorial is by thinking about it as a state machine, where each state corresponds to one step of the tutorial and where we can specify the actions allowed on each state and learn how to pass from one state to the next one.

In this section, we are going to develop a tutorial that will consist of five states or steps, where some visual elements will teach players how to implement them.

For this purpose, we will need to create an enumerated object to handle the different steps of the tutorial. So add the following lines at the top of the `GameScene` class just after the `import` section:

```swift
enum TutorialSteps : UInt32 {
    case TUTORIAL_STEP_1 = 0
    case TUTORIAL_STEP_2 = 1
    case TUTORIAL_STEP_3 = 2
```
As you can see, the enumeration consists of five tutorial steps and a step to specify that the tutorial has finished.

As we want to show some visual elements so that players understand how to play the game, we will need some variables. So add the following block of code just after

```swift
private var labelResetGame: SKLabelNode!
```

```swift
private var tutorialStep: TutorialSteps = .TUTORIAL_STEP_1
private var tutorialImage: SKSpriteNode!
private var labelTutorial: SKLabelNode!
private var tutorialFrame: SKShapeNode!
```

We initialized one variable to store the tutorial steps that will allow us to know which information needs to be shown each time (this is initialized to the first step), a sprite to show an image of a hand, and a label that shows an explanatory text. We also declared a shape node, as we are going to take advantage of it to frame the different places where players can touch.

To initialize the tutorial, we are going to perform a call to a new method. So, add the following lines at the beginning of `didMoveToView`:

```swift
if tutorialStep != .TUTORIAL_ENDED {
    self.initializeTutorial()
}
```

If the tutorial step is not the last one, which means that the tutorial has finished successfully, we will initialize it by calling the following method:

```swift
func initializeTutorial() {
    // Create action
    let pauseForTutorial = SKAction.runBlock {
        // Pause game for the tutorial
        self.view?.paused = true
        // Initialize tutorial image
        self.tutorialImage = SKSpriteNode(imageNamed: "hand")
        // Set image position
        self.tutorialImage.position = CGPoint(x:self.view!.bounds.size.width/6, y: self.view!.bounds.size.height/3)
    }
    // Pause game for the tutorial
    self.view?.paused = true
    // Initialize tutorial image
    self.tutorialImage = SKSpriteNode(imageNamed: "hand")
    // Set image position
    self.tutorialImage.position = CGPoint(x:self.view!.bounds.size.width/6, y: self.view!.bounds.size.height/3)
}
```
From Basic to Professional Games

// Specifying zPosition
self.tutorialImage.zPosition = 7
// Add the image to the scene
self.addChild(self.tutorialImage)
}
}

We'll take advantage of this method to initialize the different elements that will help us guide the player. We will stop the game and show the first state of the tutorial. This is the reason why we are going to initialize the variables in a runBlock action.

In this block, we pause the game and initialize the hand node, setting it in the middle of the first third of the screen. We specify its zPosition value, as we want it to be fully visible. Then, we add it to the scene.

Now, add the following block of code at the end of pauseForTutorial:

// Initialize tutorial frame
self.tutorialFrame = SKShapeNode(rectOfSize: CGSize(width: self.view!.bounds.width/3, height: self.view!.bounds.size.height))
// Set frame's position
self.tutorialFrame.position = CGPoint(x:self.view!.bounds.size.width/6, y: self.view!.bounds.size.height/2)
// No border
self.tutorialFrame.lineWidth = 0
// Specify zPosition
self.tutorialFrame.zPosition = 6
// Set frame color
self.tutorialFrame.fillColor = UIColor.whiteColor()
// Set alpha value
self.tutorialFrame.alpha = 0.5
// Set node's name
self.tutorialFrame.name = "tutorial_frame"
// Add frame to scene
self.addChild(self.tutorialFrame)

We initialized the frame so that it will cover the first third of the view (the one on the left), from the top to the bottom, specifying that we don't want it to have borders, and we set its zPosition property in such a way that it is above the image of the hand.

Then, we specified its color and alpha values so that we can see it as a highlighted zone, and we specified a name value to use it as an identifier for touches. Finally, we added the frame to the scene.
We still need to initialize the label. So, add the following lines of code at the end of pauseForTutorial:

```swift
// Initialize label
self.labelTutorial = SKLabelNode(fontNamed:"MarkerFelt-Thin")
// Set color, size and position
self.labelTutorial.fontColor = UIColor.blackColor()
self.labelTutorial.fontSize = 30
self.labelTutorial.position.x = self.tutorialImage.position.x
self.labelTutorial.position.y = self.tutorialImage.position.y + 50
// Specifying zPosition
self.labelTutorial.zPosition = 7
// Set text
self.labelTutorial.text = "Touch"
// Add the label to the scene
self.addChild(self.labelTutorial)
```

With these lines, we initialized the label the way we are used to doing by specifying the same font and color used in the score label. We chose a size that's large enough to make the label clearly visible, and we set it on the left-hand side of the screen, a little above the hand node. Then, we set its zPosition property and text and added it to the scene.

Once we have the tutorial elements on the screen, we just need to show them. So, add the following block at the end of initializeTutorial:

```swift
// Creating a delay action
let delayAction = SKAction.waitForDuration(1.0)
let sequence = SKAction.sequence([delayAction, pauseForTutorial])
// Running the non-ending sequence
self.runAction(sequence)
```

We created a delay action so that the tutorial will be shown one second after the game is initialized, and then we showed the information of the first state of the tutorial.
Execute the project now, and you will see how it looks:

![Game screenshot](image)

**Updating the tutorial steps**

As you can see, the game is paused, and it should start when we touch the frame to the left. So, let's add the needed code for this purpose.

Replace the following line in `touchesBegan`:

```swift
self.moveRabbitToNextLocation(location)
```

Replace the preceding line with the following block of code:

```swift
if self.tutorialStep != .TUTORIAL_ENDED && self.nodeAtPoint(location).name == "tutorial_frame" {
    self.updateTutorial()
} else if self.tutorialStep == .TUTORIAL_ENDED {
    self.moveRabbitToNextLocation(location)
}
```
With the preceding block of code, we specified that we want to update the tutorial step if the tutorial is being shown and we have touched the frame. In this way, the game will remain paused. If the condition doesn't match and we are sure that the tutorial has ended, we move the rabbit where the player has touched. Thus, we handle the game when the tutorial is finished.

Let's implement this new `updateTutorial` method. Add the following block of code in `updateTutorial()` method:

```swift
func updateTutorial() {
    // Auxiliar variables
    var moveAction: SKAction!
    var duration: CGFloat = 0.0
    var nextPosition: CGPoint
    switch tutorialStep {
        default: break
    }
}
```

We are just adding the skeleton of the method in which we create some auxiliary variables that will help us update the tutorial steps. We also added a `switch` statement, where we will add the different tutorial steps to manage what will happen in each step.

Let's start with the first one. Add the following block of code just above `default: break`:

```swift
case .TUTORIAL_STEP_1:
    // Hide tutorial elements
    self.tutorialImage.hidden = true
    self.labelTutorial.hidden = true
    self.tutorialFrame.hidden = true
    // Setting the next position
    nextPosition = CGPoint(x: leftDoor.position.x, y: rabbit.
position.y)
    // We want the rabbit to move on a constant speed
    duration = self.distanceBetween(point: self.rabbit.position,
andPoint: nextPosition) / 360.0
    // Move the rabbit to the touched position
    moveAction = SKAction.moveToX(nextPosition.x, duration:
Double(duration))
```

```swift
}```
let updateTutorialAction = SKAction.runBlock {
    // Update tutorial step
    self.tutorialStep = .TUTORIAL_STEP_2
    self.updateTutorial()
}

// Create sequence
let sequence = SKAction.sequence([moveAction, updateTutorialAction])

// Run the sequence
self.rabbit.runAction(sequence)

// Release the game for the tutorial
self.view?.paused = false
break

This case will be reached when we run the first state and the player touches on the tutorial frame. At this point, we will hide all the visual elements and move the rabbit to the center of the left door.

For this purpose, we specified the next position and the duration to reach the next position, and we created a movement action with this information.

Then, we created a runBlock action where we updated the tutorial step and called the updateTutorial method again.

Finally, we created a sequence with both the actions and we released the game so that this results in a situation where, once the player touches on the left frame, the tutorial element will disappear, the game will continue, and it will execute the code in the second step after a delay.

Let's implement this second step by adding the following code one step before default: break:

case .TUTORIAL_STEP_2:
    // Create action
    let pauseForTutorial = SKAction.runBlock {
        // Pause game for the tutorial
        self.view?.paused = true
        // Update tutorial image
        self.tutorialImage.position = CGPoint(x: 5*self.view!.bounds.size.width/6, y: self.view!.bounds.size.height/3)
        self.tutorialImage.hidden = false

        // Update tutorial frame
        self.tutorialFrame.position = CGPoint(x: 5*self.view!.bounds.size.width/6, y: self.view!.bounds.size.height/2)
        self.tutorialFrame.hidden = false
    }


// Update tutorial label
self.labelTutorial.position.x = self.tutorialImage.position.x
self.labelTutorial.hidden = false
// Update tutorial step
self.tutorialStep = .TUTORIAL_STEP_3
}
// Creating a delay action
let delayAction = SKAction.waitForDuration(4.25)
// Create sequence
let sequence = SKAction.sequence([delayAction, pauseForTutorial])
// Run the sequence
self.runAction(sequence)
break

In this step, we executed a block where we first pause the game and then show the same tutorial information as that in the first step, but on the right-hand side this time. We also updated the tutorialStep variable so that the tutorial progresses.

We want the preceding code to be triggered after a delay, which is why we created a delay action and run it in a sequence.

If you run the game now, you will see what's shown in the following screenshot:
Now, we want the game to react when we touch on the right-hand side of the screen while the tutorial is being shown. So, let's implement the third state with the following lines:

```swift
case .TUTORIAL_STEP_3:
    // Hide tutorial elements
    self.tutorialImage.hidden = true
    self.labelTutorial.hidden = true
    self.tutorialFrame.hidden = true
    // Release the game for the tutorial
    self.view?.paused = false
    // Setting the next position
    nextPosition = CGPoint(x: rightDoor.position.x, y: rabbit.
    position.y)
    // We want the rabbit to move on a constant speed
    duration = self.distanceBetween(point: self.rabbit.position,
    andPoint: nextPosition) / 360.0
    // Move the rabbit to the touched position
    moveAction = SKAction.moveToX(nextPosition.x, duration:
    Double(duration))
    // Create action
    let updateTutorialAction = SKAction.runBlock {
        // Update tutorial step
        self.tutorialStep = .TUTORIAL_STEP_4
        self.updateTutorial()
    }
    // Create sequence
    let sequence = SKAction.sequence([moveAction,
updateTutorialAction])
    // Run the sequence
    self.rabbit.runAction(sequence)
    break
```

This step is similar to the first one; we hide the tutorial elements and then release the game while we move the rabbit to the right of the screen, centered at the door, thanks to a movement action.

We created a `runBlock` action to update the tutorial step and call the `updateTutorial` method again.
Finally, we created a sequence with both the actions and ran it. At this point, we need to implement the fourth state of the tutorial to continue. So, let's add the following block of code:

```swift
case .TUTORIAL_STEP_4:
    // Create action
    let pauseForTutorial = SKAction.runBlock {
        // Set image position
        self.tutorialImage.position = CGPoint(x:self.view!.bounds.size.width/2, y: self.view!.bounds.size.height/3)
        // Update tutorial label
        self.labelTutorial.text = "RUN!"
        self.labelTutorial.position.x = self.tutorialImage.position.x
        self.labelTutorial.hidden = false
        // Update tutorial step
        self.tutorialStep = .TUTORIAL_STEP_5
        self.updateTutorial()
    }
    // Creating a delay action
    let delayAction = SKAction.waitForDuration(2.25)
    // Create sequence
    let sequence = SKAction.sequence([delayAction, pauseForTutorial])
    // Run the sequence
    self.runAction(sequence)
    break
```

When we reach this state, we will pause the game again after a delay. When the game is paused again, we move the text label to the center of the screen, change its text, and then show it. We also update the tutorial step to the next one.
In the preceding code, we also created a `delay` action and ran a `sequence` with both the actions. So, if we run the game now, we will see something that's similar to what's shown in the following screenshot:

Now, let's see what will happen in the last step of the tutorial. Add the following lines just before `default: break`:

```swift
    case .TUTORIAL_STEP_5:
        // Create action
        let endOfMovementAction = SKAction.runBlock {
            // Remove tutorial elements
            self.tutorialImage.removeFromParent()
            self.labelTutorial.removeFromParent()
            self.tutorialFrame.removeFromParent()
            // Update tutorial step
            self.tutorialStep = .TUTORIAL_ENDED
        }
```
// Creating a delay action
let delayAction = SKAction.waitForDuration(1.25)
// Create sequence
let sequence = SKAction.sequence([delayAction, endOfMovementAction])
// Run the sequence
self.runAction(sequence)
break

When we arrive at this point, we create an action where we will remove the visual elements from the view and update the tutorial step to the last one so that the game code understands that the game can continue without interruptions.

We want to execute it after a delay. Therefore, the last text is shown for a few seconds. This is the reason why we build a sequence with the aforementioned runBlock object and a delay action.

Run the game again and play it. You will see how we can play as much as we want by restarting the game, but we will see the tutorial only once:
If you develop a tutorial for another game, it will be completely different from this one, but I recommend that you break down all the concepts that you want to learn in several states so that you can ensure that the player has complete knowledge to build and start playing the game.

### Loading and saving data

When playing games, users will usually need a way to store how far they have arrived in the game or some other information, such as the best score or the main character's name. Due to this, the game will be able to load this stored data or some other data, such as the position of the enemies in each level of the game.

In this section, we are going to learn how to store the maximum score that a player has reached so far. For this purpose, we are going to take advantage of the `NSUserDefaults` class.

#### The `NSUserDefaults` class

This class provides an interface to read and write information on the default system. This default system is the place where the preferences that are chosen on your game or app, such as the user's language, sound (enabled or disabled), and so on, will be stored.

A user default values can be of different types, such as Boolean, float, integer, double, string, data, array, and so on, and the `NSUserDefaults` class provides methods for all of them.

We are going to use `NSUserDefaults` class to store the best score achieved by the player. So for this purpose, we will need some new variables. Add them to the top of `GameScene`, as follows:

```swift
private var labelBestScore: SKLabelNode!
private var bestScore: Int = 0
private var userDefaults: NSUserDefaults!
private var kUserDefaultsBestScore = "user_default_best_score"
```

We declared a new label that will show the best score, which will be kept by a new integer variable. We also declared an instance of `NSUserDefaults` and a constant that will help us identify the user's default name.

The next step is to initialize the user defaults. Add the following method call at the beginning of `didMoveToView`:

```swift
self.initializeUserDefaults()
```
Initialize this method using the following block of code:

```swift
func initializeUserDefaults() {
    // Initialize user defaults
    if (userDefaults == nil) {
        userDefaults = NSUserDefaults.standardUserDefaults()
    }
    // If the user default exists
    if userDefaults.integerForKey(kUserDefaultBestScore) > 0 {
        bestScore = userDefaults.integerForKey(kUserDefaultBestScore)
    }
}
```

We first initialized the user's default variables, if it is null, and then we got the value of the stored best score, if it has been previously updated. We performed this last action so that every time we execute the game it will show the correct value.

- We used the `integerForKey` method as we expect to store integer values in this user's default variable.

If the user's default exists, we update the `bestScore` value so that it can be used later.

The next step is to create a new label. Let's add some code to the `initializeLabels` variable and add the following lines of code at the end of the aforementioned method:

```swift
// Initialize the label as a copy
labelBestScore = labelScore.copy() as! SKLabelNode
// Set color, size and position
labelBestScore.fontColor = UIColor.orangeColor()
labelBestScore.position.y = labelScore.position.y - 30
// Set text
labelBestScore.text = "Best: \(bestScore)"
// Add the label to the scene
addChild(labelBestScore)
```

We initialized this label as a copy of the previous one so that we can reuse some of its properties. We just want it to have a different color. Therefore, we chose the orange one and placed it below the score label.

Then we set its text, which is composed by a string, and the value of the best score variable, and finally we added it to the scene.
If you run the game now, you will see this new label at the top right of the screen, as shown in the following screenshot:

Now that we have the label in the view that will get the information stored in the user's default values, we just need to update its user's default values where applicable.

The perfect moment to update it is when the game is over. Let's add the following line to `gameOver` just after `addChild(labelGameOver)`:

```swift
// Update best score
self.updateBestScore()
```

This will call a new method that we need to implement using the following lines:

```swift
func updateBestScore() {
    if score > bestScore {
        userDefaults.setInteger(score, forKey: kUserDefaultsBestScore)
        labelBestScore.text = "Best: \(score)"
    }
}
```
In this method, we first checked whether the new score that was accomplished is greater than the best one, and if this is the case, we set this new value to the user’s default value that is specified by the constant that we declared previously. Then, we updated the label too.

You have to specify the same user's default value that's used when you are storing as the one that's used when we are loading the data.

Run the game now and see what happens when the game finishes:

If you kill the game process and start playing again, you will see how the best score value is updated correctly. However, everything will be deleted if you remove the game from your device.
2-star challenge: completing the tutorial

You may have realized that the tutorial reappears every time we rerun the game even if it has been completed previously. Now that we know how to store information on the device, let's take advantage of it in order to store information about whether the tutorial has been completed.

Solution

We will need a couple of new variables to support this new behavior. Therefore, add the following lines at the top of GameScene class:

```swift
private var isTutorialCompleted: Bool = false
private var kUserDefaultTutorialCompleted = "user_default_tutorial_completed"
```

We declared a Boolean variable that will act as a flag that represents the user's default value. We also declared a constant that will be used to identify the value of the desired user default.

This new flag will be initialized in initializeUserDefaults. So add the following block of code at the end of the aforementioned method:

```swift
if userDefaults.boolForKey(kUserDefaultTutorialCompleted) {
    isTutorialCompleted = userDefaults.boolForKey(kUserDefaultTutorialCompleted)
}
```

The preceding code will get the value stored in the user's default in case value already exists. If this is not the case, it will keep the value that was set by user's default when initialized (false).

Now, we need to update the user's default value, and this will happen at the end of the fifth step of the tutorial. So, go to case .TUTORIAL_STEP_5 of updateTutorial and add the following lines just after self.tutorialStep = .TUTORIAL_ENDED:

```swift
// Update tutorial flag
self.isTutorialCompleted = true
self.userDefaults.setBool(self.isTutorialCompleted, forKey: self.kUserDefaultTutorialCompleted)
```
When the tutorial reaches the last step, we update the flag and the user’s default.

Finally, we just need to take advantage of this new flag to know when to show the tutorial. Let's make a couple of changes. In `didMoveToView`, replace the following line:

```swift
if tutorialStep != .TUTORIAL_ENDED {
```

Replace the preceding line of code with the following:

```swift
if !isTutorialCompleted && tutorialStep != .TUTORIAL_ENDED {
```

In this way, when trying to initialize the tutorial, we will also take into account the new flag.

The last change that we need to perform is in `touchesBegan`, where we check whether the tutorial has ended. Hence, replace the following lines:

```swift
if self.tutorialStep != .TUTORIAL_ENDED && self.nodeAtPoint(location).name == "tutorial_frame" {
    self.updateTutorial()
} else if self.tutorialStep == .TUTORIAL_ENDED {
    self.moveRabbitToNextLocation(location)
}
```

Replace the preceding block of code with the following lines:

```swift
if !isTutorialCompleted && self.tutorialStep != .TUTORIAL_ENDED && self.nodeAtPoint(location).name == "tutorial_frame" {
    self.updateTutorial()
} else if isTutorialCompleted || self.tutorialStep == .TUTORIAL_ENDED {
    self.moveRabbitToNextLocation(location)
}
```
In both the cases, we just add the new flag as a condition and everything will work smoothly now. So, execute the project and check out this new behavior:

![Game Image]

**The property list files**

Apart from the loading of data from user's default, there is a more powerful way to get information from an external source. It's called the known property files, which are also known as the plist files.

These files are very common in iOS development. For example, they are used to store the configuration values of projects in Info.plist.

The property list files contain a list of keys that can contain different types of values, such as dictionaries, strings, numbers, dates, or Boolean values and we are going to take advantage of them to store the information of each door for all the waves that we want to load.

This approach is the one that you will use if you create a game with several levels, and each level has some specific information that you want to load once you initialize the new scene.
In our case, we are going to simulate that we are loading the information of level 1 of the game from a `plist` file. From this file, we will get the number of waves that the level has and the distribution of the correct and wrong doors in each wave.

First of all, let’s add the corresponding file to the project by performing the following steps:

1. Right-click on `Art` and select `Add Files to InsideTheHat…`.
2. You’ll find `Level_info.plist` in the `7338_04_Resources` folder that you previously unzipped. Select this file and click on `Add`.

You will see something that’s similar to what’s shown in the following screenshot:

Note that you can create your own `plist` file using the `iOS | Resource | Property List` option in the `New File` menu.
As you can see in the preceding screenshot, this file contains a key called `numWaves` that contains the number of waves of the first level (20) and a list of dictionaries that contains three strings for each wave. Each of these strings corresponds to one of the doors in a wave, and its value can be correct or wrong, depending on how we want the wave to be loaded.

Now that we know how information is distributed in a property list file, it's time to read it. So, let's declare some variables that we will need for this purpose:

```swift
private var maxWaves: Int = 0
private var waveNumber: Int = 1
private var leftDoorsInfo: [String]!
private var centerDoorsInfo: [String]!
private var rightDoorsInfo: [String]!
```

We declared a variable to store the maximum number of waves that were read from the file and a variable that will keep the count of waves loaded.

We also declared three arrays of strings that will be used to store information of each wave for each door.

Now, let's initialize the level information. Add the following method call at the beginning of `didMoveToView`:

```swift
self.readLevelInfo()
```

Implement it using the following block of code:

```swift
self.readLevelInfo()
//Implement it using the following block of code:
func readLevelInfo() {
    // Declare dictionary variable
    var levelDictionary: NSDictionary!
    var waveInfo: NSDictionary
    leftDoorsInfo = [String]()  
    centerDoorsInfo = [String]()  
    rightDoorsInfo = [String]()  
    // Get level dictionary root
    if let path = NSBundle.mainBundle().pathForResource("Level_info", ofType: "plist") {
        levelDictionary = NSDictionary(contentsOfFile: path)
    }
```
// Initialize max number of waves
maxWaves = levelDictionary!.valueForKey("numWaves") as! Int

// Get info for all the waves
for var i: Int = 1; i <= maxWaves; i++ {
    waveInfo = levelDictionary!.valueForKey("wave - \(i)") as! NSDictionary
    leftDoorsInfo.append(waveInfo.valueForKey("leftDoor") as! String)
    centerDoorsInfo.append(waveInfo.valueForKey("centerDoor") as! String)
    rightDoorsInfo.append(waveInfo.valueForKey("rightDoor") as! String)
}

We first initialized two NSDictionary variables. The first one will be used to store the whole dictionary, and the second one will get information of each wave's dictionary. Then, we initialized the arrays so that they can begin storing values.

To get the dictionary's root, we need to get the file's path using the pathForResource method, specifying the filename and its extension. Once we have the path, we can initialize the dictionary with the contents of the file.

Then, we get the numWaves value from the file thanks to the valueForKey method, where we specify the desired key and force its type to be an integer.

We created a for loop, where we will find the information of each wave by getting each waveInfo dictionary and distributing the left, center, and right door values to the corresponding array.

Now that we have loaded the information, it's time to use it to create a level. Let's make a couple of changes to setDoorAttributes. Replace the following line of code of each case:

```swift
if (arc4random_uniform(2) == 0) {
```

Replace the preceding line of code with the corresponding lines from the following code:

```swift
if leftDoorsInfo[waveNumber-1] == "wrong" {
if centerDoorsInfo[waveNumber-1] == "wrong" {
if rightDoorsInfo[waveNumber-1] == "wrong" {
```
In this way, we created the doors, depending on the already loaded information. Finally, we need to update the wave count. Add the following lines at the end of `initializeWave`:

```swift
// Increase wave
if waveNumber < maxWaves {
    waveNumber++
} else {
    waveNumber = 1
}
```

We increase the counter if we haven’t reached the maximum value. In such a situation, we reset it to 1. Thus, the game will create an endless loop, but you can change it for things such as a **Game Over** call.

Finally, we need to reset this value when the game is restarted. Add the following line at the beginning of `restartGame`:

```swift
waveNumber = 1
```

Run the game now, and you will see how the game now loads the doors as we specified in the property list file:
Summary
This chapter helped you learn how to add some essential elements that will transform the game into a complete product that's ready to be uploaded to the App Store.

We began by using the Game Over condition to finish the game. We also added a way to restart the game the number of times that we want. Then, we added a main menu scene that helped us learn how to create transitions between the scenes of the game. We used this new scene to see how a tutorial should be created as a states machine, and we created a tutorial for the game so that the players know the mechanics of the game as soon as they play the game for the first time. Finally, we learned how to load and save data of the game by using the user's default values or property list files.

In the next chapter, we will take advantage of the iOS devices to learn some available techniques.
Where to buy this book

You can buy Getting Started with SpriteKit 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.