

Swift Game Development

Third Edition

Learn iOS 12 game development using SpriteKit, SceneKit and ARKit 2.0



Packt

www.packt.com

Siddharth Shekar and Stephen Haney

Swift Game Development

Third Edition

Learn iOS 12 game development using SpriteKit,
SceneKit and ARKit 2.0

Siddharth Shekar

Stephen Haney



BIRMINGHAM - MUMBAI

Swift Game Development

Third Edition

Copyright © 2018 Packt Publishing

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

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

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

Commissioning Editor: Ashwin Nair

Acquisition Editor: Reshma Raman

Content Development Editors: Flavian Vaz

Technical Editor: Akhil Nair

Copy Editor: Safis Editing

Project Coordinator: Sheejal Shah

Proofreader: Safis Editing

Indexers: Tejal Daruwale Soni

Graphics: Jason Monteiro

Production Coordinator: Shantanu Zagade

First published: July 2015

Second published: February 2017

Third edition: September 2018

Production reference: 1260918

Published by Packt Publishing Ltd.

Livery Place

35 Livery Street

Birmingham B3 2PB, UK.

ISBN 978-1-78847-115-2

www.packtpub.com



mapt.io

Mapt is an online digital library that gives you full access to over 5,000 books and videos, as well as industry leading tools to help you plan your personal development and advance your career. For more information, please visit our website.

Why subscribe?

- Spend less time learning and more time coding with practical eBooks and Videos from over 4,000 industry professionals
- Learn better with Skill Plans built especially for you
- Get a free eBook or video every month
- Mapt is fully searchable
- Copy and paste, print, and bookmark content

Packt.com

Did you know that Packt offers eBook versions of every book published, with PDF and ePUB files available? You can upgrade to the eBook version at www.Packt.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at customercare@packt.com for more details.

At www.Packt.com, you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on Packt books and eBooks.

Contributors

About the authors

Siddharth Shekar is a game developer and teacher with over 5 years of industry experience and 12 years of experience in C++ and other programming languages, and is adept at graphics libraries such as OpenGL and Vulkan and game engines such as Unity and Unreal. He has also published games on the iOS and Android App Stores.

Siddharth has also authored 4 books, including *Mastering Android Game Development with Unity* and *Learning iOS 8 Game Development Using Swift*, all published by Packt Publishing

He currently lives in New Zealand and is a lecturer in the Games Department at Media Design School. He teaches graphics programming and PlayStation 4 native game development, and mentors final year production students.

I would like to thank my parents for supporting me in everything that I choose to do. I would also like to thank Media Design School for encouraging me to continue working on this book. Finally, I would like to thank Packt Publishing for putting this book together and offering me the opportunity to write the book.

Stephen Haney has written two books on iOS game development. He began his programming journey at the age of 8 years on a dusty, ancient laptop using BASIC. He has been fascinated with building software and games ever since. Now well versed in multiple languages, he enjoys programming as a creative outlet the most. He believes that indie game development is an art form – an amazing combination of visual, auditory, and psychological challenges – rewarding to both the player and the creator.

He enjoyed writing this book and sincerely hopes that it directly furthers your career or hobby.

About the reviewer

Rafał Sroka is a Software Engineer with extensive expertise in iOS development, building-up mobile apps, QA and release process, API design, and complex software projects. He has worked on various high-value projects in Poland, Germany, and Switzerland. He prefers working in cross-functional start-up environments where fast delivery of well-written and tested code is crucial. He holds a Master's degree in Electronics and Telecommunications. In his free time, he's a dedicated rock climber and mountaineer. He is a doer.

Packt is Searching for Authors Like You

If you're interested in becoming an author for Packt, please visit authors.packtpub.com and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Table of Contents

Preface	ix
Chapter 1: Designing Games with Swift	1
Why you will love Swift	2
Prerequisites	3
What you will learn in this book	3
Embracing SpriteKit	4
Reacting to player input	4
Structuring your game code	5
Building UI/menus/levels	5
Integrating with Game Center	5
Maximizing fun	5
Crossing the finish line	5
Monetizing your work	6
New in Swift 4.2	6
Setting up your development environment	7
Introducing and installing Xcode	7
Creating our first Swift game	8
Navigating our project	13
Exploring the SpriteKit demo	13
Examining the demo code	16
Cleaning up	16
Summary	17
Chapter 2: Sprites, Camera, Action!	19
Preparing your project	20
Drawing your first sprite	23
Building a SKSpriteNode class	23

Table of Contents

Animation—movement, scaling, and rotation	25
Sequencing multiple animations	26
Recapping your first sprite	26
The story on positioning	27
Alignment with anchor points	27
Working with textures	29
Downloading the free assets	29
More exceptional art assets	29
Drawing your first textured sprite	29
Adding the bee image to your project	30
Loading images with SKSpriteNode	30
Designing for Retina	31
The ideal asset approach	32
Hands-on with retina in SpriteKit	32
Organizing art into texture atlases	33
Exploring Assets.xcassets	33
Collecting art into texture atlases	34
Updating our bee node to use the texture atlas	35
Iterating through texture atlas frames	35
Putting it all together	37
Centering the camera on a sprite	37
Summary	41
Chapter 3: Mix in the Physics	43
Laying the foundations	43
Adopting a protocol for consistency	44
Organizing game objects into classes	44
The icy tundra	46
Adding the player's character	50
Renovating the GameScene class	54
Physics bodies and gravity	55
Dropping like flies	55
Solidifying the ground	56
Exploring physics simulation mechanics	57
Bumping bees into bees	60
Movement with impulses and forces	61
Summary	62
Chapter 4: Adding Controls	63
Retrofitting the Player class for flight	64
The Beekeeper	64
Updating the Player class	64

Table of Contents

Moving the ground	64
Assigning a physics body to the player	65
Creating a physics body shape from a texture	65
Polling for device movement with Core Motion	66
Implementing the Core Motion code	66
Wiring up the sprite onTap events	68
Implementing touchesBegan in the GameScene	68
Larger than life	69
Teaching our penguin to fly	70
Listening for touches in GameScene	71
Fine-tuning gravity	72
Spreading your wings	72
Improving the camera	73
Pushing Pierre forward	76
Tracking the player's progress	77
Looping the ground as player the moves forward	77
Summary	79
Chapter 5: Spawning Enemies, Coins, and Power-Ups	81
Introducing the cast	82
Locating and adding the art assets	82
Adding the Power-up Star	84
Adding the Star class	84
A new enemy - the Mad Fly	86
Adding the MadFly class	86
Another terror - Bats!	87
Adding the Bat class	87
Guarding the ground with the Blade	88
Adding the Blade class	89
Adding coins	90
Creating the coin classes	90
Organizing the project navigator	91
Testing the new game objects	92
Preparing for endless flight	94
Summary	95
Chapter 6: Generating a Never-Ending World	97
Designing levels with the SpriteKit scene editor	98
Separating level data from game logic	99
Using custom classes in the scene editor	99
Building encounters for Pierre Penguin	99
Creating our first encounter	100

Table of Contents

Integrating scenes into the game	104
Looping encounters for a never-ending world	107
Building more encounters	107
Updating the EncounterManager class	108
Storing metadata in SKSpriteNodeuserData property	109
Wiring up EncounterManager in the GameScene class	111
Adding the Power-up Star at random places	112
Turning bronze coins into gold coins	114
Summary	115
Chapter 7: Implementing Collision Events	117
Learning the SpriteKit collision vocabulary	117
Collision versus contact	118
Physics category masks	118
Using category masks in Swift	119
Adding contact events to our game	120
Setting up the physics categories	120
Assigning categories to game objects	120
The player	121
The ground	121
The Power-up Star	121
Enemies	122
Coins	122
Preparing GameScene for contact events	122
Viewing console output	124
Testing our contact code	124
Player health and damage	125
Animations for damage and game over	127
The damage animation	127
The game over animation	129
Collecting coins	131
The Power-up Star logic	132
Summary	134
Chapter 8: Polishing to a Shine – HUD, Parallax	135
Backgrounds, Particles, and More	135
Adding a HUD	136
Implementing the HUD	137
Parallax background layers	141
Adding the background assets	142
Implementing a background class	142
Wiring up backgrounds in the GameScene class	144

Table of Contents

Using the particle system	146
Adding the circle particle asset	147
Creating a SpriteKit particle file	148
Configuring the path particle settings	150
Name	151
Background	151
Texture	151
Emitter	151
Lifetime	151
Position Range	152
Angle	152
Speed	152
Acceleration	153
Alpha	153
Scale	153
Rotation	153
Color Blend	153
Color Ramp	154
Adding the particle emitter to the game	154
Granting safety as the game starts	155
Summary	156
Chapter 9: Adding Menus and Sounds	157
Building the main menu	157
Creating the menu scene and menu nodes	158
Launching the main menu when the game starts	160
Wiring up the START GAME button	161
Adding the restart game menu	162
Extending the HUD	162
Wiring up GameScene for game over	163
Informing the GameScene class when the player dies	163
Implementing touch events for the restart menu	164
Adding music with AVAudio	165
Adding sound assets to the game	166
Playing background music	166
Playing sound effects	167
Adding the coin sound effect to the Coin class	167
Adding the power-up and hurt sound effects to the Player class	168
Playing sound effects with SKAction	168
Adding a mute button and volume slider	169
Adding Options to the Menu Scene	172
Summary	178

Table of Contents

Chapter 10: Standing out in the Crowd with Advanced Features	179
Adding crates to smash open	180
Creating the Crate particle effects	182
Recycling emitter nodes with particle pools	186
Wiring up crate contact events	189
Creating the health power-up crate	191
Spawning smashable crates that reward coins	193
Summary	196
Chapter 11: Introduction to SceneKit	197
Creating a Scene with SCNScene	198
Adding objects to the scene	201
Adding a sphere	201
Adding light sources	202
Adding a camera to the scene	204
Adding a floor	205
Importing scenes from an external 3D application	206
Creating the hero class and physics	208
Adding an enemy and collision detection	212
Adding a SpriteKit overlay	216
Adding labels and buttons	217
Adding touch interactivity	218
Adding the Gameloop	219
Setting a GameOver condition	221
Checking for contact	221
Adding wall and floor parallax	224
Adding particles	232
Adding character animation	233
Summary	236
Chapter 12: Choosing a Monetization Strategy	237
Developing your marketing plan	238
When to start marketing	238
Marketing checklist	238
Leveraging crowdfunding	240
Pros and cons of crowdfunding	240
Showing display ads for revenue	241
The upsides to showing ads	241
The downsides to showing ads	242
Adding an AdMob Ad in the App	242
Selling in-app purchases	250
In-app purchase strategies	251

Table of Contents

A word about farming your players	251
Adding In-App Purchases	252
Localization in foreign markets	268
Managing scope and completing projects	269
Summary	270
Chapter 13: Integrating with Game Center	271
Authenticating the player's Game Center account	271
Opening Game Center in our game	276
Updating the leaderboard from the code	281
Adding an achievement	283
Creating a new achievement in iTunes Connect	283
Updating achievements from the code	285
Summary	287
Chapter 14: Introduction to Spritekit with ARKit	289
Requirements for the project	289
Creating an AR Spritekit project	290
Adding text and crosshair	297
Adding anchors at random locations	299
Adding custom sprite	302
Registering touch controls to remove game objects	307
Summary	308
Chapter 15: Introduction to Scenekit with ARKit	309
Going through the basic Scenekit/ARKit project	311
Project setup and detecting a plane	316
Adding touches	323
Adding Game Objects	324
Stop detecting planes	324
Adding light source	324
Adding ground node	326
Adding Hero and Enemy	327
Adding Score and Gameover text	339
Finishing touches	343
Summary	345
Chapter 16: Publishing the Game on the App Store	347
Creating the Bundle ID for the app	348
Preparing the project	350
Create the App in the itunesconnect portal	352
Upload the App and submit for review	358
Summary	363

Table of Contents

Chapter 17: Multipeer Augmented Reality	365
Multipeer connectivity framework overview	366
Creating the multipeer session class	367
Creating a UI for the app	372
Setting outlets and adding variables	381
Initializing the view	383
Update session and tracking	384
Hosting and joining the session	390
Sending and receiving data	395
Initializing a multipeer session	401
Testing the application	401
Summary	402
Another Book You May Enjoy	403
Index	407

Preface

Swift is the perfect choice for game development. Developers are intrigued by Swift and want to make use of new features to develop their best games yet. Packed with best practices and easy-to-use examples, this book leads you step-by-step through the development of your first Swift game.

The book starts by introducing Swift's newest and best features for game development. Using SpriteKit, you will learn how to animate sprites and textures. Along the way, you will master the physics, animations, collision effects and required to build the UI aspects of the game.

You will then work on creating a 3D game using the SceneKit framework. You will see how to add monetization and integrate Game Center. Then you will dive into creating **augmented reality (AR)** games using SpriteKit and SceneKit.

Finally, you will see how to create a Multipeer AR project to connect two devices and send and receive data back and forth between the devices in real time.

By the end of this book, you will be able to create your own iOS games using Swift and publish them on the iOS App Store.

Who this book is for

The book is targeted at new and intermediate developers who want to update their knowledge about the changes in the new version of Swift and want to learn about the new ARKit framework for making AR games for iOS using Xcode 10.

What this book covers

Chapter 1, Designing Games with Swift, introduces you to the best features of Swift, outlines what is new in Swift 3, helps you set up your development environment, and launches your first SpriteKit project.

Chapter 2, Sprites, Camera, Action!, teaches you the basics of drawing and animating with Swift. You will draw sprites, import textures into your project, and center the camera on the main character.

Chapter 3, Mix in the Physics, covers the physics simulation fundamentals: physics bodies, impulses, forces, gravity, collisions, and more.

Chapter 4, Adding Controls, explores the various methods of mobile game controls: device tilt and touch input. We will also improve the camera and core gameplay of our example game.

Chapter 5, Spawning Enemies, Coins, and Power-ups, introduces the cast of characters we use in our example game and shows you how to create custom classes for each NPC type.

Chapter 6, Generating a Never-Ending World, explores the SpriteKit scene editor, builds encounters for the example game, and creates a system to loop encounters endlessly.

Chapter 7, Implementing Collision Events, delves into advanced physics simulation topics and adds custom events when sprites collide.

Chapter 8, Polishing to a Shine – HUD, Parallax Backgrounds, Particles, and More, adds the extra fun that makes every game shine. In this chapter, you will learn how to create parallax backgrounds, learn about SpriteKit's particle emitters, and add a heads-up display overlay to your games.

Chapter 9, Adding Menus and Sounds, builds a basic menu system and illustrates two methods of playing sounds in your games.

Chapter 10, Standing Out in the Crowd with Advanced Features, shows you how to combine the techniques you have learned to build advanced gameplay systems.

Chapter 11, Introduction to SceneKit, explains how to create a basic 3D game using the SceneKit framework.

Chapter 12, Choosing a Monetization Strategy, outlines the strategies available to indie developers who want to make money from their games.

Chapter 13, Integrating with Game Center, links our example game to the Apple Game Center for leaderboards, achievements, and friendly challenges.

Chapter 14, Introduction to SpriteKit in ArKit, gives an introduction to how to make an AR game using Arkit and SpriteKit.

Chapter 15, Introduction to SceneKit in ArKit, demonstrates how to make a 3D augmented reality game using SceneKit and Arkit.

Chapter 16, Ship It! Preparing for the App Store and Publication, covers the essentials of packaging your game and submitting it to the App Store.

Chapter 17, Multipeer Augmented Reality, shows you how to create a multiplayer AR project to connect two devices in real time so they can send and receive data.

To get the most out of this book

This book uses the Xcode IDE version 10 (Swift 4.2). If you use a different version of Xcode, you will likely encounter syntax differences; Apple is constantly upgrading Swift's syntax. You can use Xcode's **Edit | Convert | To Current Swift Syntax** to update the code examples in this book to a newer version of Xcode.

Visit <https://developer.apple.com/xcode/> to download Xcode.

You will need an Apple developer account to integrate your apps with Game Center and submit your games to the App Store.

To run ARKit games you will need an iPhone SE or higher, a fifth-generation iPad, or an iPad Pro. To run the Multipeer AR project you will need two devices because it can't be tested on an emulator.

Download the example code files

You can download the example code files for this book from your account at <http://www.packt.com>. If you purchased this book elsewhere, you can visit <http://www.packt.com/support> and register to have the files emailed directly to you.

You can download the code files by following these steps:

1. Log in or register at <http://www.packt.com>.
2. Select the **SUPPORT** tab.
3. Click on **Code Downloads & Errata**.
4. Enter the name of the book in the **Search** box and follow the on-screen instructions.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR / 7-Zip for Windows
- Zipeg / iZip / UnRarX for Mac
- 7-Zip / PeaZip for Linux

The code bundle for the book is also hosted on GitHub at <https://github.com/PacktPublishing/Swift-Game-Development-Third-Edition>. In case there's an update to the code, it will be updated on the existing GitHub repository.

We also have other code bundles from our rich catalog of books and videos available at <https://github.com/PacktPublishing/>. Check them out!

Conventions used

There are a number of text conventions used throughout this book.

CodeInText: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. For example; "Locate the `Enemies` folder in the downloadable asset bundle."

A block of code is set as follows:

```
import SpriteKit

class Coin: SKSpriteNode, GameSprite {
    var initialSize = CGSize(width: 26, height: 26)
    var textureAtlas: SKTextureAtlas =
        SKTextureAtlas(named: "Environment")
    var value = 1
```

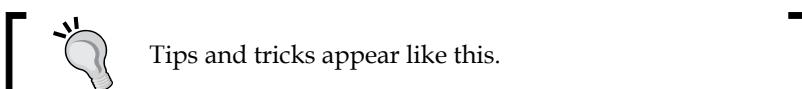
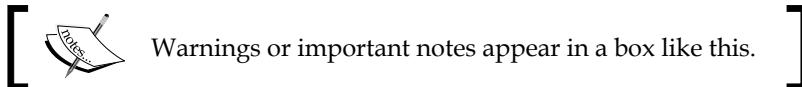
When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
import SpriteKit
import ARKit

class Scene: SKScene {

    var crosshair: SKSpriteNode!
    let scoreText = SKLabelNode(text: "00")
```

Bold: Indicates a new term, an important word, or words that you see on the screen, for example, in menus or dialog boxes, also appear in the text like this. For example: "Select **System info** from the **Administration** panel."



Get in touch

Feedback from our readers is always welcome.

General feedback: If you have questions about any aspect of this book, mention the book title in the subject of your message and email us at customercare@packt.com.

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book we would be grateful if you would report this to us. Please visit, <http://www.packt.com/submit-errata>, selecting your book, clicking on the Errata Submission Form link, and entering the details.

Piracy: If you come across any illegal copies of our works in any form on the Internet, we would be grateful if you would provide us with the location address or website name. Please contact us at copyright@packt.com with a link to the material.

If you are interested in becoming an author: If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, please visit <http://authors.packtpub.com>.

Reviews

Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential readers can then see and use your unbiased opinion to make purchase decisions, we at Packt can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about Packt, please visit packt.com.

1

Designing Games with Swift

Apple's newest version of its flagship programming language, **Swift 4.2**, is the perfect choice for game developers. As it matures, Swift is realizing its potential to be something special, a revolutionary tool for app creators. Swift is the gateway for developers to create the next big game in the Apple ecosystem. We have only started to explore the wonderful potential of mobile gaming, and Swift is the modernization we need for our toolset. Swift is fast, safe, current, and attractive to developers coming from other languages. Whether you are new to the Apple world, or a seasoned veteran of **Objective-C**, I think you will enjoy making games with Swift.

Apple's website states the following:

"Swift is a successor to the C and Objective-C languages."

My goal in this book is to guide you step by step through the creation of a 2D and a 3D game for iPhones and iPads. We will start with installing the necessary software, working through each layer of game development, ultimately publishing our new game to the **App Store**.

We will also have some fun along the way! For the 2D game, we aim to create an endless flyer game featuring a magnificent flying penguin named *Pierre*. What is an endless flyer? Picture hit games such as *iCopter*, *Flappy Bird*, *Whale Trail*, *Jetpack Joyride*, and many more – the list is quite long.

Endless flyer games are popular on the App Store, and the genre necessitates that we cover many reusable components of 2D game design. I will show you how to modify our mechanics to create many different game styles.

For the 3D game, we will be using SceneKit. Like SpriteKit, which is used to develop 2D games, SceneKit is a framework developed by Apple to make 3D games. The 3D game we will be an obstacle avoidance game, in which there will be obstacles players will have to either go under or jump over.

In later chapters, we will see how to make augmented reality games using Apple's newest framework, ARKit. With ARKit, you can make really stunning augmented reality games. We will see how to develop AR games using SpriteKit and SceneKit. Along with developing a 2D AR game, we will also see how to bring the SceneKit game we developed for 3D to life using the ARKit. In addition, we will also see how to create a Multipeer Augmented Reality App using ARKit and Multipeer Connectivity framework.

My hope is that our demo project will serve as a template for your own creative tasks. Before you know it, you will be publishing your own game ideas using the techniques we explore together.

In this chapter, we will cover the following topics:

- Why you will love Swift
- What you will learn in this book
- New in Swift 4.2
- Setting up your development environment
- Creating your first Swift game
- Exploring the SpriteKit demo

Why you will love Swift

Swift, as a modern programming language, benefits from the collective experience of the programming community; it combines the best parts of other languages and avoids poor design decisions. Here are a few of my favorite Swift features:

- **Beautiful syntax:** Swift's syntax is modern and approachable, regardless of your existing programming experience. Apple have balanced syntax with structure to make Swift concise and readable.
- **Interoperability:** Swift can plug directly into your existing projects and run side by side with your Objective-C code.
- **Strong typing:** Swift is a strongly typed language. This means the compiler will catch more bugs at compile time, instead of when your users are playing your game! The compiler will expect your variables to be of a certain type (`int`, `string`, and so on) and will throw a compile-time error if you try to assign a value of a different type. While this may seem rigid if you are coming from a weakly typed language, the added structure results in safer, more reliable code.

- **Smart type inference:** To make things easier, **type inference** will automatically detect the types of your variables and constants based upon their initial value. You do not need to explicitly declare a type for your variables. Swift is smart enough to infer variable types in most expressions.
- **Automatic memory management:** As the Apple Swift developer guide states, "*memory management just works in Swift*". Swift uses a method called **Automatic Reference Counting (ARC)** to manage your game's memory usage. Besides a few edge cases, you can rely on Swift to safely clean up and turn off the lights.
- **An even playing field:** One of my favorite things about Swift is how quickly the language is gaining mainstream adoption. We are all learning and growing together, and there is a tremendous opportunity to break new ground.
- **Open source:** From version 2.2 onwards, Apple made Swift open source, curating it through the website www.swift.org, and launched a package manager with Swift 3. This is a welcome change, as it fosters greater community involvement and a larger ecosystem of third-party tools and add-ons. Eventually, we should see Swift migrate to new platforms.

Prerequisites

I will try to make this text easy to understand for all skill levels:

- I will assume you are brand new to Swift as a language
- This book requires no prior game development experience, though any experience you have will help
- I will assume you have a fundamental understanding of common programming concepts

What you will learn in this book

By the end of this book, you will be capable of creating and publishing your own iOS games. You will know how to combine the techniques we will learn to create your own style of game and you will be well prepared to dive into more advanced topics with a solid foundation in 2D, 3D, and AR game design.

Embracing SpriteKit

SpriteKit is Apple's 2D game development framework and your main tool for iOS, macOS, tvOS, and watchOS game design. SpriteKit will handle the mechanics of our graphics rendering, physics, and sound playback. As far as game development frameworks go, SpriteKit is a terrific choice. It is built and supported by Apple and thus integrates perfectly with Xcode and iOS. You will learn to be highly proficient with SpriteKit as we will be using it exclusively in our demo game.

We will learn how to use SpriteKit to power the mechanics of our game in the following ways:

- Animating our player, enemies, and power-ups
- Painting and moving side-scrolling environments
- Playing sounds and music
- Applying physics such as gravity and impulses for movement
- Handling collisions between game objects

Similar to SpriteKit, we will also look at SceneKit. In SceneKit, instead of importing images, we will be loading 3D models, placing them in the scene and creating a game around them. We will also see how to add player movement, animation, physics, controls, collision, and scoring.

In ARKit, with 2D and 3D game demos, we will see how to take existing games and make augmented reality games out of them.

Reacting to player input

The control schemes in mobile games must be inventive. Mobile hardware forces us to simulate traditional controller inputs, such as directional pads and multiple buttons, on the screen. This takes up valuable visible area and provides less precision and feedback than with physical devices. Many games operate with only a single input method: a single tap anywhere on the screen. We will learn how to make the best of mobile input and explore new forms of control by sensing device motion and tilt.

Structuring your game code

It is important to write well-structured code that is easy to reuse and modify as your game design inevitably changes. You will often find mechanical improvements as you develop and test your games, and you will thank yourself for a clean working environment. Though there are many ways to approach this topic, we will explore some best practices to build an organized system with classes, protocols, inheritance, and composition.

Building UI/menus/levels

We will learn how to switch between scenes in our game with a menu screen. We will cover the basics of user experience design and menu layout as we build our demo game.

Integrating with Game Center

Game Center is Apple's built-in social gaming network. Your game can tie into Game Center to store and share high scores and achievements. We will learn how to register for Game Center, tie it into our code, and create a fun achievement system.

Maximizing fun

If you are like me, you will have dozens of ideas for games floating around your head. Ideas come easily, but designing fun gameplay is difficult! It is common to find that your ideas need gameplay enhancements once you see your design in action. We will look at how to avoid dead ends and see your project through to the finish line. Plus, I will share my tips and tricks to ensure your game will bring joy to your players.

Crossing the finish line

Creating a game is an experience you will treasure. Sharing your hard work will only sweeten the satisfaction. Once our game is polished and ready for public consumption, we will navigate the App Store submission process together. You will end up feeling confident in your ability to create games with Swift and bring them to the market in the App Store.

Monetizing your work

Game development is a fun and rewarding process, even without compensation, but the potential exists to start a career, or side job, selling games on the App Store. Successfully promoting and marketing your game is an important task. I will outline your options and start you down the path to monetization.

New in Swift 4.2

This year, we have the release of Swift version 4.2 with Xcode 10. Unlike previous years where we saw a full release of a Swift version, like Swift 3.0 and Swift 4.0, this year, we see the release of Version 4.2. Swift 5.0 has been moved to the sometime early next year in 2019.

Xcode 10 will be the last version to support Swift 3.0 Compatibility Mode. So, if your app uses Xcode version 3, the code needs to be migrated to at least 4.0 by next year.

Further refinements have also been introduced by Apple in Swift 4.2:

- **Speedup Debug Builds:** Code builds at least twice as fast compared to previous versions of Xcode. This will depend on the nature of the product and how much Swift code it uses. Compilation is optimized to reduce redundant work.
- **Runtime Optimization:** Swift 4.2 introduces a couple of under the hood runtime optimizations. For example, in 4.2, all intermediate retains and releases of objects are removed and the object only gets released once at the end, improving the runtime of the code. This also enables automatic code size optimization.
- **Small String:** String is now 16 bites as compared to 24 in the previous version. It also enables a small string memory optimization. If the string is within 15 bites, then the string is represented in the string type without requiring separate allocation to represent the string.
- **Reduction in Code Size:** There is a new optimization level which reduces the machine level code size generated from the compilation of the swift code. This optimization enables 10% to 30% reduction in code size with a 5% hit in runtime performance.

Swift has already made tremendous steps forward as a powerful, young language. Now, Apple is working on polishing Swift into a mature, production-ready tool. The overall developer experience improves with Swift 4.2.

With Swift 4.2, we are moving toward binary compatibility and with Swift 5.0 releasing in early 2019, this transition will be complete. With this, Apple will be able to ship the Swift runtime along with the OS itself and doesn't need to be included in the App bundle as it is currently. This will improve startup time and memory usage of the application itself. More information about binary compatibility/ abi-stability is available at swift.org/abi-stability.

[ For a detailed look at what's new in Swift, I highly recommend watching the WWDC 2018 video:
<https://developer.apple.com/videos/play/wwdc2018/401/>.]

Setting up your development environment

Learning a new development environment can be a roadblock. Luckily, Apple provides some excellent tools for iOS developers. We will start our journey by installing Xcode.

Introducing and installing Xcode

Xcode is Apple's integrated development environment (IDE). You will need Xcode to create your game projects, write and debug your code, and build your project for the App Store. Xcode also comes bundled with an iOS simulator to test your games on virtualized iPhones and iPads on your computer.

Apple praises Xcode as "*an incredibly productive environment for building amazing apps for Mac, iPhone, and iPad*".

To install Xcode, search for Xcode in the App Store, or visit <http://developer.apple.com>, select **Developer**, and then **Xcode**.

Swift is continually evolving and each new Xcode release brings changes to Swift. If you run into errors because Swift has changed, you can always use Xcode's built-in syntax update tool. Simply use Xcode's **Edit | Convert to Latest Syntax** option to update your code.

Xcode performs common IDE features to help you write better, faster code. If you have used IDEs in the past, then you are probably familiar with auto completion, live error highlighting, running and debugging a project, and using a project manager pane to create and organize your files. However, any new program can seem overwhelming at first. We will walk through some common interface functions over the next few pages. I have also found tutorial videos on YouTube to be particularly helpful if you are stuck. Most common search queries result in helpful videos.

Creating our first Swift game

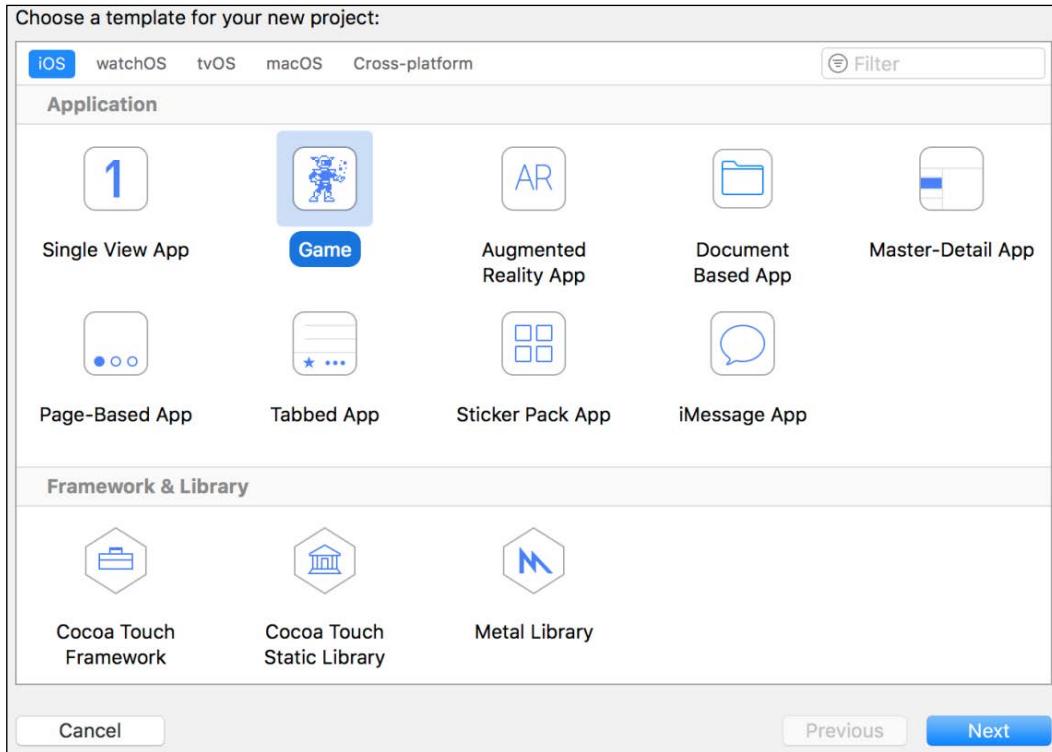
Do you have Xcode installed? Let's see some game code in action in the simulator!

We will start by creating a new project in Xcode. For our demo game, we will create a side-scrolling endless flyer featuring an astonishing flying penguin named Pierre. I am going to name this project Pierre Penguin Escapes the Antarctic, but feel free to name your project whatever you like. Follow these steps to create a new project in Xcode:

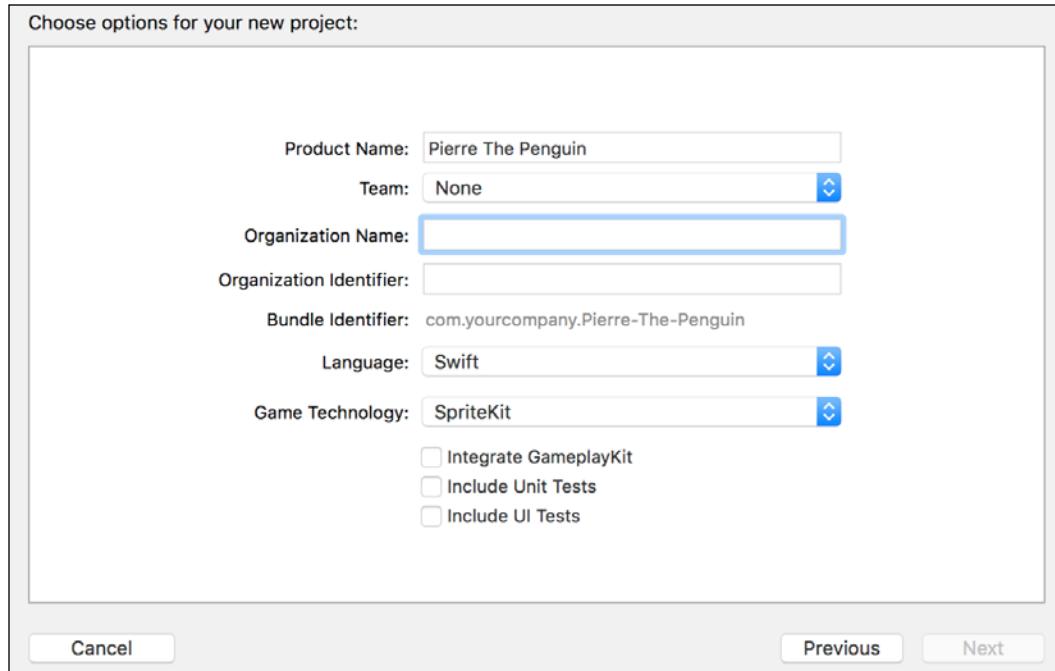
1. Launch Xcode and select **Create a new Xcode project**:



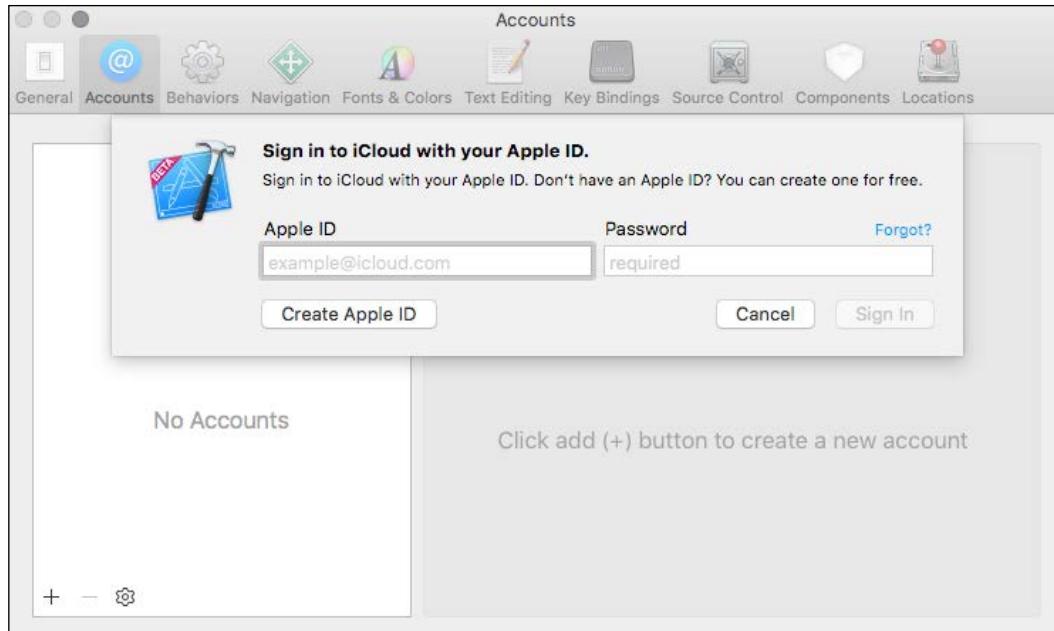
2. You will see a screen asking you to select a template for your new project. Select **iOS** and **Game** in the pane. It should look like this:



3. Once you have selected **Game**, click **Next**. The following screen asks us to enter some basic information about our project. Don't worry; we are almost at the fun bit. Fill in the **Product Name** field with the name of your game:

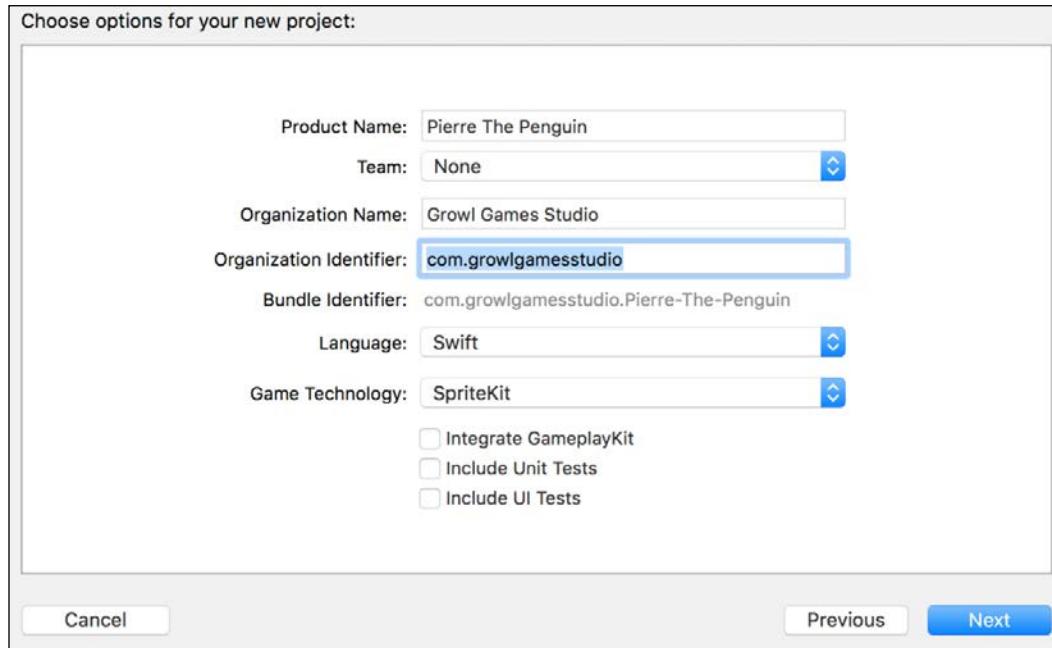


4. Let's fill in the **Team** field. Do you have an active Apple Developer account? If not, you can skip over the **Team** field for now. If you do, your **Team** is your Developer account. Click **Add Team** and Xcode will open the Accounts screen, where you can log in. Enter your Developer credentials, as shown in the following screenshot:



5. Once you're authenticated, you can close the **Accounts** screen. Your Developer account should appear in the **Team** dropdown.
6. You will want to pick a meaningful **Organization Name** and **Organization Identifier** when you create your own games for publication. Your **Organization Name** is the name of your game development studio. For me, that is Joyful Games. By convention, your **Organization Identifier** should follow a reverse domain name style. I will use com.growlgamesstudio since my website is growlgamesstudio.com.

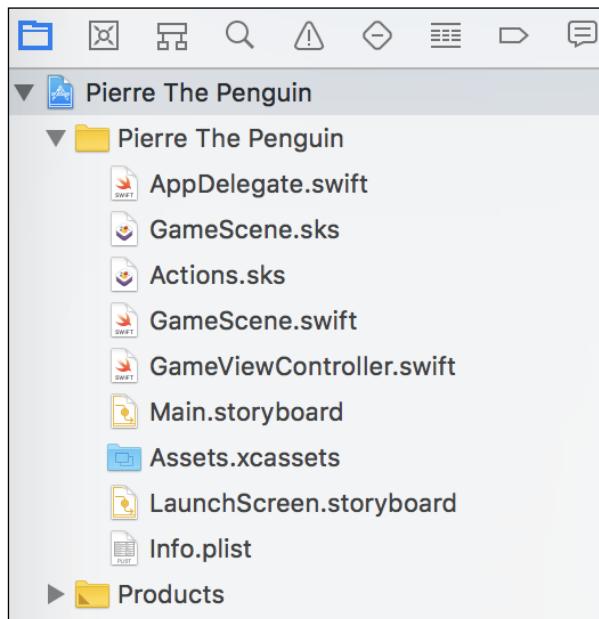
7. After you have filled out the name fields, be sure to select **Swift** for Language, **SpriteKit** for Game Technology, and **Universal** for Devices.
8. For now, uncheck **Integrate GameplayKit**, uncheck **Include Unit Tests**, and uncheck **Include UI Tests**. We will not use these features in our demo game. Here are my final project settings:



9. Click **Next** and you will see the final dialog box. Save your new project. Pick a location on your computer and click **Next**. And we are in! Xcode has pre-populated our project with a basic SpriteKit template.

Navigating our project

Now that we have created our project, you will see the project navigator on the left-hand side of Xcode. You will use the project navigator to add, remove, and rename files and generally organize your project. You might notice that Xcode has created quite a few files in our new project. We will take it slow; don't feel that you have to know what each file does yet, but feel free to explore them if you are curious:



Exploring the SpriteKit demo

Use the project navigator to open up the file named `GameScene.swift`. Xcode created `GameScene.swift` to store the default scene of our new game.

What is a scene? SpriteKit uses the concept of scenes to encapsulate each unique area of a game. Think of the scenes in a movie; we will create a scene for the main menu, a scene for the **Game Over** screen, a scene for each level in our game, and so on. If you are on the main menu of a game and you tap **Play**, you move from the menu scene to the **Level 1** scene.

SpriteKit prepends its class names with the letters "SK"; consequently, the scene class is `SKScene`.

You will see that there is already some code in this scene. The SpriteKit project template comes with a very short demo. Let's take a quick look at this demo code and use it to test the iOS simulator.

Please do not be concerned with understanding the demo code at this point. Your focus should be on learning about the development environment.

Look for the run toolbar at the top of the Xcode window. It should look something like the following:



Select the iOS device of your choice to simulate using the dropdown on the far right. Which iOS device should you simulate? You are free to use the device of your choice. I will be using an iPhone X for the screenshots in this book, so choose **iPhone X** if you want your results to match my images perfectly.

Unfortunately, expect your game to play poorly in the simulator. SpriteKit suffers from poor FPS (Frames Per Second) in the iOS simulator. Once our game becomes relatively complex, we will see our FPS drop, even on high-end computers. The simulator will get you through, but it is best if you can plug in a physical device for testing.

It is time for our first glimpse of SpriteKit in action! Press the gray play arrow on the toolbar (handy keyboard shortcut: *command + R*). Xcode will build the project and launch the simulator. The simulator starts in a new window, so make sure you bring it to the front. You should see a gray background with white text: **Hello, World**. Click around on the gray background. You will see colorful, spinning boxes spawning wherever you click:



If you have made it this far, congratulations! You have successfully installed and configured everything you need to make your first Swift game.

Once you have finished playing with the spinning squares, you can close the simulator down and return to Xcode. Note that you can use the keyboard command *command + Q* to exit the simulator, or press the stop button inside Xcode. If you use the stop button, the simulator will remain open and launch your next build faster.

Examining the demo code

Let's quickly explore the demo code. Do not worry about understanding everything just yet; we will cover each element in depth later. At this point, I am hoping you will acclimatize to the development environment and pick up a few things along the way. If you are stuck, keep going! Things will actually get simpler in the next chapter, once we clear away the SpriteKit demo and start on our own game.

Make sure you have `GameScene.swift` open in Xcode.

The demo `GameScene` class implements some functions you will use in your games. Let's examine these functions. Feel free to read the code inside each function, but I do not expect you to understand the specific code just yet:

- The game invokes the `didMove` function whenever it switches to the `GameScene`. You can think of it a bit like an `initialize`, or `main`, function for the scene. The SpriteKit demo uses it to draw the `Hello, World` text to the screen and set up the spinning square shape that shows up when we tap.
- There are seven functions involving touch, which handle the user's touch input to the iOS device screen. The SpriteKit demo uses these functions to spawn the spinning square wherever we touch the screen. Do not worry about understanding these functions at this time.
- The `update` function runs once for every frame drawn to the screen. The SpriteKit demo does not use this function, but we may have reason to implement it later.

Cleaning up

I hope that you have absorbed some Swift syntax and gained an overview of Swift and SpriteKit. It is time to make room for our own game; let's clear all of that demo code out! We want to keep a little bit of the boilerplate, but we can delete most of what is inside the functions. To be clear, I do not expect you to understand this code yet. This is simply a necessary step toward the start of our journey.

Firstly, we will remove the Hello, World text from the demo. Open the GameScene.sks file from the project navigator in Xcode. You will see a gray layout view with Hello, World written in the middle. Simply click anywhere on the Hello, World text and press your **Delete** key to remove it. Make sure you save your file before moving on.

Secondly, please replace all of the code from your GameScene.swift file with the following code:

```
import SpriteKit

class GameScene: SKScene {
    override func didMove(to view: SKView) {
    }
}
```

Once your GameScene.swift file looks like the preceding code, you are ready to move on to *Chapter 2, Sprites, Camera, Actions!* The real fun begins now!

Summary

You have already accomplished a lot. You have had your first experience with Swift, installed and configured your development environment, launched code successfully into the iOS simulator, and prepared your project for the first steps toward your own game. Great work!

We have seen enough of the "Hello World" demo; are you ready to draw your own graphics to the game screen? We will make use of sprites, textures, colors, and animation in *Chapter 2, Sprites, Camera, Action!*

2

Sprites, Camera, Action!

We will start our first game by learning how to draw shapes and textures on the screen. SpriteKit makes drawing simple by doing a lot of the hard work and exposing simple classes we can use for rendering. We are free to focus on building great gameplay experiences while SpriteKit performs the mechanical work of the game loop.

To draw an item on the screen, we need to create a new instance of a SpriteKit node. These nodes are simple; we attach a child node to our scene, or to existing nodes, for each item we want to draw. Sprites, particle emitters, and text labels are all considered nodes in SpriteKit.



The game loop is a common game design pattern used to constantly update the game many times per second and maintain the same gameplay speed on fast or slow hardware. SpriteKit wires new nodes into the game loop automatically. As you gain expertise with SpriteKit, you may wish to explore the game loop further to understand what is going on "under the hood".

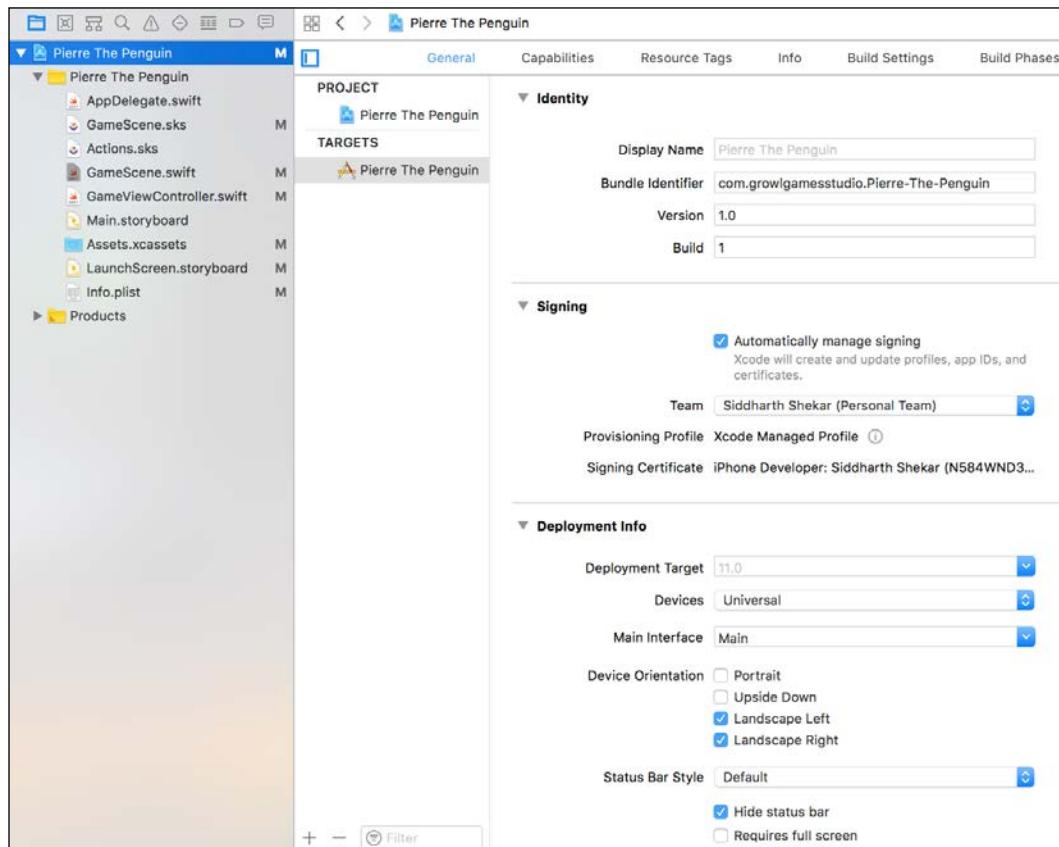
The topics in this chapter include the following:

- Preparing your project
- Drawing your first sprite
- Animation – movement, scaling, and rotation
- Working with textures
- Organizing art into texture atlases
- Centering the camera on a sprite

Preparing your project

There are three quick items to take care of before we start drawing. To begin the preparation, go through the following steps:

1. Since we will design our game to use landscape screen orientations, we will have to disable the portrait view altogether:
2. With your game project open in Xcode, select the overall Project folder in the project navigator (the uppermost item).
3. You will see your project settings in the main frame of Xcode. Under **Deployment Info**, find the **Device Orientation** section.
4. Uncheck the **Portrait** option, as shown in the following screenshot:



5. We need to resize our scene to fit the new landscape view. Follow these steps to resize the scene:

1. Open `GameViewController.swift` from the project navigator and locate the `viewDidLoad` function inside the `GameViewController` class. The `viewDidLoad` function is going to fire before the game realizes it is in landscape view, so we need to use a function that fires later in the startup process. Delete `viewDidLoad` completely, removing all of its code.
2. Replace `viewDidLoad` with a new function named `viewWillLayoutSubviews`. Do not worry about understanding every line right now; we are just configuring our project. Use this code for `viewWillLayoutSubviews`:

```
override func viewWillLayoutSubviews() {  
  
    super.viewWillLayoutSubviews()  
  
    if let view = self.view as! SKView? {  
        // Load the SKScene from 'GameScene.  
        sks'  
        if let scene = SKScene(fileNamed: "GameScene") {  
            // Set the scale mode fit the  
            window:  
            scene.scaleMode = .aspectFill  
                // Size our scene to fit the view  
                exactly:  
            scene.size = view.bounds.size  
                // Show the new scene:  
            view.presentScene(scene)  
        }  
  
        view.ignoresSiblingOrder = true  
        view.showsFPS = true  
        view.showsNodeCount = true  
    }  
}
```

3. Lastly, in `GameViewController.swift`, find the `supportedInterfaceOrientations` function and reduce it to this code:

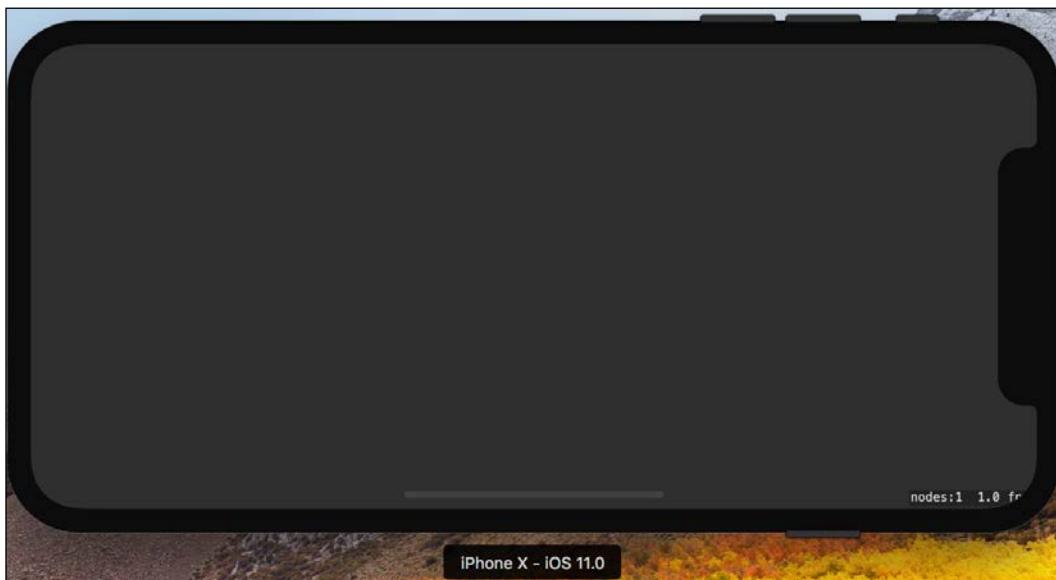
```
override var supportedInterfaceOrientations:  
UIInterfaceOrientationMask {  
    return .landscape  
}
```



Downloading the example code: You can download the example code files from your account at <http://www.packtpub.com> for all the Packt books you have purchased. If you have purchased this book elsewhere, you can visit <http://www.packtpub.com/support> and register to have the files emailed directly to you. Additionally, each chapter provides checkpoint links that you can use to download the example project to that point.

6. We should double-check that we are ready to move on. Try to run your clean project in the simulator using the toolbar play button or the *command + R* keyboard shortcut. After loading, the simulator should switch to landscape view with a blank gray background (and with the node and FPS counter at the bottom right). If the project will not run, or you still see "Hello, World", you will need to retrace your steps from the end of *Chapter 1, Designing Games with Swift*, namely, the section called *Cleaning up*, to finish your project preparation.

The following screenshot shows the output so far:



Checkpoint 2-A

I will provide checkpoints for each chapter. They are available in each chapter's code resources directory. I hope this will help you move forward if you are stuck or want to check your code against mine.

Drawing your first sprite

It is time to write some game code—fantastic! Open your `GameScene.swift` file and find the `didMove` function. Recall that this function fires every time the game switches to the `GameScene`. We will use this function to get familiar with the `SKSpriteNode` class. You will use `SKSpriteNode` extensively in your game whenever you want to add a new 2D graphic entity.



The term sprite refers to a 2D graphic or animation that moves around the screen independently from the background. Over time, the term has evolved to refer to any game object on the screen in a 2D game. We will create and draw your first sprite in this chapter: a happy little bee.

Building a `SKSpriteNode` class

Let's begin by drawing a blue square on the screen. The `SKSpriteNode` class can draw both texture graphics and solid blocks of color. It is often helpful to prototype your new game ideas with blocks of color before you spend time on artwork. To draw the blue square, add an instance of `SKSpriteNode` to the game:

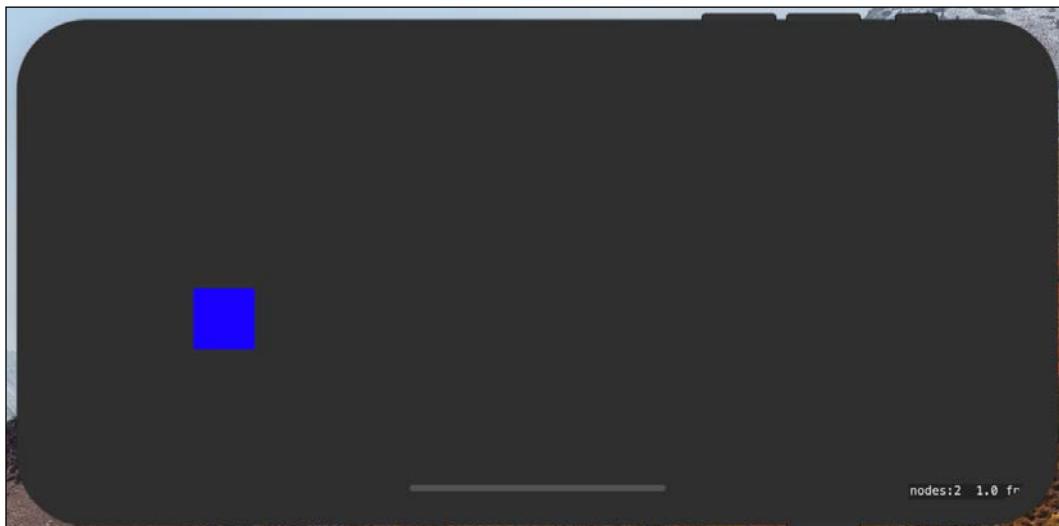
```
Override func didMove(to view: SKView) {
    // Make the scene position from its lower left
    // corner, regardless of any other settings:
    self.anchorPoint = .zero

    // Instantiate a constant, mySprite, instance of SKSpriteNode
    // The SKSpriteNode constructor can set color and size
    // Note: UIColor is a UIKit class with built-in colorpresets
    // Note: CGSize is a type we use to set node sizes
    let mySprite = SKSpriteNode(color: .blue, size:
        CGSize(width: 50, height: 50))
```

Sprites, Camera, Action!

```
// Assign our sprite a position in points, relative to its  
// parent node (in this case, the scene)  
mySprite.position = CGPointMake(x: 150, y: 150)  
  
// Finally, we need to add our sprite node into the node tree.  
// Call the SKScene's addChild function to add the node  
// Note: In Swift, 'self' is an automatic property  
// on any type instance, exactly equal to the instance itself  
// So in this case, it refers to the GameScene instance  
self.addChild(mySprite)  
}
```

Go ahead and run the project. You should see a similar small blue square appear in your simulator:



Swift allows you to define variables as constants, which can be assigned a value only once. For the best performance, use let to declare constants whenever possible. Declare your variables with var when you need to alter the value later in your code.

Animation—movement, scaling, and rotation

Before we dive back in to sprite theory, we should have some fun with our blue square. SpriteKit uses action objects to move sprites around the screen. Consider this example: if our goal is to move the square across the screen, we must first create a new action object to describe the animation. Then, we instruct our sprite node to execute the action. I will illustrate this concept with many examples in the chapter. For now, add this code in the `didMove` function, below the `self.addChild(mySprite)` line:

```
// Create a new constant for our action instance
// Use the move action to provide a goal position for a node
// SpriteKit will tween to the new position over the course of the
// duration, in this case 5 seconds
let demoAction = SKAction.move(to: CGPoint(x: 300, y: 150),
duration: 3)
// Tell our square node to execute the action!
mySprite.run(demoAction)
```

Run the project. You will see our blue square slide across the screen toward the (300, 150) position. This action is reusable—any node in your scene can execute this action to move to the (300, 150) position. As you can see, SpriteKit does a lot of the heavy lifting for us when we need to animate node properties.



Inbetweening, or tweening, uses the engine to animate smoothly between a start frame and an end frame. Our `SKAction.move` animation is a tween; we provide the start frame (the sprite's original position) and the end frame (the new destination position). SpriteKit generates the smooth transition between our values.

Let's try some other actions. The `SKAction.move` function is only one of many options. Try replacing the `demoAction` line with this code:

```
let demoAction = SKAction.scale(to: 4, duration: 5)
```

Run the project. You will see our blue square grow to four times its original size.

Sequencing multiple animations

We can execute actions together simultaneously, or one after the other, with action groups and sequences. For instance, we can easily make our sprite larger and spin it at the same time. Delete all of our animation code so far and replace it with this code:

```
// Scale up to 4X initial scale
let demoAction1 = SKAction.scale(to: 4, duration: 5)
// Rotate 5 radians
let demoAction2 = SKAction.rotate(byAngle: 5, duration: 5)
// Group the actions
let actionGroup = SKAction.group([demoAction1, demoAction2])
// Execute the group!
mySprite.run(actionGroup)
```

When you run the project, you will see a spinning, growing square. Terrific! If you want to run these actions in sequence (rather than at the same time) change `SKAction.group` to `SKAction.sequence`:

```
// Group the actions into a sequence
let actionSequence = SKAction.sequence([demoAction1, demoAction2])

// Execute the sequence!
mySprite.run(actionSequence)
```

Run the code and watch as your square first grows and then spins. Good. We are not limited to two actions; we can group or sequence as many actions together as we need.

We have only used a few actions so far; feel free to explore the `SKAction` class and try out different action combinations before moving on. You can find a full list of actions in Apple's `SKAction` Class Reference at https://developer.apple.com/library/mac/documentation/SpriteKit/Reference/SKAction_Ref/.

Recapping your first sprite

Congratulations, you have learned to draw a non-textured sprite and animate it with SpriteKit actions. Next, we will explore some important positioning concepts and then add game art to our sprites.

Before you move on, make sure that your `didMove` function matches mine and that your sequenced animation is firing properly. Here is my code up to this point:

```
override func didMove(to view: SKView) {
    // Make the scene position from its lower left
    // corner, regardless of any other settings:
```

```
self.anchorPoint = .zero

// Instantiate a constant, mySprite, instance of SKSpriteNode
let mySprite = SKSpriteNode(color: .blue, size:
    CGSize(width: 50, height: 50))

    // Assign our sprite a position
mySprite.position = CGPoint(x: 150, y: 150)

    // Add our sprite node into the node tree
self.addChild(mySprite)

    // Scale up to 4X initial scale
let demoAction1 = SKAction.scale(to: 4, duration: 5)
    // Rotate 5 radians
let demoAction2 = SKAction.rotate(byAngle: 5, duration: 5)

    // Group the actions
let actionSequence = SKAction.sequence([demoAction1,
    demoAction2])
    // Execute the group!
mySprite.run(actionSequence)
}
```

The story on positioning

SpriteKit uses a grid of points to position nodes. In this grid, the bottom left corner of the scene is (0,0), with a positive x-axis to the right and a positive y-axis to the top.

Similarly, on the individual sprite level, (0,0) refers to the bottom-left corner of the sprite, while (1,1) refers to the top-right corner.

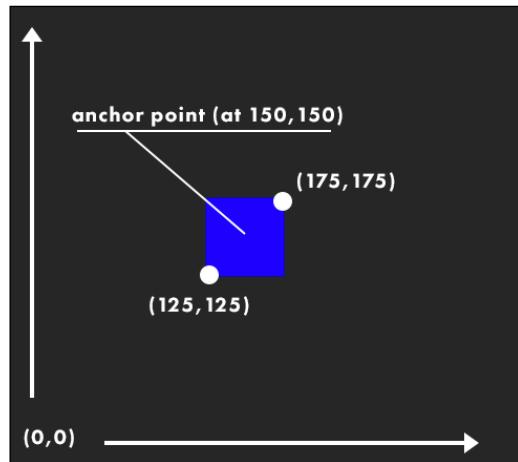
Alignment with anchor points

Each sprite has an `anchorPoint` property, or an origin. The `anchorPoint` property allows you to choose which part of the sprite aligns to the sprite's overall position.



The default anchor point is (0.5,0.5), so a new `SKSpriteNode` centers perfectly on its position.

To illustrate this, let's examine the blue square sprite we just drew on the screen. Our sprite is 50 points wide and 50 points tall, and its position is (150,150). Since we have not modified the `anchorPoint` property, its anchor point is (0.5,0.5). This means that the sprite will be perfectly centered over the (150,150) position on the scene's grid. Our sprite's left edge begins at 125 and the right edge terminates at 175. Likewise, the bottom starts at 125 and the top ends at 175. The following diagram illustrates our block's position on the grid:



Why do we prefer centered sprites by default? You may think it's simpler to position elements by their bottom-left corner with an `anchorPoint` property setting of (0,0). However, the centered behavior benefits us when we scale or rotate sprites:

- When we scale a sprite with a bottom-left `anchorPoint` property of (0,0), it will only expand up the y-axis and across the x-axis. Rotation actions will swing the sprite in wide circles around its bottom-left corner.
- A centered sprite, with the default `anchorPoint` property of (0.5, 0.5), will expand or contract equally in all directions when scaled and will spin in place when rotated, which is usually the desired effect.

There are some cases where you will want to change an anchor point. For instance, if you are drawing a rocket ship, you may want the ship to rotate around the front nose of its cone rather than its center.

Working with textures

You may want to take a screenshot of your blue box for your own enjoyment later. I absolutely love reminiscing over old screenshots of my finished games when they were nothing more than simple colored blocks sliding around the screen. Now, it is time to move past that stage and attach some fun artwork to our sprite.

Downloading the free assets

I am providing a downloadable pack for all of the art assets I use in this book. I recommend you use these assets so that you will have everything you need for our demo game. Alternatively, you are certainly free to create your own art for your game if you prefer.

More exceptional art assets

If you like the art assets, you can download over 20,000 game assets in the same style for a small donation at <http://kenney.itch.io/kenney-donation>. I do not have an affiliation with Kenney; I just find it admirable that he has released so much public domain artwork for indie game developers.

These assets are public domain, which means you can copy, modify, and distribute the art assets, even for commercial purposes, all without asking permission. You can read the full license at <https://creativecommons.org/publicdomain/zero/1.0/>.

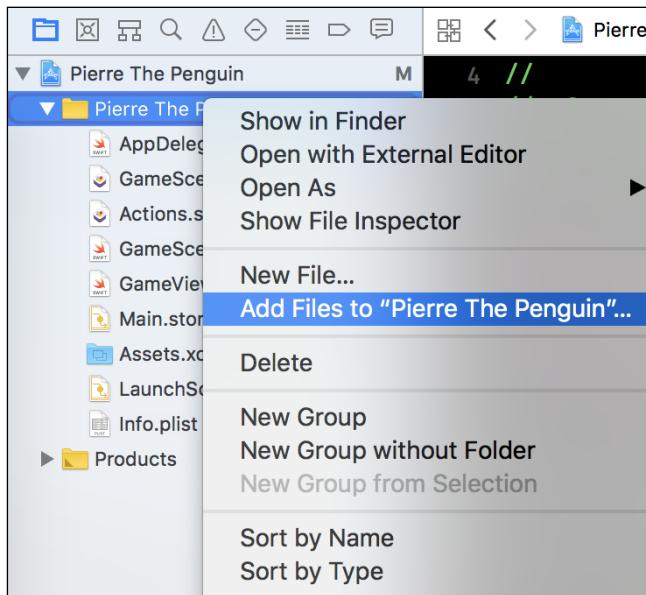
Drawing your first textured sprite

Let's use some of the graphics you just downloaded. We will start by creating a bee sprite. We will add the bee texture to our project, load the image onto a `SKSpriteNode` class, and then size the node for optimum sharpness on Retina screens.

Adding the bee image to your project

We need to add the image files to our Xcode project before we can use them in the game. Once we add the images, we can reference them by name in our code; SpriteKit is smart enough to find and implement the graphics. Follow these steps to add the bee image to the project:

1. Right-click on your project in the project navigator and click **Add Files to "Pierre Penguin Escapes the Antarctic"** (or the name of your game). Refer to this screenshot to find the correct menu item:



2. Browse to the asset pack you downloaded and locate the `bee@3x.png` image inside the `Enemies` folder.
3. Check **Copy** items if needed, then click **Add**.

You should now see `bee@3x.png` in your project navigator.

Loading images with `SKSpriteNode`

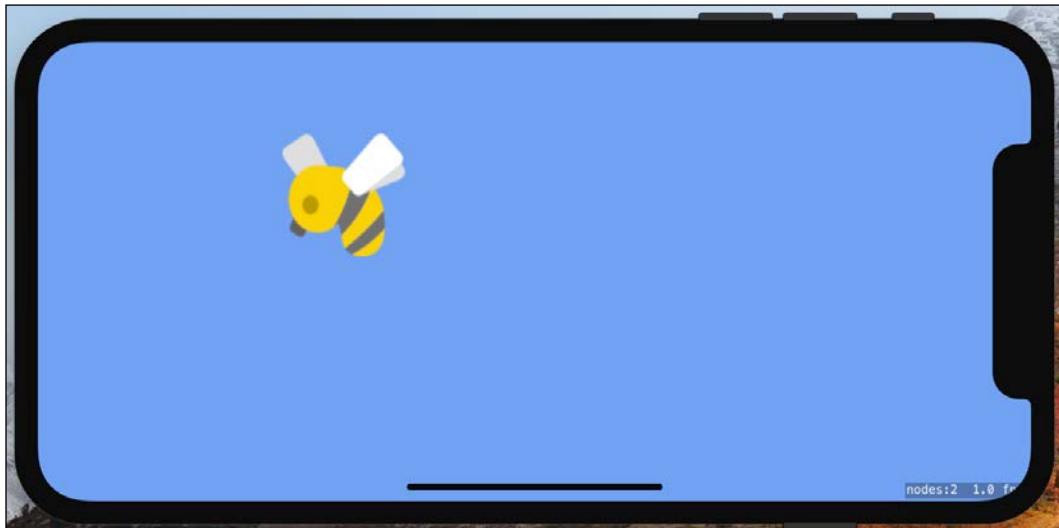
It is quite easy to draw images to the screen with `SKSpriteNode`. Start by clearing out all of the code we wrote for the blue square inside the `didMove` function in `GameScene.swift`. Replace `didMove` with this code:

```
Override func didMove(to view: SKView) {  
    // Position from the lower left corner
```

```
self.anchorPoint = .zero
    // set the scene's background to a nice sky blue
    // Note: UIColor uses a scale from 0 to 1 for its colors
    self.backgroundColor = UIColor(red: 0.4, green: 0.6, blue:
        0.95, alpha: 1.0)

    // create our bee sprite node
let bee = SKSpriteNode(imageNamed: "bee")
    // size our bee node
bee.size = CGSize(width: 100, height: 100)
    // position our bee node
bee.position = CGPoint(x: 250, y: 250)
    // attach our bee to the scene's node tree
self.addChild(bee)
}
```

Run the project and witness our glorious bee—great work!



Designing for Retina

You may notice that our bee image is quite blurry. To take advantage of retina screens, assets need to be twice the pixel dimensions of their node's size property (for most retina screens), or three times the node size for the Plus versions of the iPhone. Ignore the height for a moment; our bee node is 100 points wide, but the PNG file is only 84 pixels wide. The PNG file needs to be 300 pixels wide to look sharp on Plus-sized iPhones, or 200 pixels wide to look sharp on 2X retina devices.

SpriteKit will automatically resize textures to fit their nodes, so one approach is to create a giant texture at the highest retina resolution (three times the node size) and let SpriteKit resize the texture down for lower density screens. However, there is a considerable performance penalty, and older devices can even run out of memory and crash from huge textures.

The ideal asset approach

These double and triple-sized retina assets can be confusing to new iOS developers. To solve this issue, Xcode lets you provide three image files for each texture. For example, our bee node is currently 100 points wide and 100 points tall. You can provide the following images to Xcode and it will automatically use the image best suited to the device:

- Bee.png (100 pixels by 100 pixels)
- Bee@2x.png (200 pixels by 200 pixels)
- Bee@3x.png (300 pixels by 300 pixels)

Simplifying matters, Swift only runs on iOS7 and higher. The only non-retina devices that run iOS7 are the aging iPad 2 and iPad Mini 1st generation. If these older devices are important for your finished games, you should create all three sizes of images for your games. Otherwise, you can safely ignore non-retina assets with Swift and create only 2X and 3X sized images.

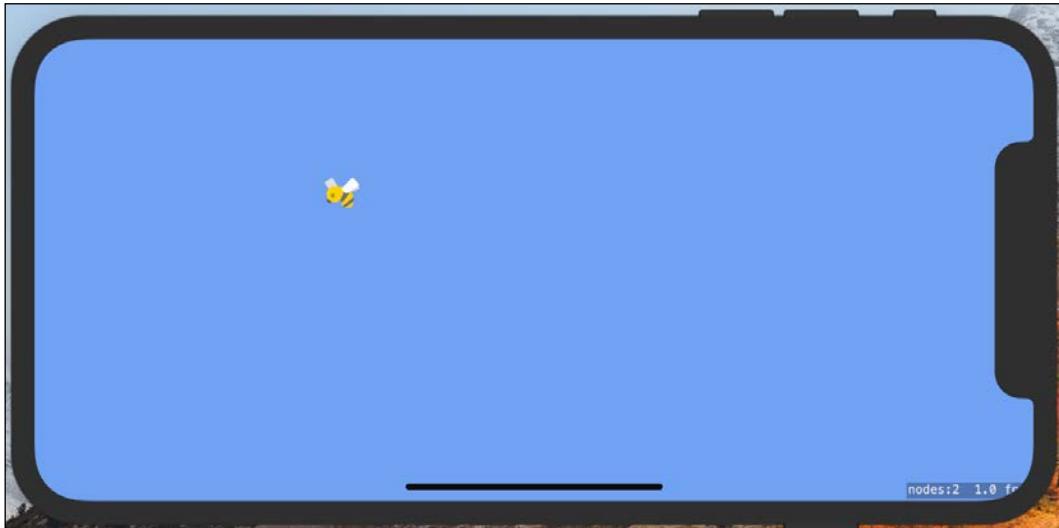
Hands-on with retina in SpriteKit

Our bee image illustrates how this all works:

- Because we set an explicit node size, SpriteKit automatically resizes the bee texture to fit our 100 point wide, 100 point tall node. This automatic size-to-fit is very handy, but notice that we have actually slightly distorted the aspect ratio of the image.
- If we do not set an explicit size, SpriteKit sizes the node to match the texture's dimensions. Go ahead and delete the line that sets the size for our bee node and rerun the project. SpriteKit maintains the aspect ratio automatically and sizes the bee correctly down to 28 points by 24 points based on the @3x suffix.
- This works wonderfully, but I still like to set my node sizes explicitly. Set the size property of your bee node to 28 points wide by 24 points tall:

```
// size our bee in points:  
bee.size = CGSizeMake(width: 28, height: 24)
```

Run the project and you will see a smaller, crystal clear bee, as in this screenshot:



Great! The important concept here is to design your art files at three times the point sizes of your sprite nodes to take full advantage of 3x retina screens. Now, we will look at organizing and animating multiple sprite frames.

Organizing art into texture atlases

We will quickly overrun our project navigator with image files if we add all our textures like we did with our bee. Luckily, Xcode provides several solutions.

Exploring Assets.xcassets

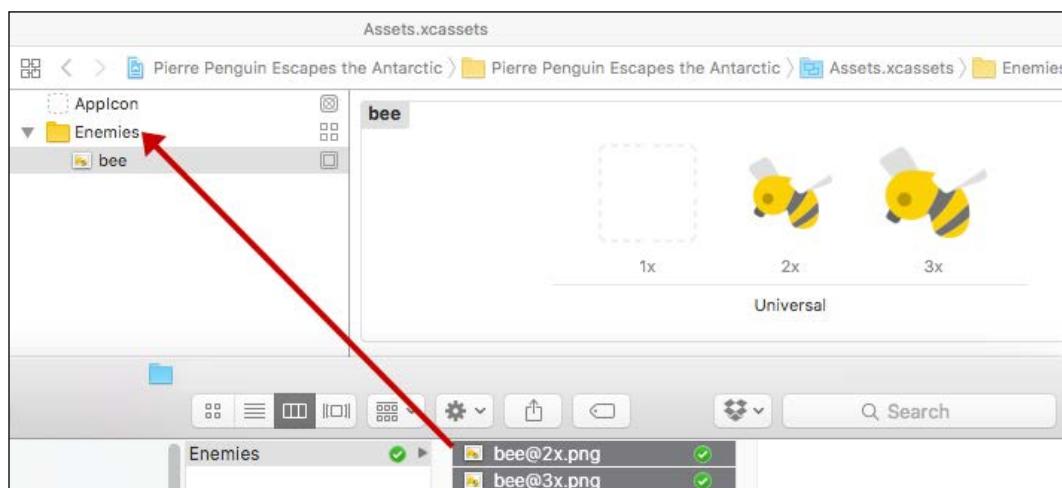
We can store images in an .xcassets file and refer to them easily from our code. Follow these steps to prepare our .xcassets file:

1. Open `Assets.xcassets` from your project navigator.
2. You will see an empty `AppIcon` entry. You can leave it there for now; we will revisit the `AppIcon` later.

Collecting art into texture atlases

We will use texture atlases for most of our in-game art. Texture atlases organize assets by collecting related artwork together. They also increase performance by optimizing all of the images inside each atlas as if they were one texture. SpriteKit only needs one draw call to render multiple images out of the same texture atlas. Plus, they are very easy to use! Follow these steps to build your bee texture atlas:

1. First, we need to remove our old bee texture. Right-click on `bee@3x.png` in the project navigator and choose **Delete**, then **Move to Trash**.
2. In `Assets.xcassets`, right-click in the left panel and select **New Sprite Atlas**.
3. You should see a new folder in the left panel called `Sprites`. This is our new texture atlas. Double-click it, or select it, and press your *Enter* key to rename it, then name it `Enemies` (the bee will eventually be an enemy in our game).
4. Inside this atlas, you will see a demo entry, called `Sprite`. We can delete this demo sprite; we will create our own entries for our game. Right-click `Sprite` and select **Remove Selected Items**.
5. Open the asset bundle that you downloaded and locate the `bee@2x.png` and `bee@3x.png` images inside the `Enemies` folder.
6. There are several ways to import files into a texture atlas. The easiest way is to select all sizes of the image in the Finder and then drag and drop them onto the `Enemies` texture atlas icon in the left panel of Xcode. You can do this now, and you should see a new sprite created inside `Enemies`, called `bee`, as shown in this screenshot:



7. We also want to create another sprite in this atlas for the second frame of our bee's flying animation. Repeat this drag and drop exercise by selecting the asset files named `bee-fly@2x.png` and `bee-fly@3x.png`, and dragging them onto the `Enemies` texture atlas in Xcode.ss

You should now have a texture atlas named `Enemies` that contains two sprites: `bee` and `bee-fly`. Each of these sprites has both a 2X and 3X version to take full advantage of retina screens. Good work; we have organized our bee assets into one collection. Xcode will now automatically create performance optimizations and select the best size for each device.

Updating our bee node to use the texture atlas

We can actually run our project right now and see the same bee as before. Our old bee texture was `bee`, and a new `bee` sprite exists in the texture atlas. Although we deleted the standalone `bee@3x.png`, SpriteKit is smart enough to find the new `bee` in the texture atlas.

We should make sure that our texture atlas is working and that we successfully deleted the old individual `bee@3x.png`. In `GameScene.swift`, change our `SKSpriteNode` instantiation line to use the new `bee-fly` graphic from the texture atlas:

```
// create our bee sprite
// notice the new image name: bee-fly
let bee = SKSpriteNode(imageNamed: "bee-fly")
```

Run the project again. You should see a different bee image, its wings held lower than before. This is the second frame of the bee animation. Next, we will learn how to animate between the two frames to create an animated sprite.

Iterating through texture atlas frames

We need to study one more texture atlas technique: we can quickly flip through multiple sprite frames to make our bee come alive with motion. We now have two frames of our bee in flight; it should appear to hover in place if we switch back and forth between these frames.

Our node will run a new `SKAction` to animate between the two frames. Update your `didMove` function to match mine (I removed some older comments to save space):

```
Override func didMove(to view: SKView) {
    self.anchorPoint = .zero
    self.backgroundColor = UIColor(red: 0.4, green: 0.6, blue:
        0.95, alpha: 1.0)

    // create our bee sprite
    // Note: Remove all prior arguments from this line:
    let bee = SKSpriteNode()
    bee.position = CGPoint(x: 250, y: 250)
    bee.size = CGSize(width: 28, height: 24)
    self.addChild(bee)

    // Find our new bee texture atlas
    let beeAtlas = SKTextureAtlas(named:"Enemies")
    // Grab the two bee frames from the texture atlas in an array
    // Note: Check out the syntax explicitly declaring beeFrames
    // as an array of SKTextures. This is not strictly necessary,
    // but it makes the intent of the code more readable, so I
    // chose to include the explicit type declaration here:
    let beeFrames:[SKTexture] = [
        beeAtlas.textureNamed("bee"),
        beeAtlas.textureNamed("bee-fly")]
    // Create a new SKAction to animate between the frames once
    let flyAction = SKAction.animate(with: beeFrames,
        timePerFrame: 0.14)
    // Create an SKAction to run the flyAction repeatedly
    let beeAction = SKAction.repeatForever(flyAction)
    // Instruct our bee to run the final repeat action:
    bee.run(beeAction)
}
```

Run the project. You will see our bee flap its wings back and forth—cool! You have learned the basics of sprite animation with texture atlases. We will create increasingly complicated animations using this same technique later in this book. For now, pat yourself on the back. The result may seem simple, but you have unlocked a major building block toward your first SpriteKit game!

Putting it all together

First, we learned how to use actions to move, scale, and rotate our sprites. Then, we explored animating through multiple frames, bringing our sprites to life. Let's now combine these techniques to fly our bee back and forth across the screen, flipping the texture at each turn.

Add this code at the bottom of the `didMove` function, beneath the `bee.run(beeAction)` line:

```
// Set up new actions to move our bee back and forth:  
let pathLeft = SKAction.moveBy(x: -200, y: -10, duration: 2)  
let pathRight = SKAction.moveBy(x: 200, y: 10, duration: 2)  
// These two scaleX actions flip the texture back and forth  
// We will use these to turn the bee to face left and right  
let flipTextureNegative = SKAction.scaleX(to: -1, duration: 0)  
let flipTexturePositive = SKAction.scaleX(to: 1, duration: 0)  
// Combine actions into a cohesive flight sequence for our bee  
let flightOfTheBee = SKAction.sequence([pathLeft,  
    flipTextureNegative, pathRight, flipTexturePositive])  
// Last, create a looping action that will repeat forever  
let neverEndingFlight =  
SKAction.repeatForever(flightOfTheBee)  
  
// Tell our bee to run the flight path, and away it goes!  
bee.run(neverEndingFlight)
```

Run the project. You will see the bee flying back and forth, flapping its wings. You have officially learned the fundamentals of animation in SpriteKit! We will build on this knowledge to create a rich, animated game world for our players.

Centering the camera on a sprite

Games often require the camera to follow the player sprite as it moves through space. We definitely want this camera behavior for Pierre, our penguin character, whom we will soon be adding to the game. With iOS9, Apple added a new `SKCameraNode` class, which makes this task easy. We will attach an `SKCameraNode` to our scene and position it directly over the player to keep their character centered in the view.

You can find the code for our camera functionality in the following code block. Read the comments for a detailed explanation. This is a quick recap of the changes:

- Our `didMove` function was becoming too crowded. We broke out our flying bee code into a new function named `addTheFlyingBee`. Later, we will encapsulate game objects, such as bees, into their own classes.
- We created two new constants on the `GameScene` class: the camera node and the bee node.
- We updated the `didMove` function. It assigns the new camera node to the scene's camera.
- We have added a new function to add a background sprite, just to show that the bee is actually moving and that the camera is centered on the bee.
- Inside the new `addTheFlyingBee` function, we removed the bee constant, as `GameScene` now declares it above as its own property.
- We are implementing a new function: `didSimulatePhysics`. SpriteKit calls this function every frame after performing physics calculations and adjusting positions. It's a great place to update our camera position. The code to change the camera position and keep the view centered on the player resides in this new function.

Please update your entire `GameScene.swift` file to match mine:

```
import SpriteKit

class GameScene: SKScene {
    // Create a constant cam as a SKCameraNode:
    let cam = SKCameraNode()
    // Create our bee node as a property of GameScene so we can
    // access it throughout the class
    // (Make sure to remove the old bee declaration below)
    let bee = SKSpriteNode()

    override func didMove(to view: SKView) {
        self.anchorPoint = .zero
        self.backgroundColor = UIColor(red: 0.4, green: 0.6, blue:
            0.95, alpha: 1.0)

        // Assign the camera to the scene
        self.camera = cam

        // Call the new bee function
        self.addTheFlyingBee()
```

```
// Add background
self.addBackground()

}

// A new function
Override func didSimulatePhysics() {
    // Keep the camera centered on the bee
    // Notice the !operator after camera. SKScene's camera
    // is an optional, but we know it is there since we
    // assigned it above in the didMove function. We can tell
    // Swift that we know it can unwrap this value by using
    // the !operator after the property name.
self.camera!.position = bee.position
}

// new function to add background
func addBackground(){

let bg = SKSpriteNode(imageNamed:"background-menu")
bg.position = CGPoint(x: 250, y: 250)
self.addChild(bg)
}

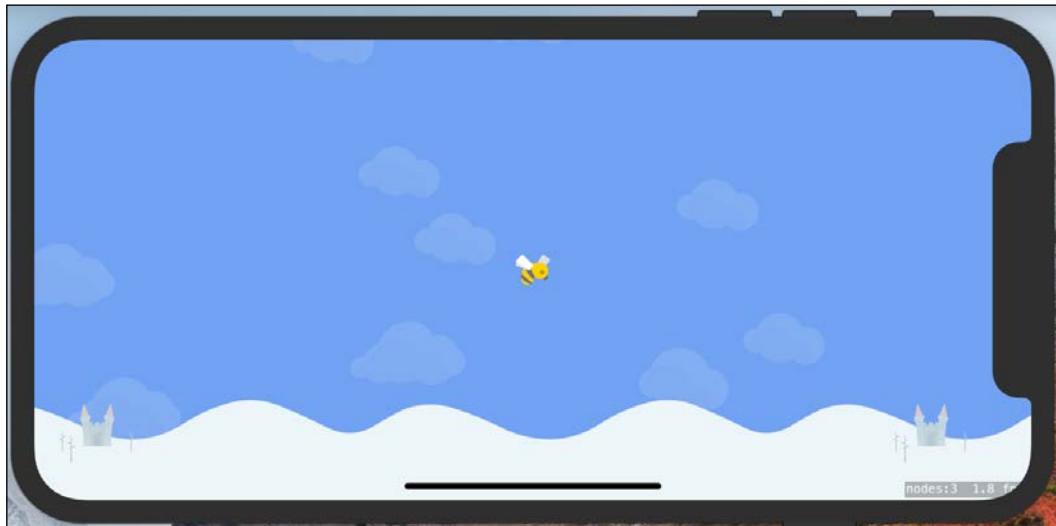
// I moved all of our bee animation code into a new function:
func addTheFlyingBee() {
    // Position our bee
bee.position = CGPoint(x: 250, y: 250)
bee.size = CGSize(width: 28, height: 24)
    // Add the bee to the scene
self.addChild(bee)

    // Find the bee textures from the texture atlas
let beeAtlas = SKTextureAtlas(named:"Enemies")
let beeFrames:[SKTexture] = [
beeAtlas.textureNamed("bee"),
beeAtlas.textureNamed("bee-fly")]
    // Create a new SKAction to animate between the frames
let flyAction = SKAction.animate(with: beeFrames,
timePerFrame: 0.14)
    // Create an SKAction to run the flyAction repeatedly
let beeAction = SKAction.repeatForever(flyAction)
```

Sprites, Camera, Action!

```
// Instruct our bee to run the final repeat action:  
bee.run(beeAction)  
  
    // Set up new actions to move our bee back and forth:  
let pathLeft =  
SKAction.moveBy(x: -200, y: -10, duration: 2)  
let pathRight =  
SKAction.moveBy(x: 200, y: 10, duration: 2)  
let flipTextureNegative =  
SKAction.scaleX(to: -1, duration: 0)  
let flipTexturePositive =  
SKAction.scaleX(to: 1, duration: 0)  
    // Combine actions into a cohesive flight sequence  
let flightOfTheBee = SKAction.sequence([  
pathLeft, flipTextureNegative, pathRight,  
flipTexturePositive])  
    // Last, create a looping action that will repeat forever  
let neverEndingFlight =  
SKAction.repeatForever(flightOfTheBee)  
  
    // Tell our bee to run the flight path, and away it goes!  
bee.run(neverEndingFlight)  
}  
}
```

Run the game. You should see our bee stuck directly at the center of the screen, flipping back and forth every two seconds:



The bee is actually changing position, just as before, but the camera is moving to keep the bee centered on the screen. When we add more game objects in *Chapter 3, Mixing in the Physics*, our bee will appear to be flying as the entire world pans past the screen.



Checkpoint 2-B

The code up to this point is available in this chapter's code resources.

Summary

You have gained foundational knowledge of sprites, nodes, and actions in SpriteKit, and have already taken huge strides toward your first game with Swift.

You configured your project for landscape orientation, drew your first sprite, and then made it move, spin, and scale. You added a bee texture to your sprite, created an image atlas, and animated through the frames of flight. Lastly, you built a camera to keep the gameplay centered on the player. Terrific work!

In the next chapter, we will use SpriteKit's physics engine to assign weight and gravity to our world, spawn more flying characters, and create the ground and sky.

3

Mix in the Physics

SpriteKit includes a fully functional physics engine. It is easy to implement and very useful, as most mobile game designs require some level of physical interaction between game objects. In our game, we want to know when the player runs into the ground, an enemy, or a power-up. The physics system can track these collisions and execute our specific game code when any of these events occur. SpriteKit's physics engine can also apply gravity to the game world – as well as the bounce and spin that can occur when sprites collide with each other – and creates realistic movement through impulses; and it does all of this before every single frame is drawn on the screen!

The topics in this chapter include the following:

- Adopting a protocol for consistency
- Organizing game objects into classes
- Adding the player's character
- Renovating the GameScene class
- Physics bodies and gravity
- Exploring physics simulation mechanics
- Movement with impulses and forces
- Bumping bees into bees

Laying the foundations

So far, we have learned through small bits of code, individually added to the GameScene class. The intricacy of our application is about to increase. To build a complex game world, we will need to construct reusable classes and actively organize our new code.

Adopting a protocol for consistency

To start, we want individual classes for each of our game objects (a bee class, a player penguin class, a power-up class, and so on). Furthermore, we want all of our game object classes to share a consistent set of properties and methods. We can enforce this commonality by creating a protocol, or a blueprint, for our game classes. The protocol does not provide any functionality on its own, but each class that adopts the protocol must follow its specifications exactly before Xcode can compile the project. Protocols are very similar to interfaces, if you are from a Java or C# background.

Add a new file to your project (right-click in the project navigator and choose **New File**, then **Swift File**) and name it `GameSprite`. Then, add the following code to your new file:

```
import SpriteKit

protocol GameSprite {
    var textureAtlas: SKTextureAtlas { get set }
    var initialSize: CGSize { get set }
    func onTap()
}
```

Now, any class that adopts the `GameSprite` protocol must implement a `textureAtlas` property, an `initialSize` property, and an `onTap` function. We can safely assume that the game objects provide these implementations when we work with them in our code.

Organizing game objects into classes

Our old bee is working wonderfully, but we want to spawn many bees throughout the game world. We will create a `Bee` class, inheriting from `SKSpriteNode`, so we can cleanly stamp as many bees to the world as we please.

It is a common convention to separate each class into its own file. Add a new `Swift` file to your project and name it `Bee`. Then, add the following code:

```
import SpriteKit

// Create the new class Bee, inheriting from SKSpriteNode
// and adopting the GameSprite protocol:
class Bee: SKSpriteNode, GameSprite {
    // We will store our size, texture atlas, and animations
    // as class wide properties.
    var initialSize: CGSize = CGSize(width: 28, height: 24)
```

```
var textureAtlas: SKTextureAtlas =
SKTextureAtlas(named: "Enemies")
Var flyAnimation = SKAction()

    // The init function will be called when Bee is instantiated:
init() {
    // Call the init function on the base class (SKSpriteNode)
    // We pass nil for the texture since we will animate the
    // texture ourselves.
super.init(texture: nil, color: .clear, size:
initialSize)
    // Create and run the flying animation:
createAnimations()
self.run(flyAnimation)
}

// Our bee only implements one texture based animation.
// But some classes may be more complicated,
// So we break out the animation building into this function:
func createAnimations() {
    let flyFrames:[SKTexture] =
        [textureAtlas.textureNamed("bee"),
textureAtlas.textureNamed("bee-fly")]
    let flyAction = SKAction.animate(with: flyFrames,
timePerFrame: 0.14)
flyAnimation = SKAction.repeatForever(flyAction)
}

// onTap is not wired up yet, but we have to implement this
// function to conform to our GameSprite protocol.
// We will explore touch events in the next chapter.
func onTap() {}

// Lastly, we are required to add this bit of boilerplate
// to subclass SKSpriteNode. We will need to do this any
// time we inherit from SKSpriteNode and use an init function
required init?(coder aDecoder: NSCoder) {
super.init(coder: aDecoder)
}
}
```

Mix in the Physics

It is now easy to spawn as many bees as we like. Switch back to `GameScene.swift` and add this code at the bottom of `didMove`:

```
// Add a second Bee to the scene:  
let bee2 = Bee()  
bee2.position = CGPoint(x: 325, y: 325)  
self.addChild(bee2)  
// ... and a third Bee:  
let bee3 = Bee()  
bee3.position = CGPoint(x: 200, y: 325)  
self.addChild(bee3)
```

Run the project Bees, bees everywhere! Our original bee is flying back and forth through a swarm. Your simulator should look like this:



Next, we will add the ground.

The icy tundra

We will add some ground at the bottom of the screen to serve as a constraint for player positioning and as a reference point for movement. We will need to create a new class named `Ground`. First, let's add the texture atlas for the ground art to our project.

Adding the ground texture to Assets.xcassets

We need to add our ground texture sprite, just as we added the bee sprites earlier. Once again, we will create a texture atlas in the `Assets.xcassets` file to hold our ground texture and other environmental textures we will use along the way. Follow these steps to add the ground texture to our project:

1. Open the `Assets.xcassets` file in Xcode, then right-click in the left panel and select **New Sprite Atlas**.
2. Change the name of the new sprite atlas from `Sprites` to `Environment` (we will use this texture atlas for all the environment textures in our game).
3. Xcode creates a new sprite—named `Sprite`—inside this atlas by default. Remove it by right-clicking it and selecting **Remove Selected Items**.
4. In **Finder**, open the asset pack you downloaded. Locate the `Environment` folder and select `ground@2x.png` and `ground@3x.png`.
5. Drag and drop these two files into Xcode on top of the `Environment` texture atlas.

Xcode should create a new sprite named `ground` inside the `Environment` atlas. When you are done, your `Assets.xcassets` should look like this:



Adding the Ground class

Next, we will add the code for the ground. Add a new Swift file to your project and name it `Ground`. Use the following code:

```
import SpriteKit

// A new class, inheriting from SKSpriteNode and
// adhering to the GameSprite protocol.
class Ground: SKSpriteNode, GameSprite {
```

```
var textureAtlas: SKTextureAtlas =
SKTextureAtlas(named: "Environment")
    // We will not use initialSize for ground, but we still need
    // to declare it to conform to our GameSprite protocol:
Var initialSize = CGSize.zero

    // This function tiles the ground texture across the width
    // of the Ground node. We will call it from our GameScene.
func createChildren() {
    // This is one of those unique situations where we use a
    // non-default anchor point. By positioning the ground by
    // its top left corner, we can place it just slightly
    // above the bottom of the screen, on any of screen size.
self.anchorPoint = CGPointMake(x: 0, y: 1)

    // First, load the ground texture from the atlas:
let texture = textureAtlas.textureNamed("ground")

var tileCount: CGFloat = 0
    // We will size the tiles in their point size
    // They are 35 points wide and 300 points tall
let tileSize = CGSize(width: 35, height: 300)

    // Build nodes until we cover the entire Ground width
while tileCount * tileSize.width < self.size.width {
    let tileNode = SKSpriteNode(texture: texture)
tileNode.size = tileSize
tileNode.position.x = tileCount * tileSize.width
    // Position child nodes by their upper left corner
tileNode.anchorPoint = CGPointMake(x: 0, y: 1)
        // Add the child texture to the ground node:
self.addChild(tileNode)

tileCount += 1
}
}

    // Implement onTap to adhere to the protocol:
func onTap() {}
}
```

Tiling a texture

Why do we need the `createChildren` function? This is one method of tiling textures. We can create a child node for each texture tile and append them across the width of the parent. Performance is not an issue; as long as we attach the children to one parent, and the textures all come from the same texture atlas, SpriteKit handles them with one draw call.

Running wire to the ground

We have added the ground art to the project and created the `Ground` class. The final step is to create an instance of `Ground` in our scene. Follow these steps to wire up the ground:

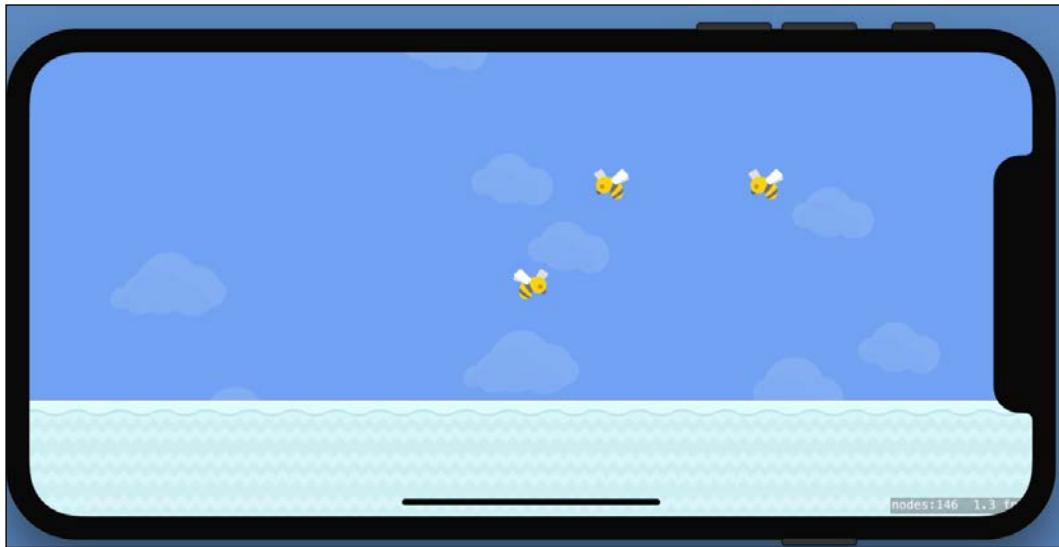
1. Open `GameScene.swift` and add a new property to the `GameScene` class to create an instance of the `Ground` class. You can place this underneath the line that instantiates the `cam` node (the new code is in bold):

```
let cam = SKCameraNode()  
let ground = Ground()
```

2. Locate the `didMove` function. Add the following code at the bottom, underneath our bee-spawning lines:

```
// Position the ground based on the screen size.  
// Position X: Negative one screen width.  
// Position Y: 150 above the bottom (remember the top  
// left anchor point).  
ground.position = CGPoint(x: -self.size.width * 2, y: 150)  
    // Set the ground width to 3x the width of the scene  
    // The height can be 0, our child nodes will create the  
height  
ground.size = CGSize(width: self.size.width * 6, height: 0)  
    // Run the ground's createChildren function to build  
    // the child texture tiles:  
ground.createChildren()  
    // Add the ground node to the scene:  
self.addChild(ground)
```

Run the project. You will see the icy tundra appear underneath our bees. This small change goes a long way toward creating the feeling that our central bee is moving through space. Your simulator should look like this:

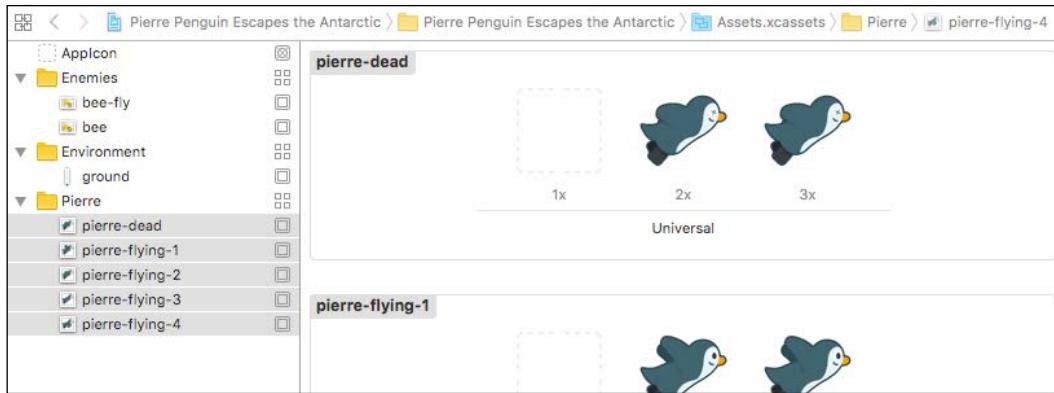


Adding the player's character

There is one more class to build before we start our physics lesson: the `Player` class! It is time to replace our moving bee with a node designated as the player.

First, we will add the sprite atlas for our penguin art. By now, you should be familiar with adding new sprite atlases and sprites to the `Assets.xcassets` file. Follow these steps to add the flying penguin art to your project:

1. Create a new sprite atlas named `Pierre` in `Assets.xcassets` by right-clicking in the left panel and selecting **New Sprite Atlas**.
2. Locate the `Pierre` folder in your downloaded asset bundle. Drag and drop all of the `.png` files from this folder onto the `Pierre` atlas in Xcode.
3. Your `Assets.xcassets` file should now look like this:



4. Now that you have Pierre Penguin's textures in a texture atlas, you can create the Player class. Add a new Swift file to your project and name it `Player.swift`. Then, add this code:

```
import SpriteKit

class Player : SKSpriteNode, GameSprite {
    var initialSize = CGSize(width: 64, height: 64)
    var textureAtlas: SKTextureAtlas =
        SKTextureAtlas(named: "Pierre")
        // Pierre has multiple animations. Right now, we will
        // create one animation for flying up,
        // and one for going down:
    var flyAnimation = SKAction()
    var soarAnimation = SKAction()

    init() {
        // Call the init function on the
        // base class (SKSpriteNode)
        super.init(texture: nil, color: .clear, size:
        initialSize)

        createAnimations()
        // If we run an action with a key,
        "flapAnimation",
        // we can later reference that
    }
}
```

```
// key to remove the action.  
self.run(flyAnimation, forKey: "flapAnimation")  
}  
  
func createAnimations() {  
    let rotateUpAction =  
    SKAction.rotate(toAngle: 0, duration: 0.475)  
    rotateUpAction.timingMode = .easeOut  
    let rotateDownAction = SKAction.rotate(toAngle:  
-1,  
                                         duration: 0.8)  
    rotateDownAction.timingMode = .easeIn  
  
    // Create the flying animation:  
    let flyFrames: [SKTexture] = [  
        textureAtlas.textureNamed("pierre-flying-1"),  
        textureAtlas.textureNamed("pierre-flying-2"),  
        textureAtlas.textureNamed("pierre-flying-3"),  
        textureAtlas.textureNamed("pierre-flying-4"),  
        textureAtlas.textureNamed("pierre-flying-3"),  
        textureAtlas.textureNamed("pierre-flying-2")  
    ]  
    let flyAction = SKAction.animate(with: flyFrames,  
timePerFrame: 0.03)  
    // Group together the flying animation with  
rotation:  
    flyAnimation = SKAction.group([  
        SKAction.repeatForever(flyAction),  
        rotateUpAction  
    ])  
  
    // Create the soaring animation,  
    // just one frame for now:  
    let soarFrames: [SKTexture] =  
    [textureAtlas.textureNamed("pierre-flying-1")]  
    let soarAction = SKAction.animate(with:  
soarFrames,  
timePerFrame: 1)  
    // Group the soaring animation with the rotation  
down:  
    soarAnimation = SKAction.group([
```

```
SKAction.repeatForever(soarAction),  
rotateDownAction  
])  
}  
  
// Implement onTap to conform to the GameSprite  
protocol:  
func onTap() {}  
  
// Satisfy the NSCoder required init:  
required init?(coder aDecoder: NSCoder) {  
super.init(coder: aDecoder)  
}  
}
```

Great! Before we continue, we need to replace our original bee with an instance of the new Player class we just created. Follow these steps to replace the bee:

1. In `GameScene.swift`, near the top, remove the line that creates a bee constant in the `GameScene` class. Instead, we want to initiate an instance of `Player`. Add the new line `let player = Player()`.
2. Completely delete the `addTheFlyingBee` function.
3. Also, remove the `addBackground` function for now.
4. In `didMove`, remove the line that calls `addTheFlyingBee`.
5. In `didMove`, at the bottom, add the following new code to position and add the `player`:

```
// Position the player:  
player.position = CGPoint(x: 150, y: 250)  
// Add the player node to the scene:  
self.addChild(player)
```

6. Further down, in `didSimulatePhysics`, replace the reference to the bee with a reference to the `player`. The new line will read: `self.camera!.position = player.position`. Recall that we created the `didSimulatePhysics` function in *Chapter 2, Sprites, Camera, Actions!*, when we centered the camera on one node.

We have successfully transformed the original bee into a penguin. Before we move on, we will make sure that our `GameScene` class includes all of the changes we have made so far in this chapter. After that, we will begin to explore the physics system.

Renovating the GameScene class

We have made quite a few changes to our project. Luckily, this is the last major overhaul of the previous animation code. Moving forward, we will use the terrific structure we built in this chapter. At this point, your `GameScene.swift` file should look something like the following:

```
import SpriteKit

class GameScene: SKScene {
    let cam = SKCameraNode()
    let ground = Ground()
    let player = Player()

    override func didMove(to view: SKView) {
        self.anchorPoint = .zero
        self.backgroundColor = UIColor(red: 0.4, green: 0.6, blue: 0.95, alpha: 1.0)

        // Assign the camera to the scene
        self.camera = cam

        // Spawn our test bees:
        let bee2 = Bee()
        bee2.position = CGPoint(x: 325, y: 325)
        self.addChild(bee2)
        let bee3 = Bee()
        bee3.position = CGPoint(x: 200, y: 325)
        self.addChild(bee3)

        // Add the ground to the scene:
        ground.position = CGPoint(x: -self.size.width * 2, y: 150)
        ground.size = CGSize(width: self.size.width * 6,
                             height: 0)
        ground.createChildren()
        self.addChild(ground)

        // Add the player to the scene:
        player.position = CGPoint(x: 150, y: 250)
        self.addChild(player)
    }
}
```

```
override func didSimulatePhysics() {  
    // Keep the camera centered on the player  
    self.camera!.position = player.position  
}  
}
```

Run the project. You will see our new penguin hovering near the bees. Great work; we are now ready to explore the physics system with all of our new nodes. Your simulator should look something like this screenshot:



Physics bodies and gravity

SpriteKit simulates physics with physics bodies. We attach physics bodies to all the nodes that need physics computations. We will set up a quick example before exploring all of the details.

Dropping like flies

Our bees need to be part of the physics simulation, so we will add physics bodies to their nodes. Open your `Bee.swift` file and locate the `init` function. Add the following code at the bottom of the function:

```
// Attach a physics body, shaped like a circle  
// and sized roughly to our bee.  
self.physicsBody = SKPhysicsBody(circleOfRadius: size.width / 2)
```

It is that easy to add a node to the physics simulation. Run the project. You will see our two bee instances drop off the screen. They are now subject to gravity, which is on by default.

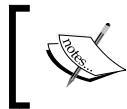
Solidifying the ground

We want the ground to catch falling game objects. We can give the ground its own physics body so that the physics simulation can stop the bees from falling through it. Open your `Ground.swift` file, locate the `createChildren` function, and add this code at the bottom of the function:

```
// Draw an edge physics body along the top of the ground node.  
// Note: physics body positions are relative to their nodes.  
// The top left of the node is X: 0, Y: 0, given our anchor point.  
// The top right of the node is X: size.width, Y: 0  
let pointTopLeft = CGPoint(x: 0, y: 0)  
let pointTopRight = CGPoint(x: size.width, y: 0)  
self.physicsBody = SKPhysicsBody(edgeFrom: pointTopLeft,  
to: pointTopRight)
```

Run the project. The bees will now quickly drop and then stop once they collide with the ground. After the bees have landed, your simulator will look like the following screenshot:





Checkpoint 3-A

The code up to this point is available in this chapter's code resources.

Exploring physics simulation mechanics

Let's take a closer look at the specifics of SpriteKit's physics system. For instance, why are the bees subject to gravity while the ground stays where it is? Though we attached physics bodies to both nodes, we actually used two different styles of physics bodies. There are three types of physics bodies, and each behaves slightly differently:

- Dynamic: Physics bodies have volume and are fully subject to forces and collisions in the system. We will use dynamic physics bodies for most parts of the game world: the player, enemies, power-ups, and others.
- Static: Physics bodies have volume but no velocity. The physics simulation does not move nodes with static bodies, but they can still collide with other game objects. We can use static bodies for walls or obstacles.
- Edge: Physics bodies have no volume and the physics simulation will never move them. They mark off the boundaries of movement; other physics bodies will never cross them. Edges can cross each other to create small containment areas.

Voluminous (dynamic and static) bodies have a variety of properties that influence how they move through space and react to collisions. This allows us to create a wide range of realistic physics effects. Each property controls one aspect of a body's physical characteristics:

- Restitution: Determines how much energy is lost when one body bounces into another. This changes the body's bounciness. SpriteKit measures restitution on a scale from 0.0 to 1.0. The default value is 0.2.
- Friction: Describes the amount of force necessary to slide one body against another body. This property also uses a scale of 0.0 to 1.0, with a default value of 0.2.
- Damping: Determines how quickly a body slows as it moves through space. You can think of damping as air friction. Linear damping determines how quickly a body loses speed, while angular damping affects rotation. Both are measured from 0.0 to 1.0, with a default value of 0.1.

- Mass: This is measured in kilograms. It describes how far colliding objects push the body and factors in momentum during movement. Bodies with more mass will move less when hit by another body and will push other bodies further when they collide with them. The physics engine automatically uses the mass and the area of the body to determine density. Alternatively, you can set the density and let the physics engine calculate mass. It is usually more intuitive to set the mass.

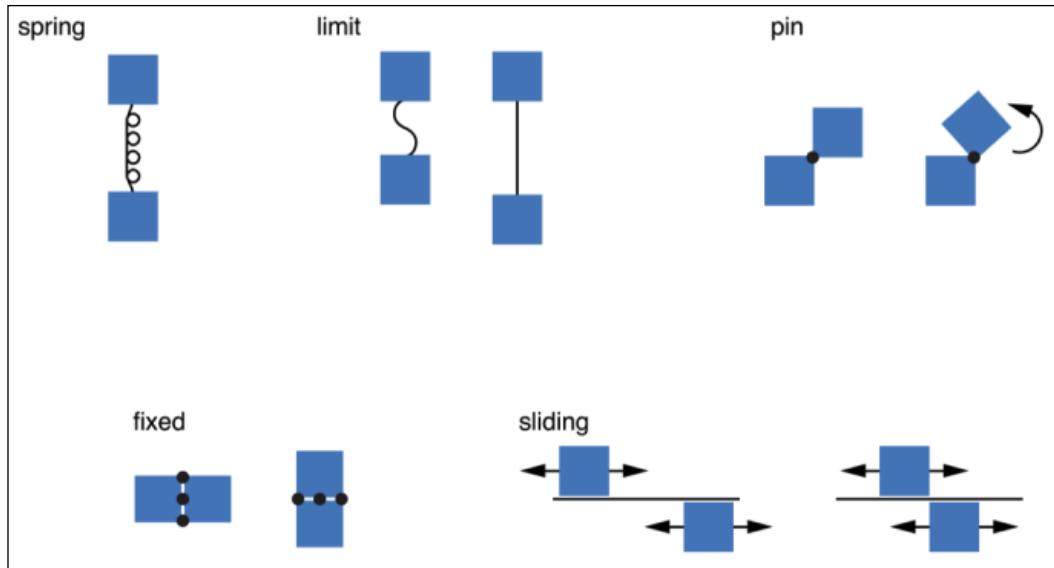
There are other physics-related classes as well, such as `PhysicsWorld`, `SKFieldNode`, and `SKPhysicsJoint`.

Physics World: You don't have to create an object of the class when you create a scene, as Physics World is already part of the scene when it is created. Physics World is responsible for the simulation of the physics objects in the scene. So, in a scene in which there are physics objects, it updates the position and collision information for all the physics objects in the scene. You can add properties such as gravity to the physics world in the scene and, depending on what you set the gravity in the scene to, the objects will behave accordingly.

Physics Field Nodes: Field Nodes can apply force on objects when they enter a certain area or field. This is similar to planets in the *Angry Birds* game, which start pulling the birds toward them when the birds go near a planet. That is one example; there are different types of `FieldNodes` that can be created and attached to scenes.

Physics Joints: Using physics joints, we can join objects together and set conditions to make them behave with each other in a certain way. There are different joint types, such as spring, limit, pin, fixed, and sliding:

- Spring Joint: The connection between the bodies is such that there is resistance between the bodies, meaning that it will take some effort to pull or push the bodies apart or together. If you pull or push the bodies and then release, they will reset to their original positions.
- Limit Joint: In this case, there is no resistance, but you can pull the bodies apart only so much, which is controlled by a limiting factor.
- Pin Joint: You can pin a body to another body so that you can only rotate the object around the other object and will not be able to pull the bodies apart.
- Fixed Joint: You fix the bodies together so that they behave like a single object.
- Sliding Joint: This only allows the movement of one body around the other object:



More about **Joint Physics** can be obtained from here: <https://developer.apple.com/documentation/spritekit/skphysicsjoint>.

All right; enough with the textbook! Let's solidify our learning with some examples.

Firstly, we want gravity to leave our bees alone. We will set their flight paths manually. We need the bees to be dynamic physics bodies in order to interact properly with other nodes, but we need these bodies to ignore gravity. For such instances, SpriteKit provides a property named `affectedByGravity`. Open `Bee.swift` and, at the bottom of the `init` function, add this code:

```
self.physicsBody?.affectedByGravity = false
```

The question mark after `physicsBody` is optional chaining. We need to unwrap `physicsBody`, since it is optional. If `physicsBody` is `nil`, the entire statement will return `nil` (instead of triggering an error). You can think of it as gracefully unwrapping an optional property with an inline statement.

Run the project. The bees should now hover in place, as they did before we added their bodies, however, SpriteKit's physics simulation now affects them; they will react to impulses and collisions. Great—let's purposefully make the bees collide.

Bumping bees into bees

You may have noticed that we positioned bee2 and bee3 at the same height in the game world. We only need to push one of them horizontally to create a collision—perfect crash test dummies! We can use an impulse to create velocity for the outside bee.

Locate the `didMove` function in `GameScene.swift`. At the bottom, after all of our spawn code, add this line:

```
bee2.physicsBody?.applyImpulse(CGVector(dx: -3, dy: 0))
```

Run the project. You will see the outermost bee fly toward the middle and crash into the inner bee. This pushes the inner bee to the left and slows the first bee from the contact.

Attempt the same experiment with a variable: increased mass. Before the impulse line, add this code to adjust the mass of bee2:

```
bee2.physicsBody?.mass = 0.2
```

Run the project. Hmm—our heavier bee does not move very far with the same impulse (it is a 200-gram bee, after all). It eventually bumps into the inner bee, but it is not a very exciting collision. We will need to crank up the impulse to propel our beefier bee. Change the impulse line to use a `dx` value of -25:

```
bee2.physicsBody?.applyImpulse(CGVector(dx: -25, dy: 0))
```

Run the project again. This time, our impulse provides enough energy to move the heavy bee in an interesting way. Notice how much energy the heavy bee transfers to the normal bee when they collide. Both bees possess enough momentum to eventually slide completely off the screen. Your simulator should look something like this screenshot, just before the bees slide off the screen:



Before you move on, you may wish to experiment with the various physics properties that I outlined earlier in this chapter. You can create many collision variations; the physics simulation offers a lot of depth without much effort.

Movement with impulses and forces

You have several options for moving nodes with physics bodies:

- An impulse is an immediate, one-time change to a physics body's velocity. In our test, an impulse gave the bee its velocity and it slowly bled speed to damping and its collision. Impulses are perfect for projectiles, such as missiles, bullets, disgruntled birds, and so on.
- A force applies velocity for only one physics calculation cycle. When we use a force, we typically apply it before every frame. Forces are useful for rocket ships, cars, or anything else that is continually self-propelled.
- You can also edit the `velocity` and `angularVelocity` properties of a body directly. This is useful for setting a manual velocity limit.



Checkpoint 3-B

The code up to this point is available in the chapter's code resources.



Summary

We have made great strides in this chapter. Our new class organization will serve us well over the course of this book. We learned how to use protocols to enforce commonality across classes, encapsulated our game objects into distinct classes, and explored tiling textures over the width of the ground node. Finally, we cleaned out some of our previous learning code from `GameScene` and used the new class system to spawn all of our game objects.

We also applied the physics simulation to our game. We have only scratched the surface of the powerful physics system in SpriteKit—we will dive deeper into custom collision events in *Chapter 7, Implementing Collision Events*—but we have already gained quite a bit of functionality. We explored the three types of physics bodies and studied the various physics properties you can use to fine-tune the physical behavior of your game objects. Then, we put all of our hard work into practice by bumping our bees together and watching the results.

Next, we will try several control schemes and move our player around the game world. This is an exciting addition; our project will begin to feel like a true game in *Chapter 4, Adding Controls*.

4

Adding Controls

Players control mobile games through a very limited number of interactions. Often, games feature only a single mechanic: tap anywhere on the screen to jump or fly. Contrast that with a console controller with dozens of button combinations. With so few actions, keeping users engaged with polished, fun controls is vital to the success of your game.

In this chapter, you will learn how to implement several popular control schemes that have emerged from the App Store. First, we will experiment with tilt controls; the physical orientation of the device will determine where the player flies. Then, we will wire up the `onTap` events on our sprite nodes. Finally, we will implement and polish a simple control scheme for flying in our game: tap anywhere on the screen to fly higher. You can combine these techniques to create unique and enjoyable controls in your future games.

The topics in this chapter include the following:

- Retrofitting the Player class for flight
- Polling for device movement with Core Motion
- Wiring up the sprite on Tap events
- Teaching our penguin to fly
- Improving the camera
- Looping the ground as the player moves forward

Retrofitting the Player class for flight

We need to perform a few quick setup tasks before we can react to player input. We will remove some of our older testing code and add a physics body to the Player class.

The Beekeeper

First, clean up the old bee physics tests from the last chapter. Open GameScene.swift, find didMove, and locate the bottom two lines; one sets a mass for bee2, and the other applies an impulse to bee2. Remove these two lines:

```
bee2.physicsBody?.mass = 0.2  
bee2.physicsBody?.applyImpulse(CGVector(dx: -25, dy: 0))
```

Updating the Player class

We need to give the Player class its own update function. We want to store player-related logic in Player and we need it to run before every frame:

1. Open Player.swift and add the following function inside Player:

```
func update() { }
```
2. In GameScene.swift, add this code at the bottom of the GameScene class:

```
override func update(_ currentTime: TimeInterval) {  
    player.update()  
}
```
3. Perfect. The GameScene class will call the Player class's update function on every update.

Moving the ground

We initially placed the ground higher than necessary to make sure it displayed for all screen sizes in the previous chapter. We can now move the ground into its lower, final position since the player will soon move up and down, bringing the ground into view.

In GameScene.swift, locate the line that sets the ground.position value and change the y value from 150 to 30:

```
ground.position = CGPoint(x: -self.size.width * 2, y: 30)
```

Assigning a physics body to the player

We will use physics forces to move our player around the screen. To apply these forces, we must first add a physics body to the player sprite.

Creating a physics body shape from a texture

When gameplay allows, you should use circles to define your physics bodies — circles are the most efficient shape for the physics simulation and result in the highest frame rate. However, the accuracy of Pierre's shape is very important to our gameplay, and a circle is not a great fit for his shape. Instead, we will assign a special type of physics body, based on his texture.

Apple introduced the ability to define the shape of a physics body with opaque texture pixels in Xcode 6. This is a convenient addition as it allows us to create extremely accurate shapes for our sprites. There is a performance penalty, however, it is computationally expensive to use these texture-driven physics bodies. You will want to use them sparingly, only on your most important sprites.

To create Pierre's physics body, add this code to `Player.swift`, at the bottom of the `init` function:

```
// Create a physics body based on one frame of Pierre's animation.  
// We will use the third frame, when his wings are tucked in  
let bodyTexture = textureAtlas.textureNamed("pierre-flying-3")  
self.physicsBody = SKPhysicsBody(  
    texture: bodyTexture, size: self.size)  
// Pierre will lose momentum quickly with a high linearDamping:  
self.physicsBody?.linearDamping = 0.9  
// Adult penguins weigh around 30kg:  
self.physicsBody?.mass = 30  
// Prevent Pierre from rotating:  
self.physicsBody?.allowsRotation = false
```

Run the project and the ground will appear to rise up to Pierre. Since we have given him a physics body, he is now subject to gravity. Pierre is actually dropping down the grid and the camera is adjusting to keep him centered. This is fine for now; we will give him the tools to fly into the sky later. Next, let's learn how to move a character, based on the tilt of the physical device.

Polling for device movement with Core Motion

Apple provides the Core Motion framework to expose precise information on the iOS device's orientation in physical space. We can use this data to move our player on the screen when the user tilts their device in the direction they want to move. This unique style of input offers new gameplay mechanics in mobile games.

You will need a physical iOS device for this Core Motion section. The iOS simulator in Xcode does not simulate device movement. However, this section is only a learning exercise and is not required to finish the game we are building. Our final game will not use Core Motion. Feel free to skip the Core Motion section if you cannot test with a physical device.

Implementing the Core Motion code

It is very easy to poll for device orientation. We will check the device's position during every update and apply the appropriate force to our player. Follow these steps to implement the Core Motion controls:

1. In `GameScene.swift`, near the very top, add a new `import` statement after the `import SpriteKit` line:

```
import CoreMotion
```

2. Inside the `GameScene` class, add a new constant named `motionManager` and instantiate an instance of `CMMotionManager`:

```
let motionManager = CMMotionManager()
```

3. Inside the `GameScene` class's `didMove` function, add the following code at the bottom. This lets Core Motion know that we want to poll the orientation data, so it needs to start reporting data:

```
self.motionManager.startAccelerometerUpdates()
```

4. Finally, add the following code to the bottom of the `update` function to poll the orientation, build an appropriate vector, and apply a physical force to the player's character:

```
// Unwrap the accelerometer data optional:  
if let accelData = self.motionManager.accelerometerData {  
var forceAmount: CGFloat  
var movement = CGVector()  
  
// Based on the device orientation, the tilt number  
// can indicate opposite user desires. The  
// UIApplication class exposes an enum that allows
```

```
// us to pull the current orientation.  
// We will use this opportunity to explore Swift's  
// switch syntax and assign the correct force for the  
// current orientation:  
switch  
    UIApplication.shared.statusBarOrientation {  
        case .landscapeLeft:  
            // The 20,000 number is an amount that felt right  
            // for our example, given Pierre's 30kg mass:  
            forceAmount = 20000  
        case .landscapeRight:  
            forceAmount = -20000  
        default:  
            forceAmount = 0  
    }  
  
    // If the device is tilted more than 15% towards  
    // vertical, then we want to move the Penguin:  
    if accelData.acceleration.y > 0.15 {  
        movement.dx = forceAmount  
    }  
    // Core Motion values are relative to portrait view.  
    // Since we are in landscape, use y-values for x-axis.  
    else if accelData.acceleration.y < -0.15 {  
        movement.dx = -forceAmount  
    }  
  
    // Apply the force we created to the player:  
    player.physicsBody?.applyForce(movement)  
}
```

Run the project. You can slide Pierre across the ice by tilting your device in the direction you want to move. Great work—we have successfully implemented our first control system.



Notice that Pierre falls through the ground when you move him too far in any direction. Later in this chapter, we will improve the ground, continuously repositioning it to cover the area beneath the player.

This is a simple example of using Core Motion data for player movement; we are not going to use this method in our final game. Still, you can extrapolate this example into advanced control schemes in your own games.



Checkpoint 4-A

The code up to this point is available in this chapter's code resources.



Wiring up the sprite onTap events

Your games will often require the ability to run code when the player taps a specific sprite. I like to implement a system that includes all the sprites in your game so that you can add tap events to each sprite without building any additional structure. We have already implemented `onTap` methods in all of our classes that adopt the `GameSprite` protocol; we still need to wire up the scene to call these methods when the player taps the sprites.



Before we move on, we need to remove the Core Motion code, since we will not be using it in the finished game. Once you finish exploring the Core Motion example, please remove it from the game by following the previous section's bullet points in reverse.



Implementing touchesBegan in the GameScene

SpriteKit calls our scene's `touchesBegan` function every time the screen is touched. We will read the location of the touch and determine the sprite node in that position. We can check whether the touched node adopts our `GameSprite` protocol. If it does, this means it must have an `onTap` function, which we can then invoke. Add the `touchesBegan` function before the `GameScene` class. I like to place it just after the `didSimulatePhysics` function:

```
override func touchesBegan(_ touches: Set<UITouch>,
    with event: UIEvent?) {
    for touch in (touches) {
        // Find the location of the touch:
        let location = touch.location(in: self)
        // Locate the node at this location:
        let nodeTouched = atPoint(location)
        // Attempt to downcast the node to the GameSprite protocol
        if let gameSprite = nodeTouched as? GameSprite {
```

```
// If this node adheres to GameSprite, call onTap:  
gameSprite.onTap()  
    }  
}  
}
```

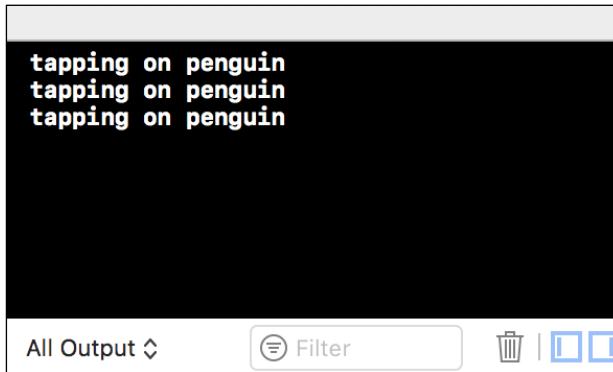
That is all we need to do to wire up all of the `onTap` functions we have implemented on the game object classes we have made. Of course, all of these `onTap` functions are empty at the moment; we will now add some functionality to illustrate this effect.

Larger than life

Open your `Player.swift` file and locate the `onTap` function. Temporarily, we will expand the penguin to giant size when tapped to demonstrate that we have wired our `onTap` functions correctly. Add this code inside the penguin's `onTap` function:

```
print(" tapping on penguin")
```

Run the project and tap on the penguin. On the debug console, you will see the text getting printed out each time you tap on the penguin:



This example shows that our `onTap` functions work. You can remove the code you added to the `Player` class in the `onTap` function. We will keep the `onTap` wire-up code in `GameScene` so that we can use tap events later.

Teaching our penguin to fly

Let's implement the control scheme for our penguin. The player can tap anywhere on the screen to make Pierre fly higher and release to let him fall. We are going to make quite a few changes--if you need help, refer to the checkpoint at the end of this chapter. Start by modifying the `Player` class; follow these steps to prepare our `Player` for flight:

1. In `Player.swift`, add some new properties directly to the `Player` class:

```
// Store whether we are flapping our wings or in free-fall:  
var flapping = false  
    // Set a maximum upward force.  
    // 57,000 feels good to me, adjust to taste:  
    let maxFlappingForce: CGFloat = 57000  
    // Pierre should slow down when he flies too high:  
    let maxHeight: CGFloat = 1000
```

2. So far, Pierre has been flapping his wings by default. Instead, we want to display the soaring animation by default and only run the flap animation when the user presses the screen. In the `init` function, remove the line that runs `flyAnimation` and, instead, run `soarAnimation`:
`self.run(soarAnimation, withKey: "soarAnimation")`
3. When the player touches the screen, we apply the upward force in the `Player` class's `update` function. Remember that `GameScene` calls the `Player` `update` function once per frame. Add this code in `update`:

```
// If flapping, apply a new force to push Pierre higher.  
if self.flapping {  
    var forceToApply = maxFlappingForce  
  
        // Apply less force if Pierre is above position 600  
        if position.y > 600 {  
            // The higher Pierre goes, the more force we  
            // remove. These next three lines determine the  
            // force to subtract:  
            let percentageOfMaxHeight = position.y / maxHeight  
            let flappingForceSubtraction =  
                percentageOfMaxHeight * maxFlappingForce  
            forceToApply -= flappingForceSubtraction  
        }  
        // Apply the final force:  
        self.physicsBody?.applyForce(CGVector(dx: 0, dy:  
            forceToApply))
```

```
    }

    // Limit Pierre's top speed as he climbs the y-axis.
    // This prevents him from gaining enough momentum to shoot
    // over our max height. We bend the physics for game play:
    if self.physicsBody!.velocity.dy > 300 {
        self.physicsBody!.velocity.dy = 300
    }
```

4. Finally, we will provide two functions on `Player` to allow other classes to start and stop the flapping behavior. The `GameScene` class will call these functions when it detects touch input. Add the following functions to the `Player` class:

```
// Begin the flap animation, set flapping to true:
func startFlapping() {
    self.removeAction(forKey: "soarAnimation")
    self.run(flyAnimation, forKey: "flapAnimation")
    self.flapping = true
}

// Stop the flap animation, set flapping to false:
func stopFlapping() {
    self.removeAction(forKey: "flapAnimation")
    self.run(soarAnimation, forKey: "soarAnimation")
    self.flapping = false
}
```

Perfect, our `Player` is ready for flight. Now, we will simply invoke the start and stop functions from the `GameScene` class.

Listening for touches in GameScene

The `SKScene` class (that `GameScene` inherits from) includes handy functions that we can use to monitor touch input. Follow these steps to wire up the `GameScene` class:

1. In `GameScene.swift`, in the `touchesBegan` function, add this code at the very bottom to start the `Player` flapping when the user touches the screen:
`player.startFlapping()`

2. After `touchesBegan`, create two new functions in the `GameScene` class. These functions stop the flapping when the user lifts his or her finger from the screen, or when an iOS notification interrupts the touch:

```
override func touchesEnded(_ touches: Set<UITouch>,
                           with event: UIEvent?) {
    player.stopFlapping()
}

override func touchesCancelled(_ touches: Set<UITouch>,
                               with event: UIEvent?) {
    player.stopFlapping()
}
```

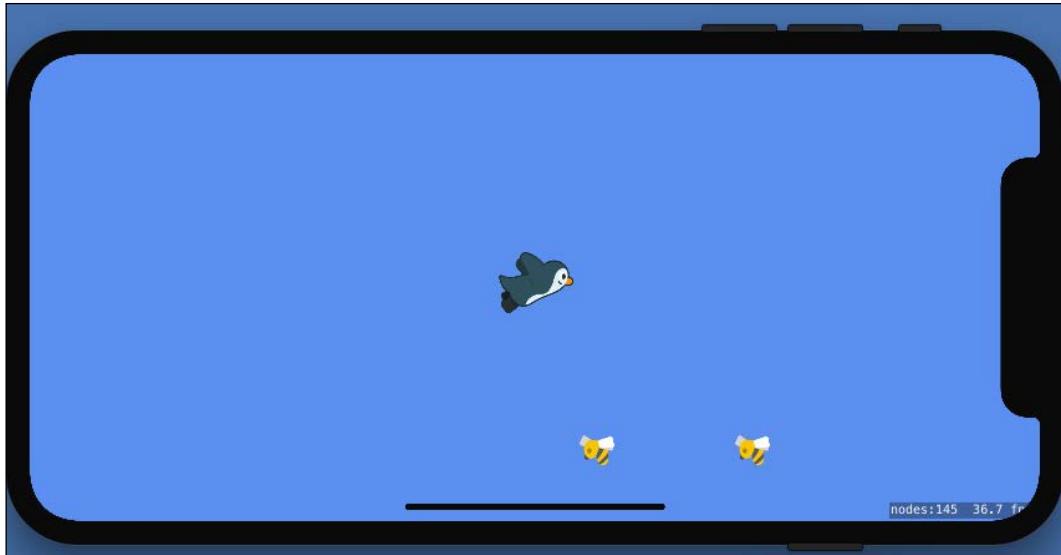
Fine-tuning gravity

Before we test out our new flying code, we need to make one adjustment. The default gravity setting of 9.8 feels too real. Pierre lives in a cartoon world; real-world gravity is a bit of a drag. We can adjust gravity in the `GameScene` class; add this line at the bottom of the `didMove` function:

```
// Set gravity
self.physicsWorld.gravity = CGVector(dx: 0, dy: -5)
```

Spreading your wings

Run the project. Tap the screen to make Pierre fly higher; release to let him fall. Play with the action; Pierre rotates toward his vector and builds or loses momentum as you tap and release. Terrific! You have successfully implemented the core mechanic of our game. Take a minute to enjoy flying up and down, as in this screenshot:



Improving the camera

Our camera code works well; it follows the player wherever they fly. However, we can improve the camera to enhance the flying experience. In this section, we will add two new features:

- Zoom the camera out as Pierre Penguin flies higher, reinforcing the feeling of increasing height.
- Suspend vertical centering when the player drops below the halfway point of the screen. This means the ground never fills too much of the screen and adds the feeling of cutting upwards into the air when Pierre flies higher and the camera starts tracking him again.

Follow these steps to implement these two improvements:

1. In `GameScene.swift`, create a new variable in the `GameScene` class to store the center point of the screen:

```
var screenCenterY:CGFloat = 0
```

2. In the `didMove` function, set this new variable with the calculated center of the screen's height:

```
// Store the vertical center of the screen:  
screenCenterY = self.size.height / 2
```

3. We need to rework the `didSimulatePhysics` function significantly. Remove the existing `didSimulatePhysics` function and replace it with this code:

```
override func didSimulatePhysics() {
    // Keep the camera locked at mid screen by default:
    var cameraYPos = screenCenterY
    cam.yScale = 1
    cam.xScale = 1

    // Follow the player up if higher than half the
    screen:
    if (player.position.y > screenCenterY) {
        cameraYPos = player.position.y
        // Scale out the camera as they go higher:
        let percentOfMaxHeight = (player.position.y -
            screenCenterY) / (player.maxHeight -
            screenCenterY)
        let newScale = 1 + percentOfMaxHeight
        cam.yScale = newScale
        cam.xScale = newScale
    }

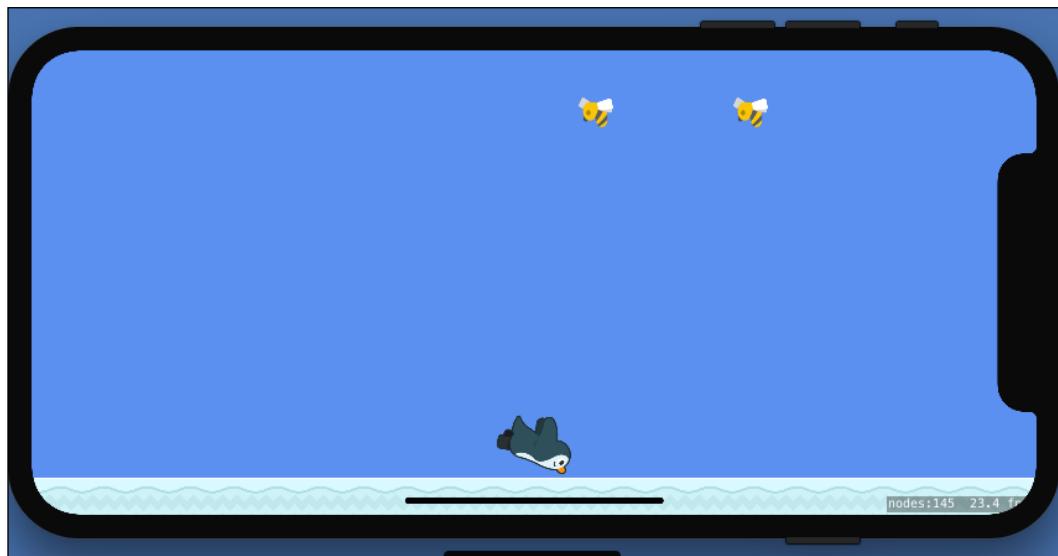
    // Move the camera for our adjustment:
    self.camera!.position = CGPoint(x: player.position.x,
                                    y: cameraYPos)
}
```

Run the project and then fly up. The world scales smaller as you gain height. The camera also now allows Pierre to dive below the center of the screen when you fly close to the ground. The following screenshot illustrates the two extremes.

Notice the smaller sprites here; Pierre flies higher and the camera zooms out:



In this screenshot, the camera stops following Pierre vertically as he approaches the ground:



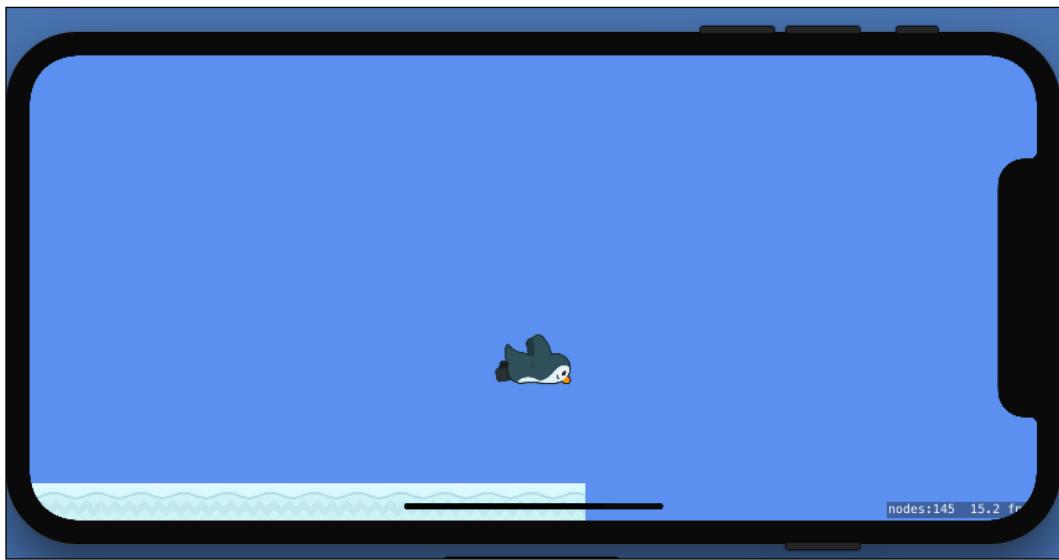
The combined effect adds a lot of polish to the game and increases the fun of flying. Our flying mechanic feels great. The next step is to move Pierre forward through the world.

Pushing Pierre forward

This style of game usually moves the world forward at a constant speed. Rather than applying force or impulse, we can manually set a constant velocity for Pierre during every update. Open the `Player.swift` file and add this code at the bottom of the `update` function:

```
// Set a constant velocity to the right:  
self.physicsBody?.velocity.dx = 200
```

Run the project. Our protagonist penguin will move forward, past the swarm of bees and through the world. This works well, but you will quickly notice that the ground runs out as Pierre moves forward, as shown in this screenshot:



Recall that our ground is only as wide as the screen width multiplied by six. Rather than extending the ground further, we will move the ground's position at well-timed intervals. Since the ground is made up of repeating tiles, there are many opportunities to jump its position forward seamlessly. We simply need to figure out when the player has travelled the correct distance.

Tracking the player's progress

First, we need to keep track of how far the player has flown. We will use this later as well, for keeping track of a high score. This is easy to implement. Follow these steps to track how far the player has flown:

1. In the `GameScene.swift` file, add two new properties to the `GameScene` class:

```
let initialPlayerPosition = CGPoint(x: 150, y: 250)
var playerProgress = CGFloat()
```

2. In the `didMove` function, update the line that positions the player to use the new `initialPlayerPosition` constant instead of the old hardcoded value:

```
// Add the player to the scene:
player.position = initialPlayerPosition
```

3. In the `didSimulatePhysics` function, update the new `playerProgress` property with the player's new distance:

```
// Keep track of how far the player has flown
playerProgress = player.position.x -
    initialPlayerPosition.x
```

Perfect! We now have access to the player's progress at all times in the `GameScene` class. We can use the distance traveled to reposition the ground at the correct time.

Looping the ground as player moves forward

There are many possible methods to create an endless ground loop. We will implement a straightforward solution that jumps the ground forward after the player has travelled over roughly one third of its width. This method guarantees that the ground always covers the screen, given that our player starts in the middle third.

We will create the jump logic on the `Ground` class. Follow these steps to implement endless ground:

1. Open the `Ground.swift` file and add two new properties to the `Ground` class:

```
var jumpWidth = CGFloat()
    // Note the instantiation value of 1 here:
var jumpCount = CGFloat(1)
```

2. In the `createChildren` function, we find the total width from one third of the children tiles and make it our `jumpWidth`. We will need to jump the ground forward every time the player travels this distance. You only need to add one line at the very bottom of the `createChildren` function:

```
// Save the width of one-third of the children nodes  
jumpWidth = tileSize.width * floor(tileCount / 3)
```

3. Add a new function named `checkForReposition` to the `Ground` class, after the `createChildren` function. The scene will call this function before every frame to check whether we should jump the ground forward:

```
func checkForReposition(playerProgress: CGFloat) {  
    // The ground needs to jump forward  
    // every time the player has moved this distance:  
    let groundJumpPosition = jumpWidth * jumpCount  
  
    if playerProgress >= groundJumpPosition {  
        // The player has moved past the jump position!  
        // Move the ground forward:  
        self.position.x += jumpWidth  
        // Add one to the jump count:  
        jumpCount += 1  
    }  
}
```

4. Open `GameScene.swift` and add this line at the bottom of the `didSimulatePhysics` function to call the `Ground` class's new logic:

```
// Check to see if the ground should jump forward:  
ground.checkForReposition(playerProgress: playerProgress)
```

Run the project. The ground will seem to stretch on forever as Pierre flies forward. This looping ground is a big step toward the final game world. It may seem like a lot of work for a simple effect, but the looping ground is important, and our method will perform well on any screen size. Great work!



Checkpoint 4-B

The code up to this point is available in this chapter's code resources.

Summary

In this chapter, we have transformed a tech demo into the beginnings of a real game. We have added a great deal of new code. You learned how to implement three distinct mobile game control methods: physical device motion, sprite tap events, and flying higher when the screen is touched. We polished the flying mechanic for maximum fun and sent Pierre flying forward through the world.

You also learned how to implement two common mobile game requirements: looping the ground and a smarter camera system. Both of these features make a big impact on our game.

Next, we will add more content to our level. Flying is already fun, but traveling past the first few bees feels a little lonely. We will give Pierre Penguin some company in *Chapter 5, Spawning Enemies, Coins, and Power-ups*.

5

Spawning Enemies, Coins, and Power-Ups

One of the most enjoyable and creative aspects of game development is building the game world for your players to explore. Our young project is starting to resemble a playable game after adding the controls; the next step is to build more content. We will create additional classes for new enemies, collectible coins, and special power-ups that give Pierre Penguin a boost as he navigates the perils of our world. We can then develop a system to spawn increasingly difficult patterns of these game objects as the player advances.

The topics in this chapter include the following:

- Adding the Power-up Star
- A new enemy – the Mad Fly
- Another terror – Bats!
- Guarding the ground with the Blade
- Adding coins
- Testing the new game objects

Introducing the cast

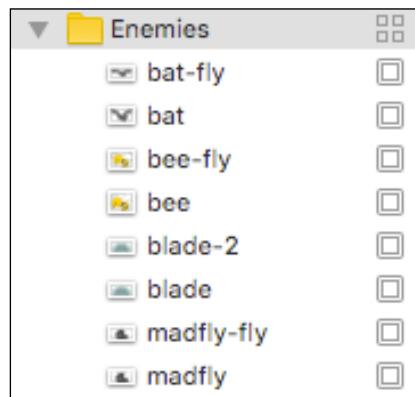
Strap on your hard hat, as we are going to be writing a lot of code in this chapter. Stick with it! The results are well worth the effort. Meet the new cast of characters we will be introducing in this chapter:



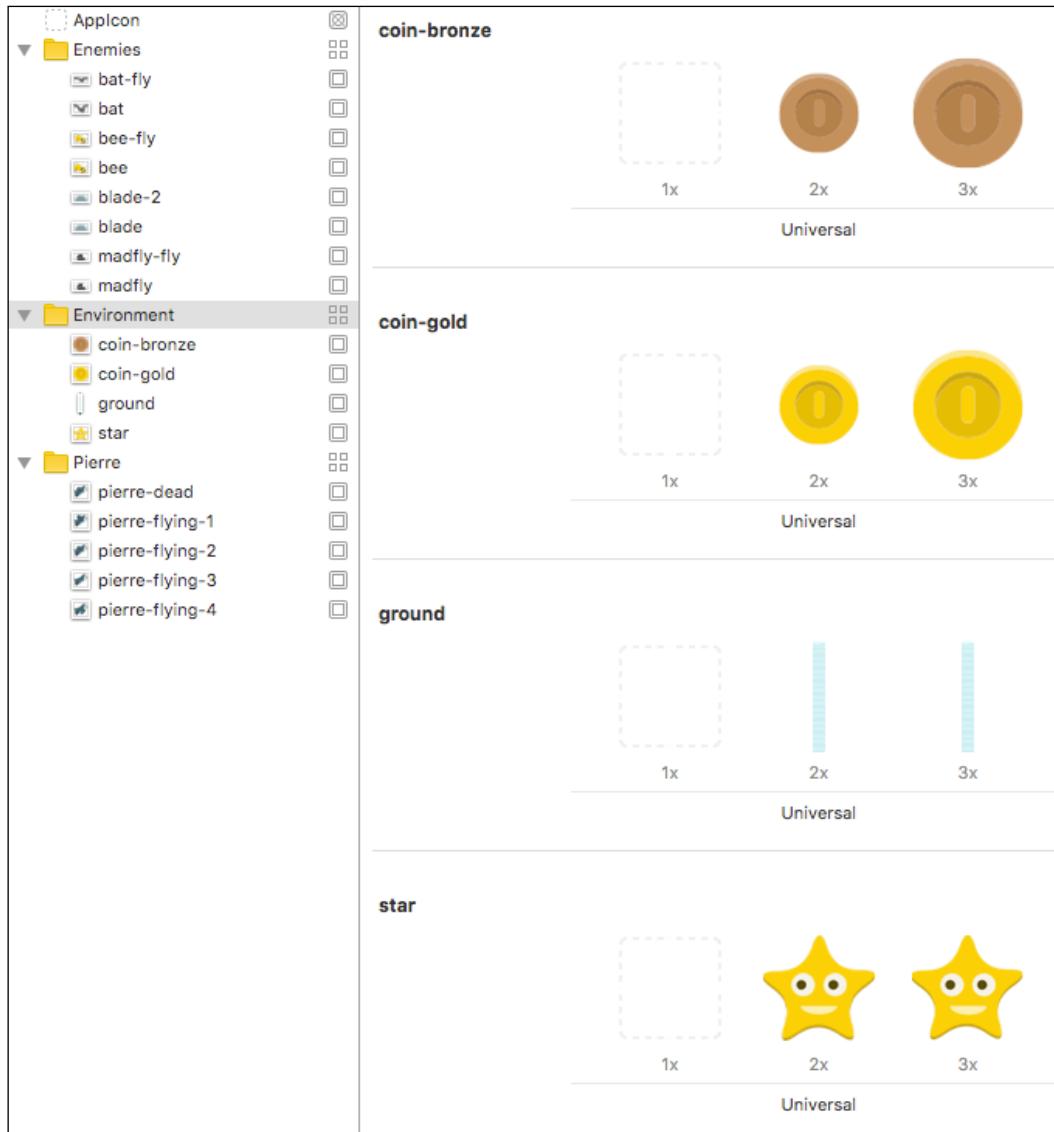
Locating and adding the art assets

Follow these steps to add these new art assets to the texture atlases in our `Assets.xcassets` file:

1. In Xcode, open the `Assets.xcassets` file and locate the texture atlases you have created. You should already have folders for `Enemies`, `Environment`, and `Pierre`.
2. Locate the `Enemies` folder in the downloadable asset bundle. You should see art for all of the enemies, including the Bat, the Blade, the Mad Fly, and the Bee.
3. We can skip the Bee art since we already added it to our project. Excluding the Bee, drag the rest of the asset files into the `Enemies` texture atlas in Xcode. You should be dragging 12 files into Xcode (two animation images per enemy, each with two resolutions). When you are done, your `Enemies` texture atlas should look like this:



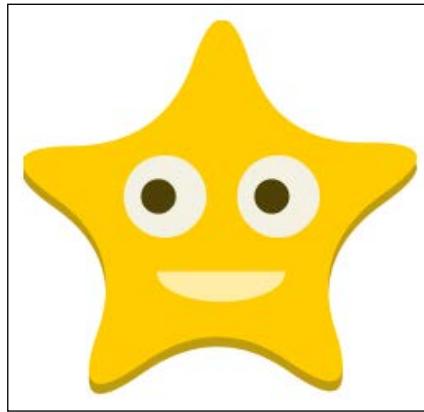
- Great work! Now, we just need to add the art for the two coins and the Power-up Star. Locate the Environment folder in the downloadable asset bundle and find the asset files for the Bronze Coin, the Gold Coin, and the Star. Just as before, drag the art for these three sprites into the Environment texture atlas in Xcode. You should be dragging six files into Xcode. You will end up with all of our power-up art in the Environment folder. Your Assets.xcassets file should look like this when you are done:



Terrific! We now have all of the art we need to create our enemies and power-ups. Next, we will add the Star into the game.

Adding the Power-up Star

Many of my favorite games grant temporary invulnerability when the player picks up a Star. We will add a hyperactive Power-up Star to our game. Meet our Star:



Adding the Star class

Now that the art is in place, you can create a new Swift file named `Star.swift` in your project; we will continue to organize classes into distinct files. The `Star` class will be similar to the `bee` class we created earlier. It will inherit from `SKSpriteNode` and adhere to our `GameSprite` protocol. The Star will add a lot of power to the player, so we will also give it a special zany animation based on `SKAction` to make it stand out.

To create the `Star` class, add the following code in your `Star.swift` file:

```
import SpriteKit

class Star: SKSpriteNode, GameSprite {
    var initialSize = CGSize(width: 40, height: 38)
    var textureAtlas: SKTextureAtlas =
        SKTextureAtlas(named: "Environment")
    var pulseAnimation = SKAction()

    init() {
```

```
let starTexture =
    textureAtlas.textureNamed("star")
super.init(texture: starTexture, color: .clear,
           size: initialSize)
// Assign a physics body:
self.physicsBody = SKPhysicsBody(circleOfRadius:
size.width / 2)
    self.physicsBody?.affectedByGravity = false
// Create our star animation and start it:
createAnimations()
self.run(pulseAnimation)
}

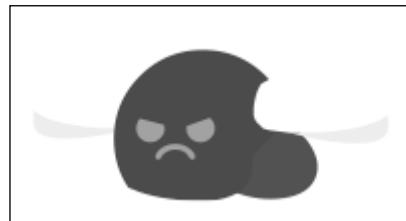
func createAnimations() {
    // Scale the star smaller and fade it slightly:
    let pulseOutGroup = SKAction.group([
SKAction.fadeAlpha(to: 0.85, duration: 0.8),
SKAction.scale(to: 0.6, duration: 0.8),
SKAction.rotate(byAngle: -0.3, duration: 0.8)
    ])
    // Push the star big again, and fade it back in:
    let pulseInGroup = SKAction.group([
SKAction.fadeAlpha(to: 1, duration: 1.5),
SKAction.scale(to: 1, duration: 1.5),
SKAction.rotate(byAngle: 3.5, duration: 1.5)
    ])
    // Combine the two into a sequence:
    let pulseSequence = SKAction.sequence([pulseOutGroup,
pulseInGroup])
pulseAnimation =
SKAction.repeatForever(pulseSequence)
}

// Required to conform to protocols:
func onTap() {}
required init?(coder aDecoder: NSCoder) {
super.init(coder: aDecoder)
}
}
```

Great! You should be familiar with most of this code at this point, since it is so similar to some of the other classes we have made. Let's continue by adding another new character: a grumpy fly.

A new enemy - the Mad Fly

Pierre Penguin will need to dodge more than just Bees to accomplish his goal. We will add a few new enemies in this chapter, starting with the `MadFly` class. The Mad Fly is quite grumpy, as you can see:



Adding the `MadFly` class

`MadFly` is another straightforward class; it looks a lot like the `bee` code. Create a new Swift file named `MadFly.swift` and enter this code:

```
import SpriteKit

class MadFly: SKSpriteNode, GameSprite {
    var initialSize = CGSize(width: 61, height: 29)
    var textureAtlas: SKTextureAtlas =
        SKTextureAtlas(named: "Enemies")
    var flyAnimation = SKAction()

    init() {
        super.init(texture: nil, color: .clear,
                   size: initialSize)
        self.physicsBody = SKPhysicsBody(circleOfRadius:
            size.width / 2)
        self.physicsBody?.affectedByGravity = false
        createAnimations()
        self.run(flyAnimation)
    }

    func createAnimations() {
        let flyFrames:[SKTexture] = [
            textureAtlas.textureNamed("madfly"),
            textureAtlas.textureNamed("madfly-fly")
        ]
    }
}
```

```
let flyAction = SKAction.animate(with: flyFrames,  
timePerFrame: 0.14)  
flyAnimation = SKAction.repeatForever(flyAction)  
}  
  
func onTap() {}  
required init?(coder aDecoder: NSCoder) {  
super.init(coder: aDecoder)  
}  
}
```

Congratulations, you have successfully implemented the Mad Fly. No time to celebrate—onward to the Bats!

Another terror - Bats!

We are getting into quite a rhythm with creating new classes. Now, we will add a Bat to swarm with the Bees. The Bat is small, but has a very sharp fang:



Adding the Bat class

To add the Bat class, create a file named `Bat.swift` and add this code:

```
import SpriteKit  
  
class Bat: SKSpriteNode, GameSprite {  
var initialSize = CGSize(width: 44, height: 24)  
var textureAtlas: SKTextureAtlas =  
SKTextureAtlas(named: "Enemies")  
Var flyAnimation = SKAction()  
  
init() {  
super.init(texture: nil, color: .clear,  
size: initialSize)  
self.physicsBody = SKPhysicsBody(circleOfRadius:  
size.width / 2)
```

```
        self.physicsBody?.affectedByGravity = false
    createAnimations()
    self.run(flyAnimation)
}

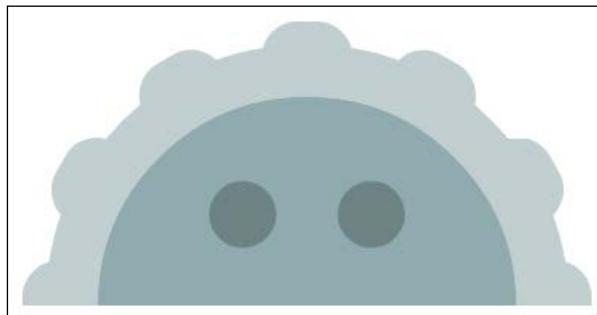
func createAnimations() {
    let flyFrames: [SKTexture] = [
textureAtlas.textureNamed("bat"),
textureAtlas.textureNamed("bat-fly")
]
    let flyAction = SKAction.animate(with: flyFrames,
timePerFrame: 0.12)
flyAnimation = SKAction.repeatForever(flyAction)
}

func onTap() {}
required init?(coder aDecoder: NSCoder) {
super.init(coder: aDecoder)
}
}
```

Now that you have created the `Bat` class, there is one more enemy to add. We will add the `Blade` class next.

Guarding the ground with the Blade

The `Blade` class will keep Pierre from flying too low. This enemy class will be similar to the others we have created, with one exception: we will generate a physics body based on the texture. The physics body circles that we have been using are very fast computationally and are usually sufficient to describe the shapes of our enemies; the `Blade` class requires a more complicated physics body, given its half-circle shape and bumpy edges:



Adding the Blade class

To add the `Blade` class, create a new file named `Blade.swift` and add the following code:

```
import SpriteKit

class Blade: SKSpriteNode, GameSprite {
    var initialSize = CGSize(width: 185, height: 92)
    var textureAtlas: SKTextureAtlas =
        SKTextureAtlas(named: "Enemies")
    var spinAnimation = SKAction()

    init() {
        super.init(texture: nil, color: .clear,
                   size: initialSize)
        let startTexture = textureAtlas.textureNamed("blade")
        self.physicsBody = SKPhysicsBody(texture: startTexture,
                                         size: initialSize)
        self.physicsBody?.affectedByGravity = false
        self.physicsBody?.isDynamic = false
        createAnimations()
        self.run(spinAnimation)
    }

    func createAnimations() {
        let spinFrames: [SKTexture] = [
            textureAtlas.textureNamed("blade"),
            textureAtlas.textureNamed("blade-2")
        ]
        let spinAction = SKAction.animate(with: spinFrames,
                                         timePerFrame: 0.07)
        spinAnimation = SKAction.repeatForever(spinAction)
    }

    func onTap() {}
    required init?(coder aDecoder: NSCoder) {
        super.init(coder: aDecoder)
    }
}
```

Congratulations! The `Blade` class was the last enemy we needed to add to our game. This process may seem repetitive—you have written a lot of boilerplate code—but separating our enemies into their own classes allows each enemy to implement unique logic and behavior later. The benefits of this structure will become apparent as your games increase in complexity.

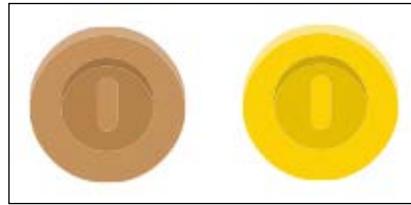
Next, we will add the class for our coins.

Adding coins

Coins are more fun if there are two value variations. We will create two coins:

- A Bronze Coin, worth one coin
- A Gold Coin, worth five coins

The two coins will be distinguishable by their color, as seen here:



Creating the coin classes

We only need a single `Coin` class to create both denominations. Everything in the `Coin` class should look very familiar at this point. To create the `Coin` class, add a new file named `Coin.swift` and then enter the following code:

```
import SpriteKit

class Coin: SKSpriteNode, GameSprite {
    var initialSize = CGSize(width: 26, height: 26)
    var textureAtlas: SKTextureAtlas =
        SKTextureAtlas(named: "Environment")
    var value = 1

    init() {
        let bronzeTexture =
            textureAtlas.textureNamed("coin-bronze")
        super.init(texture: bronzeTexture, color: .clear,
                   size: initialSize)
        self.physicsBody = SKPhysicsBody(circleOfRadius:
            size.width / 2)
```

```

        self.physicsBody?.affectedByGravity = false
    }

    // A function to transform this coin into gold!
    func turnToGold() {
        self.texture =
        textureAtlas.textureNamed("coin-gold")
        self.value = 5
    }

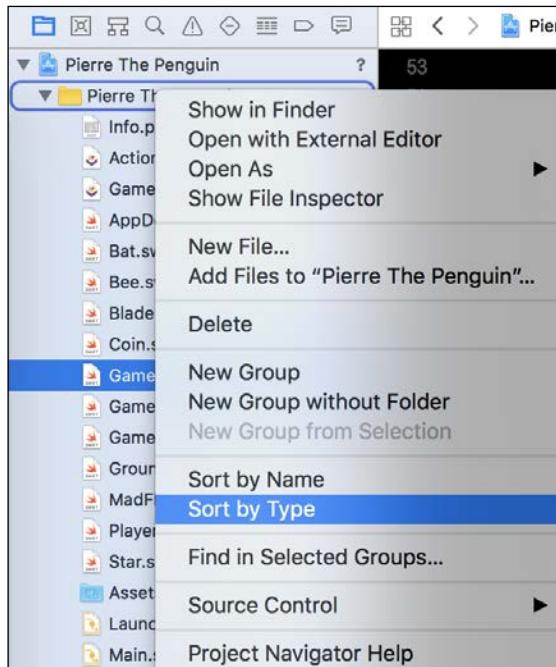
    func onTap() {}
    required init?(coder aDecoder: NSCoder) {
        super.init(coder: aDecoder)
    }
}

```

Great work—we have successfully added all of the new game objects we need for our final game!

Organizing the project navigator

You may notice that these new classes have cluttered the project navigator. This is a good time to clean up the navigator. Right-click the project in the project navigator and select **Sort by Type**, as shown in this screenshot:



Your project navigator will segment itself by file type and sort everything into alphabetical order. This makes it much easier to find files as you need them.

Testing the new game objects

It is time to see our hard work in action. We will now add one instance of each of our new classes to the game. Note that we will remove this testing code after we are done; you may want to leave yourself a comment or extra space for easy removal. Open GameScene.swift and locate the six lines that spawn the existing bee. Add this code after the bee lines:

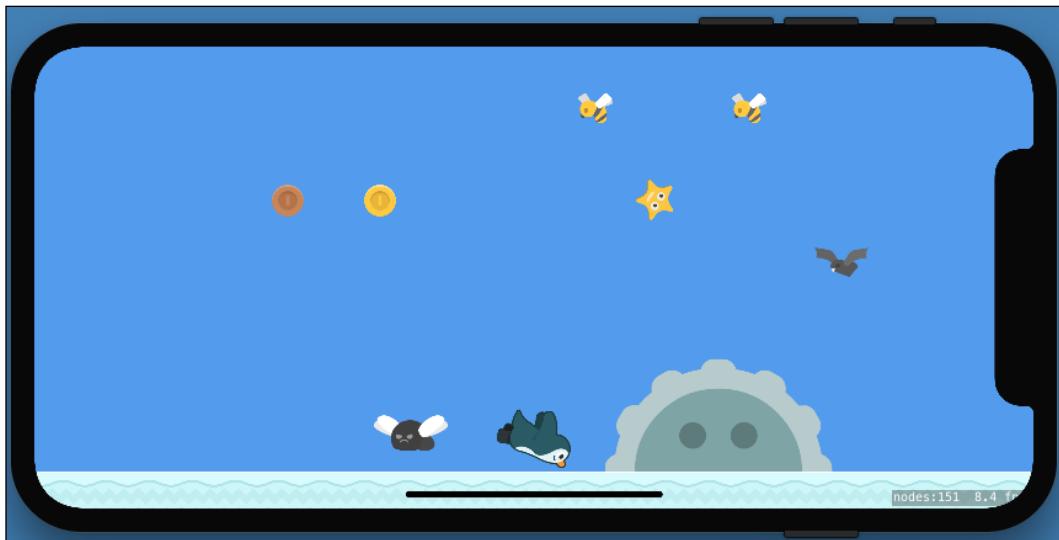
```
// Spawn a bat:  
let bat = Bat()  
bat.position = CGPoint(x: 400, y: 200)  
self.addChild(bat)  
  
// A blade:  
let blade = Blade()  
blade.position = CGPoint(x: 300, y: 76)  
self.addChild(blade)  
  
// A mad fly:  
let madFly = MadFly()  
madFly.position = CGPoint(x: 50, y: 50)  
self.addChild(madFly)  
  
// A bronze coin:  
let bronzeCoin = Coin()  
bronzeCoin.position = CGPoint(x: -50, y: 250)  
self.addChild(bronzeCoin)  
  
// A gold coin:  
let goldCoin = Coin()  
goldCoin.position = CGPoint(x: 25, y: 250)  
goldCoin.turnToGold()  
self.addChild(goldCoin)  
  
// The powerup star:  
let star = Star()  
star.position = CGPoint(x: 250, y: 250)  
self.addChild(star)
```

You may also wish to comment out the `Player` class line that moves Pierre forward, so the camera does not quickly move past your new game objects. Comment the following line in the player class:

```
// Set a constant velocity to the right:  
// self.physicsBody?.velocity.dx = 200
```

Just make sure to uncomment it when you are done.

Once you are ready, run the project. You should see the entire family, as shown in the following screenshot:



Terrific work! All of our code has paid off, and we have a large cast of characters ready for action.



Checkpoint 5-A

Code up to this point is available in the chapter's code resources.

Preparing for endless flight

In *Chapter 6, Generating a Never-Ending World*, we will build a never-ending level by spawning tactical obstacle courses full of these new game objects. We need to clear out all of our test objects to get ready for this new level-spawning system. Once you are ready, remove the new test code we just added to the `GameScene` class. Also, remove the six lines that we have been using to spawn the Bees from previous chapters. Finally, uncomment the line in `Player.swift` that sets Pierre's forward velocity (if you chose to comment it out when testing the new classes in the previous section).

When you are finished, your `GameScene` class' `didMove` function should look like this:

```
override func didMove(to view: SKView) {  
    self.anchorPoint = .zero  
    self.backgroundColor = UIColor(red: 0.4, green: 0.6, blue:  
        0.95, alpha: 1.0)  
  
    // Assign the camera to the scene  
    self.camera = cam  
  
    // Add the ground to the scene:  
    ground.position = CGPoint(x: -self.size.width * 2, y: 30)  
    ground.size = CGSize(width: self.size.width * 6, height: 0)  
    ground.createChildren()  
    self.addChild(ground)  
  
    // Add the player to the scene:  
    player.position = initialPlayerPosition  
    self.addChild(player)  
  
    // Set gravity  
    self.physicsWorld.gravity = CGVector(dx: 0, dy: -5)  
  
    // Store the vertical center of the screen:  
    screenCenterY = self.size.height / 2  
}
```

When you run the project, you should only see Pierre and the ground, as shown here:



We are now ready to build our level.

Summary

You added the complete cast of characters to our game in this chapter. Look back at all that you accomplished; you added the Power-up Star, the Bronze and Gold Coins, the Mad Fly, Bats, and the Blade. You tested all of the new classes and then removed the test code so that the project is ready for the level generation system we will put into place in the next chapter.

We put a lot of effort into building each new class. The world will come alive and reward our hard work in *Chapter 6, Generating a Never-Ending World*.

6

Generating a Never-Ending World

The unique challenge of an endless flyer-style game is in procedurally generating a rich, entertaining game world that extends as far as your player can fly. We will first explore level design concepts and tooling in Xcode; Apple added a built-in level designer to Xcode 6, allowing developers to arrange nodes visually within a scene. Once we become familiar with the SpriteKit level design methodology, we will create a custom solution to generate our world. In this chapter, you will build an entertaining world for our penguin game and learn how to design and implement levels in SpriteKit for any genre of game.

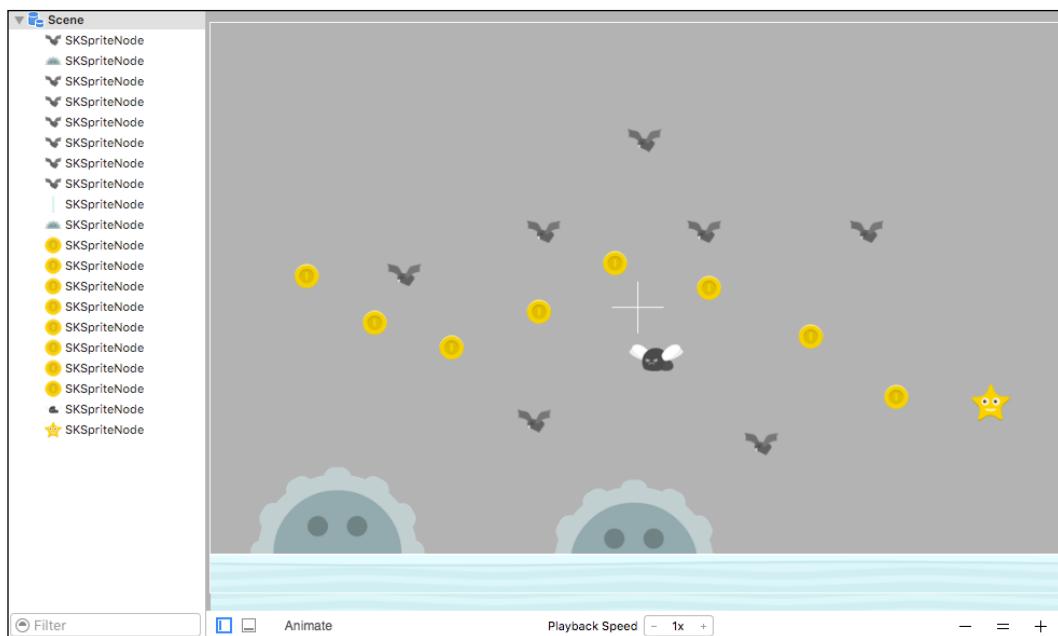
The topics in this chapter include the following:

- Designing levels with the SpriteKit scene editor
- Building encounters for Pierre Penguin
- Integrating scenes into the game
- Looping encounters for a never-ending world
- Adding the Power-up Star at random places
- Turning bronze coins into gold coins

Designing levels with the SpriteKit scene editor

The Scene editor is a valuable addition to SpriteKit. Previously, developers would be forced to hard-code positional values or rely on third-party tools or custom solutions for level design. Now, we can lay out our levels directly within Xcode by dragging and dropping sprites. We can create nodes, attach physics bodies and constraints, create physics fields, and edit properties directly from the interface.

Here is a simple example scene you might build by simply clicking and dragging:



In this example, I simply dragged and positioned sprites in the scene. If you are making an unsophisticated game, you can start in the scene editor rather than creating custom classes. By editing physics bodies in the editor, you can even create entire physics-based games in the editor, adding only a few lines of code for the controls.

Complex games require custom logic and texture animation for every object, so we will implement a system in our penguin game that only uses the scene editor as a layout generation tool. We will write code to parse the layout data from the editor and turn it into fully functioning versions of the game classes we have created throughout this book. In this way, we will separate our game logic from our level data with minimal effort.

Separating level data from game logic

The level layout is data, and it is best to separate data from code. You increase flexibility by separating the level data into scene files. The benefits include the following:

- Non-technical contributors, such as artists and designers, can add and edit levels without changing any code.
- Iteration time improves since you do not need to run the game in the simulator each time you need to view your positional changes. Scene editor layouts provide immediate visual feedback.
- Each level is in a unique file, which is ideal for avoiding merge conflicts when using source control solutions such as Git.

Using custom classes in the scene editor

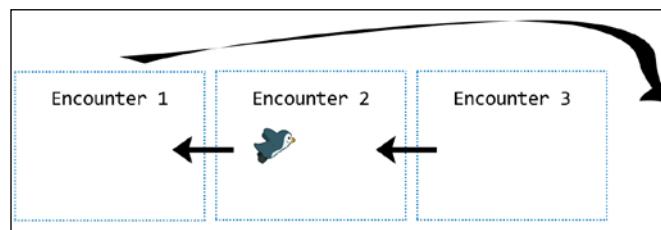
You can assign custom classes to the nodes you create in the scene editor. This assignment creates the association between scene editor node and custom class code (such as the `Bat`, `Blade`, `Coin`, and others that we created in *Chapter 5, Spawning Enemies, Coins, and Power-ups*).

You can also assign names to nodes in the scene editor and then query those names in your code. For example, we will create nodes with the custom class of `Coin` in the scene editor, and then assign the name of `GoldCoin` to the coins we want to turn gold. We will query for the name in the `Coin` class to fire the `turnToGold` function. We will see this technique in action later in this chapter.

Building encounters for Pierre Penguin

Endless flyer games continue until the player loses. They do not feature distinct levels; instead, we will design encounters (my own term) for our protagonist penguin to explore. We can create an endless world by stringing together encounters one after the other and randomly recycling from the beginning when we need more content.

The following diagram illustrates this basic concept:



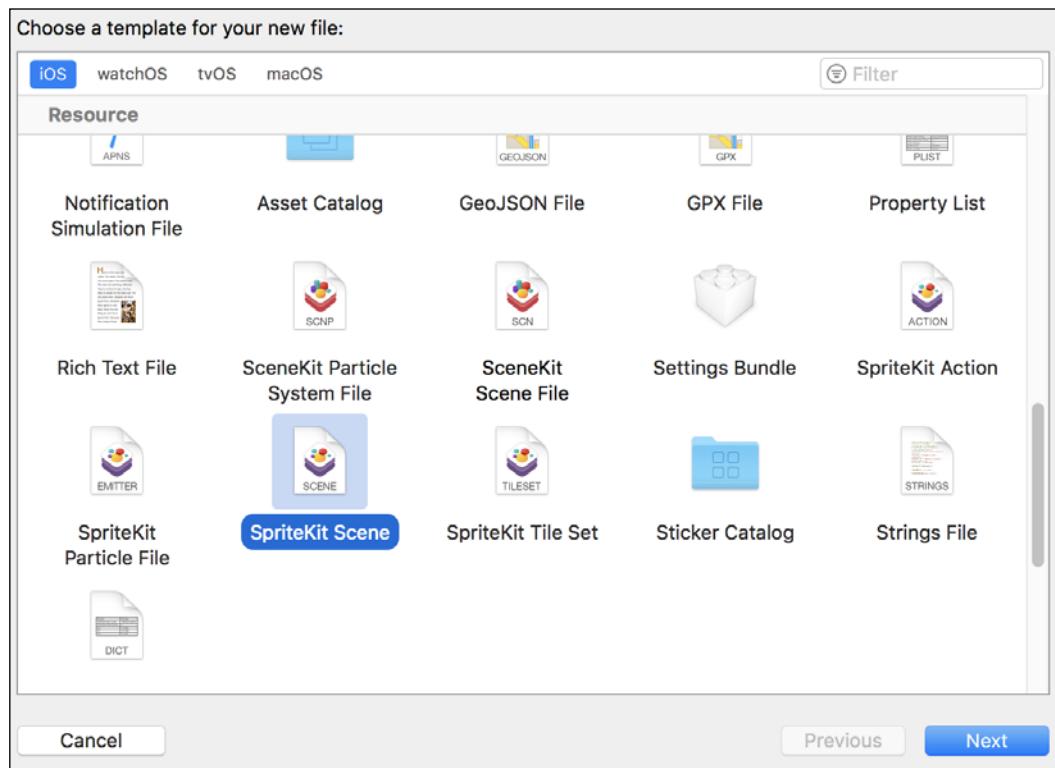
A finished game might include twenty or more encounters to feel varied and random. We will create three encounters in this chapter in order to populate the encounter recycling system.

We will build each encounter in its own scene file, in the same way we would approach a separate level in a standard platformer or physics game.

Creating our first encounter

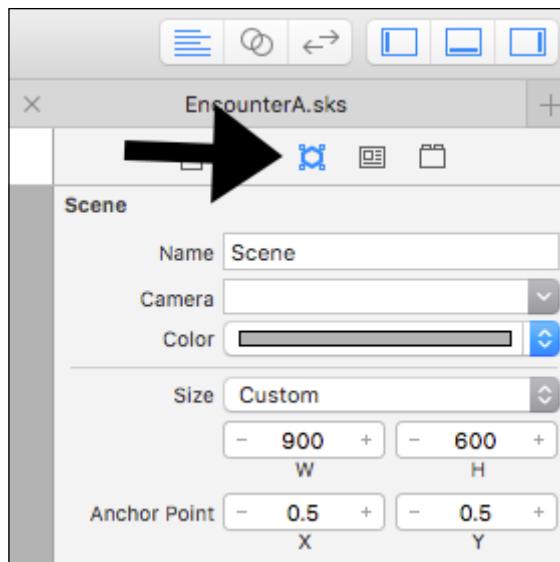
First, create an encounter folder group to keep our project organized. Right-click your project in the project navigator and create a new group named `Encounters`. Then, right-click on `Encounters` and add a new `SpriteKitScene` file (from the **iOS | Resource category**) named `EncounterA.sks`. Make sure to add a `SpriteKitScene` file and not a `SceneKit` scene (which is used in 3D games). Name this file `EncounterA`.

The `SpriteKit` scene can be seen in the following screenshot:



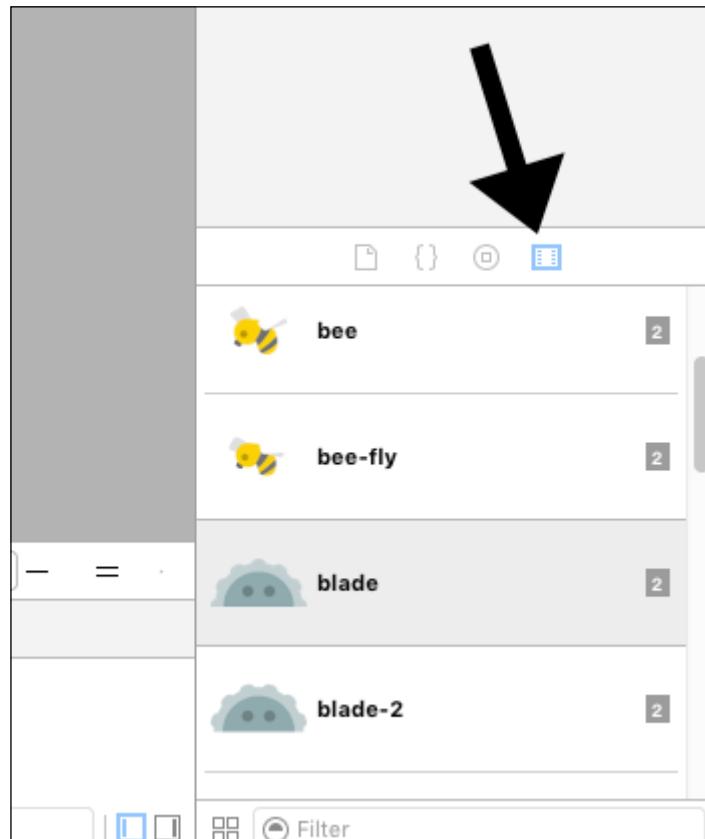
Xcode will add the new `SpriteKitScene` file to your project and open the scene editor. You should see a gray background with a white border, indicating the boundaries of the new scene. We can change the size of our scene to whatever we like; it will be easy to chain encounters together if each encounter is 900 pixels wide and 600 points tall.

You can easily change the scene's size values in the Attributes inspector. Toward the upper right corner of the scene editor, make sure that you have the Attributes inspector open by selecting the middle icon, and then change the width and height, as shown in the following screenshot:

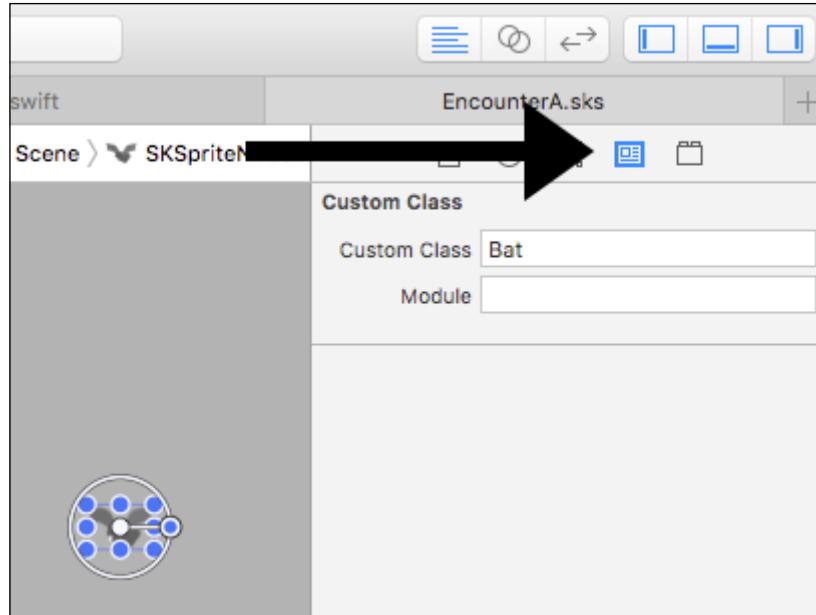


Next, we will create a sprite node by using the **Bat** custom class. Follow these steps to create a node in the scene editor:

1. You can drag textures from the media library directly onto the scene editor to create a new node. To open the media library, look toward the lower right side of the scene editor and select the movie reel icon, as shown in the following screenshot:



2. Scroll to the **bat** texture and drag it onto your scene. You will see a node appear on the gray background with the texture of the bat.
3. Using the **Custom Class** inspector on the upper right side, assign the **Bat** class to your node, as shown in this screenshot:



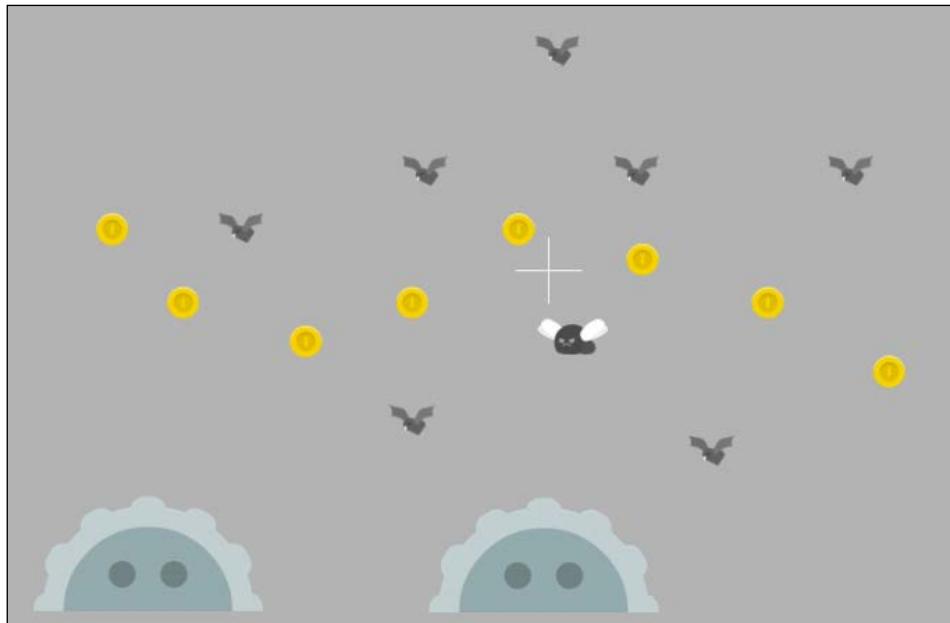
Great—you have started down the path to using the scene editor to create your level layout. We will repeat this process until we have built an entire encounter for Pierre Penguin to navigate. We can create Blades, Bees, Bats, Mad Flies, and Coins using this method.

Feel free to build out your first encounter. Add more nodes and assign custom classes until you are satisfied with the design. Try to picture the penguin character flying through the encounter. You do not need to add the ground or the Power-up Star; we will build both of those objects in our GameScene code.

You can speed up the build by holding down the option key, then clicking and dragging from an existing node to create a copy. The copy will have the same custom class attribute as the original. You can use this method to build the node layout very quickly.

It does not actually matter which texture you choose to drag onto the scene. The custom class attribute will determine which game object shows up in the final encounter. For instance, if you drag a bat texture onto the scene and assign it the custom class of `Coin`, you will see a coin in your game. However, dragging the correct texture helps you keep track of your progress and makes designing the layout easier.

In my encounter, I created a path through the Bats, filled with gold coins. You can use my encounter, shown in the following screenshot, for inspiration:



Integrating scenes into the game

Next, we will create a new class to manage the encounters in our game. Add a new Swift file to your project and name it `EncounterManager.swift`. The `EncounterManager` class will loop through our encounter scenes and use the positional data to create the appropriate game object classes in the game world. Add the following code inside the new file:

```
import SpriteKit

class EncounterManager {
    // Store your encounter file names:
    let encounterNames: [String] = [
        "EncounterA"
    ]
    // Each encounter is an SKNode, store an array:
    var encounters: [SKNode] = []

    init() {
        // Loop through each encounter scene:
```

```
for encounterFileName in encounterNames {
    // Create a new node for the encounter:
    let encounterNode = SKNode()

    // Load this scene file into a SKScene instance:
    if let encounterScene = SKScene(fileNamed:
        encounterFileName) {
        // Loop through each child node in the SKScene
        for child in encounterScene.children {
            // Create a copy of the scene's child node
            // to add to our encounter node:
            let copyOfNode = type(of: child).init()
            // Save the scene node's position to the copy:
            copyOfNode.position = child.position
            // Save the scene node's name to the copy:
            copyOfNode.name = child.name
            // Add the copy to our encounter node:
            encounterNode.addChild(copyOfNode)
        }
    }

    // Add the populated encounter node to the array:
    encounters.append(encounterNode)
}

// We will call this addEncountersToScene function from
// the GameScene to append all of the encounter nodes to the
// world node from our GameScene:
func addEncountersToScene(gameScene:SKNode) {
    var encounterPosY = 1000
    for encounterNode in encounters {
        // Spawn the encounters behind the action, with
        // increasing height so they do not collide:
        encounterNode.position = CGPoint(x: -2000,
            y: encounterPosY)
        gameScene.addChild(encounterNode)
        // Double the Y pos for the next encounter:
        encounterPosY *= 2
    }
}
```

Great—you just added the functionality to use our scene file data inside the game world. Next, follow these steps to wire up the `EncounterManager` class in the `GameScene` class:

1. Add a new instance of the `EncounterManager` class as a constant in the `GameScene` class:

```
let encounterManager = EncounterManager()
```

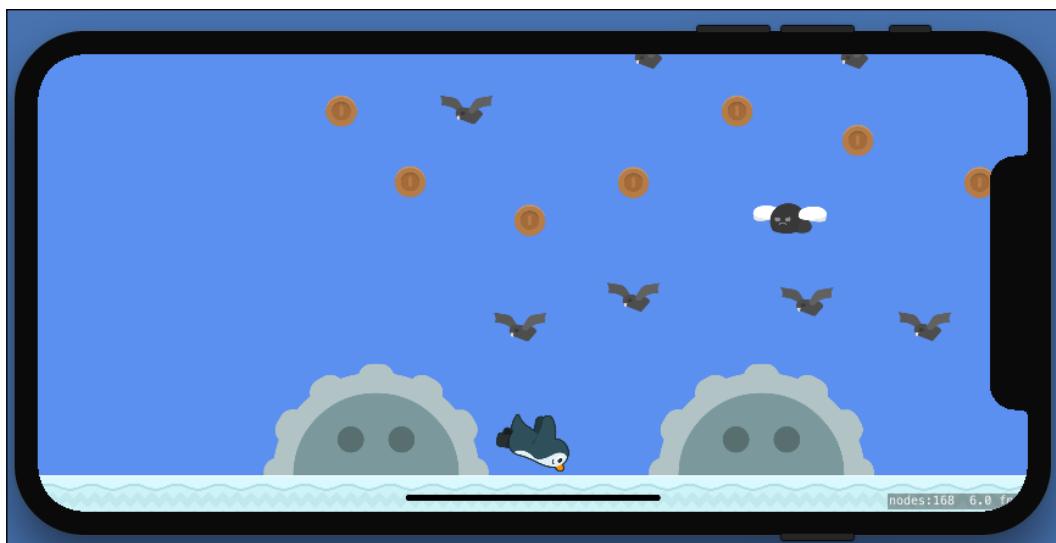
2. At the bottom of the `didMove` function, call `addEncountersToScene` to add each encounter node as a child of the `GameScene` node:

```
encounterManager.addEncountersToScene(gameScene: self)
```

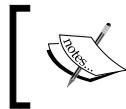
3. Since the `EncounterManager` class spawns encounters far off the screen, we will temporarily move our first encounter directly in front of the starting player position to test our code. Add this line in the `didMove` function:

```
encounterManager.encounters[0].position =  
CGPoint(x: 400, y: 330)
```

Run the project. Your game should look something like this screenshot:



Congratulations! You have implemented the core functionality of using placeholder nodes in the scene editor. You can remove the line that positions this encounter at the beginning of the game, which we added in step 3. Next, we will create a system that repositions each encounter ahead of Pierre Penguin.



Checkpoint 6-A

Code up to this point is available in this chapter's code resources.

Looping encounters for a never-ending world

We need at least three encounters to endlessly cycle and create a never-ending world; two can be on the screen at any one time, with a third positioned ahead of the player. We can track Pierre's progress and reposition the encounter nodes ahead of him.

Building more encounters

We need to build at least two more encounters before we can implement the repositioning system. You can create more if you like; the system will support any number of encounters. For now, add two more SpriteKit scene files to your game: `EncounterB.sks` and `EncounterC.sks`. Resize these scenes to 900 wide by 600 tall, the same as `EncounterA`. You can fill these encounters with Bees, Blades, Coins, and Bats – have fun! Make sure to assign the custom class attribute to the sprites that you drag into the scene editor.

For inspiration, here is my `EncounterB.sks`:



Here is my `EncounterC.sks`:



For perfect alignment with the ground, place your Blade sprites at -224 on the Y-axis.

Updating the `EncounterManager` class

We have to inform the `EncounterManager` class about these new encounters. Open the `EncounterManager.swift` file and add the new encounter names to the `encounterNames` constant:

```
// Store your encounter file names:  
let encounterNames: [String] = [  
    "EncounterA",  
    "EncounterB",  
    "EncounterC"  
]
```

We also need to keep track of the encounters that can potentially be on the screen at any given time. Add two new properties to the `EncounterManager` class:

```
var currentEncounterIndex: Int?
var previousEncounterIndex: Int?
```

Storing metadata in `SKSpriteNode.userData` property

We are going to recycle the encounter nodes as Pierre moves through the world, so we need to add the functionality to reset all of the game objects in an encounter before placing it in front of the player. Otherwise, Pierre's previous trips through the encounter will knock nodes out of place.

The `SKSpriteNode` class provides a property named `userData` that we can use to store miscellaneous data about the sprite. We will use the `userData` property to store the initial position of each sprite in the encounter so that we can reset the sprites when we reposition an encounter. Add these two new functions to the `EncounterManager` class:

```
// Store the initial positions of the children of a node:
func saveSpritePositions(node: SKNode) {
    for sprite in node.children {
        if let spriteNode = sprite as? SKSpriteNode {
            let initialPositionValue = NSValue.init(CGPoint:
                sprite.position)
            spriteNode.userData = ["initialPosition":
                initialPositionValue]
            // Save the positions for children of this node:
            saveSpritePositions(node: spriteNode)
        }
    }
}

// Reset all children nodes to their original position:
func resetSpritePositions(node: SKNode) {
    for sprite in node.children {
        if let spriteNode = sprite as? SKSpriteNode {
            // Remove any linear or angular velocity:
            spriteNode.physicsBody?.velocity = CGVector(dx: 0,
                dy: 0)
```

```
        spriteNode.physicsBody?.angularVelocity = 0
        // Reset the rotation of the sprite:
        spriteNode.zRotation = 0
        if let initialPositionVal =
            spriteNode.userData?.value(forKey:
                "initialPosition") as? NSValue {
            // Reset the position of the sprite:
            spriteNode.position =
                initialPositionVal.cgPointValue
        }

        // Reset positions on this node's children
        resetSpritePositions(node: spriteNode)
    }
}
}
```

We want to call our new `saveSpritePositions` function on `init`, when we are first spawning the encounters. Update the `init` function of `EncounterManager` at the bottom of the `for` loop, below the line that appends the encounter node to the `encounters` array (the new line is in bold):

```
// Add the populated encounter node to the array:
encounters.append(encounterNode)
// Save initial sprite positions for this encounter:
saveSpritePositions(node: encounterNode)
```

Lastly, we need a function to reset encounters and reposition them in front of the player. Add this new function to the `EncounterManager` class:

```
func placeNextEncounter(currentXPos: CGFloat) {
    // Count the encounters in a random ready type (UInt32):
    let encounterCount = UInt32(encounters.count)
    // The game requires at least 3 encounters to function
    // so exit this function if there are less than 3
    if encounterCount < 3 { return }

    // We need to pick an encounter that is not
    // currently displayed on the screen.
    var nextEncounterIndex: Int?
    var trulyNew: Bool?
    // The current encounter and the directly previous encounter
    // can potentially be on the screen at this time.
    // Pick until we get a new encounter
```

```

while trulyNew == false || trulyNew == nil {
    // Pick a random encounter to set next:
    nextEncounterIndex =
        Int(arc4random_uniform(encounterCount))
    // First, assert that this is a new encounter:
    trulyNew = true
    // Test if it is instead the current encounter:
    if let currentIndex = currentEncounterIndex {
        if (nextEncounterIndex == currentIndex) {
            trulyNew = false
        }
    }
    // Test if it is the directly previous encounter:
    if let previousIndex = previousEncounterIndex {
        if (nextEncounterIndex == previousIndex) {
            trulyNew = false
        }
    }
}
// Keep track of the current encounter:
previousEncounterIndex = currentEncounterIndex
currentEncounterIndex = nextEncounterIndex

// Reset the new encounter and position it ahead of the player
let encounter = encounters[currentEncounterIndex!]
encounter.position = CGPoint(x: currentXPos + 1000, y: 300)
resetSpritePositions(node: encounter)
}

```

Wiring up EncounterManager in the GameScene class

We will track Pierre's progress in the GameScene class and call the EncounterManager class code when appropriate. Follow these steps to wire up the EncounterManager class:

1. Add a new property to the GameScene class to track when we should next position an encounter in front of the player. We will start with a value of 150 to spawn the first encounter right away:

```
var nextEncounterSpawnPosition = CGFloat(150)
```

2. Next, we simply need to check whether the player moves past this position in the `didSimulatePhysics` function. Add this code at the bottom of `didSimulatePhysics`:

```
// Check to see if we should set a new encounter:  
if player.position.x > nextEncounterSpawnPosition {  
    encounterManager.placeNextEncounter(  
        currentXPos: nextEncounterSpawnPosition)  
    nextEncounterSpawnPosition += 1200  
}
```

Fantastic—we have added all the functionality we need to create endlessly looping encounters in front of the player. Run the project. You should see your encounters looping in front of you forever. Enjoy flying through your hard work!

If you do not see your encounters, make sure that you assigned the custom class attribute in the scene editor for each node.

Adding the Power-up Star at random places

We still need to add the Power-up Star into the world. We can randomly spawn a Star every ten encounters to add some extra excitement. Follow these steps to add the Star logic:

1. Add a new instance of the `Star` class as a constant on the `GameScene` class:

```
let powerUpStar = Star()
```

2. Anywhere inside the `GameScene didMove` function, add the Star as a child of the `GameScene` and position it:

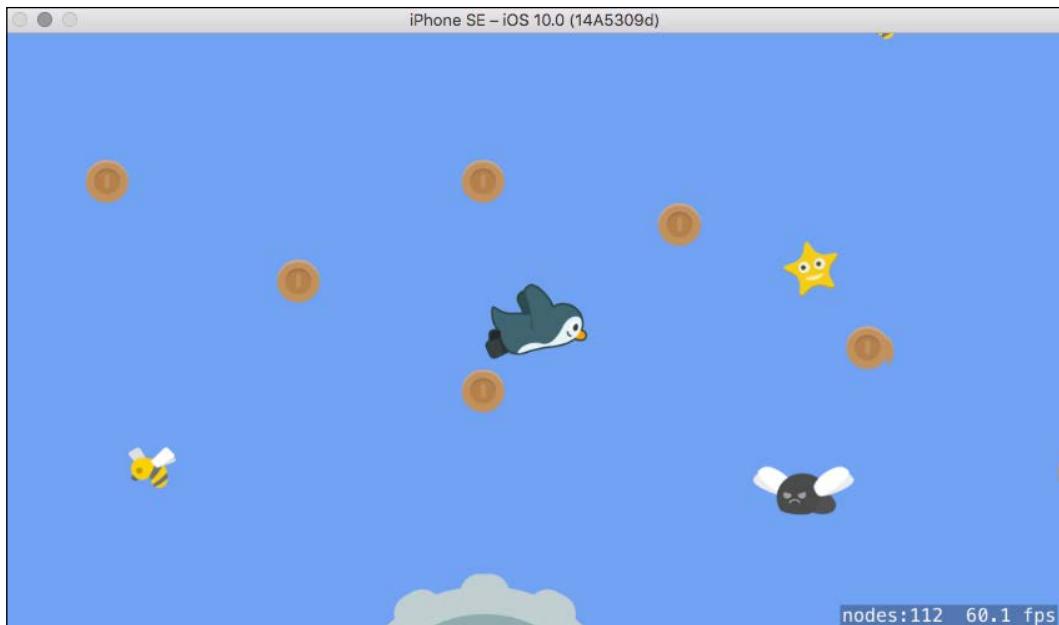
```
// Place the star out of the way for now:  
self.addChild(powerUpStar)  
powerUpStar.position = CGPointMake(x: -2000, y: -2000)
```

3. Inside the `GameScene didSimulatePhysics` function, update your new encounter code as follows (the new code is in bold):

```
// Check to see if we should set a new encounter:  
if player.position.x > nextEncounterSpawnPosition {  
    encounterManager.placeNextEncounter(  
        currentXPos: nextEncounterSpawnPosition)  
    nextEncounterSpawnPosition += 1200
```

```
// Each encounter has a 10% chance to spawn a star:  
let starRoll = Int(arc4random_uniform(10))  
if starRoll == 0 {  
    // Only move the star if it is off the screen.  
    if abs(player.position.x - powerUpStar.position.x)  
        > 1200 {  
        // Y Position 50-450:  
        let randomYPos = 50 +  
            CGFloat(arc4random_uniform(400))  
        powerUpStar.position = CGPointMake(x:  
            nextEncounterSpawnPosition, y: randomYPos)  
        // Remove any previous velocity and spin:  
        powerUpStar.physicsBody?.angularVelocity = 0  
        powerUpStar.physicsBody?.velocity =  
            CGVector(dx: 0, dy: 0)  
    }  
}  
}
```

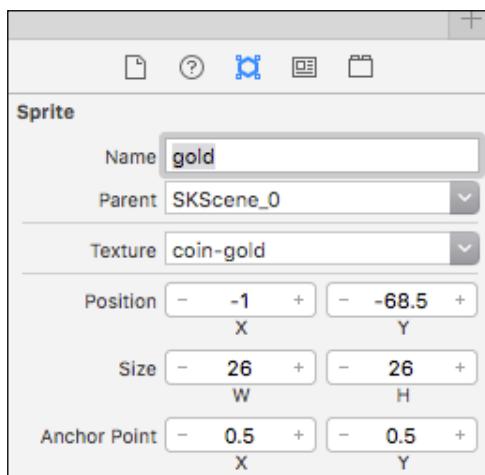
Run the game again, and you should see a Star spawn occasionally inside your encounters, as shown in the following screenshot:



Turning bronze coins into gold coins

You may notice that your gold coins in the scene editor show up as bronze coins in the game. This is because the `Coin` class defaults to using the bronze texture and value. We will specify our gold coins by setting their name attribute to `gold` in the scene editor and then checking for this name in the `Coin` `init` function. Follow these steps to implement golden coins:

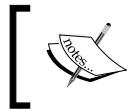
1. Open your encounters in the scene editor, select your gold coins, and use the Attributes inspector to set the name value of each node to `gold`, as shown here:



2. We can easily loop through all nodes with a certain name. We will use this functionality to run the `Coin` class's `turnToGold` function on any node with the name `gold`. Open `EncounterManager` and add the following code in the `init` function, at the bottom of the encounter's for loop (the new code is in bold):

```
// Save initial sprite positions for this encounter:  
saveSpritePositions(node: encounterNode)  
// Turn golden coins gold!  
encounterNode.enumerateChildNodes(withName: "gold") {  
    (node: SKNode, stop: UnsafeMutablePointer) in  
    (node as? Coin)?.turnToGold()  
}
```

Run your project. The gold coins should be showing up correctly. Great work! This is a useful technique for passing extra data from the scene editor to the code.



Checkpoint 6-B

Code up to this point is available in this chapter's code resources.

Summary

Great job—we have covered a lot of ground in this chapter. You learned about Xcode's new scene editor, used the scene editor to lay out sprites with custom classes, and interpreted the scene data to spawn game objects in our game world. Then, you created a system to loop encounters for our endless flyer game.

Congratulate yourself; the encounter system you built in this chapter is the most complex system in our game. You are officially in a great position to finish your first SpriteKit game!

Next, we will look at creating custom events when game objects collide. We will add health, damage, coin pick-up, invincibility, and more in *Chapter 7, Implementing Collision Events*.

7

Implementing Collision Events

So far, we have let the SpriteKit physics simulation detect and handle collisions between game objects. You have seen that Pierre Penguin sends enemies and coins flying off into space when he flies into them. This is because the physics simulation automatically monitors collisions and sets the post-collision trajectory and velocity of each colliding body. In this chapter, we will add our own game logic when two objects come into contact: taking damage from enemies, granting the player invulnerability after touching the star, and tracking points as the player collects coins. The game will become more fun to play as the game mechanics come to life.

The topics in this chapter include the following:

- Learning the SpriteKit collision vocabulary
- Adding contact events to our game
- Player health and damage
- Collecting coins
- The Power-up Star logic

Learning the SpriteKit collision vocabulary

SpriteKit uses some unique concepts and terms to describe physics events. If you familiarize yourself with these terms now, it will be easier to understand the implementation steps later in this chapter.

Collision versus contact

There are two types of interaction when physics bodies come together in the same space:

- Collision: This is the physics simulation's mathematical analysis and repositioning of bodies after they touch. Collisions include all the automatic physical interactions between bodies: preventing overlap, bouncing apart, spinning through the air, and transferring momentum. By default, physics bodies collide with every other physics body in the scene; we have witnessed this automatic collision behavior in our game so far.
- Contact: This event also occurs when two bodies touch. Contact events allow us to wire in our custom game logic when two bodies come into contact. Contact events do not create any change on their own; they only provide us with the chance to execute our own code. For instance, we will use contact events to assign damage to the player when he or she runs into an enemy. There are no contact events by default; we will manually configure contacts in this chapter.

Physics bodies collide with every other body in the scene by default, but you can configure specific bodies to ignore collisions and pass through each other without any physical reaction.

Additionally, collisions and contacts are independent; you can disable physical collision between two types of bodies and still fire custom code with a contact event when the bodies pass through each other.

Physics category masks

You can assign physics categories to each physics body in your game. These categories allow you to specify the bodies that should collide, the bodies that should be in direct contact, and the bodies that should pass through each other without any event. When two bodies try to share the same space, the physics simulation will compare each body's categories and test whether collision or contact events should fire.

Our game will include physics categories for the penguin, the ground, the coins, and the enemies.

Physics categories are stored as 32-bit masks, which allows the physics simulation to perform these tests with processor-efficient bitwise operations. It is not strictly necessary to understand bitwise operations to use physics categories, but it is a nice topic for further reading, if you are interested in enhancing your knowledge. If you are interested, try an internet search for Swift bitwise operations.

Each physics body has three properties you can use to control collisions in your game. Let's begin with a very simple summary of each property, and then explore them in depth:

- `categoryBitMask`: The physics body's own physical categories
- `collisionBitMask`: Collide with these physical categories
- `contactTestBitMask`: Contact with these physical categories

The `categoryBitMask` property stores the body's current physics categories. The default value is `0xFFFFFFFF`, equating to every category. This means that, by default, every physics body belongs to every physics category.

The `collisionBitMask` property specifies the physical categories the body should collide with, preventing two bodies from sharing the same space. The starting value is `0xFFFFFFFF`, or all bits set, meaning that the body will collide with every category by default. When one body begins to overlap with another, the physics simulation compares each body's `collisionBitMask` against the other body's `categoryBitMask`. If there is a match, a collision takes place. Note that this test works two ways; each body can independently participate or ignore a collision.

The `contactTestBitMask` property works just like the `collision` property, but specifies categories for contact events, instead of collisions. The default value is `0x00000000`, or no bits set, meaning that the body will not come into contact with anything by default.

This is a dense subject. It is OK to move forward if you do not yet fully understand this topic. Implementing category masks into our game will help you learn.

Using category masks in Swift

Apple's Adventure game demo provides a good implementation of bitmasks in Swift. You can download Apple's latest demo SpriteKit games from <https://developer.apple.com/spritekit/>. We will follow their example and use an `enum` to store our categories as `UInt32` values, writing these bitmasks in an easy-to-read manner. The following is an example of a physics category `enum` for a theoretical war game:

```
enum PhysicsCategory: UInt32 {
    case playerTank = 1
    case enemyTanks = 2
    case missiles = 4
    case bullets = 8
    case buildings = 16
}
```

It is very important to double the value for each subsequent group; this is a necessary step if you want to create proper bitmasks for the physics simulation. For example, if we were to add `fighterJets`, the value would need to be 32. Always remember to double subsequent values to create unique bitmasks that perform as expected in the physics tests.

Bitmasks are binary values that the CPU can very quickly compare to check for a match. You do not need to understand bitwise operators to complete this material, but if you are already familiar and curious, this doubling method works because 2 is equivalent to $1 \ll 1$ (binary: 10), 4 is equivalent to $1 \ll 2$ (binary: 100), 8 is equivalent to $1 \ll 3$ (binary: 1000), and so on. We opt for the manual doubling since `enum` values must be literals, and these values are easier for humans to read.

Adding contact events to our game

Now that you are familiar with SpriteKit's physics concepts, we can head into Xcode to implement physics categories and contact logic for our penguin game. We will start by adding our physics categories.

Setting up the physics categories

To create our physics categories, open your `GameScene.swift` file and enter the following code at the very top, completely outside the `GameScene` class:

```
enum PhysicsCategory: UInt32 {  
    case penguin = 1  
    case damagedPenguin = 2  
    case ground = 4  
    case enemy = 8  
    case coin = 16  
    case powerup = 32  
}
```

Notice how we double each succeeding value, as in our previous example. We are also creating an extra category for our penguin to use after he takes damage. We will use the `damagedPenguin` physics category to allow the penguin to pass through enemies for a few seconds after taking damage.

Assigning categories to game objects

Now that we have the physics categories, we need to go back through our existing game objects and assign the categories to the physics bodies. We will start with the `Player` class.

The player

Open `Player.swift` and add the following code at the bottom of the `init` function:

```
self.physicsBody?.categoryBitMask =  
    PhysicsCategory.penguin.rawValue  
self.physicsBody?.contactTestBitMask =  
    PhysicsCategory.enemy.rawValue |  
    PhysicsCategory.ground.rawValue |  
    PhysicsCategory.powerup.rawValue |  
    PhysicsCategory.coin.rawValue  
self.physicsBody?.collisionBitMask =  
    PhysicsCategory.ground.rawValue
```

We assigned the penguin physics category to the `Player` physics body, and used the `contactTestBitMask` property to set up contact logic tests with enemies, the ground, power-ups, and coins. We used the `collisionBitMask` to make the penguin only bounce off the ground while gliding through other game objects.

Also, notice how we use the `rawValue` property of our enum values. You will need to use the `rawValue` property whenever you are using the physics category bitmasks.

The ground

Next, let's assign the physics category for the `Ground` class. Open `Ground.swift`, and add the following code at the very bottom of the `createChildren` function:

```
self.physicsBody?.categoryBitMask =  
    PhysicsCategory.ground.rawValue
```

All we need to do is assign the ground bitmask to the `Ground` class physics body, since it already collides with everything by default.

The Power-up Star

Open `Star.swift` and add the following code at the bottom of the `init` function:

```
self.physicsBody?.categoryBitMask =  
    PhysicsCategory.powerup.rawValue
```

This assigns the power-up physics category to the `Star` class.

Enemies

Perform this same action in `Bat.swift`, `Bee.swift`, `Blade.swift`, and `MadFly.swift`. Add the following code inside their `init` functions:

```
self.physicsBody?.categoryBitMask = PhysicsCategory.enemy.rawValue  
self.physicsBody?.collisionBitMask =  
    ~PhysicsCategory.damagedPenguin.rawValue
```

We use the bitwise NOT operator (`~`) to remove the `damagedPenguin` physics category from collisions with enemies. Enemies will collide with all categories except the `damagedPenguin` physics category. This allows us to change the penguin's category to the `damagedPenguin` value when we want the penguin to ignore enemy collisions and pass straight through.

Coins

Lastly, we will add the coin physics category. We do not want coins to collide with other game objects, but we still want to monitor for contact events. Open `Coin.swift` and add the following code at the bottom of the `init` function:

```
self.physicsBody?.categoryBitMask = PhysicsCategory.coin.rawValue  
self.physicsBody?.collisionBitMask = 0
```

Preparing GameScene for contact events

Now that we have assigned the physics categories to our game objects, we can monitor for contact events in the `GameScene` class. Follow these steps to wire up the `GameScene` class:

1. First, we need to tell the `GameScene` class to implement the `SKPhysicsContactDelegate` protocol. SpriteKit can then inform the `GameScene` class when contact events occur. Change the `GameScene` class declaration line to look like this:

```
class GameScene: SKScene, SKPhysicsContactDelegate {
```

2. We will tell SpriteKit to inform `GameScene` of contact events by setting the `GameScene` `physicsWorld` `contactDelegate` property to the `GameScene` instance. At the bottom of the `GameScene` `didMove` function, add this line:

```
self.physicsWorld.contactDelegate = self
```

3. `SKPhysicsContactDelegate` defines a `didBegin` function that will fire when contact occurs. We can now implement this `didBegin` function in the `GameScene` class. Create a new function in the `GameScene` class named `didBegin`, as shown in the following code:

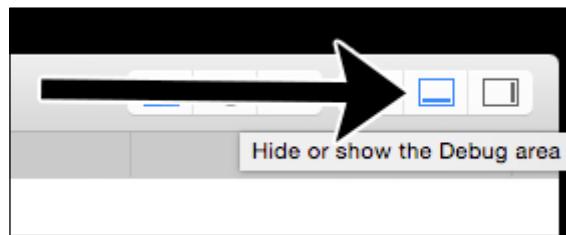
```
func didBegin(_ contact: SKPhysicsContact) {  
    // Each contact has two bodies,  
    // We do not know which is which.  
    // We will find the penguin body first, then use  
    // the other body to determine the type of contact.  
    let otherBody: SKPhysicsBody  
    // Combine the two penguin physics categories into one  
    // bitmask using the bitwise OR operator |  
    let penguinMask = PhysicsCategory.penguin.rawValue |  
        PhysicsCategory.damagedPenguin.rawValue  
    // Use the bitwise AND operator & to find the penguin.  
    // This returns a positive number if body A's category  
    // is the same as either the penguin or  
        damagedPenguin:  
    if (contact.bodyA.categoryBitMask & penguinMask) > 0 {  
        // bodyA is the penguin, we will test bodyB's  
        type:  
        otherBody = contact.bodyB  
    }  
    else {  
        // bodyB is the penguin, we will test bodyA's  
        type:  
        otherBody = contact.bodyA  
    }  
    // Find the type of contact:  
    switch otherBody.categoryBitMask {  
    case PhysicsCategory.ground.rawValue:  
        print("hit the ground")  
    case PhysicsCategory.enemy.rawValue:  
        print("take damage")  
    case PhysicsCategory.coin.rawValue:  
        print("collect a coin")  
    case PhysicsCategory.powerup.rawValue:  
        print("start the power-up")  
    default:  
        print("contact with no game logic")  
    }  
}
```

This function will serve as a central hub for our contact events. We will print to the console when our various contact events occur, to test that our code is working.

Viewing console output

You can use the `print` function to write information to the console, which is very useful for debugging. If you have not yet used the console in Xcode, follow these simple steps to view it:

1. In the upper right-hand corner of Xcode, make sure that the debug area is turned on, as shown in the following screenshot:



2. In the bottom right-hand corner of Xcode, make sure that the console is turned on, as shown in the following screenshot:



Testing our contact code

Now that you can see your console output, run the project. You should see our `print` strings appear in the console as you fly Pierre into various game objects. Your console should look something like this:

```

collect a coin
collect a coin
collect a coin
take damage
take damage

```

All Output

Congratulations— if you see the contact output in the console, you have completed the structure for our contact system.

You may notice that flying into coins produces strange collision behavior, which we will enhance later in this chapter. Next, we will add game logic for each type of contact.



Checkpoint 7-A

The code up to this point is available in this chapter's code resources.



Player health and damage

The first custom contact logic is player damage. We will assign the player health points and take them away when damaged. The game will end when the player runs out of health. This is one of the core mechanics of our gameplay. Follow these steps to implement the health logic:

1. In the `Player.swift` file, add six new properties to the `Player` class:

```

// The player will be able to take 3 hits before game
over:
var health: Int = 3
// Keep track of when the player is invulnerable:
var invulnerable = false
// Keep track of when the player is newly damaged:
var damaged = false
// We will create animations to run when the player takes
// damage or dies. Add these properties to store them:
var damageAnimation = SKAction()
var dieAnimation = SKAction()
// We want to stop forward velocity if the player dies,
// so we will now store forward velocity as a property:
var forwardVelocity: CGFloat = 200

```

2. Inside the `update` function, change the code that moves the player through the world in order to use the new `forwardVelocity` property:

```
// Set a constant velocity to the right:  
self.physicsBody?.velocity.dx = self.forwardVelocity
```

3. At the very beginning of the `startFlapping` function, add this line to prevent the player from flying higher when they die:

```
if self.health <= 0 { return }
```

4. Add the same line at the very beginning of the `stopFlapping` function to prevent the soar animation from running after death:

```
if self.health <= 0 { return }
```

5. Add a new function named `die` to the `Player` class:

```
func die() {  
    // Make sure the player is fully visible:  
    self.alpha = 1  
    // Remove all animations:  
    self.removeAllActions()  
    // Run the die animation:  
    self.run(self	dieAnimation)  
    // Prevent any further upward movement:  
    self.flapping = false  
    // Stop forward movement:  
    self.forwardVelocity = 0  
}
```

6. Add a new function named `takeDamage` to the `Player` class:

```
func takeDamage() {  
    // If invulnerable or damaged, return:  
    if self.invulnerable || self.damaged { return }  
  
    // Remove one from our health pool  
    self.health -= 1  
    if self.health == 0 {  
        // If we are out of health, run the die  
        // function:  
        die()  
    }  
    else {  
        // Run the take damage animation:  
        self.run(self.damageAnimation)  
    }  
}
```

7. Open the `GameScene.swift` file. Inside the `didBegin` function, update the switch case that fires when contact is made with an enemy:

```
case PhysicsCategory.enemy.rawValue:  
    print("take damage")  
    player.takeDamage()
```

8. We will also take damage when we hit the ground. Update the ground case in the same way:

```
case PhysicsCategory.ground.rawValue:  
    print("hit the ground")  
    player.takeDamage()
```

Good work – let's test our code to make sure everything is working correctly. Run the project and smash into some enemies. You can watch the printed output in the console to make sure everything is working correctly. After taking damage three times, the penguin should drop to the ground and become unresponsive.

You may notice that there is no way for the player to tell how many health points he or she has remaining as they play the game. We will add a health meter to the scene in the next chapter.

Next, we will enhance the feel of the game with new animations when the player takes damage and when the game ends.

Animations for damage and game over

We will use `SKAction` sequences to create fun animations when the player takes damage. By combining actions, we will grant temporary safety after the player hits an enemy and is in a damaged state. We will show a fade animation that slowly pulses at first and then speeds up as the safe state starts to wear off.

The damage animation

To add the new animation, add this code at the bottom of the `Player` class' `createAnimations` function:

```
// --- Create the taking damage animation ---  
let damageStart = SKAction.run {  
    // Allow the penguin to pass through enemies:  
    self.physicsBody?.categoryBitMask =  
        PhysicsCategory.damagedPenguin.rawValue  
}
```

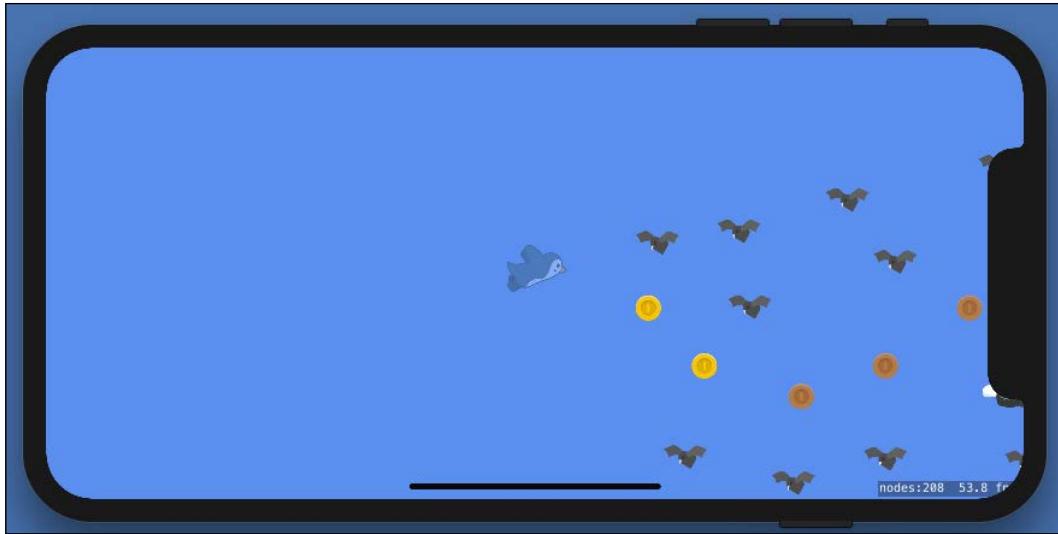
Implementing Collision Events

```
// Create an opacity pulse, slow at first and fast at the end:  
let slowFade = SKAction.sequence([  
    SKAction.fadeAlpha(to: 0.3, duration: 0.35),  
    SKAction.fadeAlpha(to: 0.7, duration: 0.35)  
])  
let fastFade = SKAction.sequence([  
    SKAction.fadeAlpha(to: 0.3, duration: 0.2),  
    SKAction.fadeAlpha(to: 0.7, duration: 0.2)  
])  
let fadeOutAndIn = SKAction.sequence([  
    SKAction.repeat(slowFade, count: 2),  
    SKAction.repeat(fastFade, count: 5),  
    SKAction.fadeAlpha(to: 1, duration: 0.15)  
])  
// Return the penguin to normal:  
let damageEnd = SKAction.run {  
    self.physicsBody?.categoryBitMask =  
        PhysicsCategory.penguin.rawValue  
    // Turn off the newly damaged flag:  
    self.damaged = false  
}  
// Store the whole sequence in the damageAnimation property:  
self.damageAnimation = SKAction.sequence([  
    damageStart,  
    fadeOutAndIn,  
    damageEnd  
])
```

Next, update the `takeDamage` function to flag the player as damaged immediately after taking a hit. The damage animation you just created will turn the damaged flag back off once it has completed. After this change, the first four lines of the `takeDamage` function should look like this (the new code is written in bold):

```
// If invulnerable or damaged, return out of the function:  
if self.invulnerable || self.damaged { return }  
// Set the damaged state to true after being hit:  
self.damaged = true
```

Run the project. Directly after taking damage, your penguin should fade and be able to pass through enemies, as shown in this image:



We are starting to see some good results from our hard work. Notice how the penguin can pass through enemies but still collides with coins, the Star, and the ground while in the invulnerable state. Next, we will add a game over animation.

The game over animation

We will create a funny, over-the-top death animation when the penguin runs out of health. When Pierre loses his last hit point, he will hang in the air, scale larger, flip over onto his back, and then finally fall to the ground. To implement this animation, add the following code at the bottom of the `Player` class' `createAnimations` function:

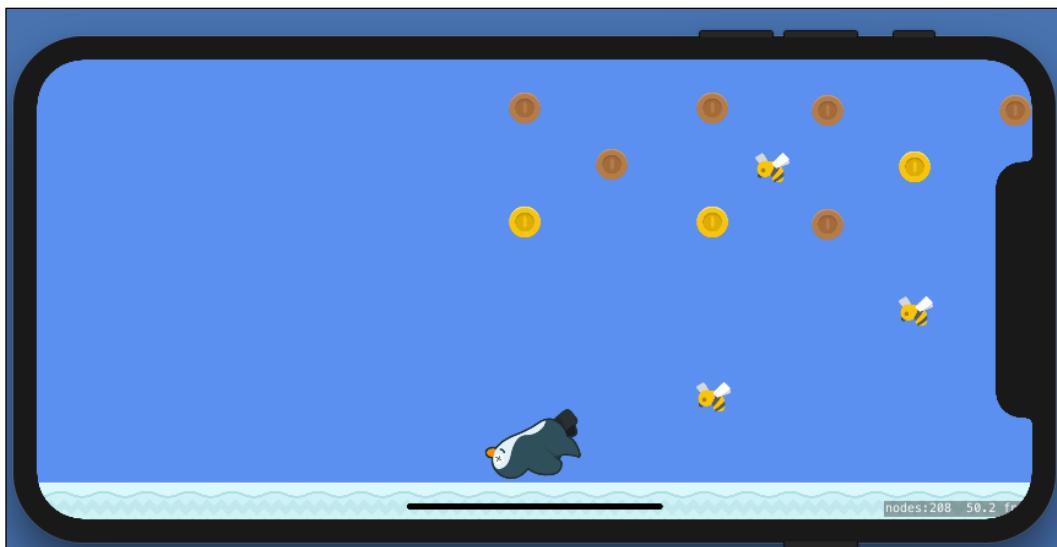
```
/* --- Create the death animation --- */
let startDie = SKAction.run {
    // Switch to the death texture with X eyes:
    self.texture =
        self.textureAtlas.textureNamed("pierre-dead")
    // Suspend the penguin in space:
    self.physicsBody?.affectedByGravity = false
    // Stop any movement:
    self.physicsBody?.velocity = CGVector(dx: 0, dy: 0)
}
```

Implementing Collision Events

```
let endDie = SKAction.run {
    // Turn gravity back on:
    self.physicsBody?.affectedByGravity = true
}

self.dieAnimation = SKAction.sequence([
    startDie,
    // Scale the penguin bigger:
    SKAction.scale(to: 1.3, duration: 0.5),
    // Use the waitForDuration action to provide a short pause:
    SKAction.wait(forDuration: 0.5),
    // Rotate the penguin on to his back:
    SKAction.rotate(toAngle: 3, duration: 1.5),
    SKAction.wait(forDuration: 0.5),
    endDie
])
```

Run the project and bump into three enemies. You will see the comedic death animation play, as shown in the following screenshot:



Poor Pierre Penguin! Good job implementing the damage and death animations. Next, we will handle coin collection with the coin contact event.

Collecting coins

As the main goal for the player, collecting coins should be one of the most enjoyable aspects of our game. We will create a rewarding animation when the player contacts a coin. Follow these steps to implement coin collection:

1. In `GameScene.swift`, add a new property to the `GameScene` class:

```
var coinsCollected = 0
```

2. In `Coin.swift`, add a new function to the `Coin` class named `collect`:

```
func collect() {
    // Prevent further contact:
    self.physicsBody?.categoryBitMask = 0
    // Fade out, move up, and scale up the coin:
    let collectAnimation = SKAction.group([
        SKAction.fadeAlpha(to: 0, duration: 0.2),
        SKAction.scale(to: 1.5, duration: 0.2),
        SKAction.move(by: CGVector(dx: 0, dy: 25),
                     duration: 0.2)
    ])
    // After fading it out, move the coin out of the way
    // and reset it to initial values until the encounter
    // system re-uses it:
    let resetAfterCollected = SKAction.run {
        self.position.y = 5000
        self.alpha = 1
        self.xScale = 1
        self.yScale = 1
        self.physicsBody?.categoryBitMask =
            PhysicsCategory.coin.rawValue
    }
    // Combine the actions into a sequence:
    let collectSequence = SKAction.sequence([
        collectAnimation,
        resetAfterCollected
    ])
    // Run the collect animation:
    self.run(collectSequence)
}
```

3. In `GameScene.swift`, call the new `collect` function from the coin contact case in the `didBegin` function:

```
case PhysicsCategory.coin.rawValue:  
    // Try to cast the otherBody's node as a Coin:  
    if let coin = otherBody.node as? Coin {  
        // Invoke the collect animation:  
        coin.collect()  
        // Add the value of the coin to our counter:  
        self.coinsCollected += coin.value  
        print(self.coinsCollected)  
    }
```

Great work! Run the project and try to collect some coins. You will see the coins perform their collection animation. The game will keep track of how many coins you are collecting and print the number to the console. The player cannot see that number yet; we will add a text counter on the game screen in the next chapter. Next, we will implement the Power-up Star game logic.

The Power-up Star logic

When the player contacts the Star, we will grant them invulnerability for a short time and give the player great speed to allow them to power through encounters. Follow these steps to implement the power-up:

1. In `Player.swift`, add a new function to the `Player` class, as shown here:

```
func starPower() {  
    // Remove any existing star power-up animation, if  
    // the player is already under the power of star  
    self.removeAction(forKey: "starPower")  
    // Grant great forward speed:  
    self.forwardVelocity = 400  
    // Make the player invulnerable:  
    self.invulnerable = true  
    // Create a sequence to scale the player larger,  
    // wait 8 seconds, then scale back down and turn off  
    // invulnerability, returning the player to normal:  
    let starSequence = SKAction.sequence([  
        SKAction.scale(to: 1.5, duration: 0.3),  
        SKAction.wait(forDuration: 8),  
        SKAction.scale(to: 1, duration: 1),  
        SKAction.run {  
            self.forwardVelocity = 200
```

```

        self.invulnerable = false
    }
])
// Execute the sequence:
self.run(starSequence, withKey: "starPower")
}

```

2. Invoke the new function from the GameScene class' didBegin function, under the power-up case:

```

case PhysicsCategory.powerup.rawValue:
    player.starPower()

```

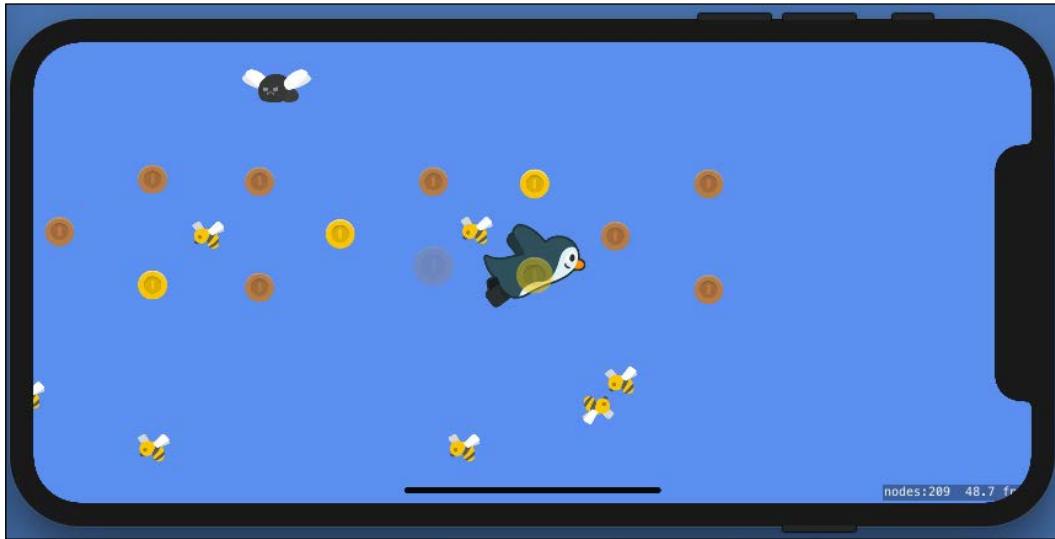
You may find it helpful to increase the spawn rate of the Power-up Star in order to test. Remember that we are generating a random number in the didSimulatePhysics function of GameScene to determine how often we spawn the star. To spawn the star more often, comment out the line that generates a random number and replace it with a hardcoded 0, as shown here (the new code is written in bold):

```

//let starRoll = Int(arc4random_uniform(10))
let starRoll = 0
if starRoll == 0 {

```

Great! Now, it will be easy to test the Power-up Star. Run the project and find a star. The penguin should scale to a large size and start charging forward, blowing enemies aside as he passes, as shown here:



Remember to change the star-spawning code back to a random number before you continue, or the star will spawn too often.



Checkpoint 7-B

The code up to this point is available in this chapter's code resources.



Summary

Our penguin game is looking great! You have brought the core mechanics to life by implementing the sprite contact events. You learned how SpriteKit handles collisions and contacts, used bitmasks to assign collision categories to different types of sprites, wired up a contact system in our penguin game, and added custom game logic for taking damage, collecting coins, and gaining the Power-up Star.

We have a playable game at this point; the next step is adding polish, menus, and features to make the game stand out. We will make our game shine by adding a HUD, background images, particle emitters, and more in *Chapter 8, Polishing to a Shine – HUD, Parallax Backgrounds, Particles, and More*.

8

Polishing to a Shine – HUD, Parallax Backgrounds, Particles, and More

Our core gameplay mechanics are in place; now we can improve the overall user experience. We will turn our focus to the non-gameplay features that make our games shine. To start, we will add a **heads-up display (HUD)** to display the player's health and coin count. Then, we will implement multiple layers of parallax background to add depth and immersion to the game world. We will also explore SpriteKit's particle system and use a particle emitter to add production value to the game. Combined, these steps will add to the fun of the gameplay experience, invite the player deeper into the game world, and impart a professional, polished feeling to our app.

The topics in this chapter include the following:

- Adding a HUD
- Parallax background layers
- Using the particle system
- Granting safety as the game starts

Adding a HUD

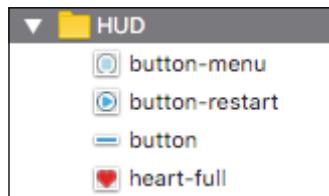
Our game needs a HUD to show the player's current health and coin score. We can use hearts to indicate health – as in classic games in the past – and draw text to the screen with `SKLabelNode` to display the number of coins collected. We will attach the HUD to the camera node, instead of to the scene itself, since it does not move as the player flies forward. We do not want to block the player's vision of upcoming obstacles to the right, so we will place the HUD elements in the top-left corner of the screen.

When we are finished, our HUD will look like this (after the player collects 110 coins and sustains one point of damage):



First, we need to add the HUD art assets into the game. Follow these steps to add the HUD textures:

1. In Xcode, open the `Assets.xcassets` file and create a new sprite atlas by right-clicking in the left pane and selecting **New Sprite Atlas**. Name this new atlas **HUD**.
2. Remove the empty example sprite that Xcode creates in the new atlas.
3. In the asset pack, locate the `HUD` folder and drag its textures into your new sprite atlas, as we have done in previous chapters. When you are finished, your `HUD` atlas should look like this:



Implementing the HUD

Now that we have our art available, follow these steps to implement the HUD:

1. We will create a `HUD` class to handle all of the `HUD` logic. Add a new Swift file to your project, `HUD.swift`, and add the following code to begin work on the `HUD` class:

```
import SpriteKit

class HUD: SKNode {
    var textureAtlas = SKTextureAtlas(named:"HUD")
    var coinAtlas = SKTextureAtlas(named:
        "Environment")
    // An array to keep track of the hearts:
    var heartNodes: [SKSpriteNode] = []
    // An SKLabelNode to print the coin score:
    let coinCountText = SKLabelNode(text: "000000")
}
```

2. We need an initializer-style function to create a new `SKSpriteNode` for each heart shape and to configure the new `SKLabelNode` for the coin counter. Add a function named `createHudNodes` to the `HUD` class, as follows:

```
func createHudNodes(screenSize: CGSize) {
    let cameraOrigin = CGPoint(
        x: screenSize.width / 2,
        y: screenSize.height / 2)
    // --- Create the coin counter ---
    // First, create and position a bronze coin icon:
    let coinIcon = SKSpriteNode(texture:
        coinAtlas.textureNamed("coin-bronze"))
    // Size and position the coin icon:
    let coinPosition = CGPoint(x:
        -cameraOrigin.x + 23, y: cameraOrigin.y - 23)
    coinIcon.size = CGSize(width: 26, height: 26)
    coinIcon.position = coinPosition
```

```
// Configure the coin text label:  
coinCountText.fontName = "AvenirNext-HeavyItalic"  
let coinTextPosition = CGPoint(x:  
    -cameraOrigin.x + 41, y: coinPosition.y)  
coinCountText.position = coinTextPosition  
// These two properties allow you to align the  
// text relative to the SKLabelNode's position:  
coinCountText.horizontalAlignmentMode =  
    SKLabelHorizontalAlignmentMode.left  
coinCountText.verticalAlignmentMode =  
    SKLabelVerticalAlignmentMode.center  
// Add the text label and coin icon to the HUD:  
self.addChild(coinCountText)  
self.addChild(coinIcon)  
  
// Create three heart nodes for the life meter:  
for index in 0 ..< 3 {  
    let newHeartNode = SKSpriteNode(texture:  
        textureAtlas.textureNamed("heart-full"))  
    newHeartNode.size = CGSize (width: 46,  
        height: 40)  
    // Position the hearts below the coins:  
    let xPos = -cameraOrigin.x +  
        CGFloat(index * 58) + 33  
    let yPos = cameraOrigin.y - 66  
    newHeartNode.position = CGPoint(x: xPos,  
        y: yPos)  
    // Keep track of nodes in an array property:  
    heartNodes.append(newHeartNode)  
    // Add the heart nodes to the HUD:  
    self.addChild(newHeartNode)  
}  
}
```

3. We also need a function that the GameScene class can call to update the coin counter label. Add a new function to the HUD class named `setCoinCountDisplay`, as follows:

```
func setCoinCountDisplay(newCoinCount: Int) {
    // We can use the NSNumberFormatter class to pad
    // leading 0's onto the coin count:
    let formatter = NumberFormatter()
    let number = NSNumber(value: newCoinCount)
    formatter.minimumIntegerDigits = 6
    if let coinStr =
        formatter.string(from: number) {
        // Update the label node with the new count:
        coinCountText.text = coinStr
    }
}
```

4. We will also need a function to update the heart graphic when the player's health changes. Add a new function to the HUD class named `setHealthDisplay`, as follows:

```
func setHealthDisplay(newHealth: Int) {
    // Create a fade SKAction to fade lost hearts:
    let fadeAction = SKAction.fadeAlpha(to: 0.2,
                                          duration: 0.3)
    // Loop through each heart and update its status:
    for index in 0 ..< heartNodes.count {
        if index < newHealth {
            // This heart should be full red:
            heartNodes[index].alpha = 1
        } else {
            // This heart should be faded:
            heartNodes[index].run(fadeAction)
        }
    }
}
```

5. Our `HUD` class is complete. Next, we will wire it up in the `GameScene` class. Open `GameScene.swift` and add a new property to the `GameScene` class, instantiating an instance of the `HUD` class:

```
let hud = HUD()
```

6. We need to add the `HUD` node to the `camera` node. Add this code at the bottom of the `GameScene didMove` function:

```
// Add the camera itself to the scene's node tree:  
self.addChild(self.camera!)  
  
// Position the camera node above the game elements:  
self.camera!.zPosition = 50  
  
// Create the HUD's child nodes:  
hud.createHudNodes(screenSize: self.size)  
  
// Add the HUD to the camera's node tree:  
self.camera!.addChild(hud)
```

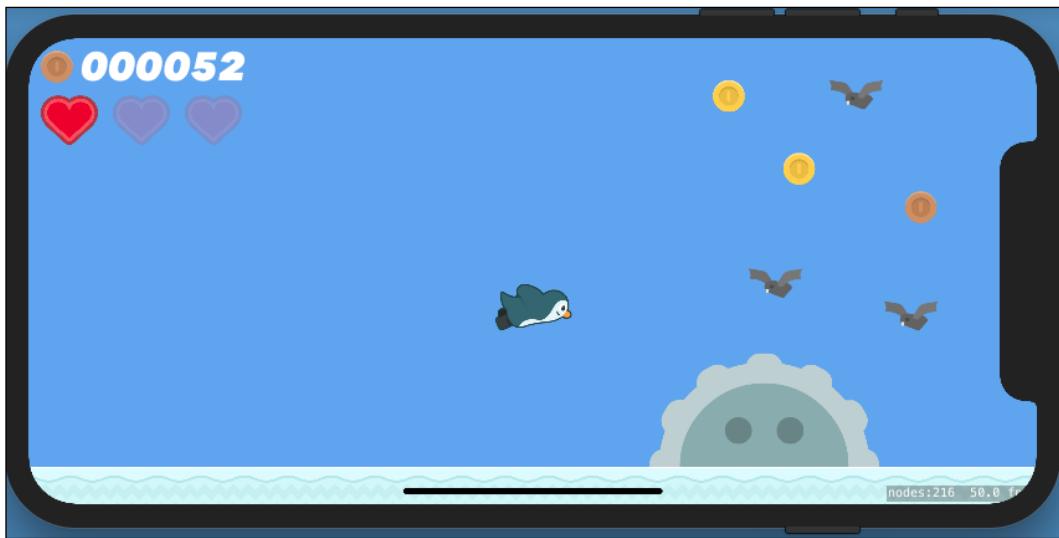
7. We are ready to send health and coin updates to the `HUD`. First, we will update the `HUD` with health updates when the player takes damage. Inside the `GameScene didBegin` function, locate the contact cases where the player takes damage—when he or she touches the ground or an enemy—and add this new code (in bold), which will send health updates to the `HUD`:

```
case PhysicsCategory.ground.rawValue:  
    player.takeDamage()  
    hud.setHealthDisplay(newHealth: player.health)  
case PhysicsCategory.enemy.rawValue:  
    player.takeDamage()  
    hud.setHealthDisplay(newHealth: player.health)
```

8. Finally, we will update the `HUD` whenever the player collects a coin. Locate the contact case where the player comes into contact with a coin and call the `HUD setCoinCountDisplay` function (the new code is in bold) as follows:

```
case PhysicsCategory.coin.rawValue:  
    // Try to cast the otherBody's node as a Coin:  
    if let coin = otherBody.node as? Coin {  
        coin.collect()  
        self.coinsCollected += coin.value  
        hud.setCoinCountDisplay(newCoinCount:  
            self.coinsCollected)  
    }
```

9. Run the project, and you should see your coin counter and health meter appear in the upper left-hand corner, as shown in the following screenshot:



Great job! Our HUD is complete. Next, we will build our background layers.

Parallax background layers

Parallax adds the feeling of depth to your game by drawing separate background layers and moving them past the camera at varying speeds. Very slow backgrounds give the illusion of distance, while fast-moving backgrounds appear to be close to the player. We can enhance this effect by painting faraway objects with increasingly desaturated colors.

In our game, we will achieve the parallax effect by attaching our backgrounds to the scene, then slowly pushing the backgrounds to the right as the camera pans right. As the camera moves to the right (making the children of the scene appear to move left), we will move the background's *x* position to the right, so that the total background node movement is less than for the other children of the scene. The result will be background layers that appear to move more slowly than the rest of our game, and thus appear farther away.

In addition, each background will only be 3,072 points wide, but will jump forward at precise intervals to loop seamlessly, in a similar way to the `Ground` class.

Adding the background assets

First, add the art by following these steps:

1. Open your project's `Assets.xcassets` file in Xcode and create a new **Sprite Atlas** named `Backgrounds`. You can delete the empty sample sprite that Xcode creates in this new atlas.
2. In the provided game assets, locate the background images in the `Backgrounds` folder.
3. Drag and drop the eight background textures into the new `BackgroundsSprite` atlas.

You should see the backgrounds appear in the left pane, as shown here:



Implementing a background class

We will need a new class to manage the repositioning logic for parallax and seamless looping. We can instantiate a new instance of a `Background` class for each background layer. To create the `Background` class, add a new Swift file, `Background.swift`, to your project, using the following code:

```
import SpriteKit

class Background: SKSpriteNode {
    // movementMultiplier will store a float from 0-1 to indicate
    // how fast the background should move past.
    // 0 is full adjustment, no movement as the world goes past
    // 1 is no adjustment, background passes at normal speed
    var movementMultiplier = CGFloat(0)
    // jumpAdjustment will store how many points of x position
    // this background has jumped forward, useful for calculating
    // future seamless jump points:
    var jumpAdjustment = CGFloat(0)
    // A constant for background node size:
    let backgroundSize = CGSize(width: 1024, height: 768)
    // Store the Backgrounds texture:
```

```
var textureAtlas = SKTextureAtlas(named: "Backgrounds")

func spawn(parentNode:SKNode, imageName:String,
          zPosition:CGFloat, movementMultiplier:CGFloat) {
    // Position from the bottom left:
    self.anchorPoint = CGPoint.zero
    // Start backgrounds at the top of the ground (y: 30)
    self.position = CGPoint(x: 0, y: 30)
    // Control the order of the backgrounds with zPosition:
    self.zPosition = zPosition
    // Store the movement multiplier:
    self.movementMultiplier = movementMultiplier
    // Add the background to the parentNode:
    parentNode.addChild(self)
    // Grab the texture for this background from the atlas:
    let texture = textureAtlas.textureNamed(imageName)

    // Build three child node instances of the texture,
    // Looping from -1 to 1 so the backgrounds cover both
    // forward and behind the player at position zero.
    // closed range operator: "..." includes both endpoints:
    for i in -1...1 {
        let newBGNODE = SKSpriteNode(texture: texture)
        // Set the size for this node from constant:
        newBGNODE.size = backgroundSize
        // Position these nodes by their lower left corner:
        newBGNODE.anchorPoint = CGPoint.zero
        // Position this background node:
        newBGNODE.position = CGPoint(
            x: i * Int(backgroundSize.width), y: 0)
        // Add the node to the Background:
        self.addChild(newBGNODE)
    }
}

// We will call updatePosition every frame to
// reposition the background:
func updatePosition(playerProgress: CGFloat) {
    // Calculate a position adjustment after loops and
    // parallax multiplier:
    let adjustedPosition = jumpAdjustment + playerProgress *
        (1 - movementMultiplier)
    // Check if we need to jump the background forward:
    if playerProgress - adjustedPosition >
```

```
        backgroundSize.width {
            jumpAdjustment += backgroundSize.width
        }
        // Adjust this background position forward as the camera
        // pans so the background appears slower:
        self.position.x = adjustedPosition
    }
}
```

Wiring up backgrounds in the GameScene class

We need to make three code additions to the GameScene class to wire up our backgrounds. First, we will create an array to keep track of the backgrounds. Next, we will spawn the backgrounds as the scene begins. Finally, we can call the Background class' updatePosition function from the GameScene didSimulatePhysics function to reposition the backgrounds before every frame. Follow these steps to wire up the backgrounds:

1. Create a new array property on the GameScene class itself to store our backgrounds, as shown here:

```
var backgrounds: [Background] = []
```

2. At the bottom of the didMove function, instantiate and spawn our three backgrounds:

```
// Instantiate three Backgrounds to the backgrounds array:
for _ in 0..<3 {
    backgrounds.append(Background())
}

// Spawn the new backgrounds:
backgrounds[0].spawn(parentNode: self,
    imageName: "background-front", zPosition: -5,
    movementMultiplier: 0.75)
backgrounds[1].spawn(parentNode: self,
    imageName: "background-middle", zPosition: -10,
    movementMultiplier: 0.5)
backgrounds[2].spawn(parentNode: self,
    imageName: "background-back", zPosition: -15,
    movementMultiplier: 0.2)
```

3. Lastly, add the following code at the bottom of the `didSimulatePhysics` function to reposition the backgrounds before each frame:

```
// Position the backgrounds:  
for background in self.backgrounds {  
    background.updatePosition(playerProgress:  
        playerProgress)  
}
```

4. Run the project. You should see the three background images as separate layers behind the action, moving past with a parallax effect. This screenshot shows the backgrounds as they should appear in your game:



If you are using the iOS simulator to test your game, it is normal to experience a lower frame rate after adding these large background textures to the game. The game will still run well on physical iOS devices.

Excellent! You have successfully implemented your background system. The background makes Pierre Penguin's world feel full, adding immersion to the game. Next, we will use a particle emitter to add a trail behind Pierre – a fun addition that helps the player master the controls.



Checkpoint 8-A

The code up to this point is available in this chapter's code resources.



Using the particle system

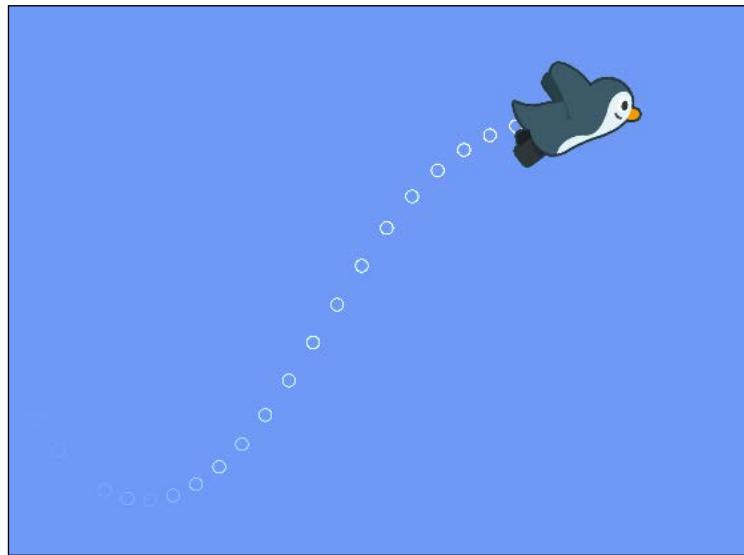
SpriteKit includes a powerful particle system that makes it easy to add exciting graphics to your game. Particle emitter nodes create many small instances of an image that are combined to create a great-looking effect. A very easy example of a particle system is Rain. Rain is a particle system, and each rain drop is a particle, and a cloud has a lot of emitters from where the droplets or particles are created.

We will create a particle system instead of just creating individual particles, because with a particle system you can create different kinds of effects using the same particle. For example, we saw Rain, which is a particle system; what if we wanted another effect, such as water coming out of the faucet? Here, the particle is the same—a water droplet—but a rain droplet behaves differently. Here, water is falling from the faucet; each drop falls with a force and is created with a single emitter—the faucet outlet. So, we can change the particle system to have one emitter and give the particles an initial downward force; this way, we will have the same particle behaving differently instead of coding the system from scratch again.

In SpriteKit, like any other framework, each particle is an image that is controlled by a particle system having one or more emitters. An emitter controls the spawning, movement, and destruction of the particle system. You can use emitter nodes to generate snow, fire, sparks, explosions, magic, and other useful effects that would otherwise require a lot of effort.

For our game, you will learn how to use an emitter node to create a trail of small dots behind Pierre Penguin as he flies, making it easier for the player to learn how their taps influence Pierre's flight path.

When we are finished, Pierre's dot trail will look something like this:

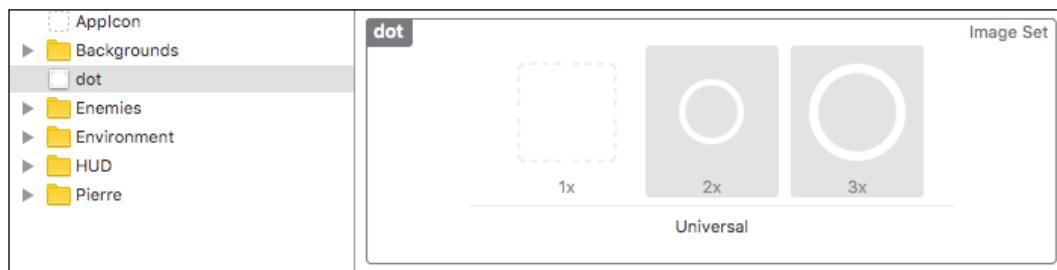


Adding the circle particle asset

Each particle system emits multiple versions of a single image in order to create a cumulative particle effect. In our case, the image is a simple circle. To add the circle image to the game, follow these steps:

1. Open the `Assets.xcassets` file in Xcode.
2. Locate the `dot@3x.png` and `dot@2x.png` images in the `Particles` folder of the downloadable game assets.
3. Drag and drop the image files into the left pane of `Assets.xcassets`.

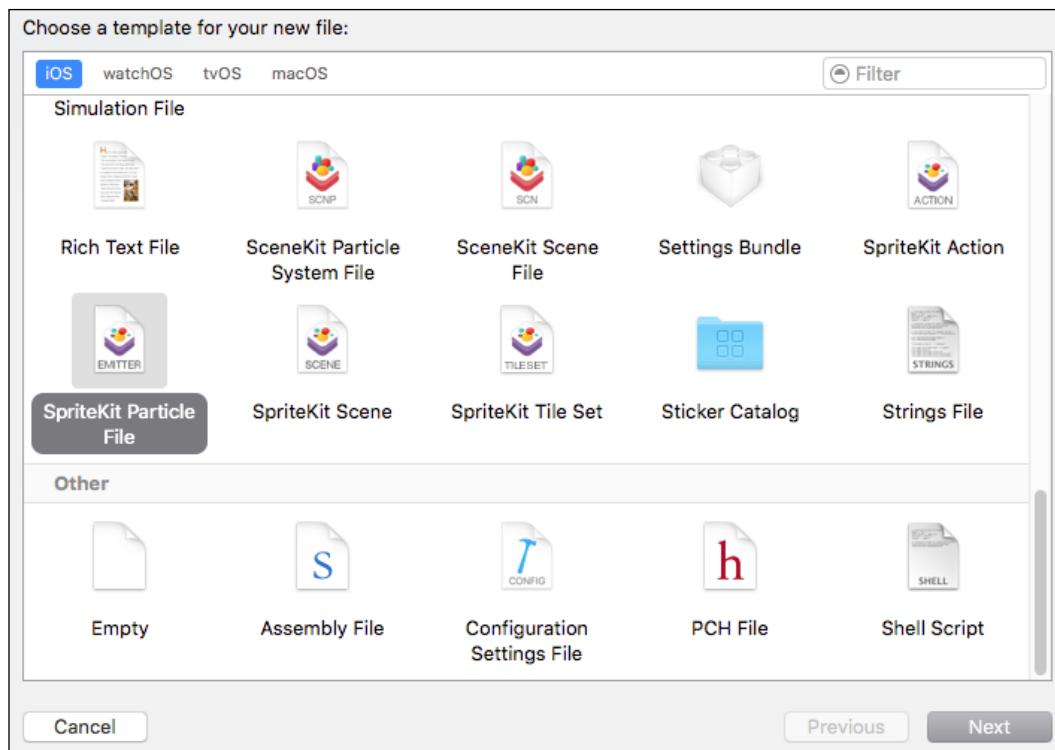
You should see the dot sprite appear in both sizes in the right pane, as shown here:



Creating a SpriteKit particle file

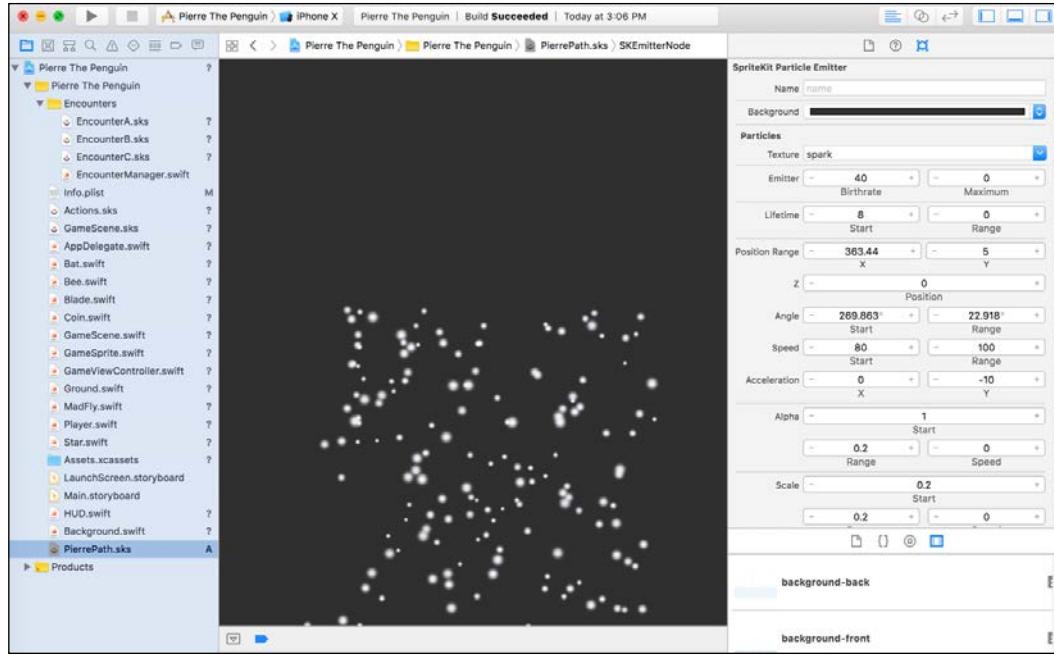
Xcode provides an excellent UI for creating and editing particle systems. To use the UI, we will add a new SpriteKit particle file to our project. Follow these steps to add the new file:

1. Start by adding a new file to your project and locating the **SpriteKit Particle File** type. You can find this template under the **Resource** category, as shown here:



2. In the following prompt, select **Snow** as the **Particle Template**.
3. Name the file `PierrePath.sks` and click **Create** to add the new file to your project.

Xcode will open the new particle emitter in the main frame, which should look something like this:

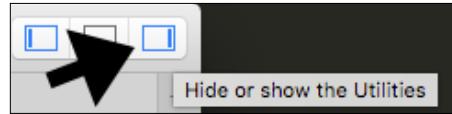


Previewing the Snow template in Xcode's particle editor

At the time of writing this book, Xcode's particle editor remains quirky. If you do not see the white snow particle effect in the middle, try clicking anywhere in the dark gray center area to reposition the particle emitter—occasionally, it does not start where expected.

This is also useful for testing setting changes without overlap from old particles—simply click anywhere in the editor to reposition the emitter.

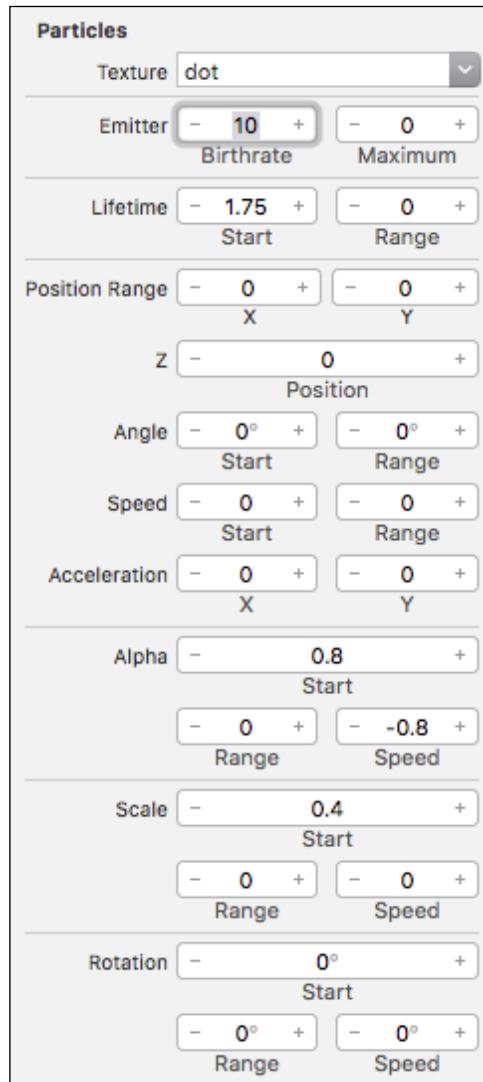
Make sure that you have the right-hand sidebar turned on by lighting up the Utilities button in the top-right corner of Xcode, as shown here:



You can use the Utilities sidebar to edit the animation qualities of the particle emitter. You can edit several properties: the number of particles, the lifetime of a particle, how fast the particles move, how they scale up or down, and so on. This is a fantastic tool because you can see immediate feedback from your changes.

Configuring the path particle settings

To create Pierre's dot trail, update your particle settings to match the settings shown in this screenshot:



Let's go through each of these basic parameters to understand what each of these variables do and how, by changing each of the parameters, the behaviors of the particle system can be changed.

Name

If you want to refer to the particle system by name in code, we can give a name here so that we can refer to it later. This is similar to how we gave a name to the `enemyNode` to check whether the node passed into the `movingSprite` class was the enemy, and then we run a certain function based on that information.

Background

This is the color of the background. Changing this doesn't affect the particle itself. This is purely for the purpose of visibility. If our particles are black, we can change the background to white so that we can see clearly how the particle looks and behaves.

Texture

This is the texture or image that will be displayed for each particle. Currently, `spark.png` is used as the texture. We can change this to the enemy, bullet, rocket, or hero image if we wanted to.

Emitter

This controls the rate at which particles are emitted and how many particles we want the emitter to emit. To control the rate, we can increase or decrease the `Birthrate` parameter. We can decrease the rate of emission to decrease the value of `Birthrate`, and if we want the particles to be emitted more quickly, we would increase it. To cap the number of particles, we change the maximum value to the number of particles we want the emitter to emit. If we want the emitter to continually emit particles, we keep the value at 0.

Lifetime

When a particle is created, Lifetime decides how long it stays on the screen before getting deleted. Here, each particle stays on the screen for 1.75 seconds. Range is used to add some randomness to the particles' behavior. If we change this range value to 1, then some particles will be on the screen for 2.0 seconds and others for 3.0 seconds before being deleted. So, the random value created is plus or minus half of the range value in addition to the initial value.

How range works is that it takes the first value and then adds or subtracts half of the range value to get the final value. In this way, it will look as if each particle has a different lifetime as in real life; not all particles behave the same.

Position Range

The default position at the beginning is the center of the screen. By looking at the Range keyword in the name of the variable itself, you would have guessed that the value that we are inputting is a range value.

Here, the value is set to 0 so that the particle is spawned in the center. If the x value is 55.65, it means that a particle is spawned anywhere between -27.825 and +27.825 in the x direction. The emission point is denoted by a small green dot in the editor view. The y value is also 0, as we want to spawn the particle in the center along the y direction. If the y value is 5, it means that the particle will be generated anywhere between -2.5 and +2.5 in the y direction.

If you set the x and y values to 0, you will see that all particles are emitted from the green dot, which is what we want.

Angle

This determines the angle at which the particles are emitted. Since we don't want the particle to be emitted at an angle, we set it to 0.

If we want the particles to start moving up, the angle is around 90 degrees. You can change this value to 45, which would make it look as if wind was acting on the particle. A range is applied to add some randomness to the initial angle. Otherwise, all particles would be going straight up, which would look very unnatural.

Speed

This is the speed with which the particles will start moving when they are emitted. Here, they start moving at an average speed of 0.

There is also a range. When it is applied, it will emit particles with different initial speeds depending on the range applied.

Acceleration

We can accelerate a particle in the x and y direction, such as in the case of a jet engine or a comet. To create this effect, you can change the x acceleration range to 5 and increase the y acceleration to around 500. Otherwise, set it to 0 in the x and y direction, which is what we want here.

Alpha

This defines the level of opacity of each particle. If the alpha value is zero, then the particle will be completely transparent, and 1 means that it will be completely opaque. There is also a range value that you can specify. The speed parameter determines the rate at which the alpha value of each particle changes per second. So, a particle is opaque when it is created, and over a period of time it becomes transparent, as the value of the alpha is reduced over a period of time. Here, we keep it constant at 0.8.

Scale

Similar to alpha, the scale value ranges from 0 to 1. At zero, the image shrinks to nothing, and at 1 it is at its original size. At 0.5, the object will be half its original size, and at 2.0 it will be double its original size in both the x and y direction. You can also specify a range and speed if you want the particle to grow or shrink during its lifetime. The speed will determine the speed at which the particle is scaled.

Since we just want the object to be smaller than its original size and have a constant value, the scale is set to 0.4, with the rest of the values set as 0.

Rotation

As soon as a particle is created, we can make it rotate by giving it a start value and range to generate random rotation speeds. We can also increase or decrease the speed of the rotation over a period of time by changing the speed parameter. In our case, all values are set to 0.

Color Blend

This is used to blend from one color to other. Here, we just keep the default values.

Color Ramp

We can use Color Ramp to specify the color of the particle to be generated. The particle texture image is always white so that we can change the color of the object in code at will.

We can also assign different colors to the particle at different stages of its life. In this case, we will keep the default values.

The remaining four variables, Blend Mode, Field Mask, Custom Shader, and Custom Shader Uniform, pertain to shaders and shader programming, which is beyond the scope of this book. But you can use shaders to create custom effects and behaviors for the particles.

You can play around with Blend Mode a little if you know how it works, as it is very similar to what you see in Photoshop. If you know Photoshop, you will be familiar with add, subtract, multiply, screen, replace, and alpha. You can select each of these and see the effect they have on the particle system.

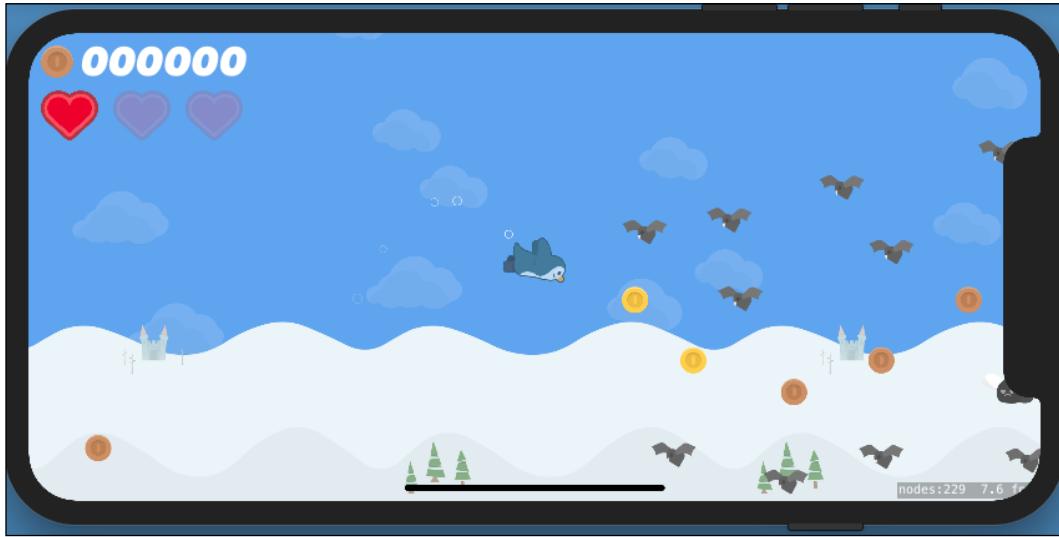
That is all the information required for now to design the particle system for our game. You have the correct settings when your editor shows a tiny white circle with no apparent movement.

Adding the particle emitter to the game

We will attach our new emitter to the `Player` node, so the emitter will create new white circles wherever the player flies. We can easily reference the emitter design we just created in the editor from our code. Open `GameScene.swift` and add this code at the bottom of the `didMove` function:

```
// Instantiate a SKEmitterNode with the PierrePath design:  
if let dotEmitter = SKEmitterNode(fileNamed: "PierrePath") {  
    // Position the penguin in front of other game objects:  
    player.zPosition = 10  
    // Place the particle zPosition behind the penguin:  
    dotEmitter.particleZPosition = -1  
    // By adding the emitter node to the player, the emitter moves  
    // with the penguin and emits new dots wherever the player is  
    player.addChild(dotEmitter)  
    // However, the particles themselves should target the scene,  
    // so they trail behind as the player moves forward.  
    dotEmitter.targetNode = self  
}
```

Run the project. You should see the white dots trailing behind Pierre, as shown here:



Good work. Now, the player can see where they have flown, which is both fun and instructive. The feedback from the dots will help the player learn the sensitivity of the control system, and thus master the game more quickly.

This is just one of many special effects you can create with particle emitter nodes. You can explore other creative possibilities now that you know how to create, edit, and place particle emitters in the world. Other fun ideas include sparks when Pierre bumps into enemies, and gentle snow falling in the background. We will use particles again later, when we add exploding crates to the game.

Granting safety as the game starts

You may have noticed that Pierre Penguin quickly falls to the ground as soon as you launch the game, which is not much fun. Instead, we can launch Pierre into a graceful looping arc as the game starts to give the player a moment to prepare for flight. To do so, open `Player.swift` and add this code at the bottom of the `init` function:

```
// Grant a momentary reprieve from gravity:  
self.physicsBody?.affectedByGravity = false  
// Add some slight upward velocity:  
self.physicsBody?.velocity.dy = 80
```

```
// Create a SKAction to start gravity after a small delay:  
let startGravitySequence = SKAction.sequence([  
    SKAction.wait(forDuration: 0.6),  
    SKAction.run {  
        self.physicsBody?.affectedByGravity = true  
    }])  
self.run(startGravitySequence)
```



Checkpoint 8-B

The code up to this point is available in this chapter's code resources.



Summary

We brought the game world to life in this chapter. We drew a HUD to show the player their remaining health and coin score, added parallax backgrounds to increase the depth and immersion of the world, and learned how to use particle emitters to create special graphics in our games. In addition, we added a small delay before gravity drags our hero down at the beginning of each flight. Our game is fun and looking great!

Next, we need a menu so that we can restart the game without rebuilding the project or manually closing the application. In *Chapter 9, Adding Menus and Sounds*, we will design a start menu, add a retry button when the player dies, and play sounds and music to create a deeper gameplay experience.

9

Adding Menus and Sounds

It is easy to overlook menu design, but the menu provides your game's first impression to the player. When used correctly, your menus reinforce the brand of your game and provide a pleasant break in the action that retains the player between gameplay tries. We will add two menus in this chapter: a main menu that shows when the game starts, and a retry menu that appears when the player loses a game.

Likewise, immersive sounds are vital to a great game. Sound is your opportunity to support the mood of the game world and emphasize key gameplay mechanics, such as coin collecting and taking damage. Additionally, every fun game deserves addictive background music! We will add background music and sound effects in this chapter to complete the mood of the game world.

The topics in this chapter include the following:

- Building the main menu scene
- Adding the restart game menu
- Adding music with AVAudio
- Playing sound effects with SKAction

Building the main menu

We can use SpriteKit components to build our main menu. We will create a new scene in a new file for our main menu, and then use code to place a background sprite node, a logo text node, and button sprite nodes. Let's start by adding the menu scene to the project and building out the nodes.

Creating the menu scene and menu nodes

To create the menu scene, follow these steps:

1. You already added the background texture assets in *Chapter 8, Polishing to a Shine – HUD, Parallax Backgrounds, Particles, and More*. To double-check, open Assets.xcassets and locate the Backgrounds Sprite Atlas. You should have a sprite called background-menu with the background textures for the menu scene. If not, you can find these two textures in the Backgrounds folder of the asset bundle.
2. Add a new Swift file to your project named MenuScene.swift.
3. Add the following code to create the MenuScene scene class:

```
import SpriteKit

class MenuScene: SKScene {
    // Grab the HUD sprite atlas:
    let textureAtlas: SKTextureAtlas =
        SKTextureAtlas(named: "HUD")
    // Instantiate a sprite node for the start button
    // (we'll use this in a moment):
    let startButton = SKSpriteNode()

    override func didMove(to view: SKView) {
    }
}
```

4. Next, we need to configure a few scene properties. Add this code inside the new scene's didMove function:

```
// Position nodes from the center of the scene:
self.anchorPoint = CGPoint(x: 0.5, y: 0.5)
// Add the background image:
let backgroundImage = SKSpriteNode(imageNamed:
    "background-menu")
backgroundImage.size = CGSize(width: 1024, height: 768)
backgroundImage.zPosition = -1
self.addChild(backgroundImage)
```

5. We need to draw the name of the game near the top of the menu. Add this code at the bottom of the didMove function to draw Pierre Penguin Escapes the Antarctic:

```
// Draw the name of the game:
let logoText = SKLabelNode(fontNamed: "AvenirNext-Heavy")
logoText.text = "Pierre Penguin"
```

```
logoText.position = CGPoint(x: 0, y: 100)
logoText.fontSize = 60
self.addChild(logoText)
// Add another line below:
let logoTextBottom = SKLabelNode(fontNamed:
    "AvenirNext-Heavy")
logoTextBottom.text = "Escapes the Antarctic"
logoTextBottom.position = CGPoint(x: 0, y: 50)
logoTextBottom.fontSize = 40
self.addChild(logoTextBottom)
```

6. Now, we will add the start button. The start button is the combination of an `SKSpriteNode` for the button graphic and an `SKLabelNode` for the "START GAME" text. Add this code at the bottom of the `didMove` function to create the button:

```
// Build the start game button:
startButton.texture = textureAtlas.textureNamed("button")
startButton.size = CGSize(width: 295, height: 76)
// Name the start node for touch detection:
startButton.name = "StartBtn"
startButton.position = CGPoint(x: 0, y: -20)
self.addChild(startButton)

// Add text to the start button:
let startText = SKLabelNode(fontNamed:
    "AvenirNext-HeavyItalic")
startText.text = "START GAME"
startText.verticalAlignmentMode = .center
startText.position = CGPoint(x: 0, y: 2)
startText.fontSize = 40
// Name the text node for touch detection:
startText.name = "StartBtn"
startText.zPosition = 5
startButton.addChild(startText)
```

7. Finally, we will make the start button text pulse in and out to add movement and excitement to the menu. Add this code at the bottom of the `didMove` function to fade the text in and out:

```
// Pulse the start text in and out gently:
let pulseAction = SKAction.sequence([
    SKAction.fadeAlpha(to: 0.5, duration: 0.9),
    SKAction.fadeAlpha(to: 1, duration: 0.9),
])
startText.run(SKAction.repeatForever(pulseAction))
```

Great work! We have created our `MenuScene` class and added all the nodes we need to build the menu. Next, we will update our app to start with the menu instead of going directly to the `GameScene` class.

Launching the main menu when the game starts

So far, our app launches directly to the `GameScene` class whenever it starts. We will now update our view controller to start with the `MenuScene` class instead. Follow these steps to launch the menu when the game starts:

1. Open `GameViewController.swift` and locate the `viewWillLayoutSubviews` function.
2. Replace the entire `viewWillLayoutSubviews` function with this code:

```
override func viewWillLayoutSubviews() {  
    super.viewWillLayoutSubviews()  
  
    // Build the menu scene:  
    let menuScene = MenuScene()  
    let skView = self.view as! SKView  
    // Ignore drawing order of child nodes  
    // (This increases performance)  
    skView.ignoresSiblingOrder = true  
    // Size our scene to fit the view exactly:  
    menuScene.size = view.bounds.size  
    // Show the menu:  
    skView.presentScene(menuScene)  
}
```

Run the project. You should see the app start with your new main menu, which looks something like this screenshot:



Terrific work! Next, we will wire up the **START GAME** button to transition to the GameScene class.

Wiring up the **START GAME** button

Just like in GameScene, we will add a touchesBegan function to the MenuScene class to capture touches on the **START GAME** button. To implement touchesBegan, open `MenuScene.swift` and, at the bottom of the class, add a new function named `touchesBegan`, as shown here:

```
override func touchesBegan(_ touches: Set<UITouch>, with event: UIEvent?) {
    for touch in (touches) {
        // Find the location of the touch:
        let location = touch.location(in: self)
        // Locate the node at this location:
        let nodeTouched = atPoint(location)
        if nodeTouched.name == "StartBtn" {
            // Player touched the start text or button node
            // Switch to an instance of the GameScene:
            self.view?.presentScene(GameScene(size: self.size))
        }
    }
}
```

Run the project and tap the **Start** button. The game should switch to the `GameScene` class, and gameplay will begin. Congratulations, you have successfully implemented your first main menu in SpriteKit! Next, we will add a simple restart menu that appears on top of `GameScene` when the player dies.

Adding the restart game menu

The restart menu is even simpler to implement. Rather than creating a new scene, we can extend our existing `HUD` class to display a restart button when the game ends. We will also include a smaller button to return the player to the main menu. This menu will appear on top of the action when the player dies.

Extending the HUD

First, we need to create and draw our new button nodes in the `HUD` class. Follow these steps to add the nodes:

1. Open the `HUD.swift` file and add two new properties to the `HUD` class, as follows:

```
let restartButton = SKSpriteNode()  
let menuButton = SKSpriteNode()
```

2. Add the following code at the bottom of the `createHudNodes` function:

```
// Add the restart and menu button textures to the nodes:  
restartButton.texture =  
    textureAtlas.textureNamed("button-restart")  
menuButton.texture =  
    textureAtlas.textureNamed("button-menu")  
// Assign node names to the buttons:  
restartButton.name = "restartGame"  
menuButton.name = "returnToMenu"  
menuButton.position = CGPoint(x: -140, y: 0)  
// Size the button nodes:  
restartButton.size = CGSize(width: 140, height: 140)  
menuButton.size = CGSize(width: 70, height: 70)
```

We purposefully did not add these nodes as children of the `HUD` yet, so they will not appear on the screen until we are ready. Next, we will add a function to make the buttons appear. We will call this function from the `GameScene` class when the player dies. Add a function named `showButtons` to the `HUD` class, as shown here:

```
func showButtons() {  
    // Set the button alpha to 0:  
    restartButton.alpha = 0
```

```
menuButton.alpha = 0
// Add the button nodes to the HUD:
self.addChild(restartButton)
self.addChild(menuButton)
// Fade in the buttons:
let fadeAnimation =
    SKAction.fadeAlpha(to: 1, duration: 0.4)
restartButton.run(fadeAnimation)
menuButton.run(fadeAnimation)
}
```

Wiring up GameScene for game over

We need to tell the `HUD` class to show the restart and main menu buttons once the player runs out of health. Open `GameScene.swift` and add a new function to the `GameScene` class named `gameOver`, as shown here:

```
func gameOver() {
    // Show the restart and main menu buttons:
    hud.showButtons()
}
```

That is all for now; we will add to the `gameOver` function in the next chapter, when we implement a high score system.

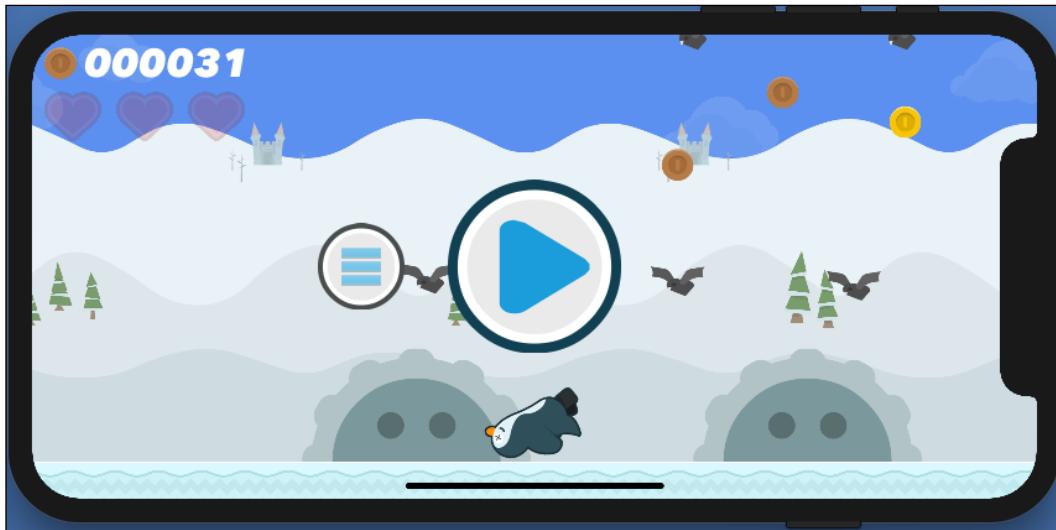
Informing the GameScene class when the player dies

So far, the `GameScene` class is oblivious to whether the player is alive or dead. We need to change that in order to use our new `gameOver` function. Open `Player.swift`, locate the `die` function, and add the following code at the bottom of the function:

```
// Alert the GameScene:
if let gameScene = self.parent as? GameScene {
    gameScene.gameOver()
}
```

We access `GameScene` by traveling up the node tree. The `Player` node's parent is the `GameScene` class.

Run the project and die. You should see the two new buttons appear after death, as shown here:



Good work. The buttons are displaying properly, but nothing happens yet when we tap on them. To complete our restart menu, we simply need to implement tap events for the two new buttons in the `GameScene` class's `touchesBegan` function.

Implementing touch events for the restart menu

Now that our buttons are displaying, we can add touch events in the `GameScene` class that are similar to the `START GAME` button in the `MenuScene` class.

To add the touch events, open `GameScene.swift` and locate the `touchesBegan` function. We will add the restart menu code at the bottom of the `for` loop. I am including the entire `touchesBegan` function in the following code, with new additions in bold:

```
override func touchesBegan(_ touches: Set<UITouch>, with event:  
    UIEvent?) {  
    for touch in (touches) {  
        // Find the location of the touch:  
        let location = touch.location(in: self)  
        // Locate the node at this location:  
        let nodeTouched = atPoint(location)
```

```

// Attempt to downcast the node to the GameSprite protocol
if let gameSprite = nodeTouched as? GameSprite {
    // If this node adheres to GameSprite, call onTap:
    gameSprite.onTap()
}

// Check for HUD buttons:
if nodeTouched.name == "restartGame" {
    // Transition to a new version of the GameScene
    // to restart the game:
    self.view?.presentScene(
        GameScene(size: self.size),
        transition: .crossFade(withDuration: 0.6))
}
else if nodeTouched.name == "returnToMenu" {
    // Transition to the main menu scene:
    self.view?.presentScene(
        MenuScene(size: self.size),
        transition: .crossFade(withDuration: 0.6))
}
}

player.startFlapping()
}

```

To test your new menu, run the project and run out of health on purpose. You should now be able to start a new game when you die, or transition back to the main menu with a tap on the menu button. Great! You have completed the two basic menus required for every game.

These simple steps go a long way toward the overall completion of the game, and the penguin game is looking terrific. Next, we will add event sounds and music to complete the game world.



Checkpoint 9-A

The code up to this point is available in this chapter's code resources.

Adding music with AVAudio

SpriteKit and Swift make it very easy to play sounds in our games. We can drag sound files into our project, just like image assets, and trigger playback with `SKActionplaySoundFileNamed`.

We can also use the `AVAudio` class from the `AVFoundation` framework for more precise audio control. We will use `AVAudio` to play our background music.

Adding sound assets to the game

Locate the `Sound` directory in the `Assets` folder and add it to your project by dragging and dropping it into the project navigator. Once you are done, you should see the `Sound` folder show up in your project, just like any other file.

Playing background music

First, we will add the background music. We want our music to play, regardless of which scene the player is currently looking at, so we will play the music from the view controller itself. To play the music, follow these steps:

1. Open `GameViewController.swift` and add the following `import` statement at the very top, just below the existing import lines, to allow us access to the `AVFoundation` classes:

```
import AVFoundation
```

2. Locate the `GameViewController` class and add the following property to store our `AVAudioPlayer`:

```
var musicPlayer = AVAudioPlayer()
```

3. At the very bottom of the `viewWillLayoutSubviews` function, add this code to play and loop the music:

```
// Start the background music:  
if let musicPath = Bundle.main.path(forResource:  
    "Sound/BackgroundMusic.m4a", ofType: nil) {  
    let url = URL(fileURLWithPath: musicPath)  
  
    do {  
        musicPlayer = try AVAudioPlayer(contentsOf: url)  
        musicPlayer.numberOfLoops = -1  
        musicPlayer.prepareToPlay()  
        musicPlayer.play()  
    }  
    catch { /* Couldn't load music file */ }  
}
```

Run the project. You should hear the background music as soon as the app starts. The music should continue as you move from the main menu to the game and back.

If you cannot hear the music, double-check that you added the Sound assets to your project, as mentioned previously. Also, Xcode does not get along with external speakers. Try using your Mac's internal speakers or demo your game on a physical iOS device to troubleshoot missing sounds.

Playing sound effects

Playing simple sounds is even easier. We will use `SKAction` objects to play sounds on specific events, such as when picking up a coin or starting the game.

Adding the coin sound effect to the Coin class

First, we will add a happy sound each time the player collects a coin. To add the coin sound effect, follow these steps:

1. Open `Coin.swift` and add a new property to the `Coin` class to cache a coin sound action:

```
let coinSound =  
    SKAction.playSoundFileNamed("Sound/Coin.aif",  
        waitForCompletion: false)
```

2. Locate the `collect` function and add the following line at the bottom of the function to play the sound:

```
// Play the coin sound:  
self.run(coinSound)
```

That is all you need to do to play the coin sound every time the player collects a coin. You can run the project now to test it out if you like.

To avoid memory-based crashes, it is important to cache each `playSoundFileNamed` action object and rerun the same object each time you want to play a sound, rather than creating a new instance of an `SKAction` object for each playback.

Adding the power-up and hurt sound effects to the Player class

We will play an excited sound when the player finds the star power-up and an injury noise when the player takes damage. Follow these steps to implement the sounds:

1. Open `Player.swift` and add two new properties to the `Player` class to cache the sound effects:

```
let powerupSound =  
    SKAction.playSoundFileNamed("Sound/Powerup.aif",  
        waitForCompletion: false)  
let hurtSound =  
    SKAction.playSoundFileNamed("Sound/Hurt.aif",  
        waitForCompletion: false)
```

2. Find the `takeDamage` function and add this line at the bottom:

```
// Play the hurt sound:  
self.run(hurtSound)
```

3. Find the `starPower` function and add this line at the bottom:

```
// Play the powerup sound:  
self.run(powerupSound)
```

Playing sound effects with SKAction

Lastly, we will play a sound when the game starts. Follow these steps to add this sound:

1. Open `GameScene.swift`. We will play this sound effect from the `didMove` function. Normally, it is vital to cache sound actions in a property, but we do not have to cache the game start sound because we will only play it once per scene load.
2. Add this line at the bottom of the `GameScene didMove` function:

```
// Play the start sound:  
self.run(SKAction.playSoundFileNamed("Sound/StartGame.  
aif",  
    waitForCompletion: false))
```

Great! We have added all the sound effects for our game. You can now run the project to test out each sound.

Adding a mute button and volume slider

We have the sounds working, but let's add some features so that the user can mute the background music if he or she doesn't want it to be playing all the time. We will also add in a volume slider for the background music so that the user can increase or decrease the volume to their liking.

To do this, let's create a new class called `BackgroundMusic.swift`. We will use this class to add the required features.

Since it is better to have all the background music stuff happening in the same place, we will also initiate the background music here.

In the `BackgroundMusic` class, add the following:

```
import AVFoundation

class BackgroundMusic: NSObject {

    // create the class as a singleton
    static let instance = BackgroundMusic()
    var mediaPlayer = AVAudioPlayer()

    func playBackgroundMusic(){
        // Start the background music:
        if let musicPath = Bundle.main.path(forResource: "Sound/
BackgroundMusic.m4a",
                                         ofType: nil) {

            let url = URL(fileURLWithPath: musicPath)

            do {
                mediaPlayer = try AVAudioPlayer(contentsOf: url)
                mediaPlayer.numberOfLoops = -1
                mediaPlayer.prepareToPlay()
                mediaPlayer.play()
            }
            catch { /* Couldn't load music file */ }
        }
    }
}

//playBackgroundMusic
}//class end
```

We will create the class as a singleton so that we can access the objects of the class without the need to create an object of the class every time. This will also ensure that only one object of the class is created.

Next, we will add functions to pause, play, and check whether the sound is already muted, and use the `setVolume` functions.

We will also be using the `UserDefault`s property to store the current state to check whether the sound is muted or not. If the user muted the sound, then we will store that information so that when they open the application next time, the application will mute the sound, adding an extra feature of convenience for the user.

Add the following functions to the class before the class end bracket:

```
func pauseMusic(){
    UserDefaults.standard.set(true, forKey:
"BackgroundMusicMuteState")
    musicPlayer.pause()
}

func playMusic(){
    UserDefaults.standard.set(false, forKey:
"BackgroundMusicMuteState")
    musicPlayer.play()
}

// Check mute state
func isMuted() -> Bool {
    if UserDefaults.standard.bool(forKey:"BackgroundMusicMuteSta
te") {
        return true
    } else {
        return false
    }
}

func setVolume(volume: Float){

    musicPlayer.volume = volume
}
```

Finally, we check whether the sound is muted and, if so, we pause the music when it is initially played. Add the following in the `playBackgroundMusic` function:

```
func playBackgroundMusic(){
    // Start the background music:
    if let musicPath = Bundle.main.path(forResource:"Sound/
BackgroundMusic.m4a", ofType: nil) {
        let url = URL(fileURLWithPath: musicPath)
```

```
do {
    musicPlayer = try AVAudioPlayer(contentsOf: url)
    musicPlayer.numberOfLoops = -1
    musicPlayer.prepareToPlay()
    musicPlayer.play()
}
catch { /* Couldn't load music file */ }
}

if isMuted(){
    pauseMusic()
}

}//playBackgroundMusic
```

Make changes to the `GameViewController` in order to play the background music using the newly created `BackgroundMusic` singleton class. Remove the old code for playing the music and replace it with the following lines of code:

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

    // Build the menu scene:
    let menuScene = MenuScene()
    let skView = self.view as! SKView
    // Ignore drawing order of child nodes
    // (This increases performance)
    skView.ignoresSiblingOrder = true
    // Size our scene to fit the view exactly:
    menuScene.size = view.bounds.size
    // Show the menu:
    skView.presentScene(menuScene)

    BackgroundMusic.instance.playBackgroundMusic()
}
```

You don't require the following line either, so you can also delete it from the `GameViewController` class:

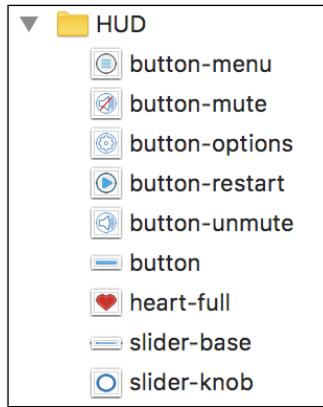
```
var musicPlayer = AVAudioPlayer()
```

We will add an **Options** menu scene so that we can add buttons and a slider to it to access the mute and volume control features.

Adding Options to the Menu Scene

Create a new file called `OptionsScene.swift`.

Next, add in the assets for the **Options Scene** menu button, mute, unmute, slider base, and slider knob assets from the chapter and add them into the `Assets.xcassts` file. Add them under the `HUD` folder as follows:



In the `OptionsScene` class, add the following properties for the texture atlas, mute button, slider base, slider knob, and menu button:

```
import SpriteKit

class OptionsScene: SKScene {

    // Grab the HUD sprite atlas:
    let textureAtlas:SKTextureAtlas = SKTextureAtlas(named:"HUD")

    let muteButton = SKSpriteNode()
    let sliderBase = SKSpriteNode()
    let sliderKnob = SKSpriteNode()

    let menuButton = SKSpriteNode()
}
```

Now, we will add the functions to the class. First, we will add the `didMove` function, which will get called as soon as the Scene is loaded:

```
override func didMove(to view: SKView) {

    // Position nodes from the center of the scene:
```

```
self.anchorPoint = CGPointMake(0.5, 0.5)

// Add the background image:
let backgroundImage = SKSpriteNode(imageNamed: "background-
menu")
backgroundImage.size = CGSizeMake(width: 1024, height: 768)
backgroundImage.zPosition = -1
self.addChild(backgroundImage)

// Draw the name of the scene
let logoText = SKLabelNode(fontNamed: "AvenirNext-Heavy")
logoText.text = "Options Scene"
logoText.position = CGPointMake(x: 0, y: 100)
logoText.fontSize = 60
self.addChild(logoText)

//Add mute button
if BackgroundMusic.instance.isMuted(){
    muteButton.texture = textureAtlas.textureNamed("button-
mute")
} else{
    muteButton.texture = textureAtlas.textureNamed("button-
unmute")
}

muteButton.name = "muteBtn"
muteButton.position = CGPointMake(x: -100, y: 0)
muteButton.size = CGSizeMake(width: 75, height: 75)
self.addChild(muteButton)

//Add slider base asset
sliderBase.texture = textureAtlas.textureNamed("slider-base")
sliderBase.position = CGPointMake(x: 50, y: 0)
sliderBase.size = CGSizeMake(width: 200, height: 20)
sliderBase.anchorPoint = CGPointMake(x: 0, y: 0.5)
self.addChild(sliderBase)

let volume = BackgroundMusic.instance.musicPlayer.volume
let pos = (Float)(sliderBase.position.x + sliderBase.frame.
size.width) * volume

//Add slider knob asset
sliderKnob.texture = textureAtlas.textureNamed("slider-knob")
sliderKnob.position = CGPointMake(x: CGFloat(pos) , y: sliderBase.
position.y)
```

Adding Menus and Sounds

```
    sliderKnob.size = CGSize(width: 25, height: 25)
    self.addChild(sliderKnob)
    sliderKnob.zPosition = 1.0

    //Add menu button
    menuButton.texture = textureAtlas.textureNamed("button-menu")
    menuButton.name = "returnToMenu"
    menuButton.position = CGPoint(x: 0, y: -120)
    menuButton.size = CGSize(width: 70, height: 70)
    self.addChild(menuButton)

}
```

We first set the background image and then add some text to show that the current scene is the **Options Menu**.

Next, we add the mute button to the scene. Before setting the texture, we check whether the audio is muted. If it's already muted, then we load the mute texture, otherwise, we load the unmute texture.

We then add the slider base texture to the scene. For the slider base, we set the anchor point to the left of the asset instead of the center.

Now, to set the position of the knob, we get the current volume (which ranges from 0 to 1) and set it with respect to where it needs to be on the slider base. We get the position of the slider base. Since its anchor is set to the extreme left, that will be its origin. To get the x position of the knob, we add the width of the sound base to the location of the knob and then multiply by the volume. So, when the volume changes, the x position changes depending upon the current volume.

We also set the slider knobs' zPosition as 1 so that it sits above the slider base.

In the end, we also add a menu button so that the user can go back to the Main Menu.

To enable the mute button, we add a touchesBegan function to the class and add the following code:

```
override func touchesBegan(_ touches: Set<UITouch>, with event: UIEvent?) {

    for touch in (touches) {
        // Find the location of the touch:
        let location = touch.location(in: self)
        // Locate the node at this location:
        let nodeTouched = atPoint(location)
```

```

        if nodeTouched.name == "muteBtn" {

            if BackgroundMusic.instance.isMuted(){
                BackgroundMusic.instance.playMusic()
                muteButton.texture = textureAtlas.
                textureNamed("button-unmute")
            }else{
                BackgroundMusic.instance.pauseMusic()
                muteButton.texture = textureAtlas.
                textureNamed("button-mute")
            }

        } else if(nodeTouched.name == "returnToMenu") {
            // Transition to the main menu scene:
            self.view?.presentScene(
                MenuScene(size: self.size),
                transition: .crossFade(withDuration: 0.6))

        }
    }
}

```

Just like when we added the play button, we do a similar check. We get the location of the touch and get the node under the touched location. If it is the mute button, then we check whether the music is muted. If so, then we play the music and change the texture of the mute button, otherwise, we do the opposite.

If the touched button is for the return menu, then we load the **Menu Scene**.

Next, to update the volume slider, we use the `touchesMoved` function and add the functionality for the volume slider. Add the following code for `touchesMoved`:

```

override func touchesMoved(_ touches: Set<UITouch>, with event:
UIEvent?) {

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

        var xpos = location.x

        if(xpos  <= sliderBase.position.x){
            xpos  = sliderBase.position.x
        }else if(xpos >= sliderBase.position.x + sliderBase.frame.
size.width) {

```

```
        xpos = sliderBase.position.x + sliderBase.frame.size.  
        width  
    }  
  
    sliderKnob.position = CGPoint(x:xpos,y: sliderKnob.  
        position.y)  
    let volume = (sliderKnob.position.x - sliderBase.  
        position.x)/sliderBase.frame.width  
  
    BackgroundMusic.instance.setVolume(volume: Float(volume))  
}  
}
```

We get the touch location and set the slider knob so that it stays on the slider base. We set the knob so that it is always between the origin of the base and the width of the base.

Depending upon the location of the knob, we calculate the value of the volume by normalizing the value of the knob position between 0 and 1.

Finally, we use the `setVolume` function to set the current volume of the background music.

To test whether this really works, we need a button on the **Main Menu Scene** to access the **Options Menu Scene**. Add an `SKSpriteNode` at the top of the class, called `optionsButton`:

```
let optionsButton = SKSpriteNode()
```

Add the following code to the bottom of the `didMove` function to add the **Options Button** to the scene:

```
//Options menu button  
optionsButton.texture = textureAtlas.textureNamed("button-options")  
optionsButton.name = "OptionsBtn"  
optionsButton.position = CGPoint(x: 0, y: -120)  
optionsButton.size = CGSize(width: 75, height: 75)  
self.addChild(optionsButton)
```

Finally, add the following code in the `MenuScene` in the `touchesBegan` function:

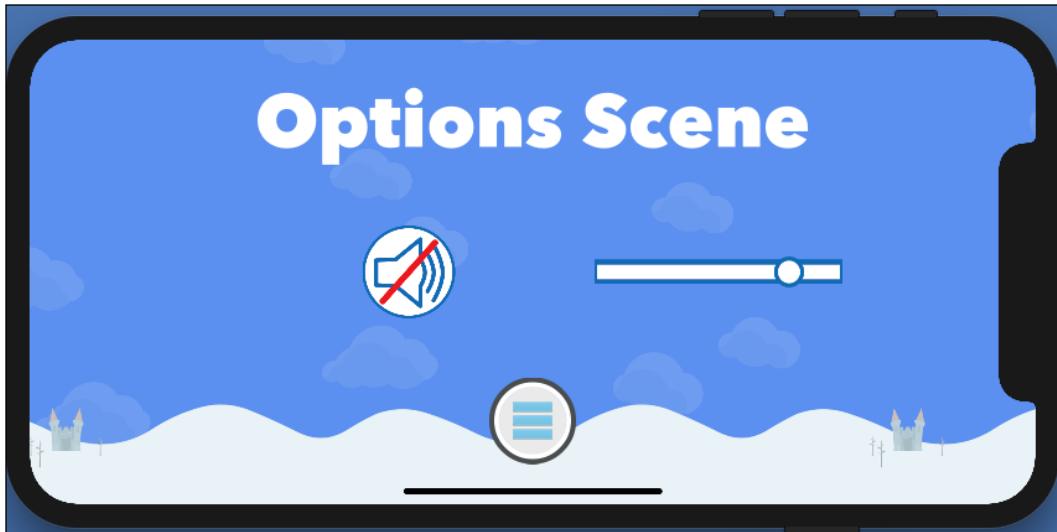
```
override func touchesBegan(_ touches: Set<UITouch>, with event: UIEvent?) {
    for touch in (touches) {
        // Find the location of the touch:
        let location = touch.location(in: self)
        // Locate the node at this location:
        let nodeTouched = atPoint(location)

        if nodeTouched.name == "StartBtn" {
            // Player touched the start text or button node
            // Switch to an instance of the GameScene:
            self.view?.presentScene(GameScene(size: self.size))
        } else if nodeTouched.name == "OptionsBtn" {
            self.view?.presentScene(OptionsScene(size: self.size))
        }
    }
}
```

The **Main Menu** should now look like this:



Click on the gear icon at the bottom to open the **Options Menu Scene**. You can now mute the background music by clicking the speaker icon and change the volume by moving the volume slider knob as follows:



Click on the menu button at the bottom to go back to the **Main Menu Scene**.

Close and restart the application to see that the background music is still muted, since you muted it the last time you opened the application.



Checkpoint 9-B

The code up to this point is available in this chapter's code resources.



Summary

We have taken huge steps toward finishing the game in this chapter. We learned how to create menus in SpriteKit, added the main menu to the game, and gave the player a way to restart the game when they run out of health. Then, we enhanced the gameplay experience with catchy background music and timely sound effects. We also added a mute and volume slider for the background music.

Next, we will explore advanced techniques that you can use to make your games more fun for your players, in *Chapter 10, Standing Out in the Crowd with Advanced Features*.

10

Standing out in the Crowd with Advanced Features

It is important to make your game stand out if you are looking for financial success or popularity. The App Store is flooded with half-baked games. In this chapter, we will add the extra bells and whistles to our game to take it to the next level. First, we will add health power-ups so that players can heal themselves after taking damage. Then, we will add smashable crates that contain coins; nothing is more fun than smashing crates! You will learn to combine the techniques you have learned in this book to create an advanced system, and these steps will make the game feel finished and ready for players.

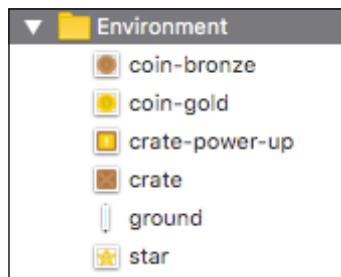
Topics in this chapter include the following:

- Adding crates to smash open
- Recycling emitter nodes with particle pools
- Creating the health power-up crate
- Spawning smashable crates that reward coins

Adding crates to smash open

Many classic games feature breakable crates. There is something very satisfying about flying into a crate and smashing it open. We will now add breakable crates to our game. Some of these crates will reward the player with coins, and some will reward the player with a health refill. Follow these steps to create the basic crate system:

1. Add the art assets to your project. Open `Assets.xcassets`, open the **Environment Sprite Atlas**, and drag the contents of the `Crates` folder from the downloadable asset bundle. When you are done, you should see `crate` and `crate-power-up` appear in your **Environment Atlas**, as shown in the following screenshot:



2. Open `GameScene.swift` and add a new physics category to the `PhysicsCategory` enum. We will create a physics category for crates with the value 64. When you are done, your `PhysicsCategory` enum should look like this:

```
enum PhysicsCategory: UInt32 {  
    case penguin = 1  
    case damagedPenguin = 2  
    case ground = 4  
    case enemy = 8  
    case coin = 16  
    case powerup = 32  
    case crate = 64  
}
```

3. Now we can add the `crate` to the `Player` class list of contact tests. This will cause a contact event to fire when the player runs into a crate. Open `Player.swift`, locate the `init` function, and update the `contactTestBitMask` to look like this (new code in bold):

```
self.physicsBody?.contactTestBitMask =  
    PhysicsCategory.enemy.rawValue |
```

```
PhysicsCategory.ground.rawValue |  
PhysicsCategory.powerup.rawValue |  
PhysicsCategory.coin.rawValue |  
PhysicsCategory.crate.rawValue
```

Next, we will add a new Crate class. Add a new Swift file to your project, named `Crate.swift`, and add the following code:

```
import SpriteKit  
  
class Crate: SKSpriteNode, GameSprite {  
    var initialSize = CGSize(width: 40, height: 40)  
    var textureAtlas: SKTextureAtlas =  
        SKTextureAtlas(named: "Environment")  
    var givesHeart = false  
    var exploded = false  
  
    init() {  
        super.init(texture: nil, color: UIColor.clear,  
                  size: initialSize)  
        self.physicsBody = SKPhysicsBody(rectangleOf:  
                                         initialSize)  
  
        // Only collide with the ground and other crates:  
        self.physicsBody?.collisionBitMask =  
            PhysicsCategory.ground.rawValue |  
            PhysicsCategory.crate.rawValue  
        self.physicsBody?.categoryBitMask =  
            PhysicsCategory.crate.rawValue  
  
        self.texture = textureAtlas.textureNamed("crate")  
    }  
  
    // A function to create a crate that gives health:  
    func turnToHeartCrate() {  
        self.physicsBody?.affectedByGravity = false  
        self.texture =  
            textureAtlas.textureNamed("crate-power-up")  
        givesHeart = true  
    }  
  
    // A function for exploding crates!  
    func explode() {  
        // Do not do anything if already exploded:  
        if exploded { return }  
        exploded = true
```

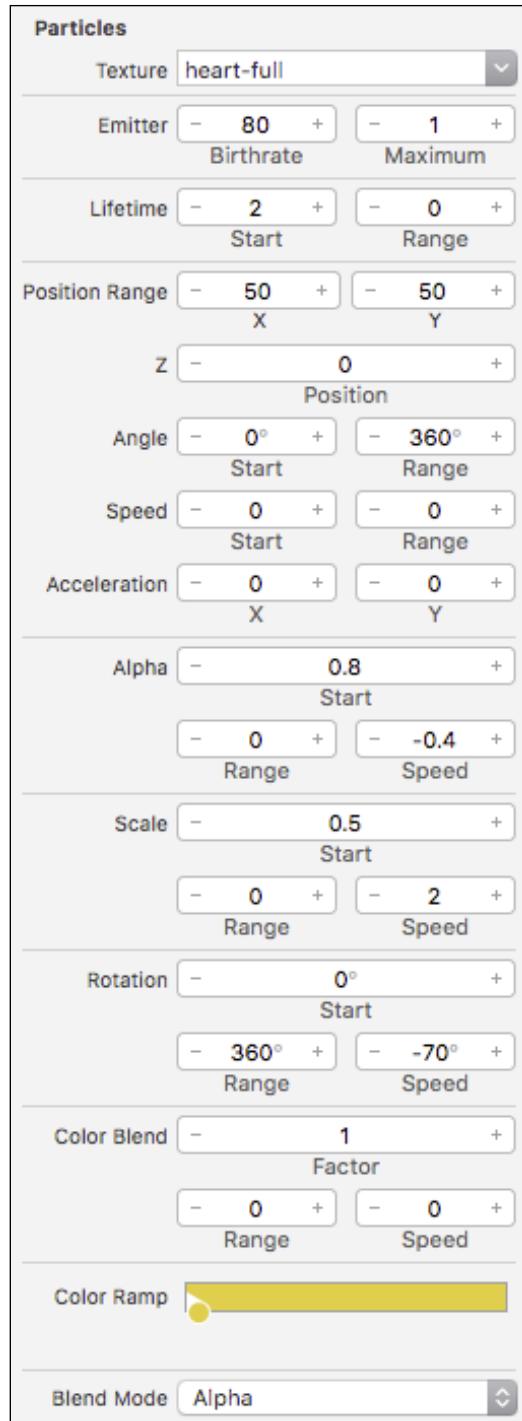
```
// Prevent additional contact:  
self.physicsBody?.categoryBitMask = 0  
// TODO: We will add more here in a bit  
}  
  
// A function to reset the crate for re-use  
func reset() {  
    self.alpha = 1  
    self.physicsBody?.categoryBitMask =  
        PhysicsCategory.crate.rawValue  
    exploded = false  
}  
// Conform to the necessary protocols:  
func onTap() {}  
required init?(coder aDecoder: NSCoder) {  
    super.init(coder: aDecoder)  
}  
}
```

Excellent work; our `Crate` class is ready. Next, we will add some supporting particle effects.

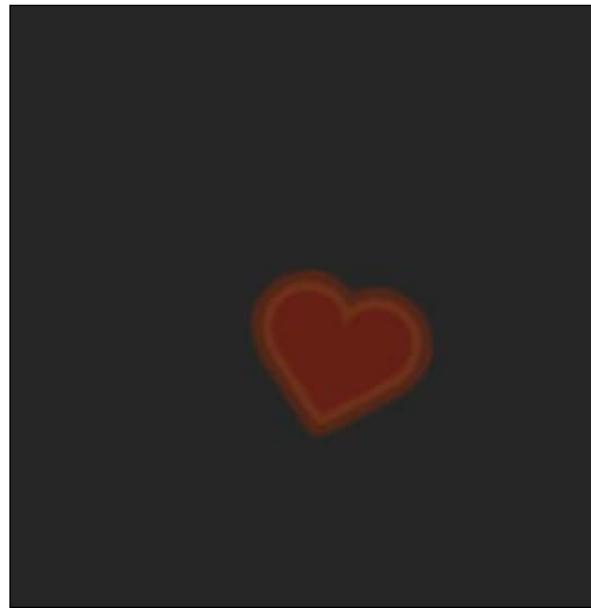
Creating the Crate particle effects

We need two fun particle effects to use when the player smashes a crate. The crates that reward hearts will display a different animation than those that reward coins. First, we will create the heart particle effect by following these steps:

1. Add a new **SpriteKit Particle File** to your project (choose the bokeh particle template) and name it `HeartExplosion.sks`. You should see the particle editor open with a relaxing bokeh animation in Xcode.
2. Using the utilities bar on the right, change the particle texture to the heart-full texture, and match the particle settings to the following screenshot. Make sure to set **Blend Mode** to **Alpha**:

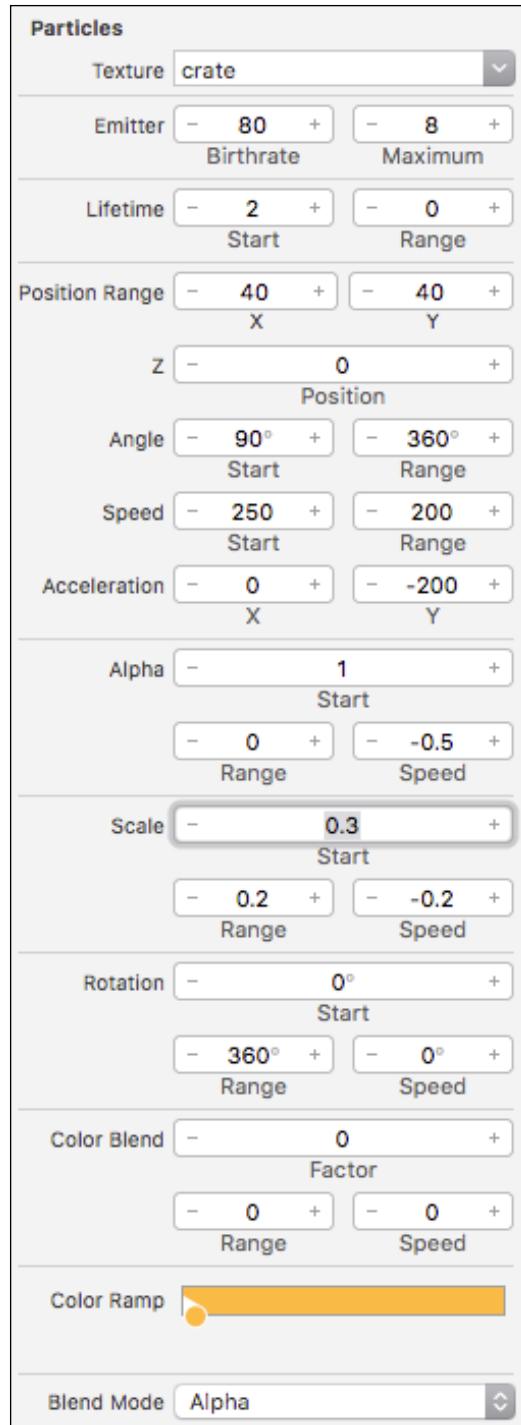


You should see a single large, rotating heart floating through the particle editor. It should look something like this:

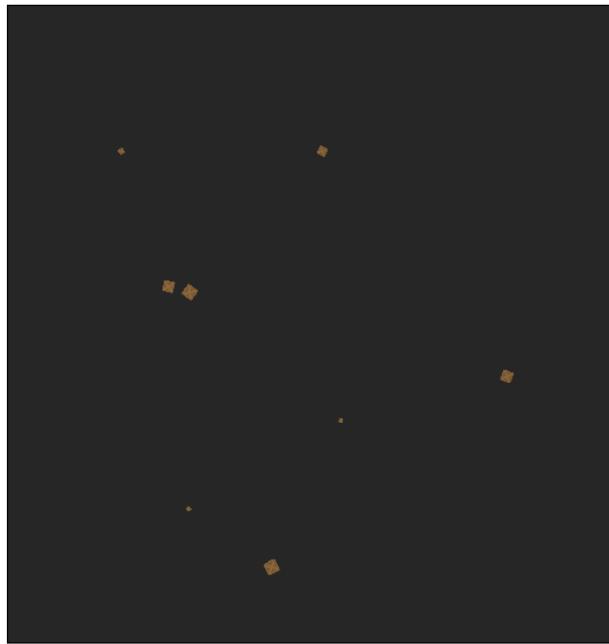


Next, we will create the particle emitter to use when the player smashes open a crate containing coins. Follow these steps to create the emitter:

1. Add a new **SpriteKit Particle File** to your project (choose the spark particle template) and name it `CrateExplosion.sks`. You should see an exciting spark particle emitter in the editor.
2. Using the utilities bar on the right, change the particle texture to the crate texture and match the particle settings to the following screenshot. Make sure to change the **Blend Mode** to **Alpha** to get the correct crate color:



You should see small crates breaking apart and shooting off in random directions, as shown in the following screenshot:



Great! We have successfully added the particle emitters we need to create fun animations when the player smashes open our crates. Next, we will wire up the contact event and fire these animations.

Recycling emitter nodes with particle pools

We do not need to create a new emitter node for every single crate in our game. Instead, we will create a small pool of emitter nodes when the game starts and reposition them when the player smashes a crate. This is a performance best practice as emitter nodes can use system resources quickly. Follow these steps to create a particle emitter node pooling system:

1. In Xcode, create a new .swift file named `ParticlePool.swift`, and add the following code to your new file:

```
import SpriteKit

class ParticlePool {
```

```
var cratePool: [SKEmitterNode] = []
var heartPool: [SKEmitterNode] = []
var crateIndex = 0
var heartIndex = 0
// A property to store a reference to the GameScene:
var gameScene = SKScene()

init() {
    // Create 5 crate explosion emitter nodes:
    for i in 1...5 {
        // Create a crate emitter node:
        let crate = SKEmitterNode(fileNamed:
            "CrateExplosion")!
        crate.position = CGPoint(x: -2000, y: -2000)
        crate.zPosition = CGFloat(45 - i)
        crate.name = "crate" + String(i)
        // Add the emitter to the crate pool array:
        cratePool.append(crate)
    }
    // Repeat these steps to create 1 heart emitter:
    for i in 1...1 {
        let heart = SKEmitterNode(fileNamed:
            "HeartExplosion")!
        heart.position = CGPoint(x: -2000, y: -2000)
        heart.zPosition = CGFloat(45 - i)
        heart.name = "heart" + String(i)
        heartPool.append(heart)
    }
}

// Called from GameScene to add emitters as children
func addEmittersToScene(scene: GameScene) {
    self.gameScene = scene
    // Add the crate emitters to the scene:
    for i in 0..
```

```
// We will use this function to reposition the
// next pooled node into the desired position
func placeEmitter(node: SKNode, emitterType: String)
{
    // Pull an emitter node from the correct pool:
    var emitter: SKEmitterNode
    switch emitterType {
        case "crate":
            emitter = cratePool[crateIndex]
            // Keep track of the next node to pull:
            crateIndex += 1
            if crateIndex >= cratePool.count {
                crateIndex = 0
            }
        case "heart":
            emitter = heartPool[heartIndex]
            heartIndex += 1
            if heartIndex >= heartPool.count {
                heartIndex = 0
            }
        default:
            return
    }

    // Find the node's position relative to GameScene:
    var absolutePosition = node.position
    if node.parent != gameScene {
        absolutePosition =
            gameScene.convert(node.position, from:
                node.parent!)
    }

    // Position the emitter on top of the node:
    emitter.position = absolutePosition
    // Restart the emitter animation:
    emitter.resetSimulation()
}
}
```

2. Open GameScene.swift and initialize your ParticlePool class as a new property on the GameScene class:

```
let particlePool = ParticlePool()
```

3. In the GameScene class, locate the didMove function. At the bottom of the function, call the ParticlePool.addEmittersToScene function to add our emitter nodes to the GameScene node tree:

```
// Add emitter nodes to GameScene node tree:  
particlePool.addEmittersToScene(scene: self)
```

4. Perfect! Now we can simply call the placeEmitter function whenever we want to display a HeartExplosion or CrateExplosion emitter animation.



Checkpoint 10-A

The code up to this point is available in the chapter's code resources.



Wiring up crate contact events

Now we can fire custom logic whenever the player runs into a crate. We will place our particle effect and award health or coins. Follow these steps to wire up the contact event:

1. In Xcode, open GameScene.swift and locate the didBegin function, where we set our physics contact logic.
2. We need to call the explode function on our Crate class any time the player runs into a Crate. Add a new case for crate contact below the star power-up contact case, as shown here (new code in bold):

```
case PhysicsCategory.powerup.rawValue:  
    player.starPower()  
case PhysicsCategory.crate.rawValue:  
    if let crate = otherBody.node as? Crate {  
        // Call the explode function with a reference  
        // to the GameScene:  
        crate.explode(gameScene: self)  
    }
```

3. Before we can award health, we need to add a new property to the Player class to set a maximum amount of health. Open Player.swift and add a new property called maxHealth, shown in the following snippet (new code in bold):

```
var health:Int = 3  
let maxHealth = 3
```

4. Open `Crate.swift` and expand the `explode` function to place a particle emitter and reward coins or health as follows:

```
func explode(gameScene:GameScene) {
    // Do not do anything if this crate already exploded:
    if exploded { return }
    exploded = true

    // Place a crate explosion at this location:
    gameScene.particlePool.placeEmitter(node: self,
        emitterType: "crate")
    // Fade out the crate sprite:
    self.run(SKAction.fadeAlpha(to: 0, duration: 0.1))

    if (givesHeart) {
        // If this is a heart crate, award a health point:
        let newHealth = gameScene.player.health + 1
        let maxHealth = gameScene.player.maxHealth
        gameScene.player.health = newHealth > maxHealth ?
            maxHealth : newHealth
        gameScene.hud.setHealthDisplay(newHealth:
            gameScene.player.health)
        // Place a heart explosion at this location:
        gameScene.particlePool.placeEmitter(node: self,
            emitterType: "heart")
    }
    else {
        // Otherwise, reward the player with coins:
        gameScene.coinsCollected += 1
        gameScene.hud.setCoinCountDisplay(newCoinCount:
            gameScene.coinsCollected)
    }

    // Prevent additional contact:
    self.physicsBody?.categoryBitMask = 0
}
```

5. Since we are fading our crates to fully transparent, we will need to add code in the `EncounterManager` to call the `Crate` class `reset` function when it recycles an encounter and resets the crate nodes. Open `EncounterManager.swift` and locate the `resetSpritePositions` function. Underneath the line that resets the `zRotation`, try to cast the node as a `Crate` and call the `reset` function (new code in bold):

```
// Reset the rotation of the sprite:  
spriteNode.zRotation = 0  
// If this is a Crate, call its reset function:  
if let crateTest = spriteNode as? Crate {  
    crateTest.reset()  
}
```

We are now ready to use our crate system. Next, we will add a health crate that randomly spawns after encounters.

Creating the health power-up crate

We have taken many steps to create our crate system. Now we can add our first crate: a crate in the `GameScene` class that will award health points to the player. Follow these steps to wire up the heart crate:

1. In `GameScene.swift`, instantiate a new instance of the `Crate` class as a property of `GameScene`:

```
let heartCrate = Crate()
```

2. At the bottom of the `GameScene didMove` function, add the `heartCrate` to the node tree and call the function that makes it award a heart:

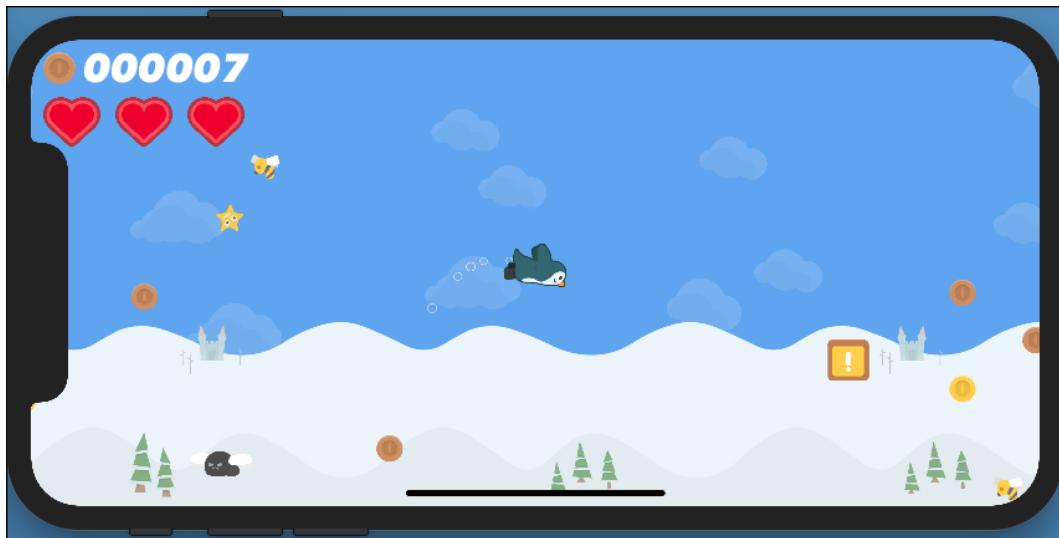
```
// Spawn the heart crate, out of the way for now  
self.addChild(heartCrate)  
heartCrate.position = CGPoint(x: -2100, y: -2100)  
heartCrate.turnToHeartCrate()
```

3. Locate the `GameScene didSimulatePhysics` function. Find the code that spawns the power-up star. We can add on to this code to spawn our heart crate randomly after some encounters. Add the following code below the `starRoll` conditional (new code in bold):

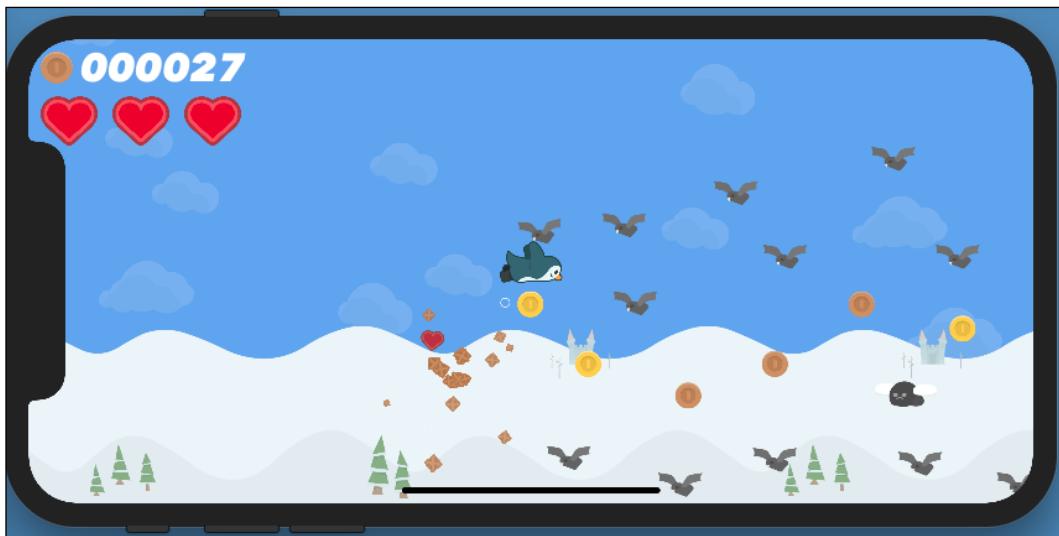
```
// Each encounter has a 10% chance to spawn a star:  
let starRoll = Int(arc4random_uniform(10))  
if starRoll == 0 {
```

```
// Note: all of the power-up star code is still here
// I'm excluding it here for brevity.
}
if starRoll == 1 {
    // Position the heart crate after this encounter:
    heartCrate.reset()
    heartCrate.position = CGPointMake(x:
        nextEncounterSpawnPosition - 600, y: 270)
}
```

We should now be randomly spawning our heart crate 10% of the time. You may wish to change the conditional to check whether `starRoll` is greater than zero temporarily for testing purposes (so it will spawn the heart crate 90% of the time). Just remember to change it back when you finish testing. Go ahead, run your project, and crash into a heart crate:



You should gain a health point and see the awesome result of our hard work, shown here:



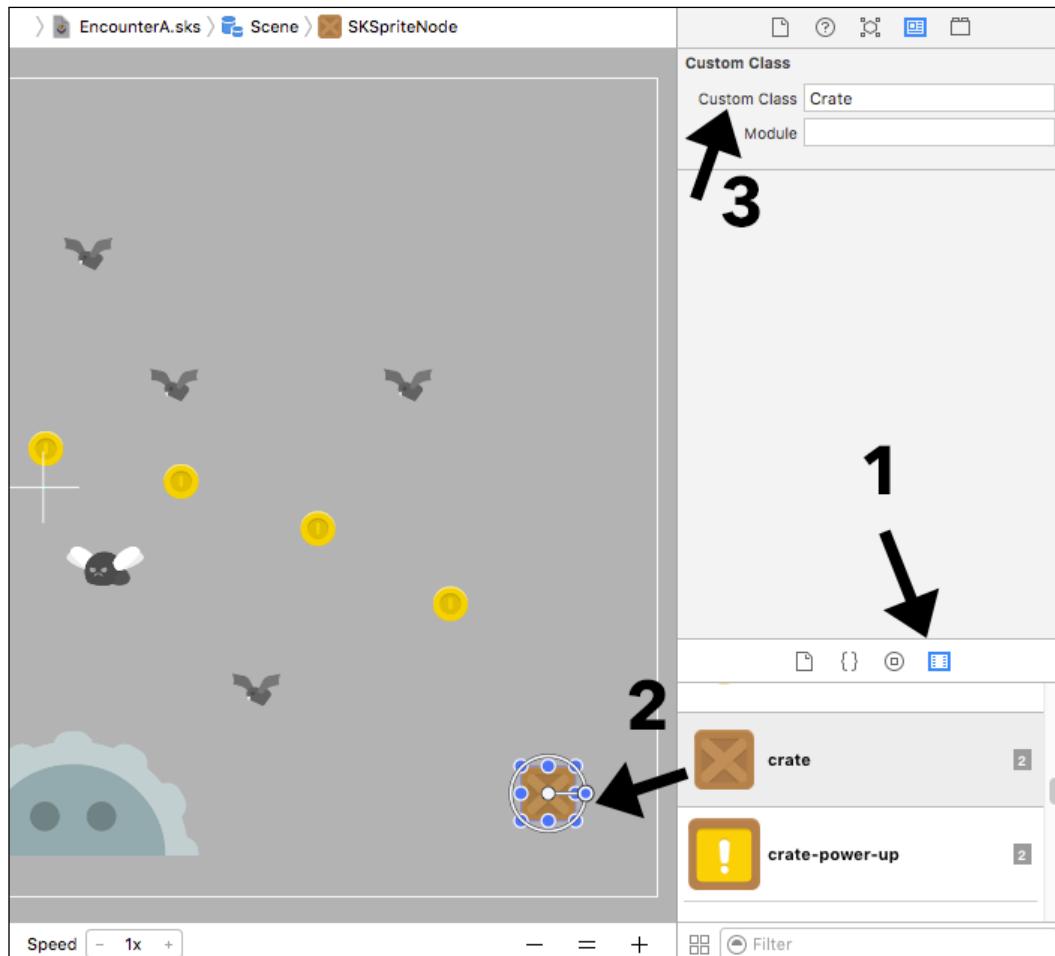
Great work! Next, let's add some smashable coin crates for the player to break apart.

Spawning smashable crates that reward coins

To add smashable coin crates, we will create new crate sprites in our encounters using the scene editor. Follow these steps to add a Crate to the game:

1. In Xcode, open one of the encounter scenes we created earlier in the book. I am using `EncounterA.sks` for this example. The scene will open in the Scene Editor. In the lower-right corner, click on the **Media Library** to see the media assets in your project.
2. Locate the crate texture and drag it into your encounter.

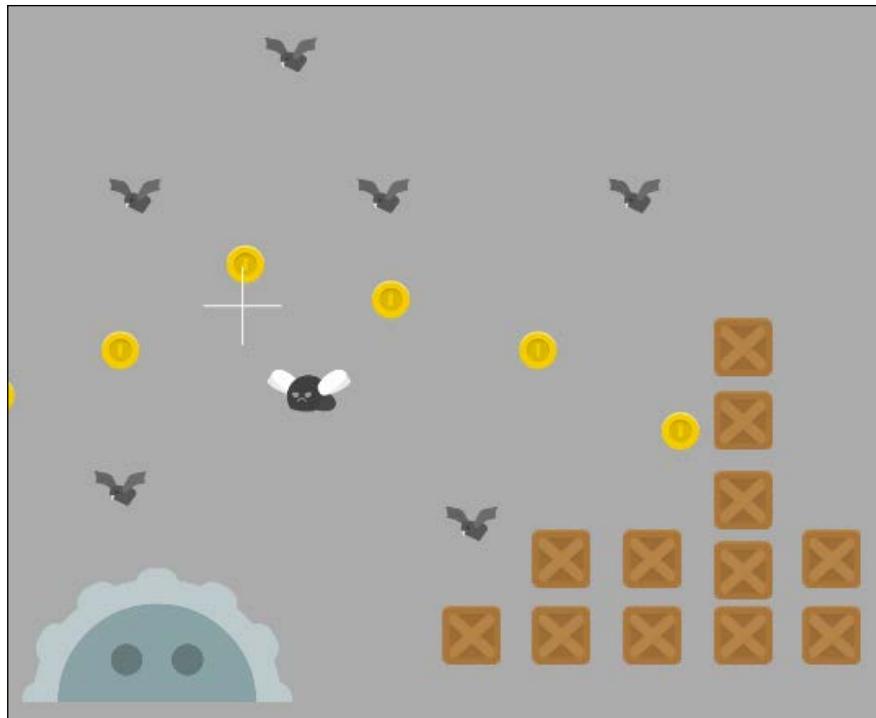
3. In the upper-right corner, click on the **Custom Class Inspector** and assign a **Custom Class** of **Crate**, as illustrated in the following screenshot:



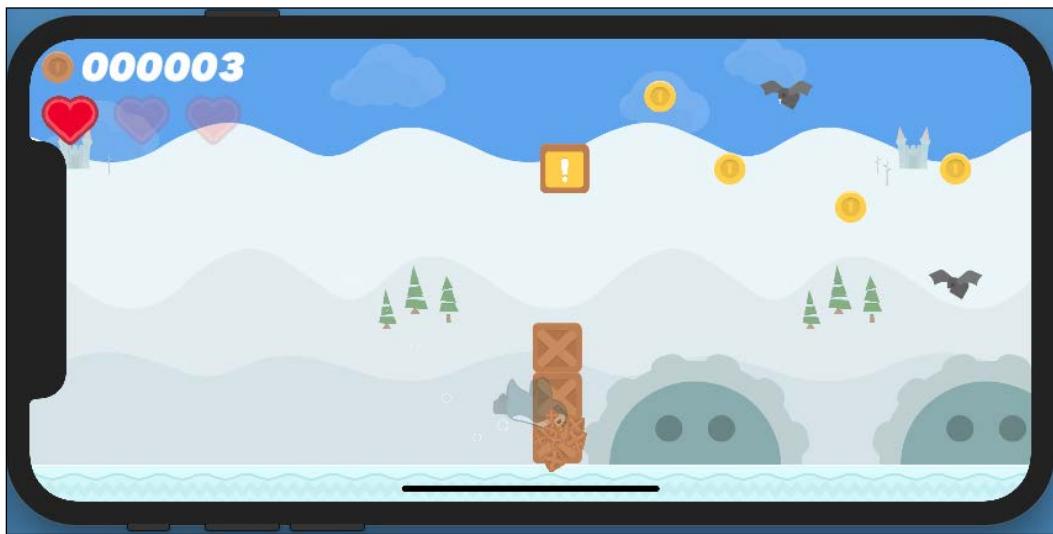
4. Repeat the process to create as many crates as you like. You can place them slightly above the ground and gravity will settle them into place.

You can always hold option and drag an existing sprite to create a clone with the same properties (persisting the **Custom Class** setting).

That is all you need to do to add smashable crates to the game. You can repeat this in all of your encounters to fill out crates throughout your game. My example scene ended up looking like this:



Run your project and try flying into some crates. You should see the crates smash apart and your coin score increase! Your smashable crates should look like this when you fly into them:





Checkpoint 10-B

The code up to this point is available in the chapter's code resources.

Summary

Great work! We have added a fun component to our game and learned some advanced techniques. We created a multi-use `Crate` class, wired it up for contact events, and created new animations with particle emitters. We learned an advanced technique to recycle our emitter nodes with a pool of particle emitters. Then, we used our new systems to create a smashable health crate and crates that reward the player with coins.

Our game is looking great and is a lot of fun to play! In *Chapter 11, Choosing a Monetization Strategy*, we will discuss publishing, advertising, and monetization options.

11

Introduction to SceneKit

We will now move from the 2D world to 3D. With SceneKit, we can make 3D games quite easily, especially since the syntax for SceneKit is quite similar to SpriteKit. When we talk about 3D games, it doesn't mean that you get to put on your 3D glasses to make the game. In 2D games, we mostly work with the *x* and *y* coordinates. In 3D games, we deal with all three axes: *x*, *y*, and *z*.

Additionally, in 3D games, we have different types of lights that we can use. Also, SceneKit has an inbuilt physics engine that takes care of forces such as gravity and also aids collision detection.

We can also use SpriteKit in SceneKit for GUI and buttons, so that we can add scores and interactivity to the game. There is a lot to cover in this chapter, so let's get started.

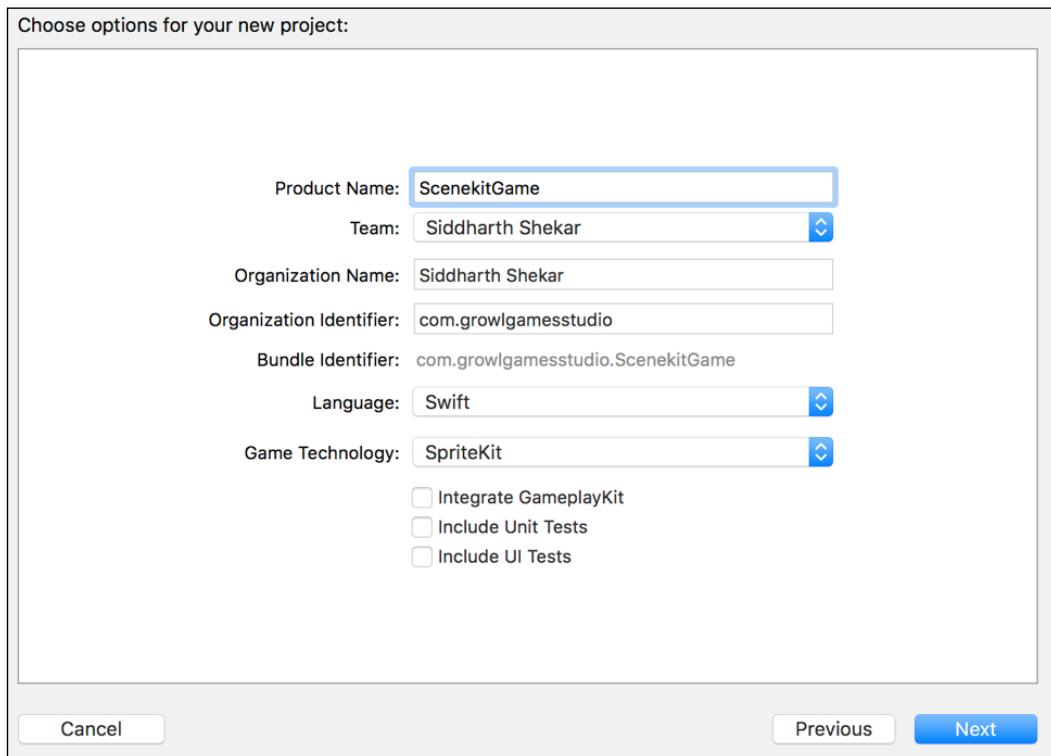
The topics covered in this chapter are:

- Creating a Scene with SCNScene
- Adding objects to the scene
- Importing scenes from an external 3D application
- Creating the Hero Class and physics
- Adding an enemy and collision detection
- Checking collision detection
- Adding a SpriteKit overlay
- Adding labels and buttons
- Adding touch interactivity
- Adding the Gameloop
- Setting a GameOver condition

- Adding wall and floor parallax
- Adding particles
- Adding character animation

Creating a Scene with SCNScene

We first create a new SceneKit project. It is very similar to creating other projects, only, this time, make sure you select SceneKit from the dropdown. Don't forget to select **Swift** when selecting the language. Uncheck the checkboxes as in the following screenshot:



Once the project is created, open it. Click on the `GameViewController` class and delete all the contents in the `viewDidLoad` and `handleTap` functions, as we will be creating a separate class and will add touch behavior to it.

Create a new class called `GameSCNScene` and import the following headers. Inherit from the `SCNScene` class and add an `init` function that takes in a parameter called `view` of type `SCNView`:

```
import SceneKit

class GameSCNScene: SCNScene {

    var scnView: SCNView!
    var _size: CGSize!

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

    init(currentview view: SCNView) {

        super.init()
    }
}
```

Also create two new constants, `scnView` and `_size`, of type `SCNView` and `CGSize` respectively.

Since we are making a SceneKit game, we have to get the current view, which is `SCNView`. Similar to how we got the view in SpriteKit, where we type cast the current view in `SpriteKit` to `SKView`, we create a `_size` constant to get the current size of the view.

Now move to the `GameViewController` class and create a global variable called `gameSCNScene` of type `GameSCNScene` and assign it to the `viewDidLoad` function:

```
var gameSCNScene: GameSCNScene!

override func viewDidLoad() {
    super.viewDidLoad()
    let scnView = view as! SCNView
    gameSCNScene = GameSCNScene(currentview: scnView)
}
```

Now we can add objects in the `GameSCNScene` class.

In the `init` function of `GameSCNScene`, add the following after the `super.init` function:

```
scnView = view
_size = scnView.bounds.size

scnView.scene = self
scnView.allowsCameraControl = true
scnView.showsStatistics = true
scnView.backgroundColor = UIColor.yellow
```

Here, we first assign the current view to the `scnView` constant. Next, we set the `_size` constant to the dimensions of the current view.

Assign the current scene to the scene of `scnView.allowCameraControls` and `showStatistics` are enabled next. This will enable us to control the camera and move it around to have a better look at the scene. Also, with statistics enabled, we will see how the game is performing and make sure that the FPS is maintained.

The `backgroundColor` property of `scnView` enables us to set the color of the view. With all this set, we can run the scene:



It is not all that awesome yet. One thing to notice is that we have not yet added a camera or a light, but we still see the scene. This is because we have not added anything to the scene, yet SceneKit automatically provides a default light and camera for the scene created.

Adding objects to the scene

Let's next add geometry to the scene. We can create basic geometry such as spheres, boxes, cones, tori, and so on in SceneKit with a lot of ease. Let's create a sphere first and add it to the scene.

Adding a sphere

Create a new function called `addGeometryNodes` in the `GameViewController` class, as follows:

```
func addGeometryNode() {  
  
    let sphereGeometry = SCNSphere(radius: 1.0)  
    sphereGeometry.firstMaterial?.diffuse.contents = UIColor.orange  
  
    let sphereNode = SCNNNode(geometry: sphereGeometry)  
    sphereNode.position = SCNVector3Make(0.0, 0.0, 0.0)  
    self.rootNode.addChildNode(sphereNode)  
}
```

We will use the `SCNSphere` class to create a sphere. We can also call `SCNBox`, `SCNCone`, `SCNTorus`, and so on to create a box, a cone, or a torus.

When creating a sphere, we have to provide the radius as a parameter, which will determine the size of the sphere. Although, to place the shape we have to attach it to a node so that we can add it to the scene.

Create a new constant called `sphereNode`, of type `SCNNNode`, and pass in the sphere as a parameter. To position the node, we have to use the `SCNvector3Make` property to place our object in 3D space by providing the *x*, *y*, and *z* values.

Finally, to add the node to the scene, we have to call `scene.rootNode` to add the `sphereNode` to the scene; unlike SpriteKit, where we would simply call `addChild` to add objects to the scene.

With the sphere added, let's run the scene. Don't forget to add `self.addGeometryNode()` to the `init` function:



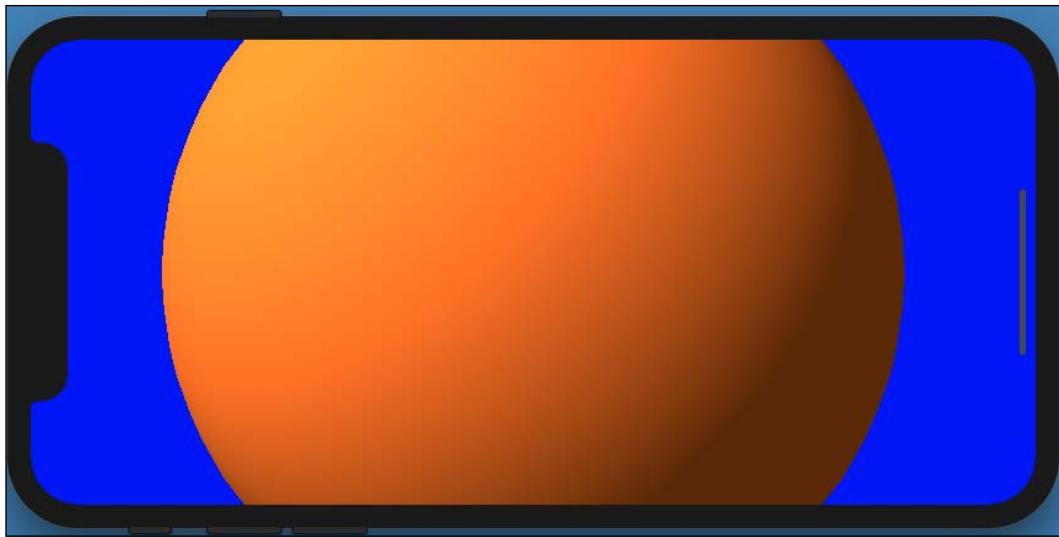
We added a sphere, so why are we getting a circle? Well, SceneKit's basic light source just enables us to see objects in the scene. If we want to see the actual sphere, we have to improve the light source of the scene. Also, the object is close to the camera – that's why you are seeing such a huge circle.

Adding light sources

Let's create a new function called `addLightSourceNode` so that we can add custom lights to our scene:

```
func addLightSourceNode() {  
  
    let lightNode = SCNNNode()  
    lightNode.light = SCNLight()  
    lightNode.light!.type = SCNLight.LightType.omni  
    lightNode.position = SCNVector3(x: 10, y: 10, z: 10)  
    self.rootNode.addChildNode(lightNode)  
  
    let ambientLightNode = SCNNNode()  
    ambientLightNode.light = SCNLight()  
    ambientLightNode.light!.type = SCNLight.LightType.ambient  
    ambientLightNode.light!.color = UIColor.darkGray  
    self.rootNode.addChildNode(ambientLightNode)  
}
```

We add some light sources to give some depth to our sphere object. Here, we add two types of light source. The first is an omni light. Omni lights start at a point and then the light is scattered equally in all directions. We also add an ambient light source. An ambient light is one that is reflected by other objects, like moonlight:



There are two more types of light sources: directional and spotlight. Spotlight is easy to understand; we usually use it if a certain object or person needs to have attention brought to it/them, such as a singer on a stage, for example. Directional lights are used if you want light to go in a single direction, like sunlight. The sun is so far from Earth that its light rays are almost parallel to each other when we see them.

To create a light source, we create a node called `lightNode` of type `SCNNode`. We then assign `SCNLight` to the `light` property of `lightNode`. We assign the `omni` light type according to the type of light. We assign the position of the light source as `10` in all three `x`, `y`, and `z` coordinates. Then, we add to the `rootNode` of the scene.

Next, we create an ambient light for the scene. The first two steps of the process are the same for creating any light source. For the type, we have to assign `SCNLightTypeAmbient` to assign an ambient type light source. Since we don't want the light source to be very strong as, after all, it is reflected; we assign `darkGrayColor` as the color. Finally, we add the light source to the scene.

There is no need to add an ambient light source to the scene but it will make the scene have softer shadows. You can remove the ambient light source to see the difference.

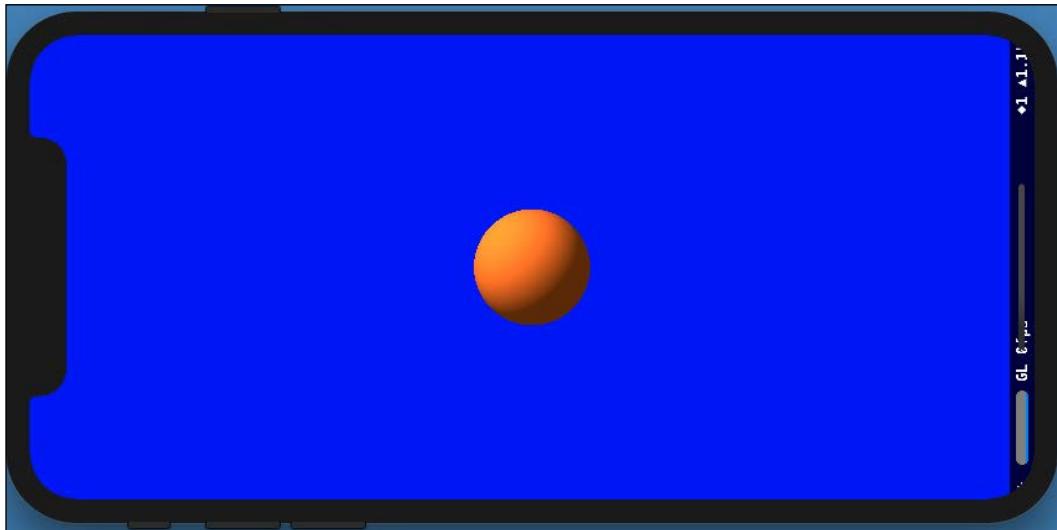
Call the `addLightSourceNode` in the `init` function. Now, build and run the scene to see an actual sphere with proper lighting.

Adding a camera to the scene

Next, let's add a camera to the scene as the default camera is very close. Add a new function called `addCameraNode` to the class and add the following code in it:

```
func addCameraNode() {  
  
    let cameraNode = SCNNNode()  
    cameraNode.camera = SCN Camera()  
    cameraNode.position = SCNVector3(x: 0, y: 0, z: 15)  
    self.rootNode.addChildNode(cameraNode)  
}
```

Here, again, we create an empty node called `cameraNode`. We assign `SCN Camera` to the `camera` property of `cameraNode`. Next, we position the camera such that we keep the `x` and `y` values at zero and move the camera back in the `z` direction by 15 units. Then, we add the camera to the root Node of the scene. Call `addCameraNode` at the bottom of the `init` function:



In this scene, the origin is at the center of the scene; unlike SpriteKit, where the origin of a scene is always at the bottom right of the scene. Here, the positive `x` and `y` are to the right and up from the center. The positive `z` direction is toward you.

We didn't move the sphere back or reduce its size here. The sphere appears smaller purely because we brought the camera back in the scene.

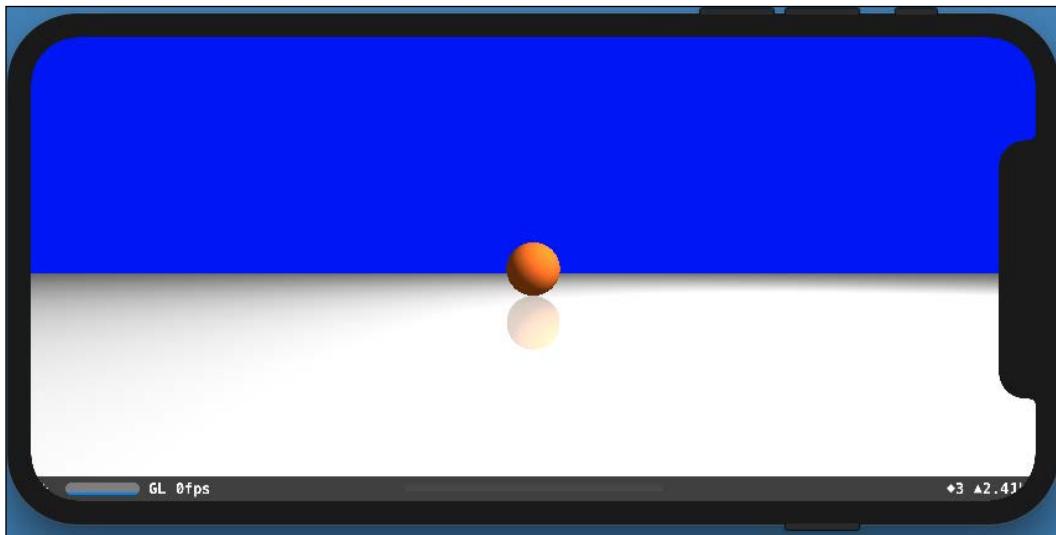
Let's next create a floor so that we will have a better understanding of the depth in the scene. Also, by doing so, we will learn how to create floors in a scene.

Adding a floor

In the `GameViewController` class, add a new function called `addFloorNode` and add the following code:

```
func addFloorNode() {  
  
    let floorNode = SCNNode()  
    floorNode.geometry = SCNPlane()  
    floorNode.position.y = -1.0  
    self.rootNode.addChildNode(floorNode)  
}
```

To create a floor, we create a variable called `floorNode` of type `SCNNode`. We then assign `SCNPlane` to the `geometry` property of `floorNode`. For the position, we assign the `y` value to negative 1 as we want the sphere to appear above the floor. Finally, as usual, we assign the `floorNode` to the root Node of the scene:



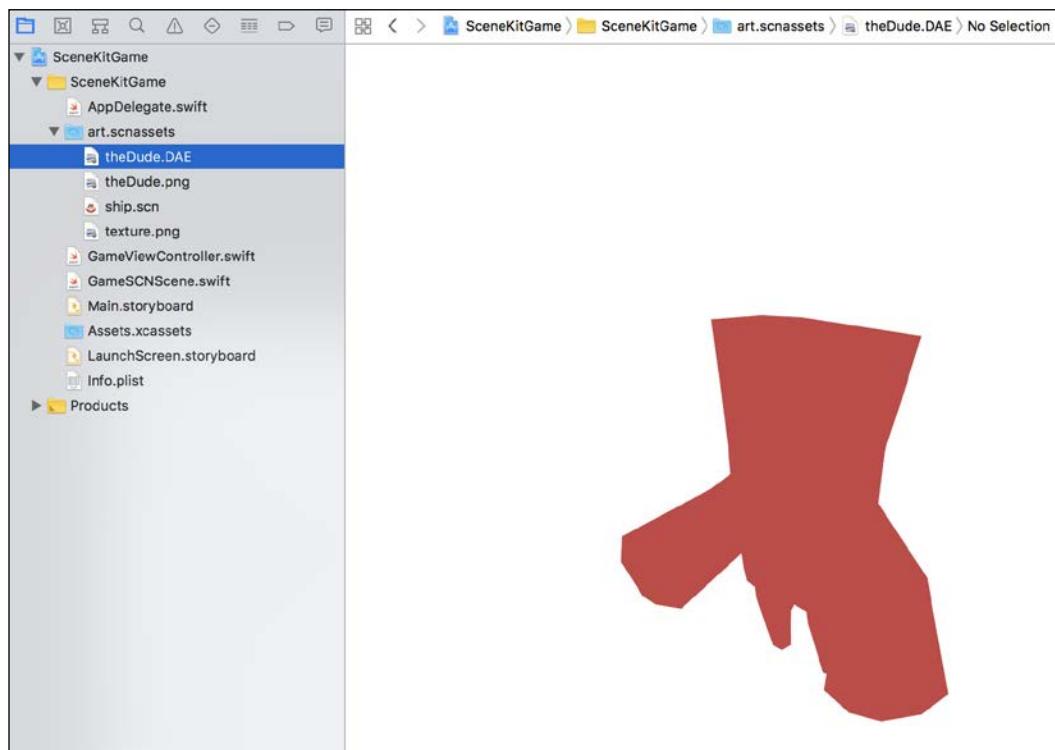
So, here, we see that with very minimal effort you can add 3D game objects to a scene. Next, we will see how to import a 3D object created externally.

Importing scenes from an external 3D application

Although we can add objects, cameras, and light through code, it will become very tedious and confusing when we have a lot of objects to add to the scene. In SceneKit, this problem can be easily overcome by importing scenes prebuilt in other 3D applications.

All 3D applications, such as 3D StudioMax, Maya, Cheetah 3D, Blender, and so on, have the ability to export scenes in Collada (.dae) and Alembic (.abc) format. We can import this scene with lighting, camera, and textured objects in SceneKit directly, without the need for setting up the scene.

In this section, we will import a Collada file into the scene. In the resources folder for this chapter, you will find the `theDude.DAE` file and `theDude.png` file. Drag the files into the current project under the `art.scnassets`:



If the model is not showing up in the scene, then you might have to drag and drop the .png onto the model in the view.

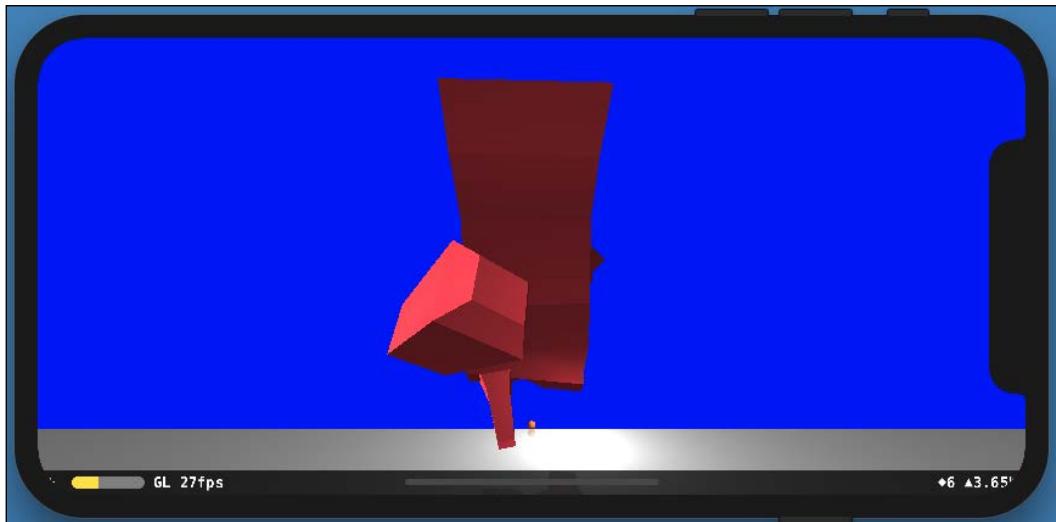
Next, in the GameSCNScene, add the following code after the code for the floor:

```
//load the monster from the collada scene
let monsterScene: SCNScene = SCNScene(named: "art.scnassets/
theDude.DAE")!
let monsterNode = monsterScene.rootNode.childNode(withName:
"CATRigHub001", recursively: false)!
self.rootNode.addChildNode(monsterNode)
```

Since theDude is too big, we just adjust the camera so that he fits in the scene. In the camera function, set the x, y, and z to the following:

```
cameraNode.position = SCNVector3(x: 0, y: 40, z: 100)
```

Run the project. The dude character will start playing the animation as soon as the scene starts:



In the game, we will only be playing the animations that are required.

Let's start building our game. We want the hero object to be in a separate class, so let's start by creating a new class, called hero.



Checkpoint A

The code up to this point is available in the chapter's code resources.



Creating the hero class and physics

Create a new class, called `Hero`, and add the following code to it:

```
import SceneKit
class Hero: SCNNNode {

    var isGrounded = false

    var monsterNode = SCNNNode()
    var jumpPlayer = SCNAccelerationPlayer()
    var runPlayer = SCNAccelerationPlayer()

    init(currentScene: GameSCNScene) {
        super.init()
        self.create(currentScene: currentScene)
    }
    required init?(coder aDecoder: NSCoder) {
        fatalError("init(coder:) has not been implemented")
    }
    func create(currentScene: GameSCNScene) {

    }
}
```

At the top of the create variables, which we will be needing later, we create a bool to check whether the character is grounded or not. Create a `SCNNNode` to access the character node and create two `SCNAccelerationPlayer` variables to access the player animations.

We create a custom `init` function, which takes in `GameSCNScene`, so that we can add the current scene to the parent scene. We call the `super.init` function in the custom `init` function. The required `init` function is a mandatory function that is required by the scene class. Lastly, we have the `create` function, in which we will be adding all the code to create the hero. The `create` function gets called in the `init` function.

Add the hero to the current node in the create function as shown here:

```
// Load the monster from the collada scene
let monsterScene: SCNScene = SCNScene(named: "monster.scnassets/
theDude.DAE")!
monsterNode = monsterScene.rootNode.childNode(withName:
"CATRigHub001", recursively: false)! //CATRigHub001
self.addChildNode(monsterNode)
```

Next, we set the anchor point of the monster to its center instead of the bottom of the body:

```
// set the anchor point to the center of the character
let (minVec, maxVec) = self.boundingBox
let bound = SCNVector3(x: maxVec.x - minVec.x, y: maxVec.y -
minVec.y, z: maxVec.z - minVec.z)
monsterNode.pivot = SCNMatrix4MakeTranslation(bound.x * 1.1, 0 , 0)
```

Let's create a collision box for the character and also set the physics properties so that we can make the character jump:

```
// Set the collision box for the character
let collisionBox = SCNBox(width: 2, height: 8, length: 2,
chamferRadius: 0)
self.physicsBody = SCNPhysicsBody(type: .dynamic,
shape:SCNPhysicsShape(geometry: collisionBox, options:nil))
self.physicsBody?.categoryBitMask = PhysicsCategory.hero.rawValue
self.physicsBody?.collisionBitMask = PhysicsCategory.ground.rawValue
self.physicsBody!.contactTestBitMask = PhysicsCategory.enemy.rawValue
// set angular velocity factor to 0 so that the character deosnt keel
over
self.physicsBody?.angularVelocityFactor = SCNVector3(0, 0, 0)
// set the mass so that the character gets affected by gravity
self.physicsBody?.mass = 20 //4.5

//set the scale and name of the current class
self.scale = SCNVector3(0.1, 0.1, 0.1)
```

We create a collision box using the `SCNBox` class and set the width, height, and length depending upon the size of the character.

Next, we set the physics body of the current node using the `SCNPhysicsBody` property and we set the body type to dynamic so that the object will be affected by an external force. Set the physics shape to the box we created earlier and set options to nil.

We then set the category, collision, and contact bit masks as we did in the SpriteKit project. Create an enum called `PhysicsCategory`, as follows:

```
enum PhysicsCategory: Int {  
    case hero = 1  
    case ground = 2  
    case enemy = 4  
}
```

We want the hero to only collide with the ground and we want him to only be in contact with the enemy.

We then set the angular velocity factor to zero so that he doesn't spin or rotate and set the mass of the node to 20. Since the character is huge, we scale the object down:

```
self.name = "hero"  
// add the current node to the parent scene  
currentScene.rootNode.addChildNode(self)
```

We name the current node hero and add the current node to the parent scene.

Let's also create two functions called `jump` and `update`. The `jump` function will apply a force in the upward direction so that the character jumps, and the `update` function will check the `y` position of the hero. If the hero is below 4.0f in the `y` direction, then the `isGrounded` bool, which will be set to true otherwise, will be set to false.

Add the `jump` and `update` functions as shown here:

```
func jump() {  
  
    //print("player jump")  
  
    if(isGrounded) {  
        //print("grounded")  
        self.physicsBody?.applyForce(SCNVector3Make(0, 2000, 0),  
        asImpulse: true) //1400  
    }  
}  
  
func update() {  
  
    //print("hero y pos: %f", self.presentation.position.y)  
  
    if(self.presentation.position.y < 4.0){  
        if(isGrounded == false){  
            isGrounded = true  
        }  
    }  
}
```

```
        }
    } else {
        if(isGrounded == true){
            isGrounded = false
        }
    }
}
```

In GameSCNScene, set the physics world property to set the gravity, set the contact delegate to self so that we can check the contact information, and set the debug options of the view so that we can see the collision box.

Add the following at the end of the init class:

```
self.physicsWorld.gravity = SCNVector3Make(0, -500, 0)
self.physicsWorld.contactDelegate = self

scnView.debugOptions = SCNDebugOptions.showPhysicsShapes
```

Create a new hero variable at the start of the class var hero: Hero! and add the hero after the preceding code:

```
self.hero = Hero(currentScene: self)
hero.position = SCNVector3Make(0, 5, 0)
```

Comment out the function that added the camera earlier and add the following code instead:

```
let cameraNode = SCNNNode()
cameraNode.camera = SCNCamera()
cameraNode.position = SCNVector3(x: -30, y: 5, z: 12)
cameraNode.eulerAngles.y -= Float(Double.pi/2)
self.rootNode.addChildNode(cameraNode)
```

Here, apart from repositioning the camera, we also rotate it so that it is perpendicular to the character.

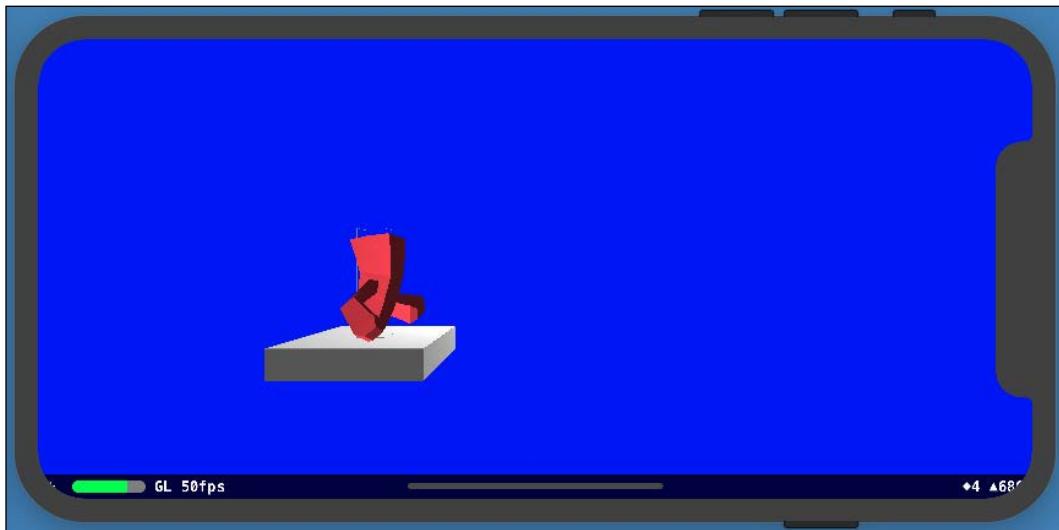
Create a ground box as follows so that the character doesn't keep on falling down:

```
let groundBox = SCNBox(width: 10, height: 2, length: 10,
chamferRadius: 0)
let groundNode = SCNNNode(geometry: groundBox)

groundNode.position = SCNVector3Make(0, -1.01, 0)
groundNode.physicsBody = SCNPhysicsBody.static()
groundNode.physicsBody?.restitution = 0.0
groundNode.physicsBody?.friction = 1.0
```

```
groundColor.physicsBody?.categoryBitMask = PhysicsCategory.ground.  
rawValue  
groundColor.physicsBody?.contactTestBitMask = PhysicsCategory.hero.  
rawValue  
  
groundColor.name = "ground"  
self.rootNode.addChildNode(groundColor)
```

Comment out the addFloor and addGeometry functions and run the application:



Adding an enemy and collision detection

Create a new file called `Enemy` and, similar to how we created the `Hero` class, we will pass the `GameSCNScene` in it:

```
import SceneKit  
  
class Enemy: SCNNNode{  
  
    var _currentScene: GameSCNScene!  
  
    init(currentScene: GameSCNScene) {  
  
        super.init()  
        self.create(currentScene: currentScene)
```

```
}

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

func create(currentScene: GameSCNScene) {
    print("spawn enemy")

    self._currentScene = currentScene

    let geo = SCNBox(width: 4.0, height: 4.0, length: 4.0,
chamferRadius: 0.0)
    geo.firstMaterial?.diffuse.contents = UIColor.yellow

    self.geometry = geo

    self.position = SCNVector3Make(0, 20.0, 60.0)
    self.physicsBody = SCNPhysicsBody.kinematic()
    self.name = "enemy"

    self.physicsBody?.categoryBitMask = PhysicsCategory.enemy.
rawValue
    self.physicsBody?.collisionBitMask = PhysicsCategory.ground.
rawValue
    self.physicsBody?.contactTestBitMask = PhysicsCategory.hero.
rawValue

    currentScene.rootNode.addChildNode(self)

}
```

We create a var of type GameSCNScene at the top of the class and, in the create function, we assign the currentScene variable to the local variable.

We create a geometry of type box and assign a yellow color to it. We set the current node geometry type to box, set the position, set the physicsBody type to kinematic and name the current node enemy.

Then, we set the category and contactTest bit mask so that only the enemy registers contact with the hero.

The current node is then added to the main scene.

Next, we will add an `update` function to update the position of the enemy:

```
func update() {  
  
    self.position.z += -0.9  
  
    if((self.position.z - 5.0) < -40){  
  
        let factor = arc4random_uniform(2) + 1  
  
        if( factor == 1 ){  
            self.position = SCNVector3Make(0, 2.0 , 60.0)  
        }else{  
            self.position = SCNVector3Make(0, 15.0 , 60.0)  
        }  
  
        _currentScene.score += 1  
        _currentScene.skScene.myLabel.text = "Score: \(_  
currentScene.score)"  
    }  
}  
  
}// end of class
```

In the `update` function, we move the enemy in the `z` direction by 0.9 every frame.

Then, we check whether the `y` position is less than -40, then we generate a random number between 1 and 2. If the random number generated is 1, then we reset the position of the enemy to the far right and set the `y` value to 2. If the random number generated is not 1, then set the `y` value to 15.

Next, we will add an `update` function, which will update the position of the enemy. In the `GameViewController`, we will inherit from `SCNSceneRendererDelegate` so that we can update the hero and the enemy.

Change the `GameViewController` class to the following:

```
class GameViewController: UIViewController, SCNSceneRendererDelegate {
```

Next, add the function as follows, which is used as the update function:

```
func renderer(_ aRenderer: SCNSceneRenderer, updateAtTime time:  
TimeInterval) {  
  
    gameSCNScene.update()  
}
```

The `rendererUpdateAtTime` function is a system function that gets called after all the objects are rendered in the scene. So, once the scene is rendered, the objects in the scene can be updated, otherwise, it might result in artifacting.

In the `viewDidLoad` function, set the delegate to self as follows:

```
override func viewDidLoad() {  
    super.viewDidLoad()  
  
    let scnView = view as! SCNView  
    scnView.delegate = self  
  
    gameSCNScene = GameSCNScene(currentview: scnView)  
}
```

Delegates are a part of code design patterns. Using delegates, a class can let different classes gain access to it and perform some of its responsibilities. Here, the `SceneRenderer` is delegating to `scnView` by assigning the delegate as `self`.

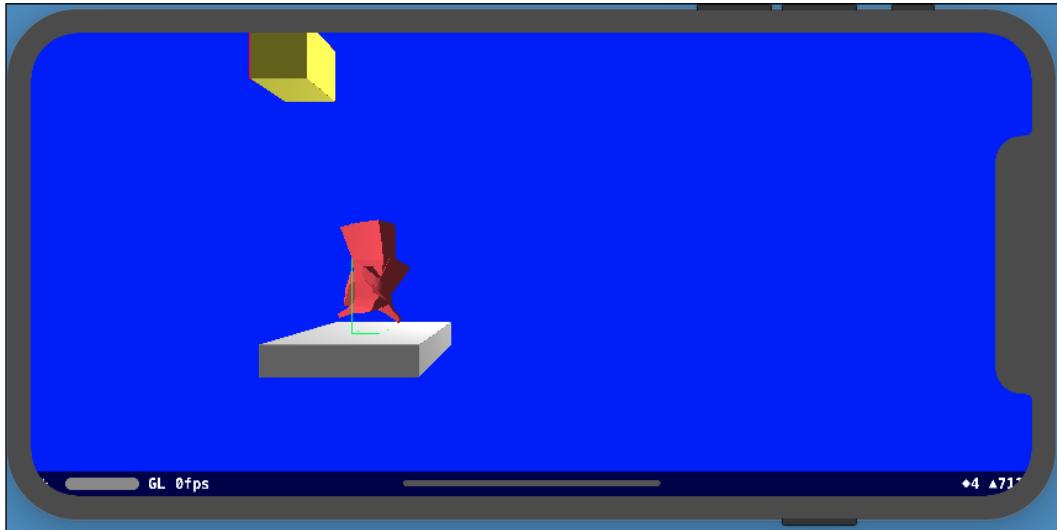
Call the `gameSCNScene.update` function. We will add an update function in the `GameSCNScene` class next:

```
func update() {  
    hero.update()  
    enemy.update()  
}
```

At the top of the class, create a new `Enemy` variable called `enemy`: `var enemy: Enemy!` and initialize it in the `init`. Add the following line after adding the `hero` to the scene:

```
self.enemy = Enemy(currentScene: self)
```

Now run the project. You should see the enemy getting updated every frame:



Adding a SpriteKit overlay

To show the score and a button, we will add it to the 2D SpriteKit layer. To add the overlay, create a class called `OverlaySKScene`. In the class, add the following:

```
import SpriteKit

class OverlaySKScene: SKScene {

    let _gameScene: GameSCNScene!
    let myLabel: SKLabelNode!
    var gameOverLabel: SKLabelNode!
    var jumpBtn: SKSpriteNode!
    var playBtn: SKSpriteNode!

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

    init(size: CGSize, gameScene: GameSCNScene) {
        super.init(size: size)

    }
}
```

We import SpriteKit as we will have to create a SubClass of SpriteKit. Create global variables of type GameSCNScene, SKLabelNodes, and SpriteNodes. Here, we create two LabelNodes: one for displaying the score and the other to show the gameover text. We also create two SpriteNodes: one for the play button and the other for the jump button.

We add the required `init` function and the default `init` function. The default `init` will take in the size of the scene and reference the `GameSCNScene` class as a parameter.

In the `init` function, we initialize the super class.

Adding labels and buttons

Next, in the `init` function, add the following code:

```
_gameScene = gameScene

myLabel = SKLabelNode(fontNamed:"Chalkduster")
myLabel.text = "Score: 0";
myLabel.fontColor = UIColor.whiteColor()
myLabel.fontSize = 65;
myLabel.setScale(1.0)
myLabel.position = CGPointMake(size.width * 0.5, size.height *
0.9)
self.addChild(myLabel)

gameOverLabel = SKLabelNode(fontNamed:"Chalkduster")
gameOverLabel.text = "GAMEOVER";
gameOverLabel.fontSize = 100;
gameOverLabel.setScale(1.0)
gameOverLabel.position = CGPointMake(size.width * 0.5, size.
height * 0.5)
gameOverLabel.fontColor = UIColor.whiteColor()
self.addChild(gameOverLabel)
gameOverLabel.hidden = true

playBtn = SKSpriteNode(imageNamed: "playBtn")
playBtn.position = CGPointMake(x: size.width * 0.15, y: size.
height * 0.2)
self.addChild(playBtn)
playBtn.name = "playBtn"
```

```
jumpBtn = SKSpriteNode(imageNamed: "jumpBtn")
jumpBtn.position = CGPoint(x: size.width * 0.9, y: size.height
* 0.15)
self.addChild(jumpBtn)
jumpBtn.name = "jumpBtn"
jumpBtn.hidden = true
```

In the `init` function, first, we set the `gameScene` passed to the `_gameScene` property of the scene.

Next, we initialize both `scoreLabel` and `gameOverLabel`. We set the `text`, `color`, `fontSize`, and `position` and add them to the scene. In the `gameOverLabel`, we set `hidden` to `true` as we only want the text to display once the game is over.

We then initialize the play and jump buttons as we have been doing in SpriteKit. We set the jump button sprite to `hidden`, as we want it to display only when the play button is clicked and the game starts. The images for the jump and play buttons are provided in the resources folder of the chapter.

Adding touch interactivity

To add touch interactivity, we will use the `touchesBegan` function, similar to how we have always used it in SpriteKit. Here, we get the location of the touch and the name of the sprite under the touch location. If the sprite name is `jumpBtn` and the `gameOver` Boolean is `false`, then we call the `heroJump` function in the `gameScene` class. If `gameOver` is `true` and the play button is clicked, then we call the `startGame` function in the `SceneKit` class.

Add the function as follows to detect touches:

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

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

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

        if(_gameScene.gameOver == false) {

            if(_node.name == "jumpBtn") {
```

```
        _gameScene.heroJump()
    }

}else{

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

        _gameScene.startGame()
    }
}
}
```

That is all we need to do for the `SpriteKit` class. We will be adding the `gameOver` Boolean, `heroJump`, and `startGame` functions in the `SceneKit` class. The code will show some errors until we create it, so ignore them for now.

Adding the Gameloop

In the `GameSCNScene` class, import `SpriteKit` at the top:

```
import UIKit
import SceneKit
import SpriteKit
```

Also create a global variable called `skScene` of type `OverlaySKScene`. Add the following code in the `GameSCNScene` at the end of the `init` function:

```
skScene = OverlaySKScene(size: _size, gameScene: self)
scnView.overlaySKScene = skScene
skScene.scaleMode = SKSceneScaleMode.Fill
```

Here, we initialize the `skScene` global variable we created earlier and pass in the size of the current scene and the current `SceneKit` class. Next, we assign the `skScene` class to the `overlaySKScene` property of `scnView`. Finally, we set the `scaleMode` of the `skScene` variable to type `SKSceneScaleMode.Fill`.

Next, create a new function called `heroJump` as follows:

```
func heroJump(){

    hero.jump()

}
```

We call the jump function of the hero class here, which will get called whenever we press the jump button.

Next, we will still need to add the Boolean and functions to the class to make our game work. At the top of the class, create a global variable called `gameOver` and set it to `true`.

Next, create a `gameStart` function as follows and add it into the class:

```
func startGame() {  
  
    gameOver = false  
    skScene.jumpBtn.hidden = false  
    skScene.myLabel.hidden = false  
    skScene.playBtn.hidden = true  
    skScene.gameOverLabel.hidden = true  
  
    score = 0  
    skScene.myLabel.text = "Score: \(score)"  
}
```

The `gameOver` Boolean is set to `false`. We set the jump button and `scoreLabel` to visible and hide the play button and `gameOverLabel`.

To keep track of and display the score, we need a `score` variable, so create a global variable called `score` of type `Int` and initialize it to `0` at the top of the class. Again, in the `startGame` function, set the value to `0` so that every time the function is called the value is reset. Also, we set the `scoreLabel` text to reflect the current score at the start of the game.

In the game, we will increment the score every time the enemy block goes beyond the screen and gets reset. If the block hits the hero, then it will be game over.

In the `Enemy`'s update function, add the following highlighted line after we check whether the enemy's `z` position is less than `-40` to update the score and `scoreLabel` text:

```
func update() {  
  
    self.position.z += -0.9  
  
    if((self.position.z - 5.0) < -40){  
  
        var factor = arc4random_uniform(2) + 1  
    }  
}
```

```
        if( factor == 1 ){
            self.position = SCNVector3Make(0, 2.0 , 60.0)
        }else{
            self.position = SCNVector3Make(0, 15.0 , 60.0)
        }
        _currentScene.score += 1
        _currentScene.skScene.myLabel.text = "Score: \(_currentScene.
score)"
    }
}
```

Setting a GameOver condition

Create a GameOver function as follows:

```
func GameOver() {

    skScene.jumpBtn.hidden = true
    skScene.playBtn.hidden = false
    skScene.gameOverLabel.hidden = false

    enemyNode.position = SCNVector3Make(0, 2.0 , 60.0)
    heroNode.position = SCNVector3Make(0, 0, 0)
}
```

Here, once the game is over, we hide the jumpButton and unhide the playButton and gameOverLabel. We then reset the positions of the enemy and hero to their initial states.

Next, we have to make sure that the enemyUpdate function is only called when gameOver is false. In the update function, enclose the enemyUpdate function in an if statement as follows:

```
if(!gameOver) {

    updateEnemy()

}
```

Checking for contact

To check for contact between the hero and enemy, we have to add a contact delegate to the current GameSCNScene class as follows:

```
class GameSCNScene: SCNScene, SCNPhysicsContactDelegate{
```

Set the contact delegate of the physicsworld to self:

```
self.physicsWorld.gravity = SCNVector3Make(0, -500, 0)
self.physicsWorld.contactDelegate = self
```

Next, add the didBeginContact function to the class as follows:

```
func physicsWorld(world: SCNPhysicsWorld, didBeginContact contact:
SCNPhysicsContact) {

    if( (contact.nodeA.name == "hero" &&
contact.nodeB.name == "enemy") ){

        contact.nodeA.physicsBody?.velocity = SCNVector3Zero
        //println("contact")
        gameOver = true
        GameOver()

    }
}
```

When two physics objects collide, the nodes are stored in the contact variable. Since we already named our physics body objects, we check whether the first body is that of the hero and the body that the hero has collided with is the enemy. If true, we print out contact and also set the velocity of the first body to zero. We also set gameOver to true and call a GameOver function.

Since ours is a small game, we can guess that body A is the hero and body B is the enemy. In bigger games, with so many collisions happening in a second, it might be hard to determine which is body A and which is body B. In such cases, we will have to check both, that is, whether body A is the enemy or the hero and vice versa for body B, then make the necessary conclusion.

Now, when you start the game, the GUI will show up with the play button:



And the **Gameover** text will show up when the game is over:



With this, the basic game play is over. Let's start refining the game. We will create a scrolling background for the scene.

Adding wall and floor parallax

What is a game without a parallax effect? In `SpriteKit`, we added parallax using sprites. In `SceneKit`, we will use planes to add it to the scene. Apart from adding parallax, we will also see how to add diffuse, normal, and specular maps to the plane. Also, we will learn what those terms mean.

Create a new file called `ScrollingBackground.swift`.

Add four global `SCNNodes`:

```
import SceneKit

class ScrollingBackground{

    var parallaxWallNode1: SCNNNode!
    var parallaxWallNode2: SCNNNode!
    var parallaxFloorNode1: SCNNNode!
    var parallaxFloorNode2: SCNNNode!
```

Add a new function, called `create`, and pass in a variable called `currentScene` as `GameSCNScene`. Add the following code to the `create` function:

```
func create(currentScene: GameSCNScene) {

    //Preparing Wall geometry
    let wallGeometry = SCNPlane(width: 250, height: 120)
    wallGeometry.firstMaterial?.diffuse.contents = "monster.
scnassets/wall.png"
    wallGeometry.firstMaterial?.diffuse.wrapS = SCNWrapMode.repeat
    wallGeometry.firstMaterial?.diffuse.wrapT = SCNWrapMode.repeat
    wallGeometry.firstMaterial?.diffuse.mipFilter = SCNFiltreMode.
linear
    wallGeometry.firstMaterial?.diffuse.contentsTransform =
SCNMatrix4MakeScale(6.25, 3.0, 1.0)

    wallGeometry.firstMaterial?.normal.contents = "monster.
scnassets/wall_NRM.png"
    wallGeometry.firstMaterial?.normal.wrapS = SCNWrapMode.repeat
    wallGeometry.firstMaterial?.normal.wrapT = SCNWrapMode.repeat
    wallGeometry.firstMaterial?.normal.mipFilter = SCNFiltreMode.
linear
    wallGeometry.firstMaterial?.normal.contentsTransform =
SCNMatrix4MakeScale(6.25, 3.0, 1.0)
```

```
wallGeometry.firstMaterial?.specular.contents = "monster.  
scnassets/wall_SPEC.png"  
    wallGeometry.firstMaterial?.specular.wrapS = SCNWrapMode.  
repeat  
    wallGeometry.firstMaterial?.specular.wrapT = SCNWrapMode.  
repeat  
    wallGeometry.firstMaterial?.specular.mipFilter =  
SCNFilterMode.linear  
    wallGeometry.firstMaterial?.specular.contentsTransform =  
SCNMatrix4MakeScale(6.25, 3.0, 1.0)  
  
    wallGeometry.firstMaterial?.locksAmbientWithDiffuse = true  
  
//Preparing floor geometry  
  
let floorGeometry = SCNPlane(width: 120, height: 250)  
    floorGeometry.firstMaterial?.diffuse.contents = "monster.  
scnassets/floor.png"  
    floorGeometry.firstMaterial?.diffuse.wrapS = SCNWrapMode.  
repeat  
    floorGeometry.firstMaterial?.diffuse.wrapT = SCNWrapMode.  
repeat  
    floorGeometry.firstMaterial?.diffuse.mipFilter =  
SCNFilterMode.linear  
    floorGeometry.firstMaterial?.diffuse.contentsTransform =  
SCNMatrix4MakeScale(12.0, 25, 1.0)  
  
    floorGeometry.firstMaterial?.normal.contents = "monster.  
scnassets/floor_NRM.png"  
    floorGeometry.firstMaterial?.normal.wrapS = SCNWrapMode.repeat  
    floorGeometry.firstMaterial?.normal.wrapT = SCNWrapMode.repeat  
    floorGeometry.firstMaterial?.normal.mipFilter = SCNFilterMode.  
linear  
    floorGeometry.firstMaterial?.normal.contentsTransform =  
SCNMatrix4MakeScale(24.0, 50, 1.0)  
  
    floorGeometry.firstMaterial?.specular.contents = "monster.  
scnassets/floor_SPEC.png"  
    floorGeometry.firstMaterial?.specular.wrapS = SCNWrapMode.  
repeat  
    floorGeometry.firstMaterial?.specular.wrapT = SCNWrapMode.  
repeat  
    floorGeometry.firstMaterial?.specular.mipFilter =  
SCNFilterMode.linear  
    floorGeometry.firstMaterial?.specular.contentsTransform =  
SCNMatrix4MakeScale(24.0, 50, 1.0)
```

```
floorGeometry.firstMaterial?.locksAmbientWithDiffuse = true

//assign wall geometry to wall nodes

parallaxWallNode1 = SCNNNode(geometry: wallGeometry)
parallaxWallNode1.rotation = SCNVector4Make(0, 1, 0, Float(-
Double.pi / 2))
parallaxWallNode1.position = SCNVector3Make(15, 0, 0)
currentScene.rootNode.addChildNode(parallaxWallNode1)

parallaxWallNode2 = SCNNNode(geometry: wallGeometry)
parallaxWallNode2.rotation = SCNVector4Make(0, 1, 0, Float(-
Double.pi / 2))
parallaxWallNode2.position = SCNVector3Make(15, 0, 250)
currentScene.rootNode.addChildNode(parallaxWallNode2)

//assign floor geometry to floor nodes
//floors

parallaxFloorNode1 = SCNNNode(geometry: floorGeometry)
parallaxFloorNode1.rotation = SCNVector4Make(0, 1, 0, Float(-
Double.pi / 2))
parallaxFloorNode1.rotation = SCNVector4Make(1, 0, 0, Float(-
Double.pi / 2))

parallaxFloorNode1.position = SCNVector3Make(15, 0, 0)
currentScene.rootNode.addChildNode(parallaxFloorNode1)

parallaxFloorNode2 = SCNNNode(geometry: floorGeometry)
parallaxFloorNode2.rotation = SCNVector4Make(0, 1, 0, Float(-
Double.pi / 2))
parallaxFloorNode2.rotation = SCNVector4Make(1, 0, 0, Float(-
Double.pi / 2))

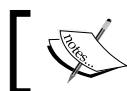
parallaxFloorNode2.position = SCNVector3Make(15, 0, 250)
currentScene.rootNode.addChildNode(parallaxFloorNode2)

}
```

OMG! That is a whole lot of code! But don't panic. We will go through it step by step. Just look at the code where it says preparing wall geometry. First, we will see how the wall geometry is set up and then, once you have an understanding of it, we will see how to set up the floor geometry.

We create a new constant called `wallGeometry` and assign an `SCNPlane` to it. The difference between an `SCNPlane` and `SCNFloor` is that, here, we can set the dimensions of the plane. So, very simply, we set the width and height of the plane to be 250 by 120 units.

Next, we assign a material to the plane. So far, we have only seen how to assign a color to an object in `SceneKit`. Here, we assign three kinds of maps to the plane. The first is a diffuse.



A diffuse material is an image or texture that you want to paste on a plane.



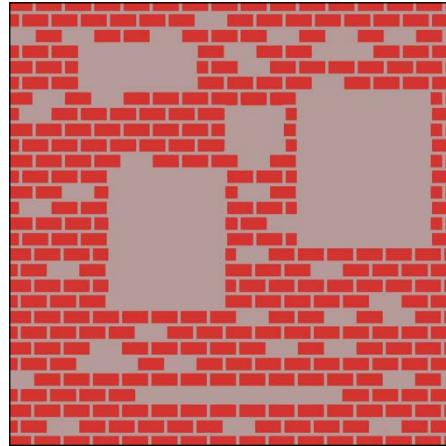
Textures are like wallpapers on walls. Imagine an unpainted wall. Now, you can either color the wall or apply wallpaper. Adding paint is done in the digital world by applying a color, as we did for the `enemyblock`, where we assigned a yellow diffuse color to it. To apply a wallpaper in the digital world, we use textures or images. Here, we apply `wall.png` to the wall plane geometry.

Notice that the wall plane is pretty big in terms of width and height. If we let it be, the `wall.png` image will have to stretch to fit the dimensions of the wall. We use the `wrapt` and `wraps` functions so that the wall texture gets repeated in both the x and y directions of the plane without stretching the wall texture. That is what happens in the next two lines. We are just repeating the wall texture over in both directions until it fills the whole plane.

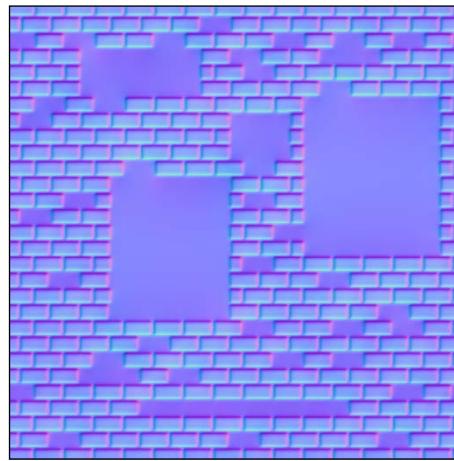
In the next step, we assign `mipFilter` to `linear`. The filter will decide how much detail needs to be added to the texture. If the camera is far enough away it will generate a lower resolution of texture to reduce the burden on the CPU. If the camera comes closer, then a higher resolution image will be created so that all the details of the texture will be visible. This is purely used for optimization purposes. Linear filter mode is the most basic filter type; there are other modes, called Bilinear and Trilinear, and so on, which give even better results but are computationally expensive. For our purposes, Linear filter will suffice. You can see the difference by changing the filter type and running the game on a device.

For the diffuse, at the end, we scale down the texture of the image itself depending upon how big or small we want the texture to appear on the plane. While scaling, we have to scale it in the same proportion as the size of the geometry. So, here, in the x and y plane, we scale it down by a factor of 1/40 times the values of the width and height of the geometry. Since we are not scaling in the z direction, we keep it at 1.

The Diffuse map of the wall is shown here:

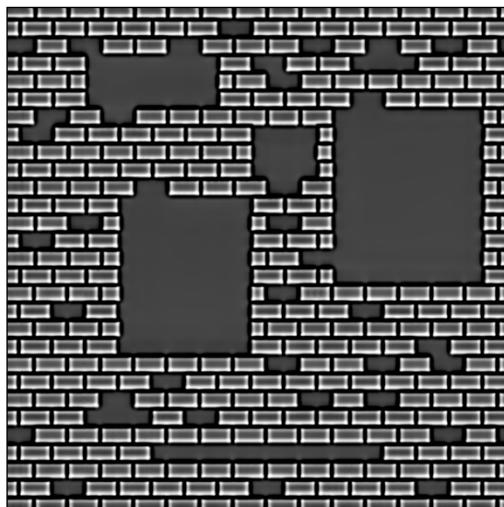


Now, the same five steps are repeated for the Normal and the Specular map. We saw that to add wallpaper to the plane we have to add a Diffuse map. Now, what if this wallpaper has some bumps and holes in it? Not all walls are so smooth. To add some roughness to the wallpaper, we use what is called a Normal map. A Normal map is shown here:



It basically calculates how light should behave once it hits the normal surface. Depending on the direction of the light and whether there is a bump or hole in the Normal map, the lighting will be calculated automatically, and all this is done using a Normal map. A normal map is nothing but an image. The code will take information from this image to create the desired effect. Here, we assign a normal map called `wall_NRM.png` and assign it as content. The next four steps are exactly the same as what we did for the diffuse map.

Next, let's look at Specular maps. This map will decide which parts of the texture are shiny and which parts are not. Imagine your wallpaper was made of stainless steel. To create this effect, we assign the specular map and to add it we use the `wall_SPEC.png` file. The remaining four steps are the same, only now we do it separately for the specular map. The specular map is shown here:



The same steps are repeated for the floor geometry too. Only, this time, we have flipped the width and height values.

Once the floor and wall geometries are ready, we assign them to the wall and floor nodes.

For the wall nodes, we assign the geometry to the node. Then, we rotate the node so that the wall is vertical. We rotate the node by -90 degrees in the y direction. Then, we place the first wall at (15, 0, 0), then add it to the scene. For the second wall node, we place it at (15, 0, 250), which will make it adjacent to the first plane.

For the floor nodes, we follow a similar process, but here we have to rotate it twice to be horizontal to the ground.

With our wall and floor nodes ready, we can update the position of the planes to create the parallax effect. So, in the update function, add the following code:

```
func update() {  
  
    parallaxWallNode1.position.z += -0.5  
    parallaxWallNode2.position.z += -0.5  
    parallaxFloorNode1.position.z += -0.5  
    parallaxFloorNode2.position.z += -0.5  
  
    if((parallaxWallNode1.position.z + 250) <= 0){  
        self.parallaxWallNode1.position = SCNVector3Make(15,  
                                                       0, 250)  
    }  
  
    if((parallaxWallNode2.position.z + 250) <= 0){  
        self.parallaxWallNode2.position = SCNVector3Make(15,  
                                                       0, 250)  
    }  
  
    if((parallaxFloorNode1.position.z + 250) <= 0){  
        self.parallaxFloorNode1.position = SCNVector3Make(15,  
                                                       0, 250)  
    }  
  
    if((parallaxFloorNode2.position.z + 250) <= 0){  
        self.parallaxFloorNode2.position = SCNVector3Make(15,  
                                                       0, 250)  
    }  
  
}  
  
}//class end
```

In the GameSCNScene class, add a new var called scrollingBackground and set it equal to the ScrollingBackground as follows:

```
var hero:Hero!  
var enemy :Enemy!  
var skScene:OverlaySKScene!  
var scrollingBackground=ScrollingBackground()
```

In the `init` function, in the `GameSCNScene` class, create the scrolling background:

```
skScene = OverlaySKScene(size: _size, gameScene: self)
scnView.overlaySKScene = skScene
skScene.scaleMode = SKSceneScaleMode.fill

self.scrollingBackground.create(currentScene: self)
```

And in the `update` function of the `GameSCNScene` class, add the line to update the scrolling background:

```
func update() {

    hero.update()
    scrollingBackground.update()

    if (!gameOver) {

        enemy.update()

    }
}
```

Run the project to see the background in action:



Adding particles

As the icing on the cake, we will include a rain particle effect. To create a particle effect in SceneKit, go to **File | New** and, under **Resource**, select **SceneKit Particle system**. Click **Next** and, on the next screen, select **Rain** from the **Particle System Template** dropdown. Click **Next** and give the file a name. I called it **rain**. Now you will have a **rain.scnp** and **spark.png** file in the project.

Add the following code at the end of the `init` function of the `GameSCNScene` class:

```
// add particle system
let rain = SCNParticleSystem(named: "rain", inDirectory:
                           nil)
rain!.warmupDuration = 10

let particleEmitterNode = SCNNNode()
particleEmitterNode.position = SCNVector3(0, 100, 0)

particleEmitterNode.addParticleSystem(rain!)
self.rootNode.addChildNode(particleEmitterNode)
```

We create a new constant called `rain`, assign `SCNParticleSystem` to it, and provide the `rain` particle system we created in it.

We use the `warmupDuration` of the particle system and assign a value of `10` to it. This is done so that, when the game starts, the rain particle effect is fast forwarded to look as if it was already raining.

A new `SCNNNode` called `particleEmitterNode` is created. Then, we assign the `rain` particle system to it. We then add the `particleEmitterNode` to the scene.

You can select the `rain.scnp` file and change the parameters to better suit your needs. Build and run to see our finished SceneKit game:



Adding character animation

To make sure that the correct animation gets played when the player runs and jumps, we have to extract the run and jump animations from the collada file.

In the Hero class's create function, after updating the anchor point, add the following code to get the run and jump animations:

```
// get the animation keys and store it in the anims
let animKeys = monsterNode.animationKeys.first
let animPlayer = monsterNode.animationPlayer(forKey:
animKeys!)
let anims = CAAnimation(scnAnimation: (animPlayer?.
animation)!)

// get the run animation from the animations
let runAnimation = Hero.animation(from: anims,
startingAtFrame: 31, endingAtFrame: 50)
runAnimation.repeatCount = .greatestFiniteMagnitude
runAnimation.fadeInDuration = 0.3
runAnimation.fadeOutDuration = 0.3
```

```
//set the run animation to the player
runPlayer = SCNAccelerationPlayer(animation:
SCNAcceleration(caAnimation: runAnimation))
monsterNode.addAnimationPlayer(runPlayer, forKey: "run")

// get the jump animation from the animations
let jumpAnimation = Hero.animation(from: anims,
startingAtFrame:81, endingAtFrame: 100)
jumpAnimation.repeatCount = .greatestFiniteMagnitude
jumpAnimation.fadeInDuration = 0.3
jumpAnimation.fadeOutDuration = 0.3

//set the jump animation to the player
jumpPlayer = SCNAccelerationPlayer(animation:
SCNAcceleration(caAnimation: jumpAnimation))
monsterNode.addAnimationPlayer(jumpPlayer, forKey: "jump")

//remove all the animations from the character
monsterNode.removeAllAnimations()

// play the run animation at start
monsterNode.animationPlayer(forKey: "run")?.play()
```

Also add the following functions to extract the animation information:

```
//Converts frames to time
static func time(atFrame frame:Int, fps:Double = 30) ->
TimeInterval {
    return TimeInterval(frame) / fps
}

// Given the frame range returns the offset and range of the animation
static func timeRange(forStartingAtFrame start:Int,
endingAtFrame end:Int, fps:Double = 30) -> (offset:TimeInterval,
duration:TimeInterval) {
    let startTime = self.time(atFrame: start, fps: fps)
    let endTime = self.time(atFrame: end, fps: fps)
    return (offset:startTime, duration:endTime - startTime)
}
// Returns the animation
static func animation(from full:CAAnimation, startingAtFrame
start:Int, endingAtFrame end:Int, fps:Double = 30) -> CAAnimation {
    let range = self.timeRange(forStartingAtFrame: start,
endingAtFrame: end, fps: fps)
    let animation = CAAnimationGroup()
```

```
    let sub = full.copy() as! CAAcceleration
    sub.timeOffset = range.offset
    animation.animations = [sub]
    animation.duration = range.duration
    return animation
}
```

Add `playRunAnim` and `playJumpAnim` to load and play the correct animations:

```
func playRunAnim() {
    monsterNode.removeAllAnimations()
    monsterNode.addAnimationPlayer(runPlayer, forKey: "run")
    monsterNode.animationPlayer(forKey: "run")?.play()
}

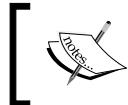
func playJumpAnim() {
    monsterNode.removeAllAnimations()
    monsterNode.addAnimationPlayer(jumpPlayer, forKey: "jump")
    monsterNode.animationPlayer(forKey: "jump")?.play()
}
```

Finally, change the `update` function so that the correct animation is played when the character is grounded and in the air:

```
func update() {
    //print("hero y pos: %f", self.presentation.position.y)

    if(self.presentation.position.y < 4.0) {
        if(isGrounded == false){
            playRunAnim()
            isGrounded = true
        }
    }else{
        if(isGrounded == true){
            playJumpAnim()
            isGrounded = false
        }
    }
}
```

Now the correct animation should play when the hero runs and jumps.



Checkpoint B

The code up to this point is available in the chapter's code resources.



Summary

In this chapter, we saw how to make a 3D game in SceneKit. From making simple geometries to making floors, we created a fully-fledged game with a complete game loop. We added a scene already created in a 3D software package with animation and imported it into SceneKit.

We imported the COLLADA objects into the scene and saw how to access the objects through code. We added an enemy and physics to the scene. We used SceneKit's physics engine to calculate collision and also applied force to hero objects.

Additionally, we also saw how to integrate SpriteKit into SceneKit to display the score and buttons on the scene. We also used SpriteKit's `touchBegan` function to detect touches on the screen and created the play and jump buttons.

Parallax scrolling was also added to the scene using planes. Also, we looked at different types of maps, such as diffuse, normal, and specular maps, and the functionality of each. Finally, we added a rain particle system and character animation to the scene.

12

Choosing a Monetization Strategy

It is a great time to be an indie game developer. There are an abundance of monetization options and game marketplaces. We can easily connect our projects with people looking to play our genre and style. To stand out, you will need to spend time and energy on marketing your project, while considering alternative routes such as crowdfunding.

There are more freemium games than ever before; it is now very possible to make money strictly from ad revenue or in-app purchases. Additionally, options exist to expand your market by making your app available in other regions. You can set yourself up for success by considering the market and aligning your game's style to the demand that exists. It is also important to manage your projects wisely so that you can complete and publish your games. Each of these topics is very in-depth, but we will spend time exploring the basics to give you an idea of how to spend your marketing time.

Topics in this chapter include the following:

- Developing your marketing plan
- Leveraging crowdfunding
- Showing display ads for revenue
- Adding an Admob Ad in the App
- Selling in-app purchases
- Localization in foreign markets
- Managing scope and completing projects

Developing your marketing plan

Indie game development has never been more accessible. This is fantastic news, but it also means that we face a sea of noise and competition when we try to attract attention to our efforts. Further complicating matters, creating and executing a marketing plan requires different skills than those required to create a video game. Many developers would rather spend their time coding or drawing versus building awareness through social outreach. Luckily, there are many resources available to help us get started.

When to start marketing

Many games have their best sales day on the day they are launched. This means that you need to start building public awareness of your project early in development so your popularity peaks as you release your game. You need to time your publicity to coincide with your release date in order to maximize your sales window.

As soon as you have some representative material, publish it! Take it to social media, Reddit, and game forums. You will get valuable feedback and start to create awareness of your project. The indie developer community is very friendly and welcoming of new material.

Marketing checklist

Here are some proven steps to build awareness of your game and yourself as an indie developer:

- Website and development blog: You will need a basic website so that you show up in search engine results. It is beneficial to keep a regular development blog where you post about interesting challenges you are solving or simply share your latest progress. Though it can be hard to spend time writing a blog post instead of implementing a new system in your game, it will pay off as you build a fan base and awareness. For instance, my development blog led to me writing this book!

- Social media: The game developer community on Twitter is bursting with activity. Engaging with other developers on Twitter is a fantastic way to gauge interest in your genre and spread awareness of your project. Try to build authentic relationships with other members to grow your fan base. This can pay off if someone with a large following retweets a post about your game. And if you are using hashtags (such as #gamedev) to spread awareness, try to time your posts during Twitter's busiest hours—usually weekends and evenings.
- Video trailers: On macOS, you can use QuickTime to easily record screen-capture videos and then edit them with iMovie. You do not need anything fancy, just a simple video showing off the unique mechanics and fun of your game. Post your trailers to YouTube and link them from Twitter and your developer blog. Additionally, you can take small snippets and make GIFs to share. Gifs are one of the best formats to communicate your game quickly.
- Screenshot Saturday: #screenshotsaturday is a widely used hashtag in the indie game community. You can find it on Twitter, Facebook, Reddit (there is a weekly thread in /r/gamedev), and others. You should definitely be making gifs of your weekly progress and sharing them with the community each Saturday. It's a fun way to get feedback and create buzz.
- Game journalists and blogs: Cold calling journalists, a socially awkward activity, does not come easily to many game developers. However, it is vitally important to get the exposure from features on game websites and blogs, especially leading up to your launch date. One article on the right blog can be your primary source of traffic. There is no perfect formula, but try to reach out to blogs that write about similar games and genres, be friendly and humble, and make sure to give them plenty of material to write about. You can also create an easy-to-use press kit that you can hand off with all of the relevant information a journalist will need to write a story.
- `presskit()`: It is a simple, open source tool to generate a press-ready site where you can consolidate media and information about your game to distribute to journalists. Many game journalists will already be familiar with this format, which may make things easier for them. Making it easy for a journalist or blogger to write about you is a winning strategy. You can find `presskit()` at <http://dopresskit.com/>.

Leveraging crowdfunding

Many developers are turning to crowdfunding platforms such as Kickstarter and Indiegogo to fund their projects. These sites allow you to put your unfinished product before the world and ask supporters to chip in monetarily to make it a reality. You ask for a small amount of money from a great number of people. This is an excellent way to gather funds if your idea resonates well with the public.

Crowdfunding opens up many new possibilities, but successful crowdfunding campaigns create their own success by hyping their projects through the press and social media. Crowdfunding campaigns that launch with pre-existing fan bases are most likely to succeed.

Pros and cons of crowdfunding

One of the best aspects of crowdfunding is that it forces developers to create all the necessary marketing materials for their games. Even if your crowdfunding campaign is unsuccessful, you will gain all the materials and assets you need to market your game in the process. On the other hand, you will need to devote extensive time to your campaign, potentially distracting from development (especially if you are a solo developer).

Crowdfunding campaigns need a lot of outside publicity to be successful. Simply having a Kickstarter page is not enough to get a campaign noticed. The campaigns that manage to attract the attention of the game media are campaigns that gain traction. Once you do have some awareness, however, you may gain extra press just for having a crowdfunding campaign running.

An additional advantage to crowdfunding is that it lets you quickly test the demand for your genre and style of game. If you are able to find backers for your project, it is likely something that gamers would be willing to purchase. This bodes well for the financial success of your game.

If you do decide to use crowdfunding, spend some time researching campaigns that have succeeded and try to mimic what they did well. Crowdfunding takes serious time and effort; do not expect it to be an easy source of money. It pays off if you are able to communicate a great idea and find a willing group of fans who are looking for what you are making.

Showing display ads for revenue

The iOS App Store market favors free games that show ads or provide in-app purchases to make money. Free games tend to get more attention than games that cost money up front, and going with a freemium model is a good idea if you do not already have massive exposure for your project. Mobile advertising is a large, growing business; there are plenty of options if you choose to shows ads in your games.

The upsides to showing ads

Ads can be the most straightforward approach to monetizing your game, assuming you can build the traffic volume to make an impact. Here are some reasons to use ads in your game:

- When done right, users do not mind ads. Some developers work them into the gameplay by giving in-game rewards for watching a video ad. This seems to be a winning formula: it results in higher revenue per user, and users are generally not annoyed by this style of ad. Unity conducted a study that showed that *71 percent of players choose watching video ads as their preferred way to 'pay' for game content* (full study available here: <http://response.unity3d.com/in-game-advertising-the-right-way-monetize-engage-retain-whitepaper>).
- Ads do not take much development work, compared to quality in-app purchases. All you need to do is configure and implement your ad networks (the companies you partner with to deliver your ad content) and you are free to focus on making the best game possible. Contrast that with the work of constantly creating new game content to sell to your players in small bits, even after launch, for in-app purchases. Most ad networks provide an SDK to speed up the implementation process.
- Ads allow you to provide your full game up front and free, meaning you will receive more installations and your players will get to use the full game without limits or additional purchases (unless you choose to use both ads and in-app purchases together). Free games experience a tremendous amount of downloads over games that charge, so allowing your players to download your game freely is a solid strategy if you do not have an existing fan base.

The downsides to showing ads

There are definitely reasons to skip the ad networks and choose another monetization strategy, including the following:

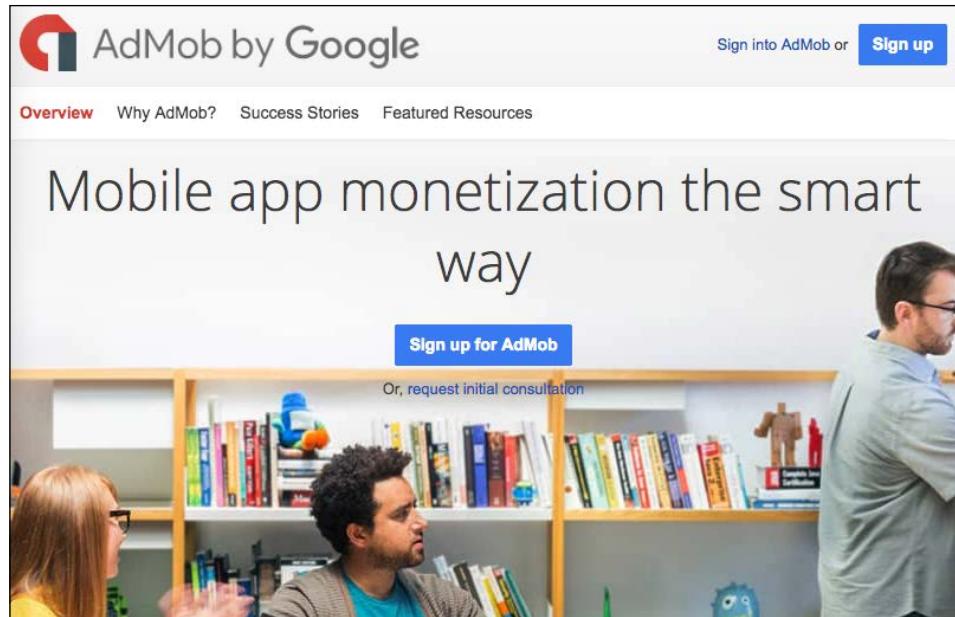
- Showing ads can damage your own brand—especially banner ads that take up screen real estate during gameplay. Misuse of ads makes a game look cheap and mass produced. Think about how your ads may influence your fans' view of your game itself, assuming you want to carry a fan base forward to your next game.
- Ads can be annoying if implemented hastily. If you do choose to use ads, please do so in a tactful manner. For instance, interstitial video ads between levels are less offensive than ads that take up screen space during gameplay.
- You need a high volume of players to make ads lucrative. No monetization strategy succeeds without volume, but ads are particularly worthless if you cannot generate a thriving fan base.
- If you do choose to go with ads, you can choose from many ad networks. New ad networks are constantly springing up, while many developers are finding success with some of the most popular networks (such as Google's AdMob and Chartboost).

Adding an AdMob Ad in the App

Let's see how to implement ads in the game. We will be using AdMob to integrate an ad into the game.

You have to sign up for AdMob. If you don't already have an account, you can sign up at www.google.com/admob. Signing up is easy and it's free. If you have a Gmail account, you can use it to sign in as well.

www.google.com/admob



To implement the ad, we have to first create an app on the AdMob site and then download the AdMob framework to implement the ad in the game. Click on the Apps button on the sidebar of the site and click **Add App**:

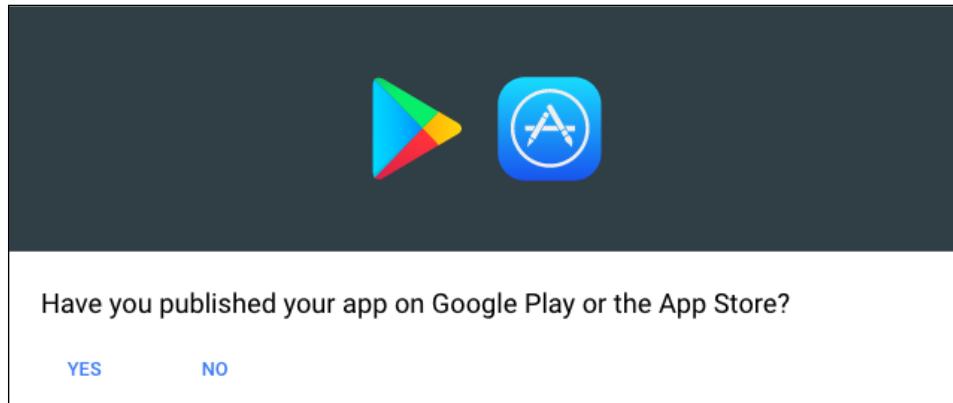
The screenshot shows the AdMob 'Home' page. The sidebar on the left contains icons for Home, Library, Network, Monetization, Settings, and Help. The main content area has a red header bar with the 'Home' title and a yellow banner at the top right with a warning about a policy update for iPhone X and a 'LEARN MORE' link. Below this, there's a section for 'Network earnings' showing '\$0.00' for yesterday and last month. To the left of the earnings, a list of apps is shown:

App Name	Platform
Unknown	
Jumpy Jumps	iOS
lingerers	Free Android
lingerers	Free iOS
pizZap Mania Free	Free iOS
pizZapMania	Free Android

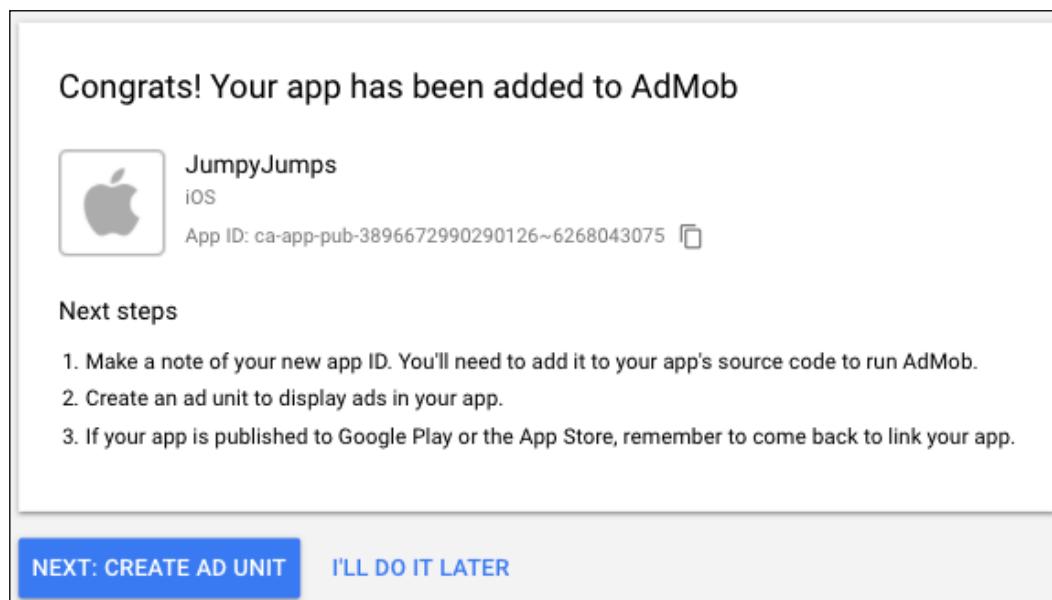
At the bottom of the sidebar, there are 'ADD APP' and 'VIEW ALL APPS' buttons. The bottom right corner of the main content area says 'work performance'.

Choosing a Monetization Strategy

Next, you will be asked if the app is already published. Click **NO** as we have not yet published the app on the App Store:



Next, enter the app name and select the platform. I named the App JumpyJumps and selected the platform as iOS. You will be greeted with a screen giving you the App ID. Make a note of this as it will be required when we add the ad in the game:



Congrats! Your app has been added to AdMob

 JumpyJumps
iOS
App ID: ca-app-pub-3896672990290126~6268043075

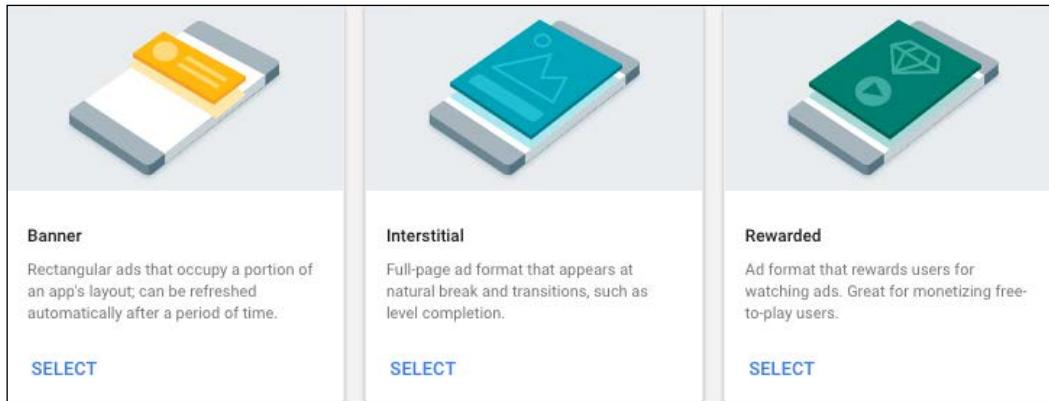
Next steps

1. Make a note of your new app ID. You'll need to add it to your app's source code to run AdMob.
2. Create an ad unit to display ads in your app.
3. If your app is published to Google Play or the App Store, remember to come back to link your app.

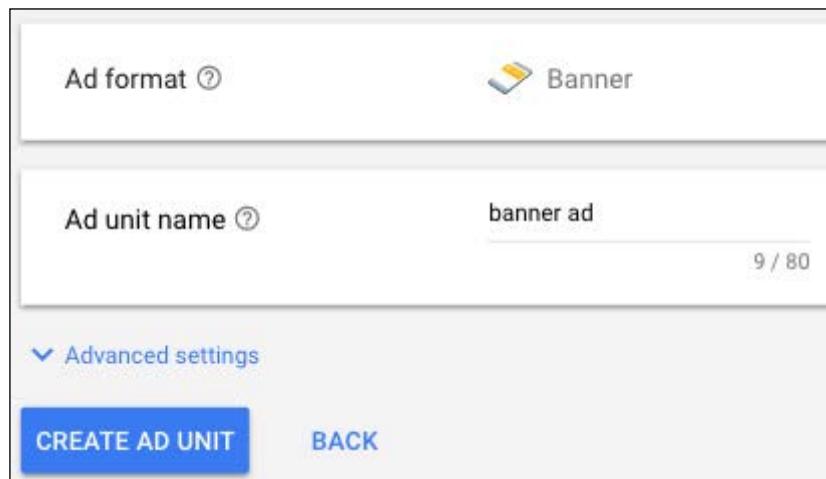
NEXT: CREATE AD UNIT **I'LL DO IT LATER**

Next, we have to create an Ad unit. Click **Next**.

Here, we will have to select what type of ad we want to create for the application. There are three options: **Banner**, **Interstitial**, and **Rewarded**:



For the game, we will create a banner ad that will sit at the bottom of the game. Click **Select** under **Banner**. Then, on the next screen, give the ad a name so that it will be possible to search for it later if required. Click on **Create Ad Unit** to proceed:



Choosing a Monetization Strategy

Great! Your ad unit has been created. You will get the ad unit ID, which will also be required to implement the ad, so make note of it as well:

Next, place the ad unit inside your app

Follow these instructions:

1. Complete the instructions in the [Google Mobile Ads SDK guide](#) using this app ID:



ca-app-pub-3896672990290126-6268043075

2. Follow the [banner implementation guide](#) to integrate the SDK. You'll specify ad type, size, and placement when you integrate the code using this ad unit ID:



ca-app-pub-3896672990290126/5697977684

3. Review the [AdMob policies](#) to ensure your implementation is compliant.

[EMAIL INSTRUCTIONS](#)

We still have to download the Google Mobile Ads SDK. Go to <https://developers.google.com/admob/ios/download> and click on the `googlemobileadssdkios.zip` link to start downloading the SDK:

Google Mobile Ads SDK

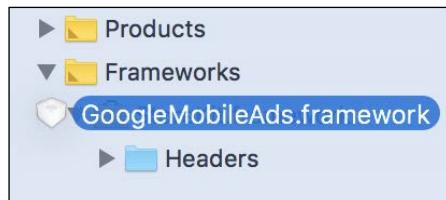
★★★★★

The Google Mobile Ads SDK is offered as a download for iOS. Your use of the Google Mobile Ads SDK is governed by the Google Developers Site [Terms of Service](#).

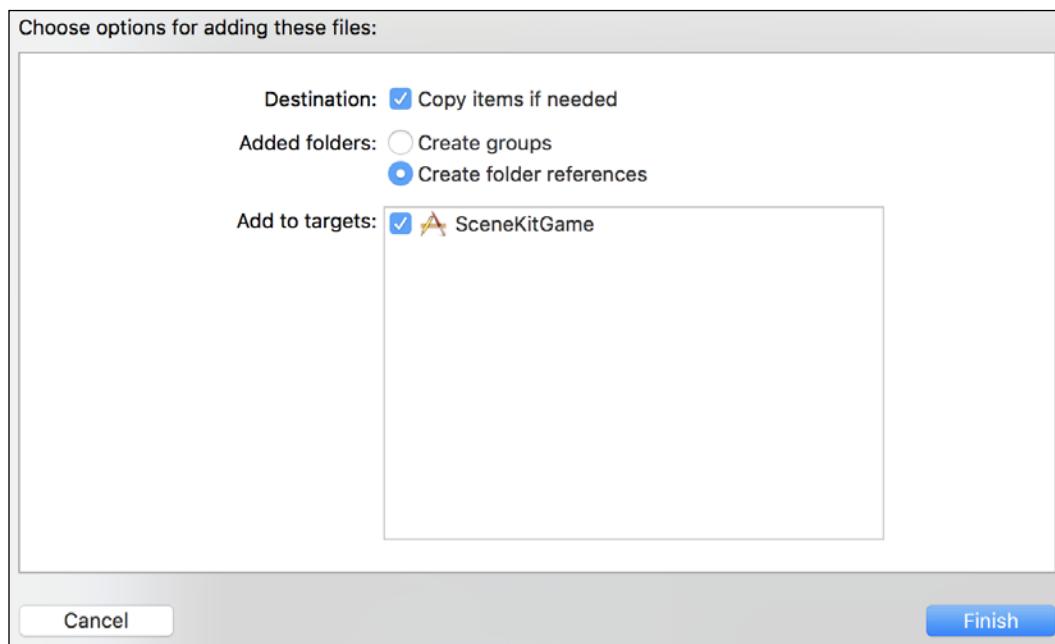
★ Note: Instead of downloading and installing the SDK directly, we recommend using [CocoaPods](#) to make it easier to manage library dependencies for your Xcode projects.

Version	
	7.26.0
Package	
	googlemobileadssdkios.zip

Once downloaded, drag and drop the SDK under the frameworks section in the project:



Make sure that **Copy items if needed** is checked:



Next comes the fun part of actually adding the ad in the app.

We have to first set the App ID in the application. In the Project, go to `AppDelegate.swift` and import the `GoogleMobileAds` framework at the top. In the `didFinishLaunchingWithOptions` function application **Options**, set the **Application ID**. The top of the `AppDelegate` class should look like this:

```
import UIKit  
import GoogleMobileAds  
  
@UIApplicationMain
```

```
class AppDelegate: UIResponder, UIApplicationDelegate {  
  
    var window: UIWindow?  
  
    func application(_ application: UIApplication,  
didFinishLaunchingWithOptions launchOptions:  
[UIApplicationLaunchOptionsKey: Any]?) -> Bool {  
        // Override point for customization after application launch.  
  
        GADMobileAds.configure(withApplicationID: "ca-app-  
pub-3896672990290126~8662417917")  
  
        return true  
    }  
}
```

Next, we will set the ad unit ID and create a banner view so that the ad displays. In the `GameViewController.swift` file, import the `GoogleMobileAds` framework. Make the class inherit from `GADBannerViewDelegate` so that the current class can receive call backs and, in the `viewDidLoad` function, set the banner type, set the ad unit ID, set the root view to self, load the `GADRequest`, and set the delegate to self:

```
import UIKit  
import QuartzCore  
import SceneKit  
  
import GoogleMobileAds  
  
class GameViewController: UIViewController, SCNSceneRendererDelegate,  
GADBannerViewDelegate{  
  
    var gameSCNScene: GameSCNScene!  
    var bannerView: GADBannerView!  
  
    override func viewDidLoad() {  
        super.viewDidLoad()  
  
        let scnView = view as! SCNView  
        scnView.delegate = self  
  
        gameSCNScene = GameSCNScene(currentview: scnView)
```

```
bannerView = GADBannerView(adSize:  
kGADAdSizeSmartBannerLandscape)  
bannerView.adUnitID = "ca-app-pub-3896672990290126/1081287272"  
bannerView.rootViewController = self  
bannerView.load(GADRequest())  
bannerView.delegate = self  
  
}
```

Once an ad is received, the `adViewDidReceive` function will be called, in which we'll add a banner ad.

Add the `adViewDidReceieveAd` function as follows, which in turn will call an `addBannerView` function:

```
// Tells the delegate an ad request loaded an ad.  
func adViewDidReceiveAd(_ bannerView: GADBannerView) {  
  
    print("adViewDidReceiveAd")  
    addBannerView(bannerView)  
  
}  
  
func addBannerView(_ bannerView: GADBannerView){  
  
    bannerView.translatesAutoresizingMaskIntoConstraints = false  
    view.addSubview(bannerView)  
    view.addConstraints(  
        [NSLayoutConstraint(item: bannerView,  
                           attribute: .bottom,  
                           relatedBy: .equal,  
                           toItem: bottomLayoutGuide,  
                           attribute: .top,  
                           multiplier: 1,  
                           constant: 0),  
        NSLayoutConstraint(item: bannerView,  
                           attribute: .centerX,  
                           relatedBy: .equal,  
                           toItem: view,  
                           attribute: .centerX,  
                           multiplier: 1,  
                           constant: 0)  
    ]  
}
```

And that's it. If you run the application now, you will see an ad at the bottom of the screen:



Selling in-app purchases

In-app purchases have grown at an astounding rate since Apple launched the App Store. Love them or hate them, in-app purchases are a winning monetization strategy that works for many developers and companies. When implemented well, in-app purchases encourage players to spend extra time playing your game, exploring every expanse and subplot.

On the other hand, in-app purchases can corrupt gameplay for the sake of monetization. There are entire books on the subject of using psychological gameplay tricks to keep players hooked on in-app purchases, and we have seen this style used a lot in the App Store (with financial success). Still, it is possible to create an in-app purchase system that is both successful and healthy for your players. Let's explore some strategies.

In-app purchase strategies

Successfully implementing an in-app purchase system is a very deep topic, but we will briefly examine some proven strategies:

- Make sure to provide enough free content that your players are hooked and are willing to pay more for additional levels, skins, or game time on top of the already solid foundation of gameplay.
- If you allow players to work toward in-app purchases using in-game currency, randomize the rewards they receive in-game. Randomized rewards are more compelling and keep the player hooked, hoping for the chance of a great reward roll on their next opportunity.
- Provide samples of in-app purchasable content. By mixing purchasable content into free gameplay, users will learn how the special content works and will be more interested in purchasing it.
- If you have a semi-successful in-app purchase game, keep developing! Once you have proven that gamers are interested in paying for your system, you are likely to make more money by adding onto the existing game than by starting a new project. This means that if you choose in-app purchases, you should continue to support your existing games with new content buildout after launch.
- Run daily features, limited-time content, and sales around holidays. These types of sales tactics increase sales.

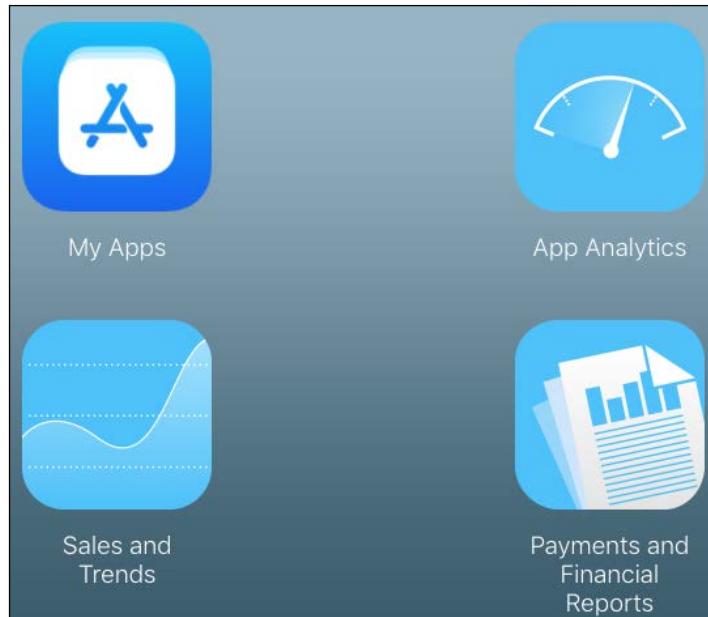
A word about farming your players

Without getting too philosophical, I prefer not to use gameplay-based in-app purchases, because they create a conflict of interest for the game developer. Rather than focusing on creating a great game, you are incentivized to create a game that evokes specific psychological and emotional responses in your players in order to prompt repeated purchases. Some of that is true for any monetization strategy, but in-app purchases in particular can end up dictating gameplay design in order to make the most money.

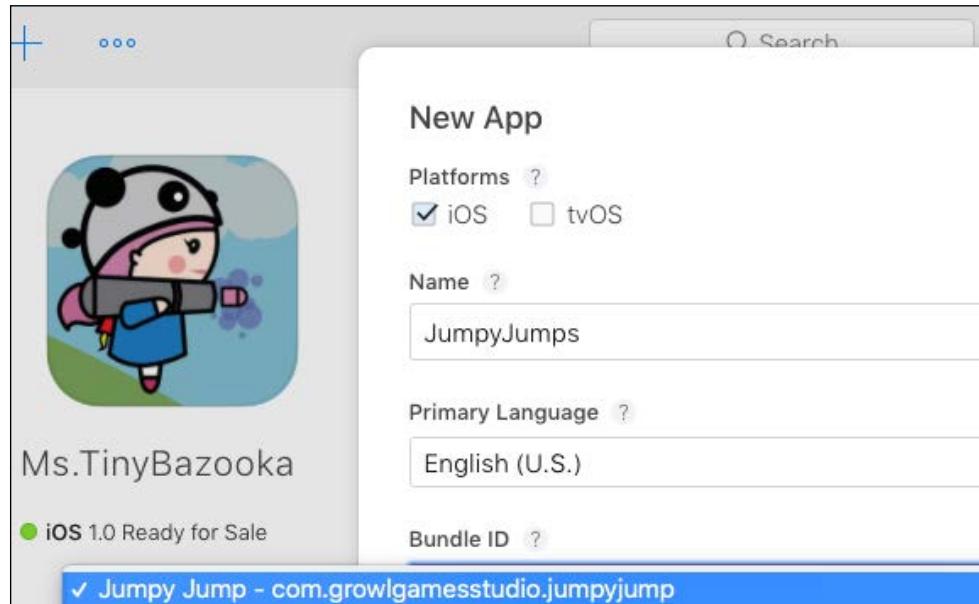
Adding In-App Purchases

We will add a very simple in-app purchase in the game so that, if the player wishes, they can pay the minimum amount to remove the ad at the bottom of the screen.

For this, we need a paid developer account to add in-app purchases. The annual fee is 99 USD, and with this you can publish on the iOS / macOS App Store. Once you have paid Apple the amount, signed the contract, and updated your tax and banking information, go to itunesconnect.apple.com and **My Apps**:



Create a new app by clicking on the + sign in the upper-left corner of the page:



Select the **Platform**, **Name**, **Primary Language**, **Bundle ID**, and **SKU**.

If you don't have a **Bundle ID**, you will need to create one. For **SKU**, since this is the fifth game, I chose **005**.

To create a new in-app purchase, from the tabs at the top of the screen, select **Features** and select **In-App Purchases**. Click on the + sign to create a new In-App Purchase.

Here, you will select the type of In-App purchase:

Select the in-app purchase you want to create.

Consumable
A product that is used once, after which it becomes depleted and must be purchased again.
Example: Fish food for a fishing app.

Non-Consumable
A product that is purchased once and does not expire or decrease with use.
Example: Race track for a game app.

Auto-Renewable Subscription
A product that allows users to purchase dynamic content for a set period. This type of subscription renews automatically unless cancelled by the user.
Example: Monthly subscription for an app offering a streaming service.

Non-Renewing Subscription
A product that allows users to purchase a service with a limited duration. The content of this in-app purchase can be static. This type of subscription does not renew automatically.
Example: One-year subscription to a catalog of archived articles.

[Learn more about In-App Purchases.](#) [Cancel](#) [Create](#)

We want a Consumable In-App purchase so that it can be purchased just once.

Next, we have to provide a **Reference Name, Product Name, Pricing, Display Name, and Description**:

The screenshot shows the 'In-App Purchases > New In-App Purchase' screen. On the left, a sidebar lists 'IN-APP PURCHASES' (selected), 'In-App Purchases' (highlighted in blue), 'App Store Promotions', 'Game Center', 'Encryption', and 'Promo Codes'. The main area has tabs for 'Reference Name' (containing 'removead'), 'Product ID' (containing 'removead'), 'Availability' (checkbox checked for 'Cleared for Sale'), 'Pricing' (showing 'NZD 1.49 (Tier 1)' and currency dropdown for 'Other Currencies' with 'Dec 10, 2017' start date and 'No End Date'), and 'App Store Information' (with 'Display Name' 'Remove ads' and 'Description' 'Remove Ads'). A 'Save' button is at the top right.

For the **Reference name** and **Product ID**, put in `removead`. For pricing, I chose the lowest tier, which is equal to 1 USD, and for **Display Name** and **Description**, I put in **Remove ads**.

Once done, click **Save**. Now `removead` will show as one of your In-App purchases. Now let's go back to the project and implement the In-App purchase.

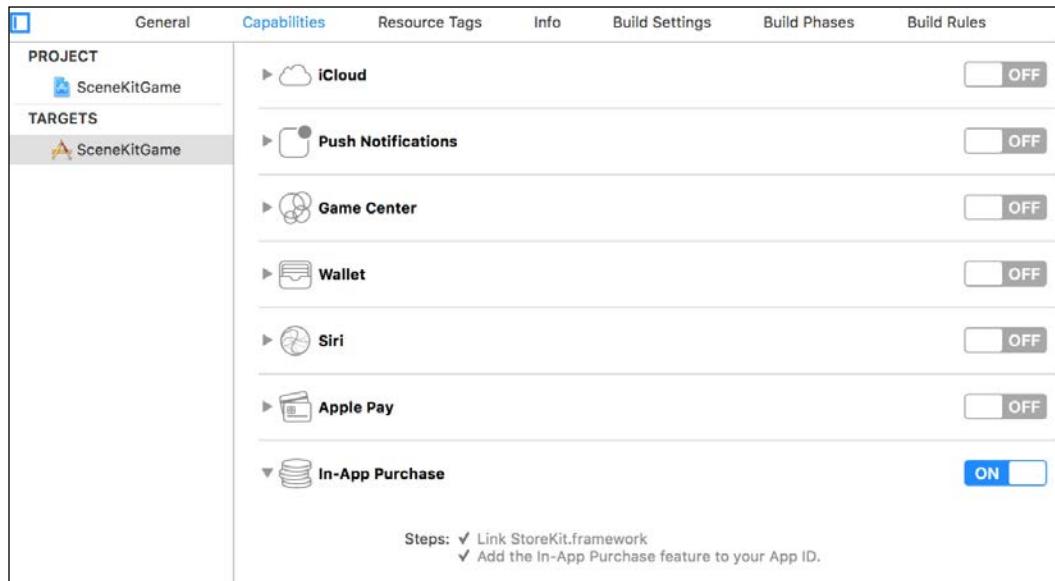
Choosing a Monetization Strategy

First of all, since we changed the **Bundle ID**, we have to make sure the **Bundle ID** matches:



Go to **Project | General Setting | Bundle Identifier** and change it to reflect the Bundle ID you just created. It is okay if the name doesn't match—we will change it later—but the bundle ID is most important for Apple to identify the right app.

Since we are here, go to the **Capabilities** tab and turn the **In-App Purchases** on:



This will link the StoreKit framework to the project.

Next, in the `OverlaySKScene`, we'll add a button called `noAds` so that when the player wishes to purchase the `removead` In-App purchase, we can initialize the purchase.

Create a global variable called `noAdsBtn` at the top of the class:

```
var jumpBtn: SKSpriteNode!
var playBtn: SKSpriteNode!
var noAdsBtn: SKSpriteNode!
```

And at the bottom of the `init` function, add the following:

```
noAdsBtn = SKSpriteNode(imageNamed: "noAdsBtn")
noAdsBtn.position = CGPoint(x: size.width * 0.9, y: size.height *
0.8)
noAdsBtn.setScale(0.75)
self.addChild(noAdsBtn)
noAdsBtn.name = "noAdsBtn"
```

We only want to show the button when the game is not running so, in the `startGame` function, we hide the button and, in the `GameOver` function, we unhide the button as follows:

```
func startGame() {
    gameOver = false
    skScene.jumpBtn.isHidden = false
    skScene.myLabel.isHidden = false
    skScene.playBtn.isHidden = true
    skScene.gameOverLabel.isHidden = true

    skScene.noAdsBtn.isHidden = true

    score = 0
    skScene.myLabel.text = "Score: \(score)"

}

func GameOver() {
    skScene.jumpBtn.isHidden = true
    skScene.playBtn.isHidden = false
    skScene.gameOverLabel.isHidden = false

    skScene.noAdsBtn.isHidden = false
}
```

```
// Reset hero and enemy position  
enemy.position = SCNVector3Make(0, 2.0, 60.0)  
hero.position = SCNVector3Make(0, 5, 0)  
}
```

The assets for the button are provided in the project. Add the button image to the project as we have done before.

In the `touchesBegan` function, add the functionality to press the **no ads** button:

```
override func touchesBegan(_ touches: Set<UITouch>, with event:  
UIEvent?) {  
  
    for touch: AnyObject in touches {  
        let location = touch.location(in: self)  
  
        let _node: SKNode = self.atPoint(location);  
  
        if (_gameScene.gameOver == false){  
            if(_node.name == "jumpBtn"){  
                _gameScene.heroJump()  
            }  
            }else{  
                if(_node.name == "playBtn"){  
                    _gameScene.startGame()  
                } else if (_node.name == "noAdsBtn"){  
                    inAppPurchase()  
                }  
            }  
        }  
    }  
}
```

At the top of the class, we create three variables:

```
var request: SKProductsRequest!  
var products: [SKProduct] = []  
var noAdsPurchased = false
```

`SKProductRequest` is for requesting any product information, `products` is an array to store all the products that we have for this app, and the third is a Boolean to check whether the purchase was made or not.

We create the `inAppPurchase` function as follows:

```
func inAppPurchase() {  
  
    let alert = UIAlertController(title: "In App Purchases",  
        message: "", preferredStyle: UIAlertControllerStyle.alert)  
  
    for i in 0 ..< products.count{  
        let currentProduct = products[i]  
  
        if(currentProduct.productIdentifier == "removead" &&  
            noAdsPurchased == false){  
  
            print("removead found")  
  
            let numberFormatter = NumberFormatter()  
            numberFormatter.numberStyle = .currency  
            numberFormatter.locale = currentProduct.priceLocale  
  
            alert.addAction(UIAlertAction(title:currentProduct.localizedTitle + " : " + numberFormatter.string(from: currentProduct.price)!, style: UIAlertActionStyle.default){_ in  
  
                self.buyProduct(product: currentProduct)  
  
            })  
        }  
    }  
  
    if(noAdsPurchased == false){  
        alert.addAction(UIAlertAction(title: "Restore", style: UIAlertActionStyle.default){_ in  
            self.restorePurchaseProducts()  
        })  
    }  
  
    alert.addAction(UIAlertAction(title:"Cancel",style:UIAlertActionStyle.default){_ in  
        print("cancelled purchase")  
    })  
  
    _gameScene.scnView?.window?.rootViewController?.present(alert,  
        animated: true, completion: nil)  
  
} //inapppurchase
```

Once the **no ads** button is pressed, the function gets called. In the function, we create three buttons. The first one loops through all the In-App purchases for the app. We check whether the current product removeads if true, and we also check whether we have already purchased the product. It shows the Remove Ads In-App purchase with the description and the amount in the local currency and value. If the button is pressed, then the `buyProduct` function is called.

If the app has not been purchased, then we add the second button, which gives an option to restore the purchase. The restore option is usually relevant to Non-Consumable products, but I added it here anyway. If you wish to add it to Non-Consumable products, if clicked, it will call the `selfRestorePurchaseProduct` function.

The third button is the cancel button, enabling the user to cancel the scene.

Finally, we add the `Alert` to the `gameScene`'s root view controller for the Alert to display.

When the player clicks the button to purchase the product, `buyProduct` gets called, which in turn calls the `paymentQueue` function, which starts the product's payment process. Add the `buyProduct` and `paymentQueue` functions as follows:

```
// Buy the product
func buyProduct(product: SKProduct) {
    let payment = SKPayment(product: product)
    SKPaymentQueue.default().add(payment)
}

//Called when processing the purchase
func paymentQueue(_ queue: SKPaymentQueue, updatedTransactions transactions: [SKPaymentTransaction]) {

    for transaction in transactions as! [SKPaymentTransaction] {

        switch(transaction.transactionState) {

            case .purchasing:
                print("Transaction State: Purchasing")
            case .purchased:
                if transaction.payment.productIdentifier == "removead" {
                    print("Transaction State: Purchased")
                    handleNoAdsPurchased()
                }
        }
    }
}
```

```
queue.finishTransaction(transaction)
case .failed:
    print("Payment Error: %@", transaction.error!)
    queue.finishTransaction(transaction)
case .restored:
    if transaction.payment.productIdentifier == "removead"
{
    print("Transaction State: Restored")
    handleNoAdsPurchased()
}
queue.finishTransaction(transaction)
case .deferred:
    print("Transaction State: %@", transaction.
transactionState)
} //switch
} //for loop
} //payment que
```

Once the purchase is in the payment queue, you can check which stage the payment is at. Is it Purchasing, purchased, failed, restored, or deferred? If the payment is successful, we call our `handleNoAdsPurchased` function, in which we can set the logic for what to do if the payment is made. Add `handleNoAdsPurchased` as follows:

```
func handleNoAdsPurchased() {
    noAdsPurchased = true
    noAdsBtn.isHidden = true
    UserDefaults.standard.set(true, forKey: "removeAdsKey")
    let controller = _gameScene scnView?.window?.
rootViewController as! GameViewController
    controller.removeAd()
}
```

Here, we set the `noAdsPurchased` Boolean to true, hide the `noAds` button, set the user default key to true, so that next time the app loads we won't add the `noAds` button to the game, and finally, we will remove the banner ad from the game.

In the GameViewController class, add a new function as follows to remove the banner ad:

```
func removeAd() {
    print("removed ad")

    bannerView.removeFromSuperview()
}
```

Back in the OverlaySKScene, we need to add a couple more functions as follows:

```
// Initialize the App Purchases
func initInAppPurchases() {
    print("In App Purchases Initialized")

    SKPaymentQueue.default().add(self)

    // Get the list of possible purchases
    if self.request == nil {
        self.request = SKProductsRequest(productIdentifiers:
Set(["removead"]))
        self.request.delegate = self
        self.request.start()
    }
}

//Called when appstore responds and populates the products array
func productsRequest(_ request: SKProductsRequest, didReceive
response: SKProductsResponse) {

    print("products request received")

    self.products = response.products
    self.request = nil

    print("products count: ", products.count)

    if(response.invalidProductIdentifiers.count != 0) {
        print(" *** products request not received ***")
        print(response.invalidProductIdentifiers.description)
    }
}
```

```
//Restore purchases
func restorePurchaseProducts() {
    SKPaymentQueue.default().restoreCompletedTransactions()
}

//Called when an error happens in communication
func request(_ request: SKRequest, didFailWithError error: Error)
{
    print(error)
    self.request = nil
}
```

The `initInAppPurchase` needs to be called to initialize an In-App purchase at the start of the application. The `Product Request` function gets called by the app to get all the In-App Purchases for the current app. Then, we have the `restorePurchase` function, which gets called when we restore a purchase, and finally, we have the `request` function, which checks for errors in communication during the In App Purchase process.

Next, we have to make changes to the project so that the In-App purchase is initialized only when the no ads In-App purchase is still available and doesn't show up once the purchase has been made.

In the `OverlaySKScene`, add the **no ads** button and initialize the `InAppPurchase` only if the `removeAdsKey` is false, as follows:

```
if(!UserDefaults.standard.bool(forKey: "removeAdsKey")) {

    initInAppPurchases()

    noAdsBtn = SKSpriteNode(imageNamed: "noAdsBtn")
    noAdsBtn.position = CGPoint(x: size.width * 0.9, y: size.height * 0.8)
        noAdsBtn.setScale(0.75)
        self.addChild(noAdsBtn)
        noAdsBtn.name = "noAdsBtn"
    }
```

In the `GameViewController` class, add the banner ad only if the no ads purchase has not been made:

```
if(!UserDefaults.standard.bool(forKey: "removeAdsKey")) {
    bannerView = GADBannerView(adSize:
kGADAdSizeSmartBannerLandscape)
    bannerView.adUnitID = "ca-app-
pub-3896672990290126/1081287272"
    bannerView.rootViewController = self
```

```
    bannerView.load(GADRequest())
    bannerView.delegate = self
}
```

The noAds button will be either hidden or visible in the startGame and Gameover function of the GameSCNScene if the purchase hasn't been made yet:

```
func startGame() {

    gameOver = false
    skScene.jumpBtn.isHidden = false
    skScene.myLabel.isHidden = false
    skScene.playBtn.isHidden = true
    skScene.gameOverLabel.isHidden = true

    if(!UserDefaults.standard.bool(forKey: "removeAdsKey")) {
        skScene.noAdsBtn.isHidden = true
    }

    score = 0
    skScene.myLabel.text = "Score: \(score)"

}

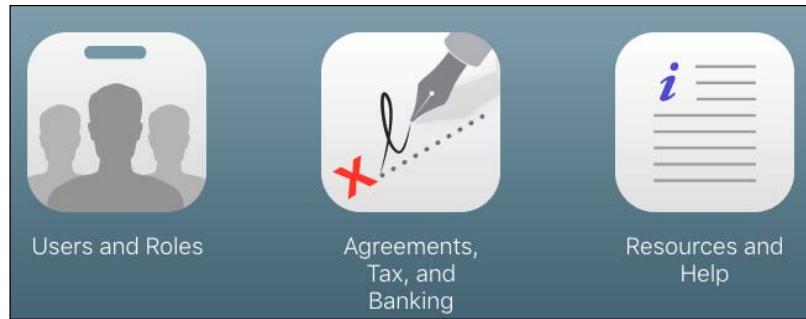
func GameOver() {

    skScene.jumpBtn.isHidden = true
    skScene.playBtn.isHidden = false
    skScene.gameOverLabel.isHidden = false

    if(!UserDefaults.standard.bool(forKey: "removeAdsKey")) {
        skScene.noAdsBtn.isHidden = false
    }

    //reset hero and enemy position
    enemy.position = SCNVector3Make(0, 2.0, 60.0)
    hero.position = SCNVector3Make(0, 5, 0)
}
```

To actually test whether the In-App purchase really works, we have to create sandbox testers. On the main page of [itunesconnect](#), go to the **Users and Roles** section:

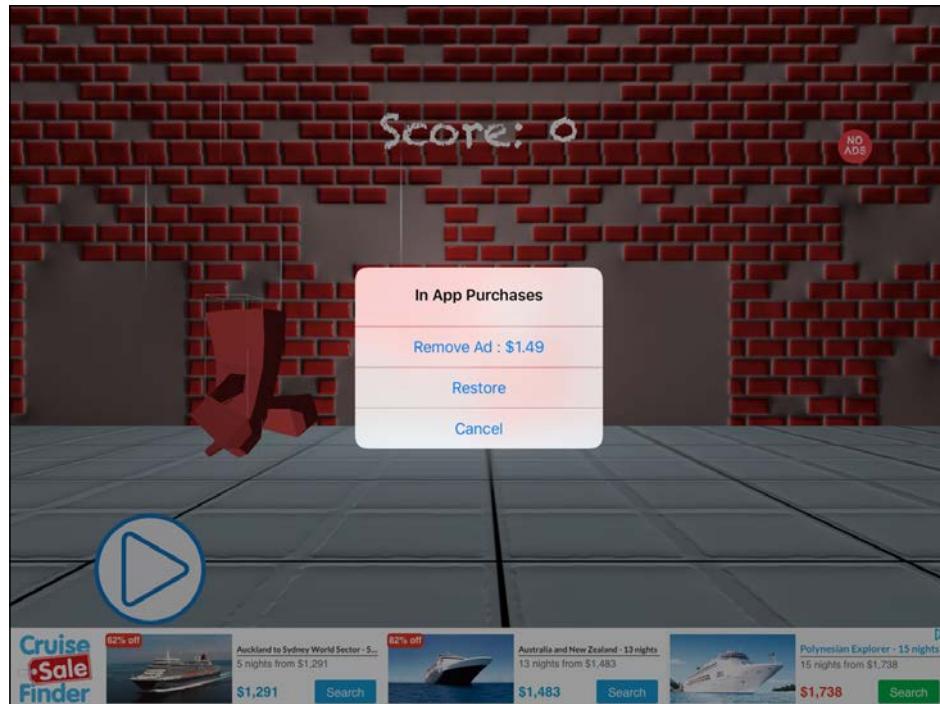


Under the **Sandbox Testers** tab, create a new tester. This shouldn't be a valid Apple ID. You can create an email ID, such as `siddharth@phony.com`, which is not real, and type in a password and secret answer. Select a date of birth and country.

You will need to sign in on the device with this username and password to test the In-App purchases.

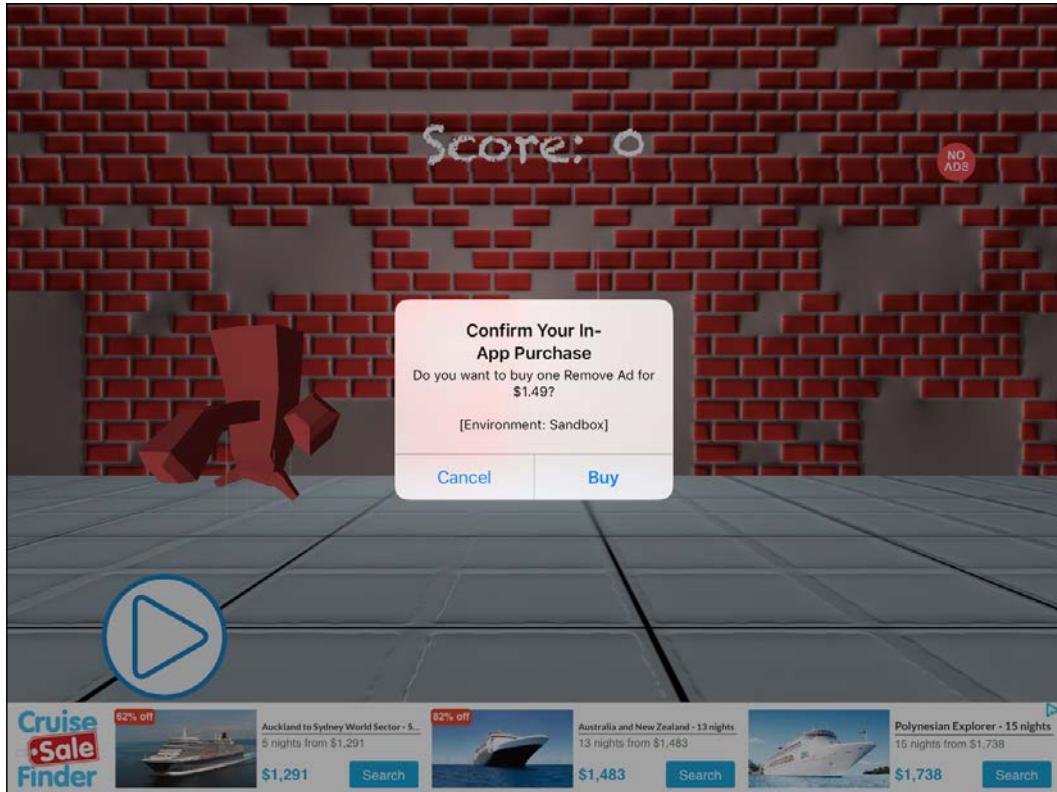
After that, log in with that email ID and password on an actual device to test the In-App purchase.

Click on the no ads button for the **In App purchase** button to pop up:

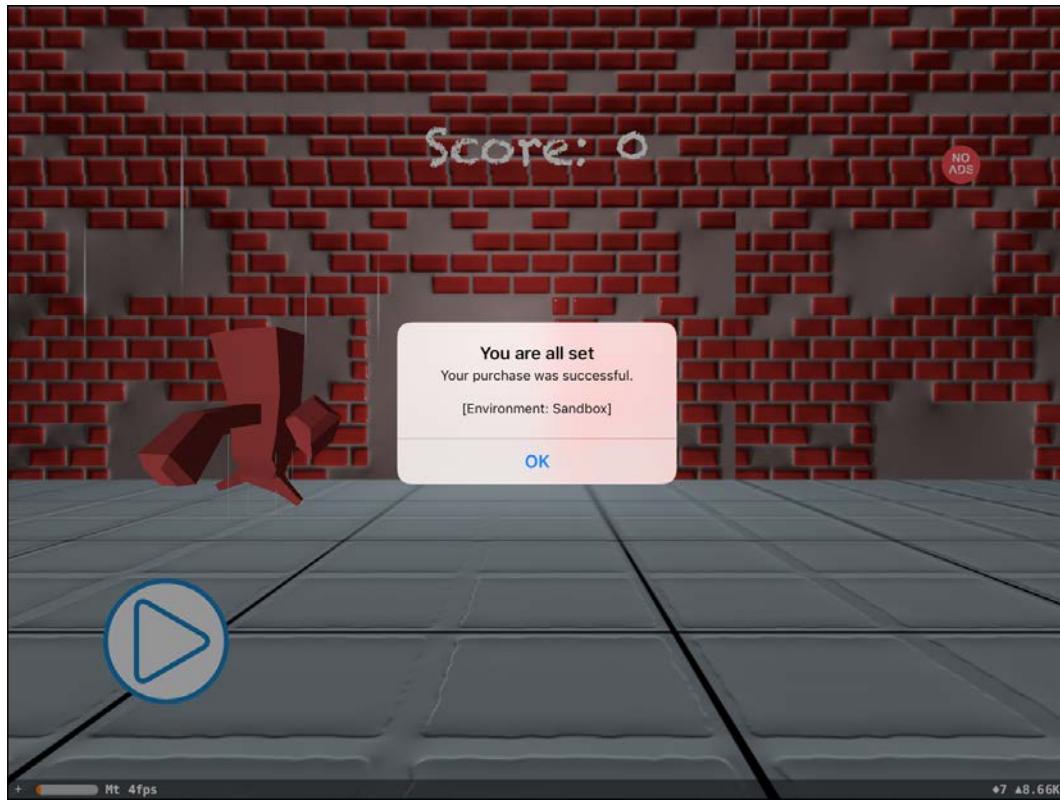


Choosing a Monetization Strategy

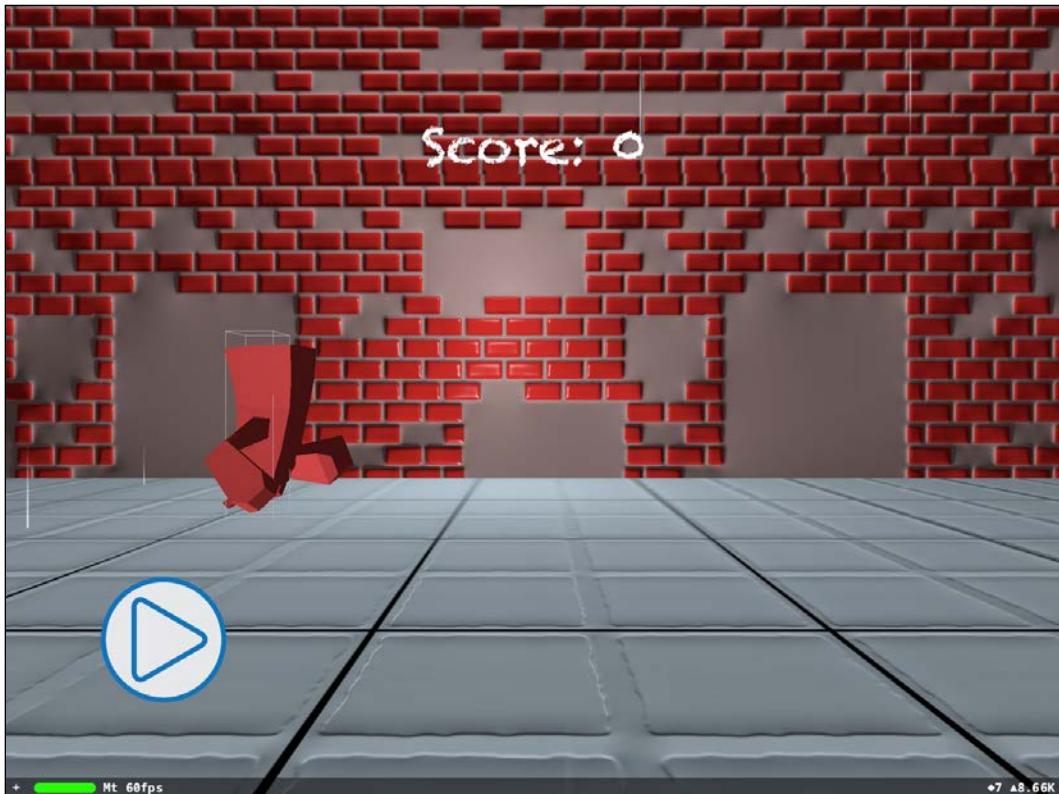
Click on the **Remove Ad** button to start a purchase:



Click on the **Buy** button to buy the product:



Once the purchase is successful, the ad is removed from the bottom, and once you press **OK**, the **NO ADS** button also gets removed from the scene:



Localization in foreign markets

While localization can be expensive, it can also open up an entirely new market for your game, which has a multiplying effect on your user base. You may want to consider translating your game into other languages, especially if your game does not use many words.

Sometimes, localization requires more than just language translation. Games in English can usually be literally translated to mainstream European languages and find success, but countries such as China or Japan can be more difficult. Besides cultural differences, these countries have strong game development industries already, and it can be tough to gain attention. Still, the market is massive, and it may be worth the effort.

Despite the large amount of effort required and the possibility of making mistakes, expanding into new markets can quickly multiply the number of people who are potentially interested in your project. It is worth a look, especially once you have proven that your game has demand in the marketplace. For indies, localization is likely most practical for simpler puzzle games and more difficult for storied games, where cultural differences will take extra work and care.

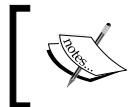
Managing scope and completing projects

Finishing and publishing projects is hard for everyone. Video games are expansive works of art and even simple games can take months to polish. The success stories in the media often portray a lone developer making millions of dollars with their first idea, but that is not often the reality. Rather than attaching to one idea, most pros build many quick prototypes and iterate on their best ideas. It is like any artistic endeavor—who ends up the better painter: the student who takes 30 days to meticulously paint one picture, or the student who paints a new picture every day for 30 days? The student who paints 30 paintings has the opportunity to learn far more.

It is for this reason that I recommend starting with simple puzzle games. Make a Snake clone, a Tetris clone, and a gem game clone. If you can finish and publish these simple games, you are probably in great shape to take on more challenging artistic pursuits. You will become a better game maker with every game you make, so do not worry about making the perfect game on your first try.

I also recommend building out a menu system that you can reuse between different projects. It is easy to finish gameplay, work on a game, and then let the project languish because working on menus and title sequences is tedious (at least for me). Make the dull work as easy as possible and you will have a much higher chance of success.

Finally, stay organized. Use pen and paper, create time charts, plan your schedule, and measure yourself against your plans. Make a list of everything you need to do to finish your game (including your marketing efforts) and assign a day and time to tackle each task. Assigning a specific amount of time for tasks is far more effective than simply dumping items onto a to-do list.



Checkpoint 12-A

The code up to this point is available in the chapter's code resources.



Summary

We have learned about basic approaches to marketing games and finding players who are interested in our projects. This can be a challenging area for indie developers. While some industry pros recommend spending as much time on marketing as you do on making your game, coding and art probably come more easily than cold calling journalists or hiring translators. However, marketing is a necessary part of successful indie game publishing. This marketing work gives you the best chance of success when you publish your game.

I hope that this chapter has given you some ideas for how to build a fan base. First, we looked at a basic checklist of a solid marketing plan. Then, we explored how to approach crowdfunding. Thirdly, we looked at using ads to generate revenue, and contrasted that with selling in-app purchases. Fourthly, we took a brief look at the power of localization and exposing your games to new markets. Finally, we reviewed some useful tips for making sure you bring your games to completion and publication. In the next chapter, we will add Game Center functionality for adding leaderboards and achievements to our game.

13

Integrating with Game Center

Apple provides an online social gaming network called Game Center. Your players can share high scores, track achievements, challenge friends, and start matchmaking for multiplayer games with Game Center. In this chapter, we will use Apple's iTunes Connect website to register our app with Apple. Then, we can integrate with Game Center to add leaderboards and achievements in our game.

You will need an active Apple developer account (which costs \$99 per year) to register your app with Apple, access the iTunes Connect website with Game Center, and publish your game to the App Store.

The topics in this chapter include the following:

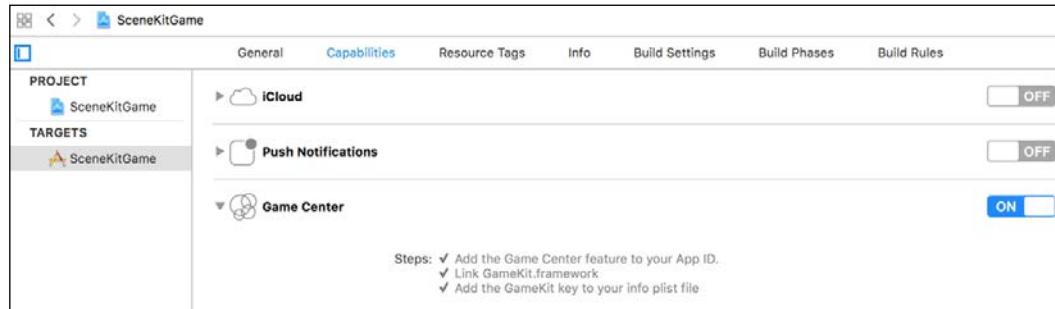
- Authenticating the player's Game Center account
- Opening Game Center in our game
- Updating the leaderboard from the code
- Adding an achievement
- Updating achievements from the code

Authenticating the player's Game Center account

As soon as our app starts, we will check whether the player is already logged in to their Game Center account. If not, we will give them a chance to log in. Later, when we want to submit high scores or achievements, we can use the authentication information we gathered when the app launched, instead of interrupting their gaming session to collect their Game Center information.

Follow these steps to authenticate the player's Game Center account when the app starts:

1. First, we will turn on **Game Center** for your project in Xcode. Open the **Capabilities** tab of your project settings and make sure **Game Center** is flipped to **ON**, as shown in the following screenshot:



2. We will now be working in the `GameViewController` class, so open `GameViewController.swift` in Xcode.
3. Add a new `import` statement at the top of the file so we can use the `GameKit` framework:

```
import GameKit
```

4. In the `GameViewController` class, add a new function called `authenticateLocalPlayer` with this code:

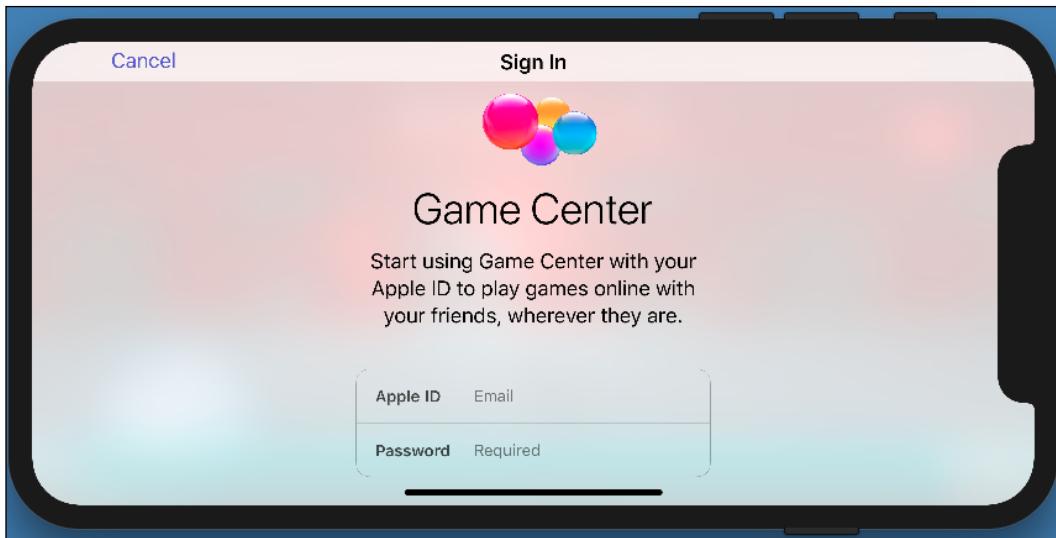
```
// We pass in the menuScene instance so we can create a
// leaderboard button in the menu when the player is
// authenticated with Game Center
func authenticateLocalPlayer(menuScene: MenuScene) {
    // Create a new Game Center localPlayer instance:
    let localPlayer = GKLocalPlayer.localPlayer()
    // Create a function to check if they authenticated
    // or show them the log in screen:
    localPlayer.authenticateHandler =
        { (viewController, error) -> Void in
            if viewController != nil {
                // They are not logged in, show the log in:
                self.present(viewController!, animated: true,
                             completion: nil)
            }
        }
```

```
        else if localPlayer.isAuthenticated {
            // They authenticated successfully!
            // We will be back later to create a
            // leaderboard button in the MenuScene
        }
        else {
            // Not able to authenticate, skip Game Center
        }
    }
}
```

5. At the bottom of the `GameViewController` class' `viewDidLoad` function, add a call to the new `authenticateLocalPlayer` function you just created:

```
authenticateLocalPlayer()
```

Run your project. You should see **Game Center** appear, asking for your credentials, as shown in the following screenshot:



Integrating with Game Center

Next, we have to create a leaderboard to make sure that the app is recognized by Game Center. Go to the Apple developer account and create a new leaderboard. First, go to the features tab and select **Game Center**:

The screenshot shows the 'Features' tab in the iTunes Connect interface for the app 'JumpJumps'. Under the 'Game Center' section, there is a 'Move to Group' section with a 'Move to Group' button. Below it is a 'Leaderboards (0)' section with a note about adding leaderboards. A 'Save' button is visible at the top right.

Click on the + sign to create a new leaderboard. Next, we can choose between a single or combined leaderboard. For this game, we will use the single leaderboard:

The screenshot shows the 'Game Center - Add Leaderboard' dialog box. It has two options: 'Single Leaderboard' (selected) and 'Combined Leaderboard'. The 'Single Leaderboard' option allows creating a new leaderboard. The 'Combined Leaderboard' option requires at least two single leaderboards with the same score format type and sort order. Both options have a 'Choose' button. At the bottom left is a 'Cancel' button.

Add a new single leaderboard as follows:

Single Leaderboard

Leaderboard Reference Name [\(?\)](#)

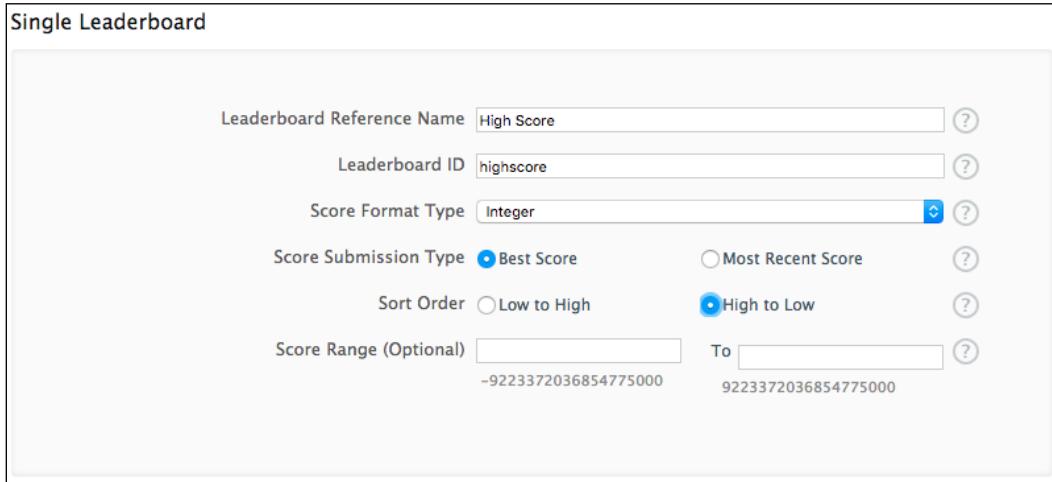
Leaderboard ID [\(?\)](#)

Score Format Type [\(?\)](#)

Score Submission Type Best Score Most Recent Score [\(?\)](#)

Sort Order Low to High High to Low [\(?\)](#)

Score Range (Optional) [\(?\)](#) To [\(?\)](#)



And in the next section, we have to specify at least one language, so add the details as follows:

Add Language

Language [\(?\)](#)

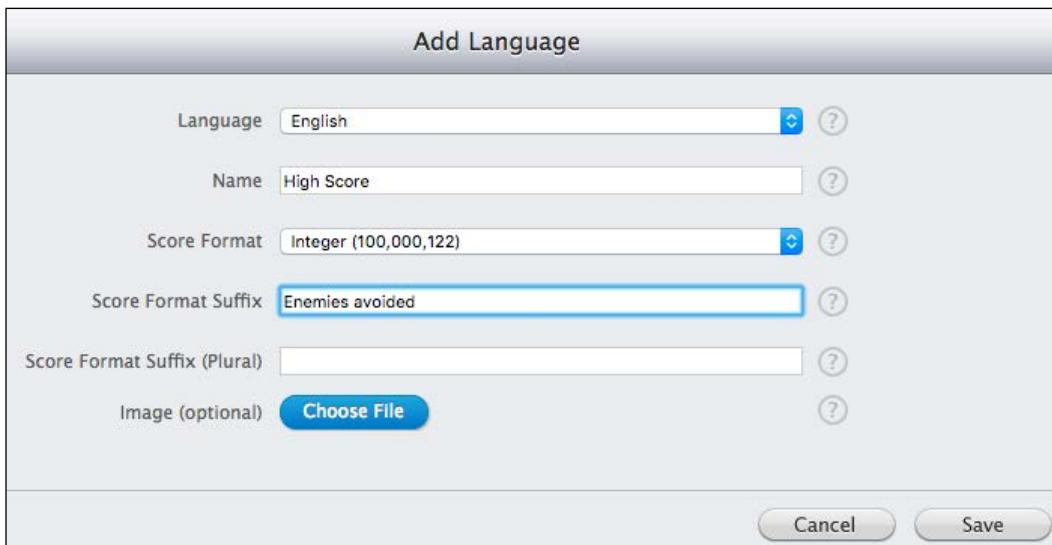
Name [\(?\)](#)

Score Format [\(?\)](#)

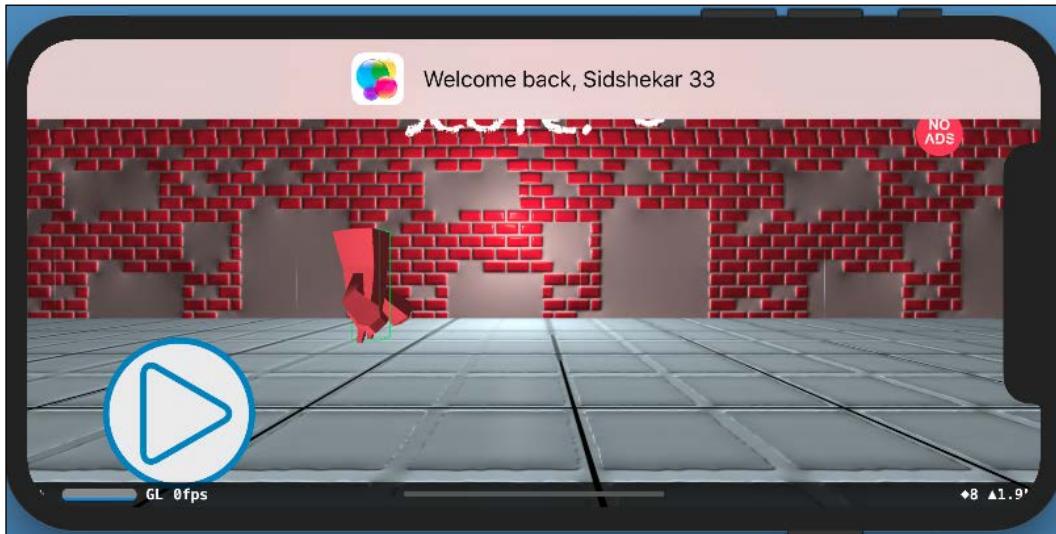
Score Format Suffix [\(?\)](#)

Score Format Suffix (Plural) [\(?\)](#)

Image (optional) [\(?\)](#)



Great! Now we can log into Game Center. Remember to use your new sandbox account. The first time you log in, Game Center will ask you to agree to the terms and conditions, and create a new Nickname. Once you finish with the Game Center form, you will see a small banner appearing at the top of the screen, letting you know you are signed in. The banner looks something like this:



If you see this Welcome back banner, you have successfully implemented the Game Center authentication code. Next, we will add a leaderboard button to the menu so that the player can see their progress within our app.

Opening Game Center in our game

If the user is authenticated, we will add a button to the `MenuScene` class so that they can open the leaderboard and view achievements from within our game. Alternatively, players can always use the Game Center app in iOS to view their progress.

Follow these steps to create a leaderboard button in the menu scene:

1. Open `OverlaySKScene.swift` in Xcode.
2. Add a new import statement at the top of the file so we can use the GameKit framework:

```
import GameKit
```

3. Update the line that declares the `OverlaySKScene` class so that our class adopts the `GKGameCenterControllerDelegate` protocol. This allows the Game Center screen to inform our scene when the player closes the Game Center:

```
class OverlaySKScene: SKScene,
SKPaymentTransactionObserver, SKProductsRequestDelegate,
GKGameCenterControllerDelegate {
```

Add two new variables at the top of the `OverlaySKScene` to add the leaderboard text and to check if the player was authenticated:

```
var leaderboardText: SKLabelNode!
var isPlayerAuthenticated = false
```

4. We need a function that will create the leaderboard button and add it to the scene. We will call this function once the Game Center authenticates the player. Add a new function to the class, named `createLeaderboardButton`, as shown in the following snippet:

```
func createLeaderboardButton() {
    // Add some text to open the leaderboard
    leaderboardText = SKLabelNode(fontNamed:
        "AvenirNext")
    leaderboardText.text = "Leaderboard"
    leaderboardText.name = "LeaderboardBtn"
    leaderboardText.position = CGPoint(x: 0, y: -100)
    leaderboardText.fontSize = 20
    self.addChild(leaderboardText)
}
```

5. Next, we will create the function that actually opens the Game Center. Add a new function named `showLeaderboard`, as shown in the following snippet:

```
func showLeaderboard() {
    // A new instance of a game center view controller:
    let gameCenter = GKGameCenterViewController()
    // Set this scene as the delegate (helps enable the
    // done button in the game center)
    gameCenter.gameCenterDelegate = self
    // Show the leaderboards when the game center opens:
    gameCenter.viewState =
        GKGameCenterViewControllerState.leaderboards
    // Find the current view controller:
    if let gameViewController =
        self.view?.window?.rootViewController {
        // Display the new Game Center view controller:
        gameViewController.show(gameCenter, sender: self)
```

```
        gameViewController.navigationController?
            .pushViewController(gameCenter, animated:
                true)
    }
}
```

6. We need to add another function to adhere to the `GKGameCenterControllerDelegate` protocol. This function is named `gameCenterViewDidFinish`, and Game Center will invoke it when the player clicks the Done button in Game Center. Add the function to the `MenuScene` class, as shown in the following snippet:

```
// This hides the game center when the user taps
'done'
func gameCenterViewControllerDidFinish
    (_ gameCenterViewController:
     GKGameCenterViewController) {
    gameCenterViewController.dismiss(animated: true,
        completion: nil)
}
```

7. We need to check for presses on our leaderboard button in the `touchesBegan` function to invoke `showLeaderboard`. Update the `touchesBegan` function's if block, as shown in the following snippet:

```
override func touchesBegan(_ touches: Set<UITouch>, with
event: UIEvent?) {

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

        let _node:SKNode = self.atPoint(location);

        if(_gameScene.gameOver == false){
            if(_node.name == "jumpBtn"){
                _gameScene.heroJump()
            }
        }else{
            if(_node.name == "playBtn"){
                _gameScene.startGame()
            } else if (_node.name == "noAdsBtn"){
                inAppPurchase()
            } else if _node.name == "LeaderboardBtn" {
```

```
        showLeaderboard()
    }

}

}
```

8. Next, make sure the button is hidden when the game is running:

```
func startGame() {

    gameOver = false
    skScene.jumpBtn.isHidden = false
    skScene.myLabel.isHidden = false
    skScene.playBtn.isHidden = true
    skScene.gameOverLabel.isHidden = true

    skScene.leaderboardText.isHidden = true

    if (!UserDefaults.standard.bool(forKey: "removeAdsKey")) {
        skScene.noAdsBtn.isHidden = true
    }

    score = 0
    skScene.myLabel.text = "Score: \(score)"

}
```

9. And show it again when the game is over:

```
func GameOver() {

    skScene.jumpBtn.isHidden = true
    skScene.playBtn.isHidden = false
    skScene.gameOverLabel.isHidden = false

    skScene.leaderboardText.isHidden = false

    if (!UserDefaults.standard.bool(forKey: "removeAdsKey")) {
        skScene.noAdsBtn.isHidden = false
    }

    //reset hero and enemy position
    enemy.position = SCNVector3Make(0, 2.0, 60.0)
    hero.position = SCNVector3Make(0, 5, 0)
}
```

10. Finally, in the GameSCNScene update function, we will check whether the player is authenticated so that we can add the create leaderboard button. So, at the top of the update function in GameSCNScene, add the following code:

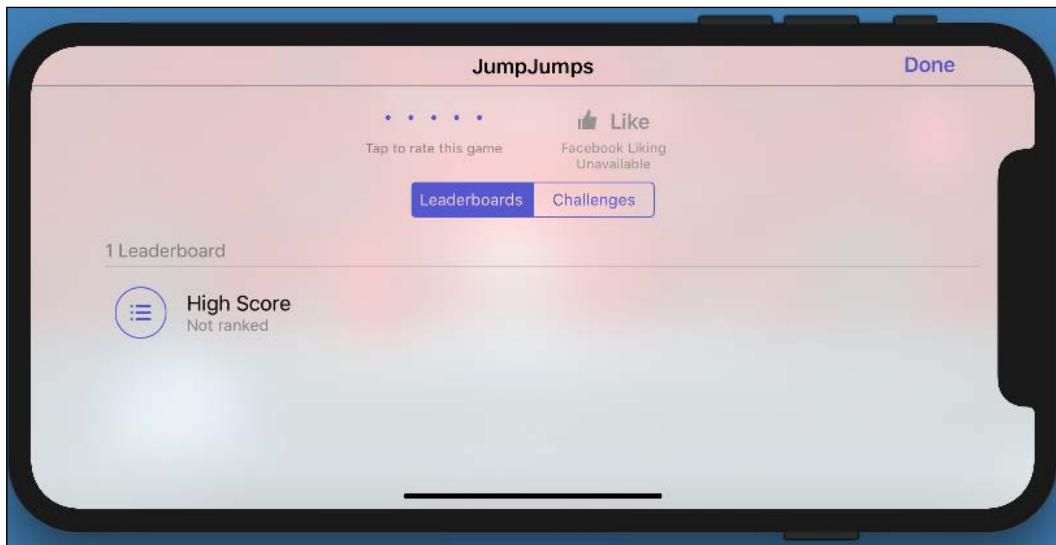
```
if (!skScene.isPlayerAuthenticated) {  
  
    // If they're logged in, create the leaderboard button  
    // (This will only apply to players returning to the  
    // menu)  
    if GKLocalPlayer.localPlayer().isAuthenticated {  
  
        print(" +++ player is authenticated +++")  
        skScene.isPlayerAuthenticated = true  
        skScene.createLeaderboardButton()  
    }  
}
```

Good work. Run the project, and you should see a leaderboard button appear in the menu after Game Center authenticates, as shown in the following screenshot:



Terrific! If you tap on the **Leaderboard** text, Game Center will open within the game. Now your players will be able to view leaderboards and achievements directly from your game. Next, we will create a leaderboard and an achievement in iTunes Connect to populate Game Center.

Click on the **Leaderboard** button to open up the leaderboard and see the **High Score** leaderboard:



Checkpoint 13-A

The code up to this point is available in the chapter's code resources.

Updating the leaderboard from the code

It is simple to send a new score to the leaderboard from the code. Follow these steps to send the number of coins collected to the leaderboard every time a game ends:

1. In Xcode, open `GameSCNScene.swift`.
2. Add an import statement at the top so we can use the GameKit framework in this file:

```
import GameKit
```

3. Add a new function in the `GameSCNScene` class named `updateLeaderboard`, as shown here:

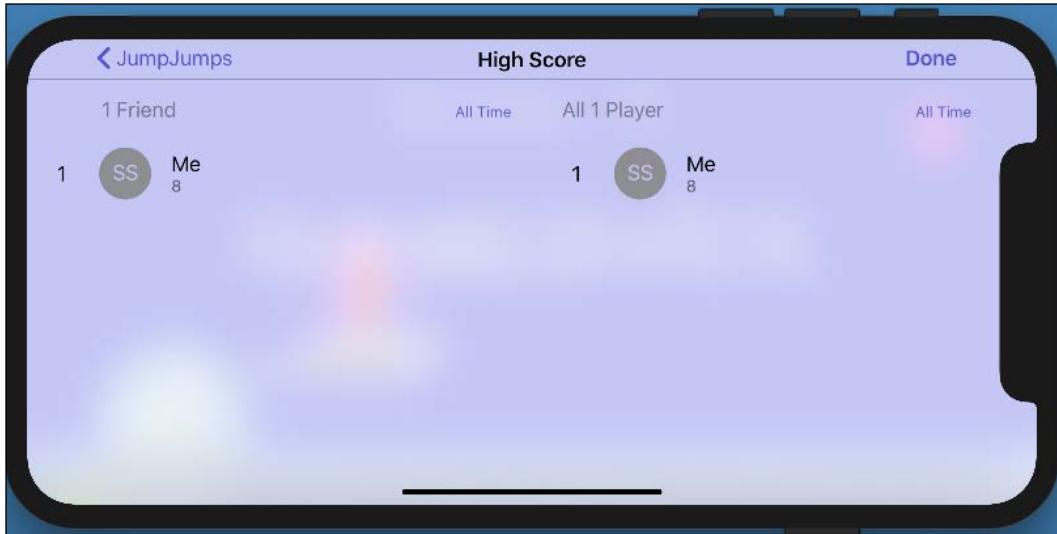
```
func updateLeaderboard() {  
    if GKLocalPlayer.localPlayer().isAuthenticated {  
        // Create a new score object, with our leaderboard:  
        let highScore = GKScore(leaderboardIdentifier:  
    "highscore")
```

```
// Set the score value to our coin score:  
highScore.value = Int64(self.score)  
  
// Report the score (wrap the score in an array)  
GKScore.report([highScore], withCompletionHandler:  
    {(error : Error?) -> Void in  
        // The error handler was used more in old  
        // versions of iOS, it would be unusual to  
        // receive an error now:  
        if error != nil {  
            print(error!)  
        }  
    })  
}
```

4. In the GameSCNScene class in the GameOver function, call the new updateLeaderboard function:

```
// Push their score to the leaderboard:  
updateLeaderboard()
```

Run the project and play through a game. Click the **Leaderboard** button to open Game Center within your game. You should see your first score appear on the leaderboard! It will look something like this:



Sometimes it can take a minute before your score shows up on the leaderboard. If you can't see your score, try restarting the simulator before troubleshooting your code.

Great work! You have implemented your first Game Center leaderboard. Next, we will follow a similar series of steps to create an achievement.

Adding an achievement

Achievements add a second layer of fun to your game and create replay value. To demonstrate a Game Center achievement, we will add a reward for getting a score of 15.

Creating a new achievement in iTunes Connect

Just like the leaderboard, we first need to create an iTunes Connect record for our achievement. Follow these steps to create a record:

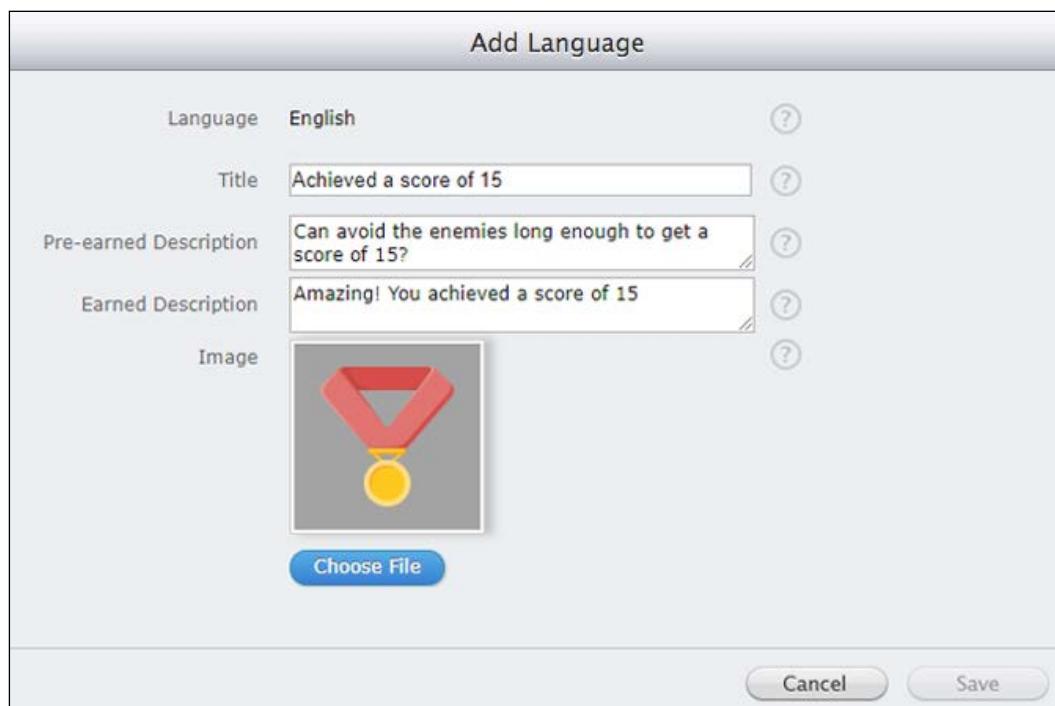
1. Log into **iTunes Connect** and navigate to the **Game Center** page for your app.
2. Underneath the leaderboards list, locate and click on the **Add Achievement** button.
3. Fill out the information for your achievement. Here are my values:

The screenshot shows the 'Achievement' creation form in iTunes Connect. The fields filled in are:

- Achievement Reference Name: Scored 15
- Achievement ID: score15
- Point Value: 10
990 of 1000 Points Remaining
- Hidden: No (radio button selected)
- Achievable More Than Once: No (radio button selected)

Let's take a look at each field:

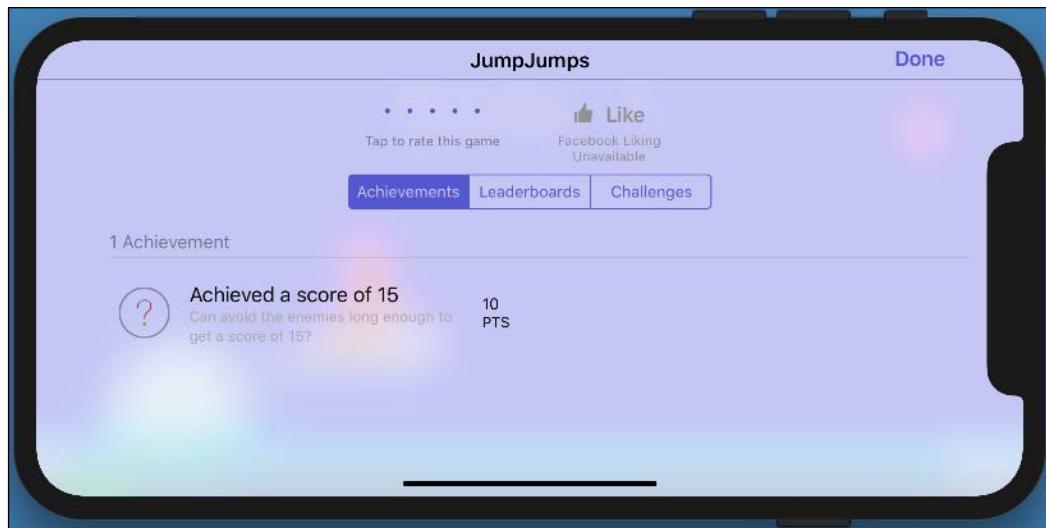
- **Achievement Reference Name** is the name iTunes Connect will use internally to refer to the achievement.
- **Achievement ID** is the unique identifier we will use to reference this achievement in our code.
- You can assign a **Point Value** to each achievement so that players can earn more achievement points as they collect new achievements.
- **Hidden** and **Achievable More Than Once** are self-explanatory, but you can use the question mark buttons on the right for additional information from Apple.
- Click the **Add Language** button. We will name the achievement and give it a description, as in the leaderboard process. Additionally, an image is required for achievements. The image dimensions can be 512 x 512 or 1024 x 1024. You can find the one I used in our Assets bundle download, in the Extras folder, `gold-medal.png`. Here are my values:



- Click **Save** twice (once for the language dialog and once on the achievement screen).

Terrific! You should be back on the main iTunes Connect Game Center page for your app, with your new achievement listed in the **Achievements** section.

When you open the leaderboard in the game, you will see an **Achievement** tab with our achievement in the list:



Next, we will integrate this achievement into the game.

Updating achievements from the code

Much like sending leaderboard updates, we can send achievement updates to Game Center from GameScene. Follow these steps to integrate our 200-coin achievement:

1. Open GameSCNScene.swift in Xcode.
2. If you skipped over the leaderboard section, you will need to add a new import statement at the top of the file so that we can use GameKit. If you have already implemented the leaderboard, you can skip this:

```
import GameKit
```

3. Add a new function to the GameScene class, named checkForAchievements, as shown in the following snippet:

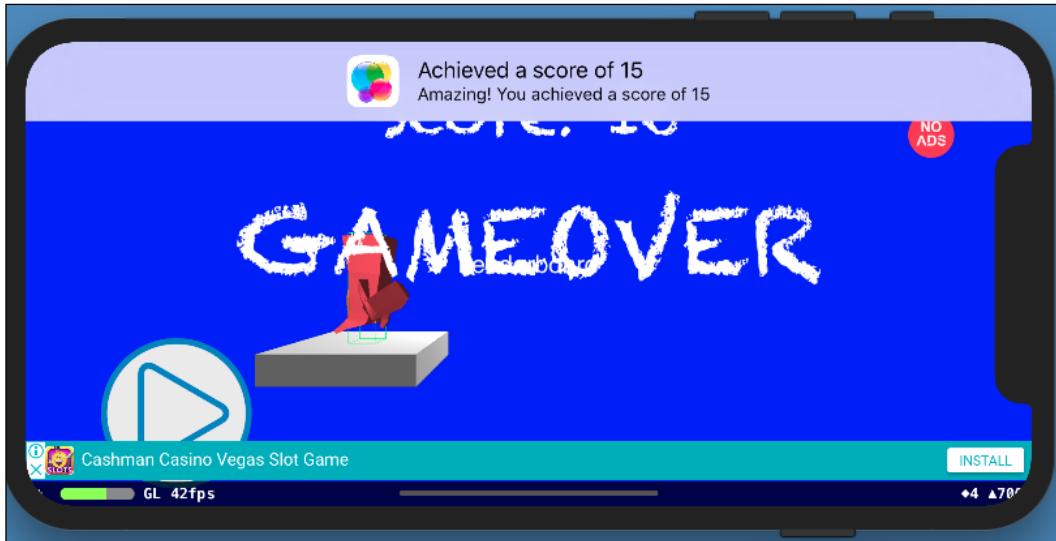
```
func checkForAchievements() {
    if GKLocalPlayer.localPlayer().isAuthenticated {
        // Check if they earned 200 coins in this game:
        if self.score >= 15 {
```

```
let achieve = GKAchievement(identifier: "score15")
// Show a notification that they earned it:
achieve.showsCompletionBanner = true
achieve.percentComplete = 100
// Report the achievement!
GKAchievement.report([achieve],
    withCompletionHandler:
    {(error : Error?) -> Void in
        if error != nil {
            print(error!)
        }
    }
)
}
}
```

4. At the bottom of the `GameOver` function, invoke the new `checkForAchievements` function:

```
// Check if they earned the achievement:
checkForAchievements()
```

Run the project, and, when your game ends, you should see a banner at the top proclaiming your new achievement, as shown in the following screenshot:



I commented out the code for running the background and particles so that the game is playable in the simulator. Uncomment the lines when you are done testing it.

Great work! You have implemented Game Center leaderboards and achievements into your game.



Checkpoint 13-B

The code up to this point is available in the chapter's code resources.



Summary

Integrating with Game Center is a great feature for your players. In this chapter, we learned how to authenticate Game Center users in our code, create new leaderboards and achievements on iTunes Connect, and then integrate those leaderboards and achievements within our game. We have made a lot of progress! In the next chapter, we will see how to make an Augmented Reality Game!

14

Introduction to Spritekit with ARKit

ARKit is a new framework developed by Apple to create 2D and 3D **augmented reality (AR)** apps for iOS. AR apps take real-world information using various sensors and place virtual objects in tandem with the real-world environment's position and lighting conditions. When viewed from the screen of the device, the virtual objects are anchored in the real world, and even if you move the device the virtual objects will scale and light up as if they are part of the real world.

In this chapter, we will see how to create a small ARKit app using SpriteKit.

The following topics are covered in the chapter:

- Requirements for the project
- Creating an AR SpriteKit project
- Adding text and crosshair
- Adding anchors at random location
- Adding custom sprites
- Registering Touch Controls to remove game objects

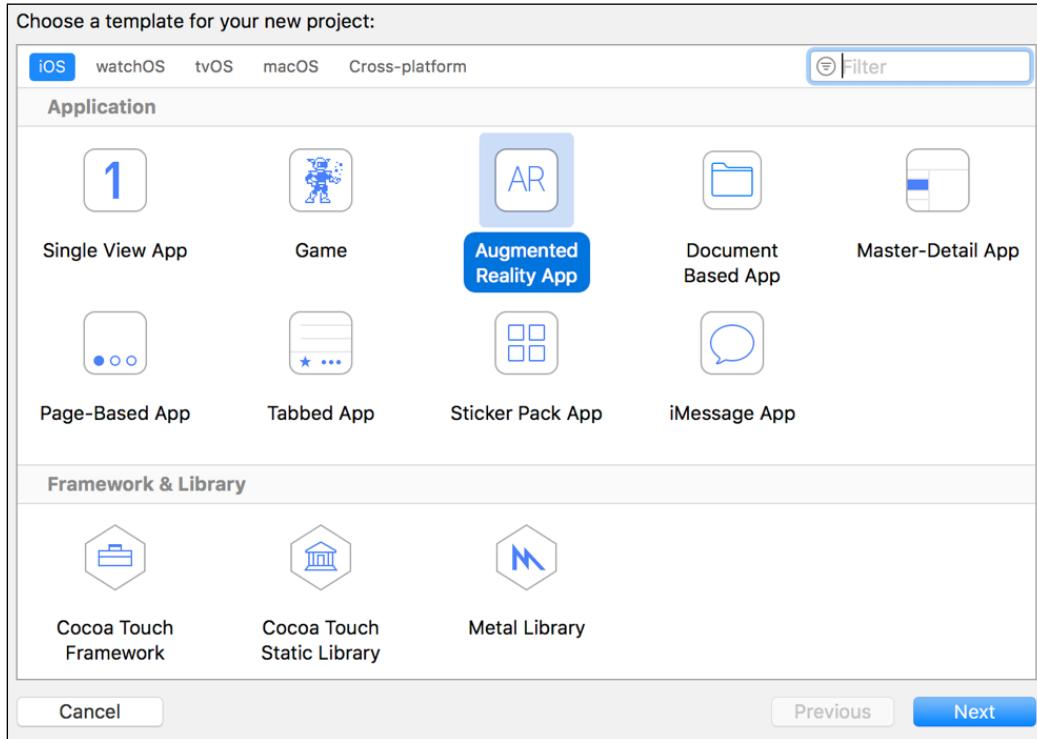
Requirements for the project

ARKit requires an iOS device with an A9 or newer processor, as you cannot run the AR project on a simulator. So, you will need at least an iPhone 6s to run the project.

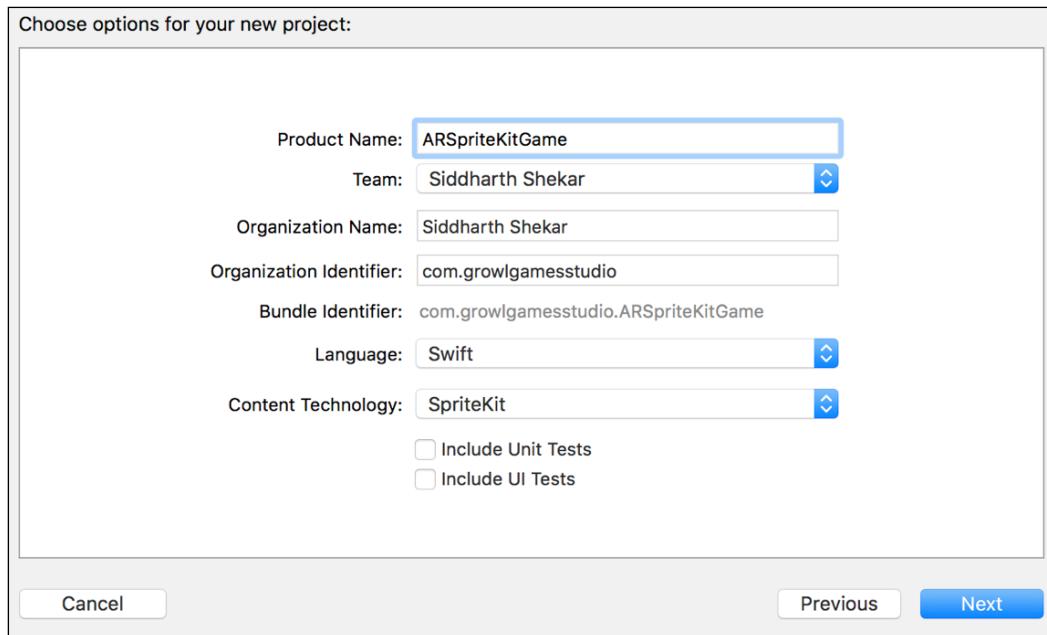
ARKit is a feature of iOS11, so you will need the latest version of Xcode 9 to develop AR applications. Xcode version 9.2 is available to all and can be downloaded from the Apple Developer website.

Creating an AR Spritekit project

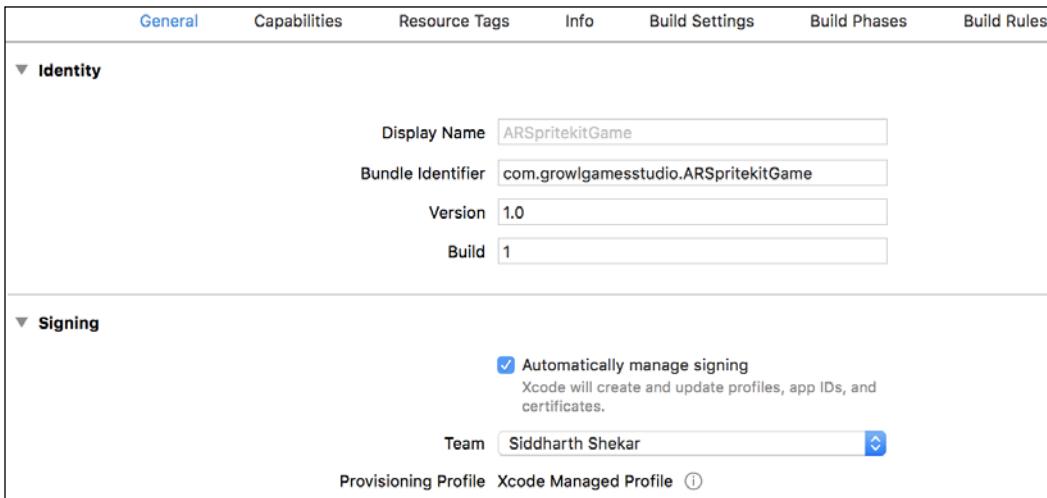
Open Xcode and create a new project. In the **Application** tab, select **Augmented Reality App**:



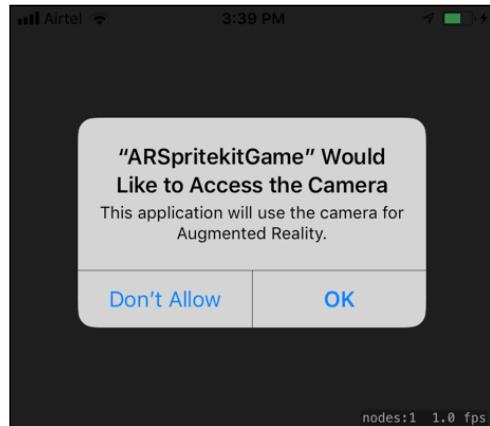
Next, set the name of the project, the development team, the language, and the content technology:



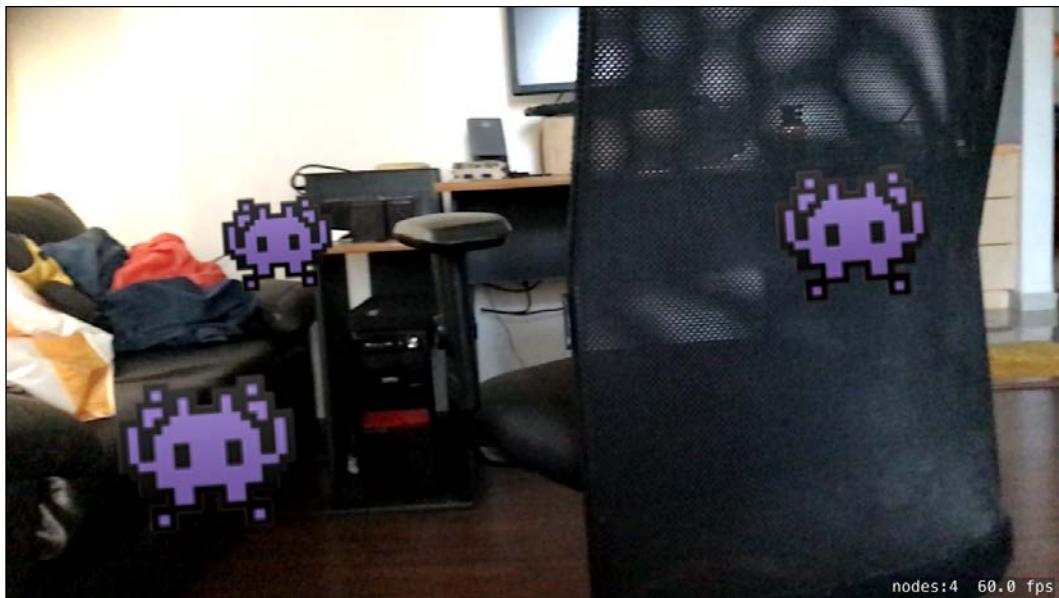
Once the project is created, sign in with your developer account to run it on the device:



When you run the application, it will need permission to access the camera.
Click **OK**:



Now, move around in the room with the camera and touch the screen. You will see objects created on the screen, and they will scale depending upon how close you are to the object:



The objects in `SpriteKit ARKit` are all 2D objects, so they do not have perspective or depth. The object is made to face the camera irrespective of the user's location and orientation. This is called bill boarding.

Let's see how all of this is set in code to make it work. In terms of class structure, you will see that it remains the same. We still have the same `Main.Storyboard`, `ViewController.swift`, `Scene.swift`, and `Scene.sks` files in the folder.

In the `Main.Storyboard` file, instead of `SKView` we now have `ARSKView`, which is the subclass of `SKView`. This fills the screen and renders the live video from the device as the background image for the scene.

The view also automatically scales and rotates the virtual objects to the scene and makes it look as if it is fixed in its location. This location is called an Anchor. It is a virtual location of the physical space in the world. This Anchor location is stored in a class called `ARAnchor`, which has the position, scale, and rotation corresponding to the physical world.

The `ViewController` class controls the view. It has the same `viewDidLoad` function, which gets called when the view loads. In this class, the `sceneView` delegate is set to `self`, show FPS, and the Node count. The `Scene.sks` file is loaded.

In addition to `viewDidLoad`, there are two new functions that are integral to this project, `viewWillAppear` and `viewNodeForAnchor`.

In `viewWillAppear`, the AR world tracking configuration needs to be created, and for the current `sceneView` session, the configuration needs to be run. So let's add it in the function:

```
override func viewWillAppear(_ animated: Bool) {  
  
    super.viewWillAppear(animated)  
  
    // Create a session configuration  
    let configuration = ARWorldTrackingConfiguration()  
  
    // Run the view's session  
    sceneView.session.run(configuration)  
}
```

There are two types of tracking configuration: `ARWorldTrackingConfiguration` and `AROrientationTrackingConfiguration`.

`ARWorldTrackingConfiguration` provides tracking information in six degrees that is three **Degrees of Freedom (DOF)** is rotation, in x , y and z axis and three degree is translation along the x , y and z axis.

`AROrientationTrackingConfiguration` provides tracking in only three degrees and is generally better used for tracking in low-end devices.

Next, let's look at the `viewNodeForAnchor` function. This function creates a node at a specific anchor and returns it as an `SKNode` object. Here, a `SKLabelNode` is created and returned because `SKLabelNode` is a child class of `SKNode`. Similarly, we can create `SKSpriteNode` or a custom node that is a child of it and add it to the scene:

```
func view(_ view: ARSKView, nodeFor anchor: ARAnchor) -> SKNode? {
    // Create and configure a node for the anchor added to the
    view's session.
    let labelNode = SKLabelNode(text: "")
    labelNode.horizontalAlignmentMode = .center
    labelNode.verticalAlignmentMode = .center
    return labelNode;
}
```

The anchor node will be created automatically when you tap the screen and the `touchesBegan` function is called in the `Scene.swift` file. Let's see how the anchor is created in the `touchesBegan` function:

```
override func touchesBegan(_ touches: Set<UITouch>, with event:
UIEvent?) {

    guard let sceneView = self.view as? ARSKView else {
        return
    }

    // Create anchor using the camera's current position
    if let currentFrame = sceneView.session.currentFrame {

        // Create a transform with a translation of 0.2 meters in front of the
        // camera
        var translation = matrix_identity_float4x4
        translation.columns.3.z = -0.2
        let transform = simd_mul(currentFrame.camera.transform,
        translation)

        // Add a new anchor to the session
        let anchor = ARAnchor(transform: transform)
        sceneView.session.add(anchor: anchor)
    }
}
```

In the `touchesBegan` function, the first thing that is done is to get the current `sceneView`. With this, we can get the `currentFrame` information. The `currentFrame` has information regarding the position and rotation of the camera in the current location. It also has other information, such as lighting conditions.

What we want to do is to place the object in front of the device's camera. So, we get the device's current location. Then we place the object right in front of the device. We create a new 4×4 matrix. We use matrices to store rotation, scale, and position information for 3D location so that all the required information is available in one place.

In the 4×4 matrix, the fourth column (in blue) represents translation in the x, y, z directions. The red 3×3 matrix is used for rotation, and the green part of the matrix is used to specify the scale.

The last location, in purple, is used to specify whether the values are of a vector or a position in 3D space:

$$\begin{bmatrix} A_{11} & A_{12} & A_{13} & A_{14} \\ A_{21} & A_{22} & A_{23} & A_{24} \\ A_{31} & A_{32} & A_{33} & A_{34} \\ A_{41} & A_{42} & A_{43} & A_{44} \end{bmatrix}$$

So, since we want to place the object in front of the camera, we have to translate the object. We focus on the fourth column of the matrix (Apple has its matrices with 0th index, so the four columns are numbered 0, 1, 2, and 3 and that's why it is column 3) and we want to change the z value of the position, so we go down three places from the top. The top row represents the x axis, the second row is the y axis, and the third row is the z axis.

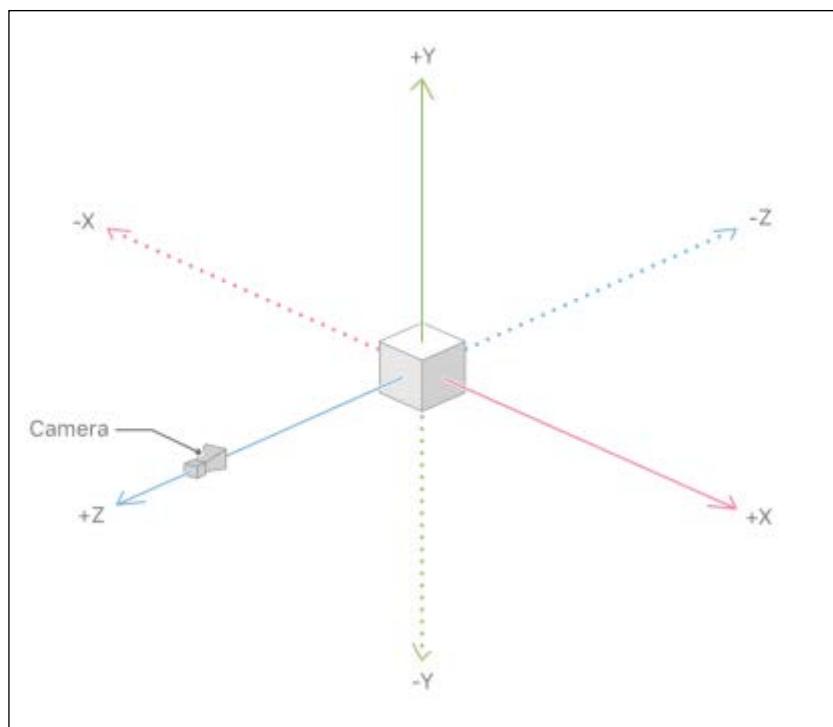
So, that's why we change the value in the fourth column's third row if we want to translate in the z axis.

The values that are specified here need to be in meters, because that is how the API corresponds virtual distances to real-world distances. Here, we are placing the object 0.2 meters from the device:

```
translation.columns.3.z = -0.2
```

You may be wondering why there is a negative sign. The following diagram is the coordinate system of the 3D world that Apple uses. When you have the screen of the device looking towards you, the positive x and y direction are to the right and up, respectively. The position z direction starts at the device and points towards you, so if you want to place an object in front of the camera you have to place the object in the negative z direction.

Refer to the following diagram. Imagine the camera is the device's camera, with the main camera facing away from you. The positive z axis is towards you, the positive x axis is to the right, and the positive y axis is upward:



To set the distance with respect to the camera's location, we multiply our translation matrix by the camera's current transformation matrix.

Next, we create a new anchor and add that anchor to the current session.

This is performed each time you press the screen and a new virtual object is created with its anchor in the real world.

Now that we have some understanding of this, let's create a basic application.

Adding text and crosshair

Let's see how to add basic game objects, such as a regular sprite and text, to the scene.

At the top of the `Scene.swift` class, add `crosshair` of type `SKSpriteNode` and `scoreText` of type `SKLabelNode` called `crosshair` and `scoreText` as follows:

```
import SpriteKit
import ARKit

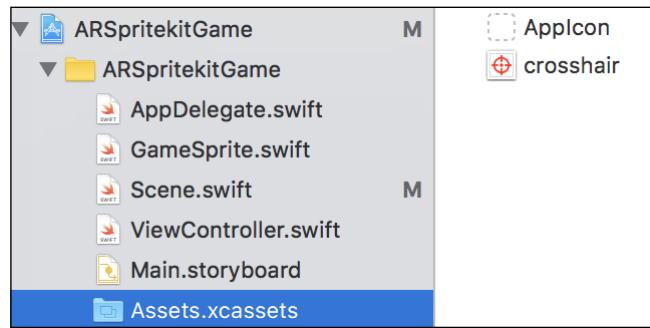
class Scene: SKScene {

    var crosshair: SKSpriteNode!
    let scoreText = SKLabelNode(text: "00")
```

Also create a new `ARSKView` called `sceneView`, which will return the current view as `ARSKView`. This is because we need to access the current view often:

```
var sceneView: ARSKView {
    return view as! ARSKView
}
```

The crosshair image is provided in the assets for the chapter. Copy and include it in the current project's assets, as follows:



In the `viewDidLoad` function, add the following lines of code to add the crosshair and text to the scene:

```
override func didMove(to view: SKView) {
    // Setup your scene here

    crosshair = SKSpriteNode(imageNamed: "crosshair")
```

```
        addChild(crosshair)

        scoreText.fontName = "AvenirNext-HeavyItalic"
        let coinTextPosition = CGPoint(x: view.bounds.width * 0.4,
                                       y: view.bounds.height * 0.4)
        scoreText.position = coinTextPosition
        scoreText.horizontalAlignmentMode =
        SKLabelHorizontalAlignmentMode.right
        scoreText.verticalAlignmentMode =
        SKLabelVerticalAlignmentMode.center
        self.addChild(scoreText)

    }

}
```

In the `viewController.swift` class, let's load the `Scene.swift` file directly. Replace the `viewDidLoad` function in the `viewController` class with the following code:

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

    if let view = self.view as? ARSKView {

        // Set the view's delegate
        sceneView.delegate = self

        // Show statistics such as fps and node count
        sceneView.showsFPS = true
        sceneView.showsNodeCount = true

        let scene = Scene(size: view.bounds.size)
        scene.scaleMode = .resizeFill
        scene.anchorPoint = CGPoint(x: 0.5, y: 0.5)
        view.presentScene(scene)

    }
}
```

Here is the output:



You can see that the assets are not actually created in the 3D world, but are like any of the regular sprites that we have created in SpriteKit so far.

Adding anchors at random locations

We want to create a new anchor node randomly every few seconds, and by using the crosshair we should be able to center the crosshair on the virtual 3D object and tap the screen to remove the virtual object from the scene. We will then track the score to see how many virtual objects are left in the scene.

At the top of the `Scene` class, create two new variables of type `TimeInterval` and `Int` called `creationTime` and `score`, and initialize both of them to 0:

```
var creationTime: TimeInterval = 0  
var score: Int = 0
```

Also add a new function called `randomFloat`, which will take in a minimum and maximum float value and generate a random float value between the maximum and minimum values:

```
func randomFloat(min: Float, max: Float) -> Float {  
    return (Float(arc4random()) / 0xFFFFFFFF) * (max - min) + min  
}
```

Create another function called `createAnchorNode`. In this function, we will create an anchor node in the scene:

```
func createAnchorNode() {  
  
    var translation = matrix_identity_float4x4  
    translation.columns.3.x = randomFloat(min: -1.0, max: 1.0)  
    translation.columns.3.y = randomFloat(min: -1.0, max: 1.0)  
    translation.columns.3.z = randomFloat(min: -1.0, max: -0.5)  
  
    // Add a new anchor to the session  
    let anchor = ARAnchor(transform: translation)  
    sceneView.session.add(anchor: anchor)  
    score+=1  
    self.scoreText.text = "\\"(score)"  
  
}
```

You will see that this code is very similar to the code we saw in the `touchesBegan` function previously. The only difference here is that the positions of the anchor points are generated relative to the world's coordinate system and not the camera's current position. The world's coordinate system is set as soon as the app is launched and is stored in the device.

Each time a new anchor is created, the `score` variable and `scoreText` are increased.

Still in the `Scene.swift` file, let's add code to randomly generate new anchor nodes in the `update` function:

```
override func update(_ currentTime: TimeInterval) {  
    // Called before each frame is rendered  
  
    if currentTime > creationTime {
```

```
        createAnchorNode()
        creationTime = currentTime + TimeInterval(randomFloat(min:
3.0, max: 6.0))
    }
}
```

The update function checks whether the current time is greater than the creation time. If so, then we call the `createAnchorNode` function and then set the creation time to a random value between two and six seconds after the current time.

To generate a proper random number, we have to call the `srand48` function, which will get the interval between the date object and 00:00:00 UTC on January 1, 2000, and generate a random number accordingly. This function needs to be called once at the start of the application. Include it at the end of the `viewDidLoad` function of the class as follows:

```
override func didMove(to view: SKView) {
    // Setup your scene here

    crosshair = SKSpriteNode(imageNamed: "crosshair")
    addChild(crosshair)

    scoreText.fontName = "AvenirNext-HeavyItalic"
    let coinTextPosition = CGPoint(x: view.bounds.width * 0.4,
                                    y: view.bounds.height * 0.4)
    scoreText.position = coinTextPosition
    scoreText.horizontalAlignmentMode =
    SKLabelHorizontalAlignmentMode.right
    scoreText.verticalAlignmentMode =
    SKLabelVerticalAlignmentMode.center
    self.addChild(scoreText)

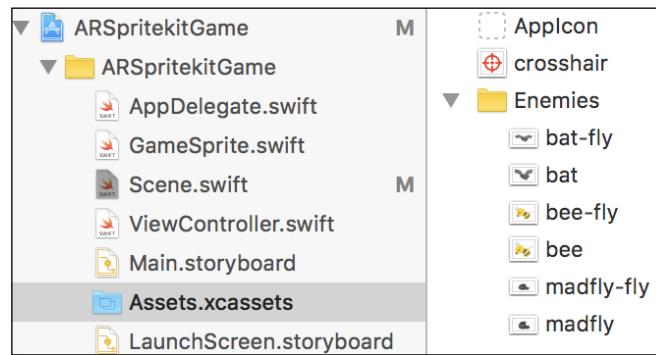
    srand48(Int(Date.timeIntervalSinceReferenceDate))
}
```

If you build and run it on the device, you will see that the critters are generated at random intervals in the scene:



Adding custom sprite

Next, let's add our custom sprite to the scene. We will use the same asset that we used to make the regular SpriteKit game. One difference is that this time we will use the bee as one of the enemies. So, add the bat-fly, bat, bee-fly, bee, madfly-fly, and madfly assets to the assets in the Enemies folder:



Also, add the `GameSprite.swift`, `Bee.swift`, `Bat.swift`, and `Madfly.swift` files and make changes to them as explained in this section.

Here is the `GameSprite.swift` file:

```
import SpriteKit

protocol GameSprite {
    var textureAtlas: SKTextureAtlas { get set }
    var initialSize: CGSize { get set }
}
```

Here is the modified `Bee.swift` file:

```
import SpriteKit

// Create the new class Bee, inheriting from SKSpriteNode
// and adopting the GameSprite protocol:
class Bee: SKSpriteNode, GameSprite {
    // We will store our size, texture atlas, and animations
    // as class wide properties.
    var initialSize: CGSize = CGSize(width: 28, height: 24)
    var textureAtlas:SKTextureAtlas =
        SKTextureAtlas(named: "Enemies")
    var flyAnimation = SKAction()

    // The init function will be called when Bee is instantiated:
    init() {
        // Call the init function on the base class (SKSpriteNode)
        // We pass nil for the texture since we will animate the
        // texture ourselves.
        super.init(texture: nil, color: UIColor.clear, size:
initialSize)

        // Create and run the flying animation:
        createAnimations()
        self.run(flyAnimation)
    }

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

```
// Our bee only implements one texture based animation.  
// But some classes may be more complicated,  
// So we break out the animation building into this function:  
func createAnimations() {  
    let flyFrames: [SKTexture] =  
        [textureAtlas.textureNamed("bee"),  
         textureAtlas.textureNamed("bee-fly")]  
    let flyAction = SKAction.animate(with: flyFrames,  
                                     timePerFrame: 0.14)  
    flyAnimation = SKAction.repeatForever(flyAction)  
}  
  
}  
}
```

Here is the `Bat.swift` file:

```
import SpriteKit  
  
class Bat: SKSpriteNode, GameSprite {  
    var initialSize = CGSize(width: 44, height: 24)  
    var textureAtlas: SKTextureAtlas = SKTextureAtlas(named: "Enemies")  
    var flyAnimation = SKAction()  
  
    init() {  
        super.init(texture: nil, color: UIColor.clear, size:  
                  initialSize)  
  
        createAnimations()  
        self.run(flyAnimation)  
    }  
  
    required init?(coder aDecoder: NSCoder) {  
        fatalError("init(coder:) has not been implemented")  
    }  
  
    func createAnimations() {  
        let flyFrames: [SKTexture] = [  
            textureAtlas.textureNamed("bat"),  
            textureAtlas.textureNamed("bat-fly")  
        ]  
        let flyAction = SKAction.animate(with: flyFrames,  
                                         timePerFrame: 0.12)
```

```
        flyAnimation = SKAction.repeatForever(flyAction)
    }

}
```

And finally, here is the `Madfly.swift` file:

```
import SpriteKit

class MadFly: SKSpriteNode, GameSprite {
    var initialSize = CGSize(width: 61, height: 29)
    var textureAtlas: SKTextureAtlas =
        SKTextureAtlas(named: "Enemies")
    var flyAnimation = SKAction()

    init() {
        super.init(texture: nil, color: UIColor.clear,
                   size: initialSize)

        createAnimations()
        self.run(flyAnimation)
    }

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

    func createAnimations() {
        let flyFrames: [SKTexture] = [
            textureAtlas.textureNamed("madfly"),
            textureAtlas.textureNamed("madfly-fly")
        ]

        let flyAction = SKAction.animate(with: flyFrames,
                                         timePerFrame: 0.14)
        flyAnimation = SKAction.repeatForever(flyAction)
    }

}
```

Next, we want these assets to be created when the anchor is created. We have to make changes to the `viewNodeForAnchor` function in the `ViewController` class. In the function, remove all the contents of the function and replace them with the following code:

```
func view(_ view: ARSKView, nodeFor anchor: ARAnchor) -> SKNode? {  
  
    var gameNode: SKSpriteNode!  
    let random = randomInt(min: 1, max: 3)  
  
    switch random {  
        case 1:  
            gameNode = Bat()  
        case 2:  
            gameNode = Bee()  
        case 3:  
            gameNode = MadFly()  
        default:  
            print(" Cannot Load enemy ")  
    }  
  
    gameNode.name = "enemy"  
  
    return gameNode  
}
```

Here, we create a `gameNode` of the type `SKSpriteNode`, as well as a `switch` statement. The `switch` statement takes a random number between 1 and 3, and then a Bat, Bee, or Mad Fly is generated. The `gameNode` is assigned the name `enemy`, and then `gameNode` is returned.

Since we are generating a random number, we create a custom function called `randomInt` to generate an integer between a minimum and maximum value. Include the function in the `viewController` file above the `viewNodeForAnchor` function:

```
func randomInt(min: Int, max: Int) -> Int {  
    return min + Int(arc4random_uniform(UInt32(max - min + 1)))  
}
```

Registering touch controls to remove game objects

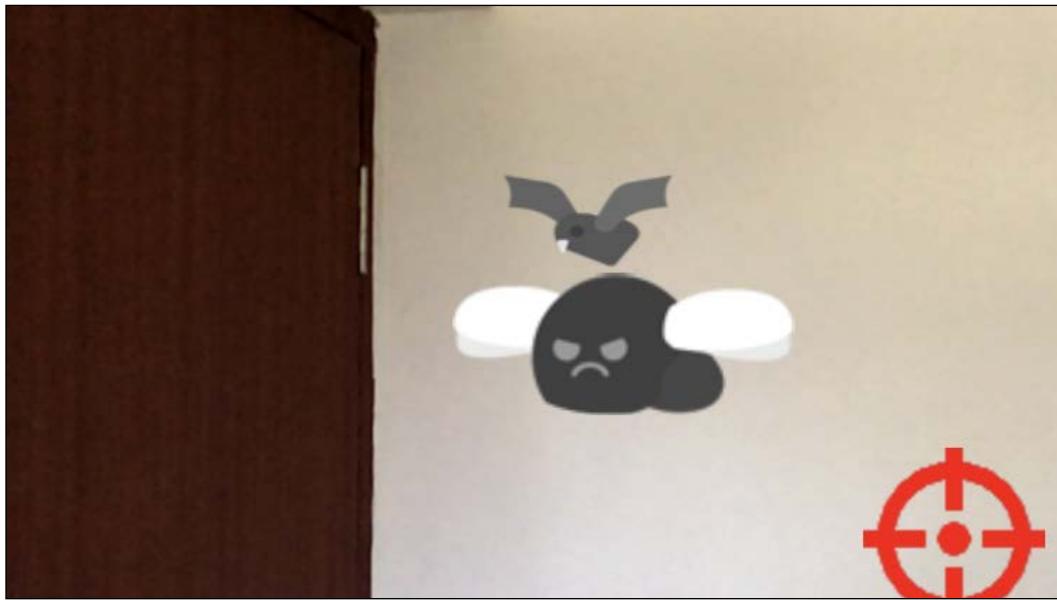
Next, let's add the functionality to remove the anchor node (that is, at the crosshair) and decrease the score.

In the `touchesBegan` function, remove all the current code and add the following:

```
override func touchesBegan(_ touches: Set<UITouch>, with event: UIEvent?) {  
  
    let location = crosshair.position  
    let hitNodes = nodes(at: location)  
  
    for node in hitNodes {  
  
        if(node.name == "enemy") {  
  
            node.removeFromParent()  
  
            score-=1  
            self.scoreText.text = "\\(score)"  
  
            break  
        }  
    }  
}
```

Here, we set the location to the crosshair to the center of the screen. We get all the nodes that are present at that location. Then, we cycle through all the nodes in `hitNodes`. If the name of the node is `enemy`, then we remove the node from the parent, decrease the score, and update the `scoreText` label.

When you build and run it, you will see the familiar animated sprites sitting in the virtual 3D world:



You can click on them to see them disappear from the scene and see the score getting updated.

Summary

We have covered the basics of ARKit and SpriteKit. You can develop the level we have created by adding scenes, sound, ads, and achievements, as the rest of the implementation remains the same as a regular SpriteKit game.

In the next chapter, we will dive into the world of ARKit with Scenekit. We will see how to import a 3D object and make it appear to be part of the real world and have fun with it.

15

Introduction to Scenekit with ARKit

In the previous chapter, we saw how to integrate ARKit into Sprite. Although it is good for placing a game object in the real world, we also quickly saw the limitations of ARKit with SpriteKit, in that all the sprites are attached to a 2D plane and that plane will always look at the player. This removes the reality and makes the app look unnatural. In the real world, you don't see planes facing wherever you go. That's spooky.

With ARkit included with SceneKit, we can add 3D objects to a scene and look around the scene, and it will look as if the object is actually present in the real world. You can get real-world lighting information and make the objects have real-life lighting effects. Also, since we have real-world lighting information, we can have shadows and make the objects look as if they are part of the real world.

The popular AR mobile game Pokemon Go uses this to stunning effect:



Image courtesy of Niantic Labs

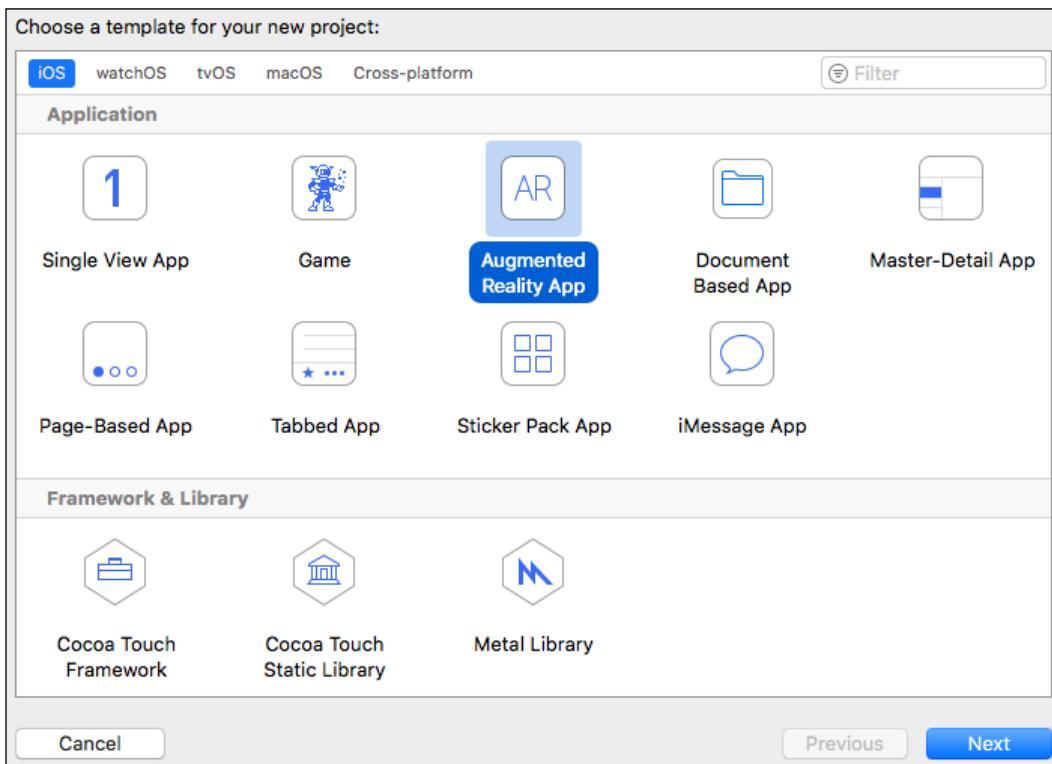
This makes a far more compelling framework than working with SpriteKit and ARKit. So, let's see how we can create the same 3D game we created in *Chapter 11, Introduction to SceneKit*, and convert it into an AR experience. The following topics will be covered in this chapter:

- Going through the basic SceneKit/ARKit project
- Project setup and detecting a plane
- Adding touches

- Adding game objects
- Adding Score and Gameover text
- Finishing touches

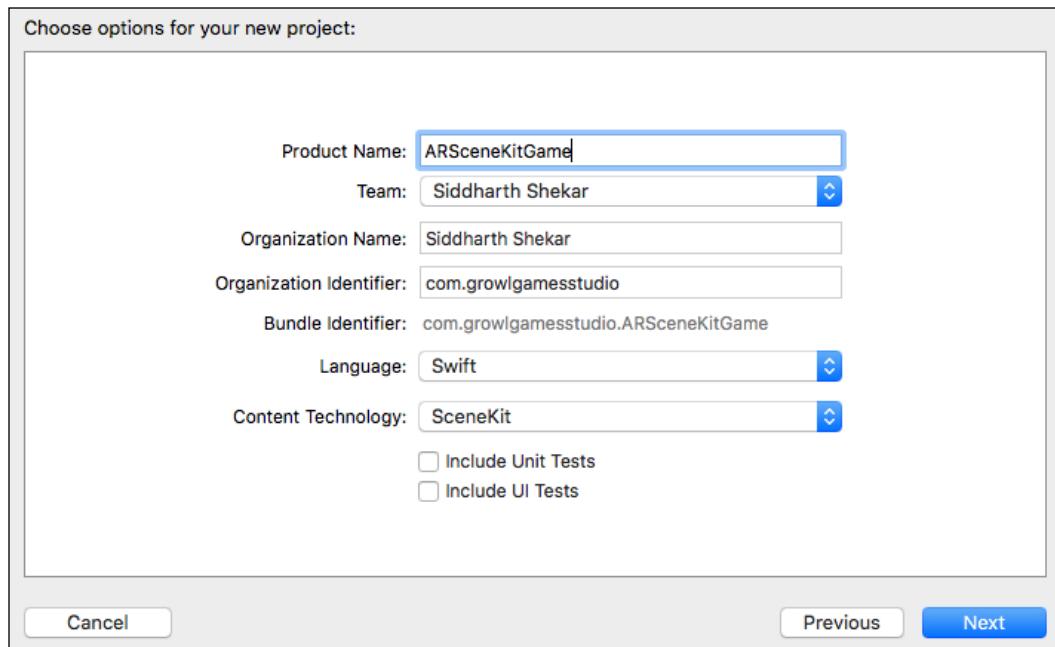
Going through the basic SceneKit/ARKit project

Let's create the SceneKit/ ARKit project. Open up Xcode and click **Create New Project**. As before, select the **Augemented Reality App** template:



Click the **Next** button.

Give it a suitable name and enter the **Team**, **Organization Name**, and **Identifier**. Set the language as **Swift**. Most importantly, select **SceneKit** as the **Content Technology**:



Click **Next**, and select where you want to place the project. Once the project has been created, open it up. Let's look at the `ViewController.swift` file and see what's different here:

```
import UIKit
import SceneKit
import ARKit

class ViewController: UIViewController, ARSCNViewDelegate {

    @IBOutlet var sceneView: ARSCNView!

    override func viewDidLoad() {
        super.viewDidLoad()

        // Set the view's delegate
        sceneView.delegate = self

        // Show statistics such as fps and timing information
        sceneView.showsStatistics = true
```

```
// Create a new scene
let scene = SCNScene(named: "art.scnassets/ship.scn")!

// Set the scene to the view
sceneView.scene = scene
}

override func viewWillAppear(_ animated: Bool) {
    super.viewWillAppear(animated)

    // Create a session configuration
    let configuration = ARWorldTrackingConfiguration()

    // Run the view's session
    sceneView.session.run(configuration)
}

override func viewWillDisappear(_ animated: Bool) {
    super.viewWillDisappear(animated)
    // Pause the view's session
    sceneView.session.pause()
}

override func didReceiveMemoryWarning() {
    super.didReceiveMemoryWarning()
    // Release any cached data, images, etc that aren't in use.
}
// MARK: - ARSCNViewDelegate

/*
    // Override to create and configure nodes for anchors added to the
view's session.
    func renderer(_ renderer: SCNSceneRenderer, nodeFor anchor:
ARAnchor) -> SCNNNode? {
        let node = SCNNNode()
        return node
    }
*/
func session(_ session: ARSession, didFailWithError error: Error)
{
    // Present an error message to the user
}

func sessionWasInterrupted(_ session: ARSession) {
    // Inform the user that the session has been interrupted, for
example, by presenting an overlay
```

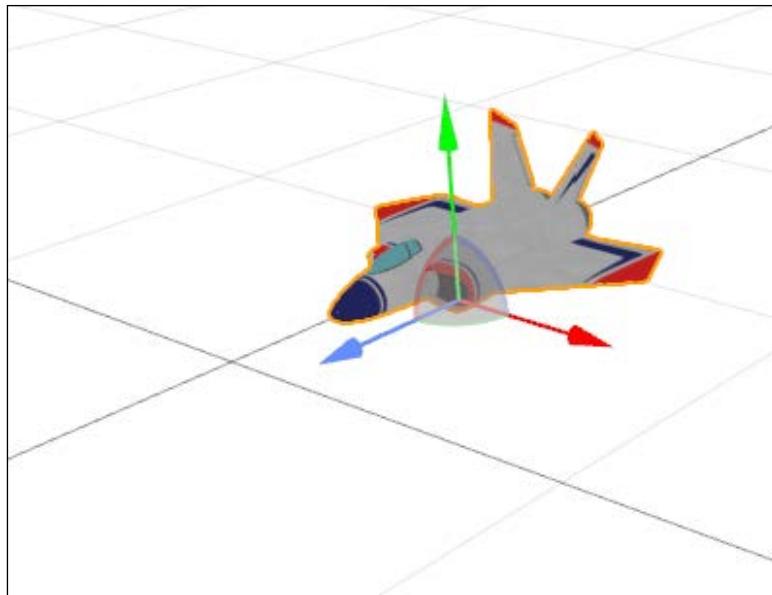
```
    }

    func sessionInterruptionEnded(_ session: ARSession) {
        // Reset tracking and/or remove existing anchors if consistent
        tracking is required
    }
}
```

Here, you can see that the project imports SceneKit, UIKit, and ARKit, as before, but the class inherits UIViewController and implements or conforms to the ARSCNViewDelegate protocol. ARSCNViewDelegate is inherited from SCNViewDelegate, so it has certain properties that are specific to AR.

You also have the regular viewDidLoad function, which sets the sceneView delegate to self, enables statistics to see how many nodes are added to the current scene, loads the ship.scn, and assigns it to the current scene.

Let's analyze the 3D model:



If you look at the ship.scn file, shown in the preceding diagram, you will notice that the center of the ship is slightly behind the origin. This is to make sure that, when the scene loads, the ship will be visible, because the camera will be placed at the origin. If the ship is also placed at the origin, you won't see anything.

Inside `viewWillAppear`, you will see a variable called `ARWorldTrackingConfiguration`. Just as in the SpriteKit/ARKit project, this sets the configuration for tracking the world information. Then the AR session is run with the configuration.

In `viewWillDisappear`, the AR session is paused, and in `didReceiveMemoryWarning`, if there are any memory warnings then the program will automatically try to minimize memory usage by releasing cached data.

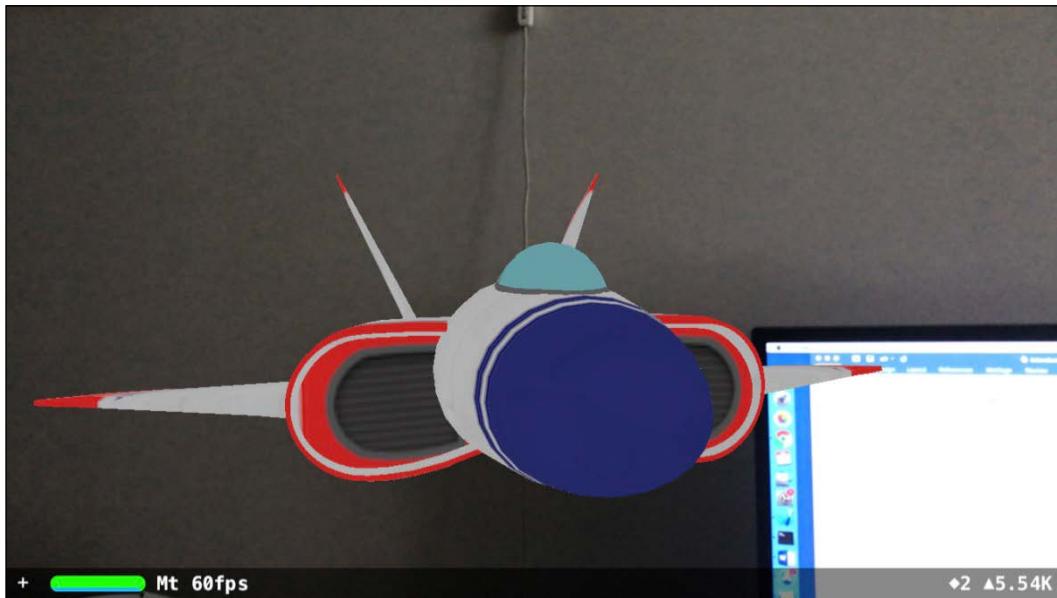
Then, similar to the `nodeForAnchor` function in the SpriteKit project, we have a similar function here that can be used to add objects to the anchors in the scene, but here, the objects that are returned are of type `SCNNode` instead of `SKNode`.

Then there are three session-related functions. The `session(_ session: ARSession, didFailWithError error: Error)` function tells the user if the session failed with an error.

The `sessionWasInterrupted(_ session: ARSession)` function tells the user if the session was interrupted.

The `sessionInterruptionEnded(_ session: ARSession)` functions tells the user if the session interruption ended.

Let's now run the project. You need an actual device to run the project, so I will run the project on my trusty iPhone 7:



Here is the ship in the virtual world. This is nice, but not very exciting. Let's start making our game now.

Project setup and detecting a plane

The first thing that we have to do, before we even set up the project, is change some settings and detect a plane so that our game object will have a plane to start on. In the 3D game in *Chapter 11, Introduction to SceneKit*, we created the ground on which the player character was placed and can jump from. When we make an AR game, we first need to detect a physical real-world plane on which we want our game objects to be placed.

ARKit for SceneKit provides a specific function that does just that. In fact, there are three functions. The first one is `didAddAnchor`, then we have `didUpdateAnchor`, and finally we have `didDeleteAnchor`:

```
func renderer(_ renderer: SCNSceneRenderer, didAdd node: SCNNNode,  
for anchor: ARAnchor) {  
  
}  
func renderer(_ renderer: SCNSceneRenderer, didUpdate node:  
SCNNNode, for anchor: ARAnchor) {  
  
}  
func renderer(_ renderer: SCNSceneRenderer, didDelete node:  
SCNNNode, for anchor: ARAnchor) {  
  
}
```

`DidAddAnchor` checks whether a new anchor/plane was detected in the scene. If it was, then we can get the dimensions of the plane. Next, there is `didUpdateAnchor`. Here, the system checks whether the current anchor needs to be updated and updates the details of the anchor.

For example, we will detect whether there is a table on which we can place the objects. We start scanning from the left of the table so that the system can detect the edge of the table . The system will see that there is a flat surface, but it will only see a part of the table. So it will create a flat surface from the left and create it till which it can detect the right edge of the table. As you pan the camera to the right to see the full table, the system will see that the plane is actually much bigger and updates the value of the current plane to fit the whole table.

`didDeleteAnchor` is called when an anchor is deleted. Suppose, instead of slowly panning to the right, you just jump to the right. The system sees that there is a plane but it doesn't know that it is part of the same plane, so it will create a separate plane at the same height. Now you scan the center of the table. The system realizes that the plane it created on the left and right are the same, and that there is also a plane in the middle. The system will update the plane on the left, extend it to the middle, extend it to the right of the table, and then delete the second plane because it is all the same plane.

That's cool, but we need a visual debugging tool to see what the system is actually thinking. So we will create a `Plane` class, which will visually represent what is being done by the system.

So, as before, create a new Swift file called `plane.swift` and add the following code to it:

```
import Foundation
import SceneKit
import ARKit

class Plane: SCNNode{

    var planeGeometry: SCNPlane!

    init(_ anchor: ARPlaneAnchor){

        super.init()

        let material = SCNMaterial()
        material.diffuse.contents = UIImage(named: "plane.png")

        self.planeGeometry = SCNPlane(width: CGFloat(anchor.extent.x),
                                      height: CGFloat(anchor.extent.z))
        planeGeometry.materials = [material]

        let planeNode = SCNNode(geometry: self.planeGeometry)
        planeNode.position = SCNVector3Make(anchor.center.x, 0,
                                             anchor.center.z)
        planeNode.transform = SCNMatrix4MakeRotation(Float(-'Double.pi' / 2.0), 1.0, 0.0, 0.0)

        setTextureScale()

        self.addChildNode(planeNode)
```

```
        print("plane added")

    }

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

func update(_ anchor: ARPlaneAnchor) {

    self.planeGeometry.width = CGFloat(anchor.extent.x)
    self.planeGeometry.height = CGFloat(anchor.extent.z)
    self.position = SCNVector3Make(anchor.center.x, 0, anchor.
center.z);
    setTextureScale()

}

func setTextureScale() {

    let width = self.planeGeometry.width
    let height = self.planeGeometry.height

    let material = self.planeGeometry.materials.first
    material?.diffuse.contentsTransform = SCNMatrix4MakeScale(Float(width), Float(height), 1);
    material?.diffuse.wrapS = SCNWrapMode.repeat
    material?.diffuse.wrapT = SCNWrapMode.repeat

}

}
```

We import `SceneKit` and `ARKit` to the class. We make the current class inherit from `SCNNode`. We also create a new `SCNPlane` called `planeGeometry`.

In the `init` function, we get the newly created anchor information as `ARPlaneAnchor`, because we want to get the plane information. We create a new material and assign a `plane.png` file to it so that when the plane is created, we can see it.

We set the size of the plane using the `anchor.extends` property and set it to the width and height of `planeGeometry`. We also assign the material to `planeGeometry`.

We then create a `planeNode` and set it at the center of the anchor's location. The plane also needs to be rotated so that it sits flat on the ground. Next, we have a `scale` function, which scales the texture to the size of the `planeGeometry`, which is handled by the `setTextureScale` function.

There is also an `update` function, which we will call in the `didAnchorUpdate` so that when the anchor's width and height values change, the width and height information of `planeGeometry` is also updated.

Go back to the `ViewController` class so that we can make some changes to the file.

Change the `viewDidLoad` function as follows:

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

    // Set the view's delegate
    sceneView.delegate = self

    // Show statistics such as fps and timing information
    sceneView.showsStatistics = true

    // Create a new scene
    //let scene = SCNScene(named: "art.scnassets/ship.scn")!

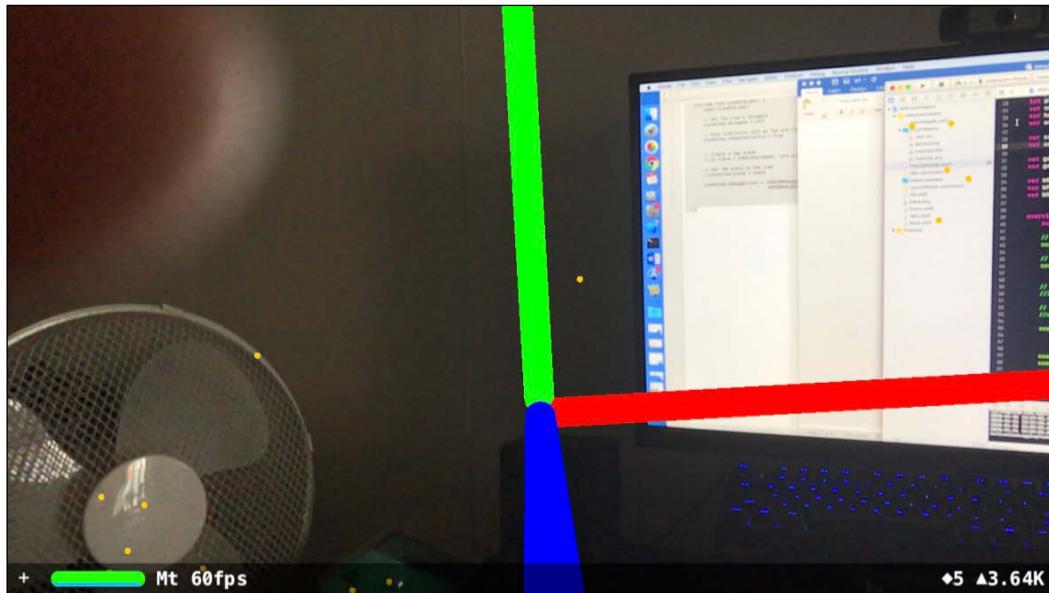
    // Set the scene to the view
    //sceneView.scene = scene

    sceneView.debugOptions = [ARSCNDebugOptions.showWorldOrigin,
                            ARSCNDebugOptions.showFeaturePoints]
}
```

Comment out the lines so that the ship doesn't appear, and add `sceneView.debugOptions`. The debug option will show the world origin, which will show the world's origin location and also the directions of the *x*, *y*, and *z* axes.

It will also show feature points, which are small yellow dots that the system uses to track the scene when you move the camera.

As shown in the following diagram, the X axis is red in color. The green axis is Y and the blue axis is Z. Where all three meet is the world origin:



Here, we see the origin and the three axes. The x axis is red, the z axis is blue, and the y axis is green. The small dots show the tracking information.

Let's add some code to include plane detection.

In the `viewWillAppear` function, add the highlighted code to enable plane detection:

```
override func viewWillAppear(_ animated: Bool) {
    super.viewWillAppear(animated)

    // Create a session configuration
    let configuration = ARWorldTrackingConfiguration()
    configuration.planeDetection = .horizontal

    // Run the view's session
    sceneView.session.run(configuration)

}
```

So far, only horizontal plane detection is available, so we shall enable it.

Also create three variables. Add a plane to create the plane. Create a `NSMutable` array to store all the planes that are created in the scene, and finally, create a global configuration so that we can access it when needed:

```
var plane: Plane!
let planes: NSMutableDictionary! = [:]
var configuration: ARWorldTrackingConfiguration! = nil
var bPlaneAdded = false
```

Create the three functions to detect, update, and remove planes as follows:

```
func renderer(_ renderer: SCNSceneRenderer, didAdd node: SCNNNode,
for anchor: ARAnchor) {

    if(!anchor .isKind(of: ARPlaneAnchor.self)){
        return
    }

    plane = Plane(anchor as! ARPlaneAnchor)
    planes.setObject(plane, forKey: anchor.identifier as
NSCopying)
    node.addChildNode(plane)

    bPlaneAdded = true
}

func renderer(_ renderer: SCNSceneRenderer, didUpdate node:
SCNNNode, for anchor: ARAnchor) {

    plane = planes.object(forKey: anchor.identifier) as! Plane

    if(plane == nil){ return }

    plane.update(anchor as! ARPlaneAnchor)
}

func renderer(_ renderer: SCNSceneRenderer, didRemove node:
SCNNNode, for anchor: ARAnchor) {

    planes.removeObject(forKey: anchor.identifier )
}
```

In the `didAddAnchor`, we first check whether the detected anchor is of the plane type. If not, then we don't do anything. Otherwise, we create a new plane and pass the anchor as the `ARPlaneAnchor` type. We add it to the planes dictionary with the plane and Key as the `anchor.identification` to identify the current plane anchor. Finally, we add the plane to the current node graph and set the `bPlaneAdded` Boolean to `true`.

In the `didUpdateAnchor`, we check whether the current anchor is in the planes dictionary. If so, then we update the current anchor's width and height.

In the `didRemoveAnchor` function, if an anchor was deleted then we remove the anchor from the planes dictionary.

Run the project, and you will see the plane being detected.

In order for the plane to be detected, make sure you use a rough surface and that the room is well lit. Only then will the plane be easy to generate.



Now that we have our plane, we can add objects to the scene.

Adding touches

Before we can add objects to the scene, we have to include touches so that an object will be placed where we touch on the scene. Basically, wherever we touch on the scene, a ray is cast onto the plane created in the previous step. Wherever the ray touches the plane, we get an intersection. Using this point as reference, we can place objects in the scene.

We will add some Booleans at the top of the `viewConctrller` class:

```
var bGameSetup = false  
var bGameOver = false
```

`bGameSetup` will check whether the game is ready to start. `bGameOver` is a flag we set when the game is over.

To add touches to the scene, add the following function to the `ViewController` class:

```
override func touchesBegan(_ touches: Set<UITouch>, with event:  
UIEvent?) {  
  
    let location = touches.first!.location(in: sceneView)  
  
    var hitTestOptions = [SCNHitTestOption: Any]()  
    hitTestOptions[SCNHitTestOption.boundingBoxOnly] = true  
  
    let hitTestArray = sceneView.hitTest(location,  
types: [.estimatedHorizontalPlane, .existingPlane,  
.existingPlaneUsingExtent])  
  
    for result in hitTestArray {  
  
        if (!bGameSetup) {  
  
            let spawnPos = SCNVector3(x: result.worldTransform.  
columns.3.x,  
                           y: result.worldTransform.  
columns.3.y,  
                           z: result.worldTransform.  
columns.3.z)  
            if (bPlaneAdded) {  
  
                setupGame(spawnPos)  
            }  
        }  
    }  
}
```

```
    plane.isHidden = true

    bGameSetup = true
}
}

}

}
```

In the `TouchesBegan` function, we first get the location of the touches on the screen.

We then set the `HitTestOptions`. Here, we set the option that checks for collisions with bounding boxes to `true`.

Next, we get the `hitTests` array. The `hitTest` array will cast rays to the surface check if anything is hit in this case the horizontal plane

Next, we loop through all the `hitTests` in the scene. We create a new global Boolean and initialize it to `false`. If the `bGameSetup` Boolean is `false`, we get the location of the intersection of the ray with the plane in the world. If the plane is added, we start setting up the game by calling the `setupGame` function. We also hide the plane we created and set the `bGameSetup` Boolean to `true`.

Now we can see how to add the game objects to the scene.

Adding Game Objects

Create a new `setupGame` function. We start adding game objects to the scene. To do this, add the following to the `setupGame` function.

Stop detecting planes

The first thing we do in the `setupGame` function is disable the checking of planes in the scene. Add the following code to change the configuration and run the new configuration:

```
configuration.planeDetection = ARWorldTrackingConfiguration.
PlaneDetection(rawValue: 0)
sceneView.session.run(configuration)
```

Adding light source

Then we add a light source to the scene:

```
let directionLight = SCNLight()
directionLight.type = SCNLight.LightType.directional
```

```
directionLight.castsShadow = true
directionLight.shadowRadius = 200
directionLight.shadowColor = UIColor(red: 0,
                                      green: 0,
                                      blue: 0,
                                      alpha: 0.3)

directionLight.shadowMode = .deferred

let directionLightNode = SCNNNode()
directionLightNode.light = directionLight
directionLightNode.transform = SCNMatrix4MakeRotation(Float(-
    Double.pi / 2.0), 1.0, 0.0, 0.0)
directionLightNode.position = SCNVector3(spawnPos.x + 0.2,
                                         spawnPos.y + 0.5,
                                         spawnPos.z + 0.0)

sceneView.scene.rootNode.addChildNode(directionLightNode)

let ambientLightNode = SCNNNode()
ambientLightNode.light = SCNLight()
ambientLightNode.light!.type = SCNLight.LightType.ambient
ambientLightNode.light!.color = UIColor.darkGray
sceneView.scene.rootNode.addChildNode(ambientLightNode)
```

We have added a directional light to the scene. We created an `SCNLight` called `directionLight`. The type is set to `directional`. We set the `castsShadow` property to `true`, set the `shadowRadius` to `200`, and set the `shadowColor` to an alpha value of `0.3`. Finally, we set the `shadowMode` to `deferred`.

Then, we created a node called `directionLightNode` of type `SCNNNode` and set the `light` to `direteinLight`. We rotated the light so that it looks down and positioned the camera in the `x`, `y`, and `z` direction. Finally, we added the light source to the scene.

While positioning anything in the scene, keep in mind that all these values need to correspond to real-world dimensions, and 1 unit in the game is equal to 1 meter in the real world. So here, when we placed the light source, it is 0.2 meters in the `x` direction, 0.5 meters in the `y` direction, and 0 meters along the `z` direction from the location of the touch.

Adding ground node

This is where we will be adding a ground physics object and setting its category. Make the `ViewController` class inherit from `SCNPhysicsContactDelegate` as follows:

```
class ViewController: UIViewController, ARSCNViewDelegate, SCNPhysicsC  
ontactDelegate
```

Set the `PhysicsCategory` of the different objects in the scene. Add the following to the top of the `ViewController` class:

```
enum PhysicsCategory:Int {  
    case hero = 1  
    case ground = 2  
    case enemy = 4  
}
```

Next, we add the ground node so that we can place the hero on top of it:

```
let ground = SCNPlane()  
ground.firstMaterial?.diffuse.contents = UIColor.gray  
let groundNode = SCNNNode(geometry: ground)  
groundNode.physicsBody = SCNPhysicsBody(type: .static,  
shape:SCNPhysicsShape(geometry: ground, options:nil))  
groundNode.position = SCNVector3(spawnPos.x + 0.2, spawnPos.y,  
spawnPos.z)  
  
groundNode.physicsBody?.categoryBitMask = PhysicsCategory.ground.  
rawValue  
groundNode.physicsBody?.collisionBitMask = PhysicsCategory.hero.  
rawValue  
groundNode.physicsBody!.contactTestBitMask = PhysicsCategory.hero.  
rawValue  
  
groundNode.physicsBody?.restitution = 0.0  
groundNode.name = "ground"  
groundNode.castsShadow = true  
  
sceneView.scene.rootNode.addChildNode(groundNode)
```

We created a `SCNPlane` with the name of `ground` and set the color of the floor to be gray. We then created a `groundNode` of type `SCNNNode` and passed in the `ground` as geometry. We set the physics body to be static and the set the shape to be the same as the ground.

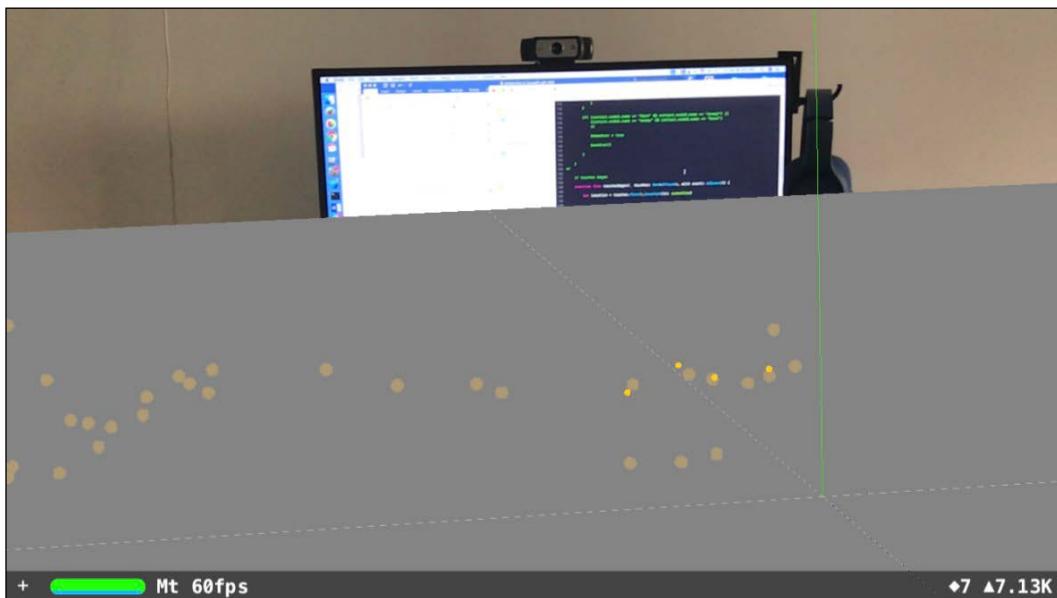
The position of the `groundNode` is set according to the `spawnPos` variable.

The category, collision, and `contactTestBitMasks` are set for the `groundNode` in the same way as in the SceneKit chapter.

We set the restitution to 0, named the ground node as ground, and set the `castsShadow` Boolean to true.

Finally, we added the `groundNode` to the current scene.

Run the project once the yellow plane is created. Tap on the screen to see the light source and ground added to the scene:



Adding Hero and Enemy

To add the player and enemy to the scene, I just copied the `Hero` and `Enemy` classes from the SceneKit chapter and made minor changes to them. The `Hero` class is as follows:

```
import SceneKit  
import ARKit  
  
class Hero:SCNNode{  
  
    var isGrounded = false
```

```
var monsterNode = SCNNode()
var jumpPlayer = SCNAccelerationPlayer()
var runPlayer = SCNAccelerationPlayer()

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

static func time(atFrame frame:Int, fps:Double = 30) -> TimeInterval {
    return TimeInterval(frame) / fps
}
static func timeRange(forStartingAtFrame start:Int,
endingAtFrame end:Int, fps:Double = 30) -> (offset:TimeInterval,
duration:TimeInterval) {
    let startTime = self.time(atFrame: start, fps: fps)
    let endTime = self.time(atFrame: end, fps: fps)
    return (offset:startTime, duration:endTime - startTime)
}

static func animation(from full:CAAnimation, startingAtFrame start:Int, endingAtFrame end:Int, fps:Double = 30) -> CAAnimation {
    let range = self.timeRange(forStartingAtFrame: start,
endingAtFrame: end, fps: fps)
    let animation = CAAnimationGroup()
    let sub = full.copy() as! CAAnimation
    sub.timeOffset = range.offset
    animation.animations = [sub]
    animation.duration = range.duration
    return animation
}

init(_ currentScene: SCNScene, _ spawnPosition: SCNVector3) {
    super.init()

    //load the monster from the collada scene
    let monsterScene:SCNScene = SCNScene(named: "art.scnassets/
theDude.DAE")!
    monsterNode = monsterScene.rootNode.childNode(withName:
"CATRigHub001", recursively: false)! //CATRigHub001
    self.addChildNode(monsterNode)
```

```
// set the anchor point to the center of the character
let (minVec, maxVec) = self.boundingBox
let bound = SCNVector3(x: maxVec.x - minVec.x, y: maxVec.y -
minVec.y, z: maxVec.z - minVec.z)
monsterNode.pivot = SCNMatrix4MakeTranslation(bound.x * 1.1, 0
, 0)

// get the animation keys and store it in the anims
let animKeys = monsterNode.animationKeys.first
let animPlayer = monsterNode.animationPlayer(forKey:
animKeys!)
let anims = CAAnimation(scnAnimation: (animPlayer?.
animation)!)

// get the run animation from the animations
let runAnimation = Hero.animation(from: anims,
startingAtFrame: 31, endingAtFrame: 50)
runAnimation.repeatCount = .greatestFiniteMagnitude
runAnimation.fadeInDuration = 0.0
runAnimation.fadeOutDuration = 0.0

//set the run animation to the player
runPlayer = SCNAccelerationPlayer(animation:
SCNAcceleration(caAnimation: runAnimation))
monsterNode.addAnimationPlayer(runPlayer, forKey: "run")

// get the jump animation from the animations
let jumpAnimation = Hero.animation(from: anims,
startingAtFrame: 81, endingAtFrame: 100)
jumpAnimation.repeatCount = .greatestFiniteMagnitude
jumpAnimation.fadeInDuration = 0.0
jumpAnimation.fadeOutDuration = 0.0

//set the jump animation to the player
jumpPlayer = SCNAccelerationPlayer(animation:
SCNAcceleration(caAnimation: jumpAnimation))
monsterNode.addAnimationPlayer(jumpPlayer, forKey: "jump")

//remove all the animations from the character
monsterNode.removeAllAnimations()

// play the run animation at start
monsterNode.animationPlayer(forKey: "run")?.play()
```

```
// set the collision box for the character
let collisionBox = SCNBox(width: 2/100.0, height: 8/100.0,
length: 2/100.0, chamferRadius: 0)
collisionBox.firstMaterial?.diffuse.contents = UIColor.orange

self.physicsBody = SCNPhysicsBody(type: .dynamic,
shape:SCNPhysicsShape(geometry: collisionBox, options:nil))
self.physicsBody?.categoryBitMask = PhysicsCategory.hero.
rawValue
self.physicsBody?.collisionBitMask = PhysicsCategory.ground.
rawValue
self.physicsBody!.contactTestBitMask = PhysicsCategory.enemy.
rawValue | PhysicsCategory.ground.rawValue

// set angular velocity factor to 0 so that the character
deosnt keel over
self.physicsBody?.angularVelocityFactor = SCNVector3(0, 0, 0)
self.physicsBody?.restitution = 0.0
// set the mass so that the character gets affected by gravity
self.physicsBody?.mass = 20/100.0 //4.5

self.transform = SCNMatrix4MakeRotation(Float(M_PI / 2.0),
0.0, 1.0, 0.0)

//set the scale and name of the current class
self.scale = SCNVector3(0.1/100.0, 0.1/100.0, 0.1/100.0)

self.name = "hero"

self.position = SCNVector3(spawnPositon.x, spawnPositon.y +
0.25, spawnPositon.z)

// add the current node to the parent scene
currentScene.rootNode.addChildNode(self)

}

func jump(){

//print("player jump")
```

```
        if(isGrounded == true) {
            //print("grounded")
            self.physicsBody?.applyForce(SCNVector3Make(0, 0.2, 0),
asImpulse: true) //1400
            isGrounded = false
            playJumpAnim()
        }
    }

func playRunAnim(){

    monsterNode.removeAllAnimations()
    monsterNode.addAnimationPlayer(runPlayer, forKey: "run")
    monsterNode.animationPlayer(forKey: "run")?.play()

}

func playJumpAnim(){

    monsterNode.removeAllAnimations()
    monsterNode.addAnimationPlayer(jumpPlayer, forKey: "jump")
    monsterNode.animationPlayer(forKey: "jump")?.play()

}

// end class
```

The major change is the size of the game object. It has been considerably reduced from its original size to fit in the real world. All the values regarding the size have been reduced to 1% of their original value.

All the value changes have been highlighted in bold, and the `update` function has been completely removed because it is not required.

The `jump` function is changed so that when the screen is tapped to cause the player to jump, the `isGrounded` Boolean is set to `false` and the jump animation is played. Once the player makes contact with the ground, we will set it back to `true` and set the animation to run.

In the `ViewController` class, add a new variable of type `Hero` and call it `hero`:

```
var hero:Hero!
```

In the `setupGame` function, initialize the hero:

```
hero = Hero(sceneView.scene, spawnPos)
hero.castsShadow = true
```

To register physics, we have to make the class conform to the `SCNPhysicsContactDelegate` protocol, add gravity to the current scene, and set the `contactDelegate` to `self`. So, make the following changes in the `viewDidLoad` function:

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

    // Set the view's delegate
    sceneView.delegate = self

    // Show statistics such as fps and timing information
    sceneView.showsStatistics = true

    // Create a new scene
    //let scene = SCNScene(named: "art.scnassets/ship.scn")!

    // Set the scene to the view
    //sceneView.scene = scene

    sceneView.debugOptions = [ARSCNDebugOptions.showWorldOrigin,
                            ARSCNDebugOptions.showFeaturePoints]

    sceneView.scene.physicsWorld.gravity = SCNVector3Make(0,
-500/100.0, 0)
    sceneView.scene.physicsWorld.contactDelegate = self

}
```

Just as in the SceneKit project, add the `didBeginContact` function and the functionality to check whether the player is grounded. If they are grounded, we set the `isGrounded` Boolean to `true` and set the animation to run for the player:

```
func physicsWorld(_ world: SCNPhysicsWorld, didBegin contact:
SCNPhysicsContact) {

    if( (contact.nodeA.name == "hero" && contact.nodeB.name ==
"ground") ||
        (contact.nodeA.name == "ground" && contact.nodeB.name ==
"hero"))

```

```
        ) {  
  
            if(hero.isGrounded == false){  
                hero.isGrounded = true  
                hero.playRunAnim()  
                //print("ground in contact with hero")  
            }  
        }  
    }  
}
```

We also add code to the touchesBegan function so that once the game has been set up we can control the character. Add the highlighted code to the function:

```
override func touchesBegan(_ touches: Set<UITouch>, with event:  
UIEvent?) {  
  
    let location = touches.first!.location(in: sceneView)  
  
    var hitTestOptions = [SCNHitTestOption: Any]()  
    hitTestOptions[SCNHitTestOption.boundingBoxOnly] = true  
  
    let hitTestArray = sceneView.hitTest(location,  
types: [.estimatedHorizontalPlane, .existingPlane,  
.existingPlaneUsingExtent])  
  
    for result in hitTestArray {  
  
        //print(result.anchor)  
  
        if(!bGameSetup){  
  
            let spawnPos = SCNVector3(x: result.worldTransform.  
columns.3.x,  
                           y: result.worldTransform.  
columns.3.y,  
                           z: result.worldTransform.  
columns.3.z)  
            if(bPlaneAdded){  
  
                setupGame(spawnPos)  
  
                plane.isHidden = true  
            }  
        }  
    }  
}
```

```
        bGameSetup = true
    }

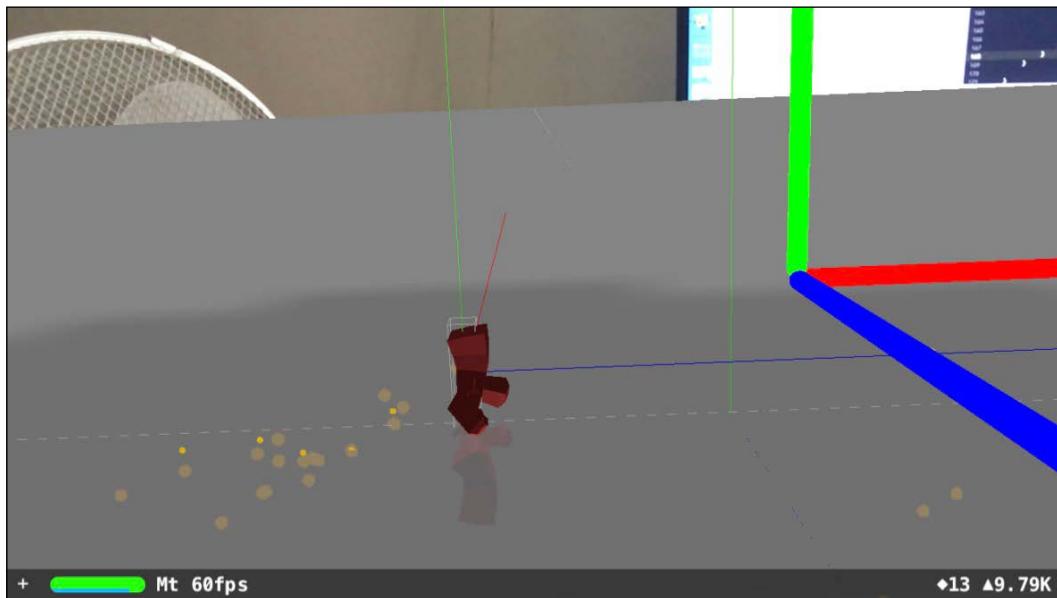
}else{

    if(!bGameOver){
        hero.jump()
    }
}
}
```

Enable physics debugging so that you can see the player collision box information. In the `viewDidLoad` function, add the following line to the code:

```
sceneView.debugOptions = [ARSCNDebugOptions.showWorldOrigin,
                         ARSCNDebugOptions.showFeaturePoints,
                         SCNDebugOptions.showPhysicsShapes]
```

Run the project, and once the plane is added, tap on the screen to add the plane and player. And now when you tap, the player will jump:



Next, we will add the enemy. Here is the `Enemy.swift` file:

```
import SceneKit

class Enemy:SCNNode{

    var _currentScene: SCNScene!
    var spawnPos: SCNVector3!

    var score:Int = 0

    init(_ currentScene: SCNScene, _ spawnPositon: SCNVector3) {

        super.init()

        print("spawn enemy")

        self._currentScene = currentScene
        self.spawnPos = spawnPositon

        spawnPos = SCNVector3(spawnPos.x + 0.8, spawnPos.y + 2.0/100,
        spawnPos.z)

        let geo = SCNBox(width: 4.0/100.0, height: 4.0/100.0, length:
        4.0/100.0, chamferRadius: 0.0)
        geo.firstMaterial?.diffuse.contents = UIColor.yellow

        self.geometry = geo

        self.position = spawnPos
        self.physicsBody = SCNPhysicsBody.kinematic()
        self.name = "enemy"

        self.physicsBody?.categoryBitMask = PhysicsCategory.enemy.
rawValue
        self.physicsBody?.contactTestBitMask = PhysicsCategory.hero.
rawValue

        currentScene.rootNode.addChildNode(self)

    }

}
```

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

func update() {
    self.position.x += -0.9/100.0

    if((self.position.x - 5.0/100.0) < -60/100.0) {
        let factor = arc4random_uniform(2) + 1

        if( factor == 1 ) {
            self.position = spawnPos
        }else{
            self.position = SCNVector3Make(spawnPos.x, spawnPos.y
+ 0.1 , spawnPos.z)
        }

        score += 1
    }
}

func reset() {
    self.position = spawnPos
    self.score = 0
}
```

As with the hero, we place the enemy based on where the hero is placed, and we adjust the size of the object so that it is scaled down to the physical world.

We also calculate the score locally. We have added a variable called `score` of type `int` and increment it if the player doesn't hit the enemy. There is also a new `reset` function, which resets the score and spawns the position of the enemy.

In the `ViewController.swift` file, we create a new global variable called `enemy` of type `Enemy`:

```
var hero:Hero!
var enemy:Enemy!
```

We initialize it in the `setupGame` function:

```
hero = Hero(sceneView.scene, spawnPos)
hero.castsShadow = true

enemy = Enemy(sceneView.scene, spawnPos)
enemy.castsShadow = true
```

Add a new update function so that we can update the location of the enemy. Create a new Boolean called `bGameOver` and set it to `false`. We will only update the game if the `bGameOver` condition is `false`. Add the `update` function as follows:

```
func renderer(_ aRenderer: SCNSceneRenderer, updateAtTime time:
TimeInterval) {

    if (bGameSetup) {

        if (!bGameOver) {

            enemy.update()

        }
    }
}
```

In the `didBeginContact` function, add functionality to check whether contact has been made between the hero and the enemy, and set the `gameover` condition to `true` if the contact occurs:

```
func physicsWorld(_ world: SCNPhysicsWorld, didBegin contact:
SCNPhysicsContact) {

    //print("begin contact")

    if( (contact.nodeA.name == "hero" && contact.nodeB.name ==
"ground") ||
        (contact.nodeA.name == "ground" && contact.nodeB.name ==
"hero"))
    ) {

        if(hero.isGrounded == false){
            hero.isGrounded = true
            hero.playRunAnim()
```

```
        //print("ground in contact with hero")
    }
}

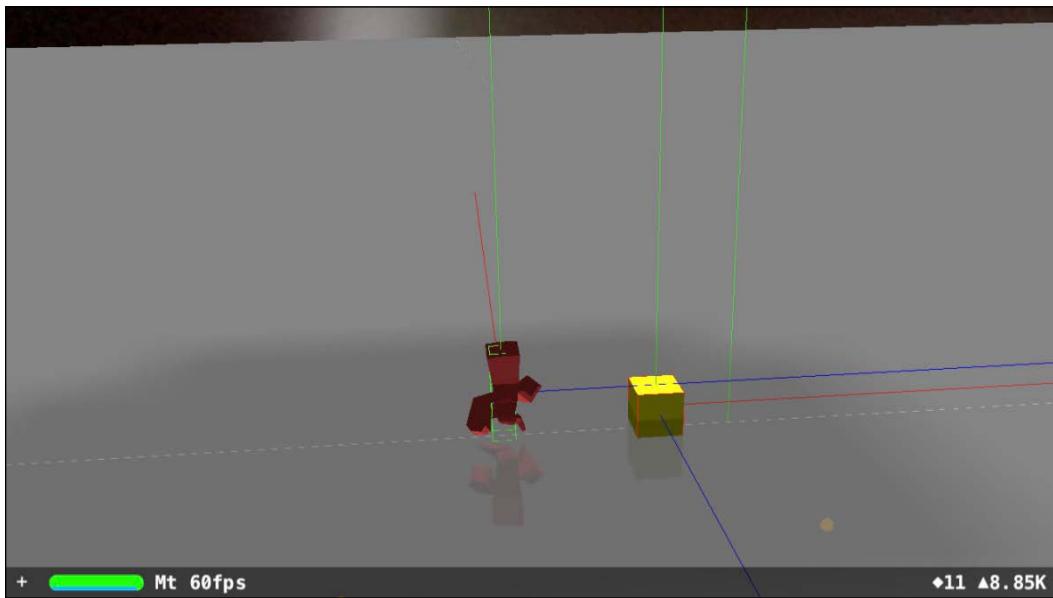
if( (contact.nodeA.name == "hero" && contact.nodeB.name ==
"enemy") ||
    (contact.nodeA.name == "enemy" && contact.nodeB.name ==
"hero"))
{
    bGameOver = true

    enemy.reset()

}
```

}

Now run the project to see the game in action. The game will start, and you can see the enemy moving towards the player:



Adding Score and Gameover text

Just as we can have 3D game objects, we can also have 3D text. We will create 3D text and make it part of the world instead of adding it to the screen space, which is what we have done so far.

Add the following lines to add the `SCNText` and `SCNNNode` to the current class:

```
var scoreLabel: SCNText!
var scoreNode: SCNNNode!

var gameOverLabel: SCNText!
var gameOverNode: SCNNNode!
```

In the `setupGame` function, add the following lines of code to add the 3D text to the scene:

```
scoreLabel = SCNText(string: "Score: 0", extrusionDepth: 0.2)
scoreLabel.font = UIFont(name: "Arial", size: 4)
scoreLabel.firstMaterial!.diffuse.contents = UIColor.blue

let (scoreLabelMinVec, scoreLabelMaxVec) = scoreLabel.
boundingBox
let scoreBound = SCNVector3(x: scoreLabelMaxVec.x -
scoreLabelMinVec.x,
                            y: scoreLabelMaxVec.y -
scoreLabelMinVec.y,
                            z: scoreLabelMaxVec.z -
scoreLabelMinVec.z)
scoreNode = SCNNNode(geometry: scoreLabel)
scoreNode.pivot = SCNMatrix4MakeTranslation(scoreBound.x *
0.5, 0 , 0)
scoreNode.position = SCNVector3(spawnPos.x + 0.2, spawnPos.y +
0.16, spawnPos.z)
scoreNode.scale = SCNVector3(0.0125, 0.0125, 0.0125)
sceneView.scene.rootNode.addChildNode(scoreNode)

// gameover label
gameOverLabel = SCNText(string: "GAMEOVER !!!",
extrusionDepth: 0.2)
gameOverLabel.font = UIFont(name: "Arial", size: 4)
gameOverLabel.firstMaterial!.diffuse.contents = UIColor.red
```

```
let (gameOverLabelMinVec, gameOverLabelMaxVec) =  
gameOverLabel.boundingBox  
let goBound = SCNVector3(x: gameOverLabelMaxVec.x -  
gameOverLabelMinVec.x,  
y: gameOverLabelMaxVec.y -  
gameOverLabelMinVec.y,  
z: gameOverLabelMaxVec.z -  
gameOverLabelMinVec.z)  
gameOverNode = SCNNNode(geometry: gameOverLabel)  
gameOverNode.pivot = SCNMatrix4MakeTranslation(goBound.x *  
0.5, 0 , 0)  
gameOverNode.position = SCNVector3(spawnPos.x + 0.2,  
spawnPos.y + 0.08, spawnPos.z)  
gameOverNode.scale = SCNVector3(0.02, 0.02, 0.2)  
sceneView.scene.rootNode.addChildNode(gameOverNode)  
  
gameOverNode.isHidden = true
```

We add two texts; one is for displaying score and the other is for displaying the gameOver text.

We initially hide the gameOver text and unhide it when the game is over:

```
func physicsWorld(_ world: SCNPhysicsWorld, didBegin contact:  
SCNPhysicsContact) {  
  
    //print("begin contact")  
  
    if( (contact.nodeA.name == "hero" && contact.nodeB.name ==  
"ground") ||  
        (contact.nodeA.name == "ground" && contact.nodeB.name ==  
"hero")  
    ) {  
  
        if(hero.isGrounded == false){  
            hero.isGrounded = true  
            hero.playRunAnim()  
            //print("ground in contact with hero")  
        }  
    }  
  
    if( (contact.nodeA.name == "hero" && contact.nodeB.name ==  
"enemy") ||  
        (contact.nodeA.name == "enemy" && contact.nodeB.name ==  
"hero")  
    ) {
```

```
) {  
    bGameOver = true  
  
    enemy.reset()  
    gameOverNode.isHidden = false  
  
}  
}
```

We hide it again when we restart the game:

```
override func touchesBegan(_ touches: Set<UITouch>, with event:  
UIEvent?) {  
  
    let location = touches.first!.location(in: sceneView)  
  
    var hitTestOptions = [SCNHitTestOption: Any]()  
    hitTestOptions[SCNHitTestOption.boundingBoxOnly] = true  
  
    let hitTestArray = sceneView.hitTest(location,  
types: [.estimatedHorizontalPlane, .existingPlane,  
.existingPlaneUsingExtent])  
  
    for result in hitTestArray {  
  
        //print(result.anchor)  
  
        if(!bGameSetup){  
  
            let spawnPos = SCNVector3(x: result.worldTransform.  
columns.3.x,  
                           y: result.worldTransform.  
columns.3.y,  
                           z: result.worldTransform.  
columns.3.z)  
            if(bPlaneAdded){  
  
                setupGame(spawnPos)  
  
                plane.isHidden = true  
            }  
        }  
    }  
}
```

```
        bGameSetup = true
    }

}else{

    if (!bGameOver) {
        hero.jump()
    }else{

        bGameOver = false
        gameOverNode.isHidden = true

    }
}

}
```

The game score is updated in the update function as follows:

```
func renderer(_ aRenderer: SCNSceneRenderer, updateAtTime time:
TimeInterval) {

    if (bGameSetup) {

        if (!bGameOver) {

            enemy.update()

            scoreLabel.string = "Score: \(enemy.score)"
        }
    }
}
```

Run the game, and you will see the updated score and GAMEOVER!!! text as follows:



Finishing touches

Although the game is done, we can see that the plane is completely covering the scene and breaking the AR experience. We will disable the rendering of the floor and make sure that only the shadow is preserved so that it looks like part of the real world.

To do this, after creating the floor, set the `colorBufferWriteMask` value to 0 as follows:

```
let ground = SCNPlane()
ground.firstMaterial?.diffuse.contents = UIColor.gray

ground.firstMaterial?.colorBufferWriteMask = .init(rawValue: 0)
```

We don't need the debug information anymore, so we can comment out the debugOption line in the viewDidLoad function:

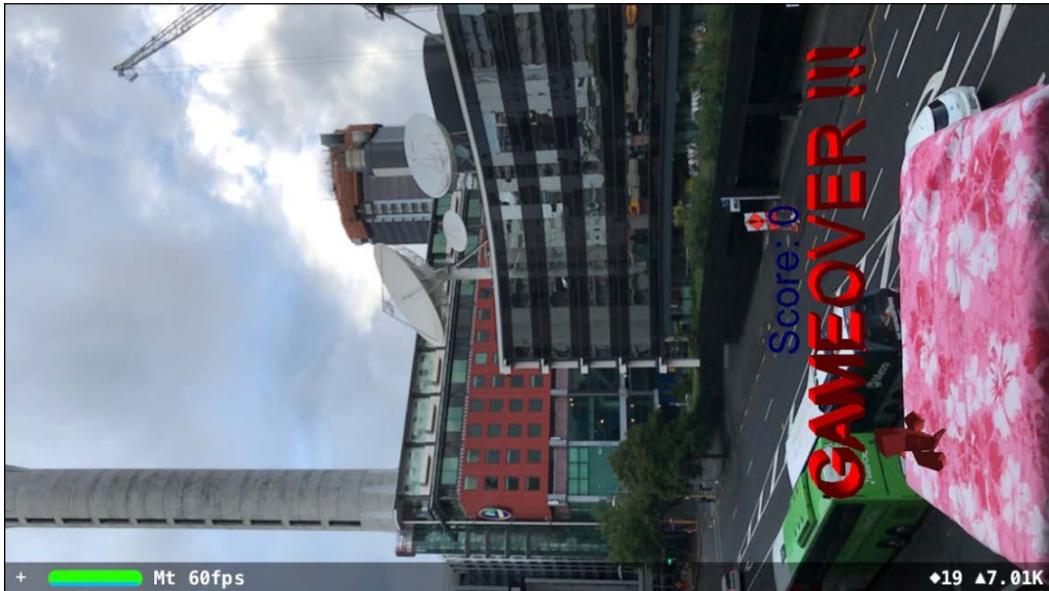
```
//sceneView.debugOptions = [ARSCNDebugOptions.showWorldOrigin,  
//                           ARSCNDebugOptions.showFeaturePoints,  
//                           SCNDebugOptions.showPhysicsShapes]  
  
    sceneView.scene.physicsWorld.gravity = SCNVector3Make(0,  
-500/100.0, 0)  
    sceneView.scene.physicsWorld.contactDelegate = self
```

Run the application now:



You can see that the shadows look as if they are being cast in the real world.

Now you can take the game and play it wherever you want:



Summary

In this chapter, we saw how to create a 3D AR game using SceneKit and ARKit. We saw what changes have to be made to the regular SceneKit scene to make it into an AR game. The major difference here is the detection of the plane and making sure the objects are in the same scale as the surrounding objects.

We are done with the majority of this book. In the next and final chapter, we will learn how to upload the game to the App Store and publish our creation to the world.

16

Publishing the Game on the App Store

The moment is finally here. We have put our time and effort into making this awesome AR game, and now we want the world to know about our creation.

The road to publishing a game is actually quite straightforward, provided you follow the steps correctly.

First, we will go to the Apple Developer portal and create a suitable Bundle ID for the game.

Then, we will make changes to the project to make it upload-ready. We will change the name of the app, add an app icon, change the Bundle ID to the one we created in the Developer portal, and remove the debug code from the main project.

Once the project is ready to upload, we will create the app in the `itunesconnect` portal. This is where you will give the app a name and a description, add screenshots, and add an icon. You will also have the opportunity to decide whether you would like to give the game away for free or put it up for a price.

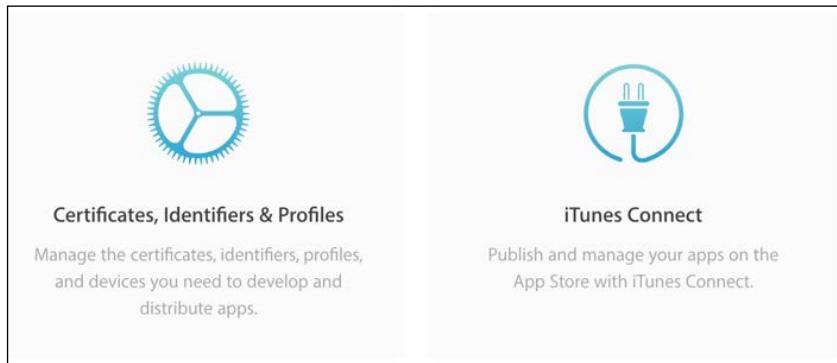
Finally, you will upload the app file from Xcode onto the App Store and submit the app for review.

We will cover the following in this chapter:

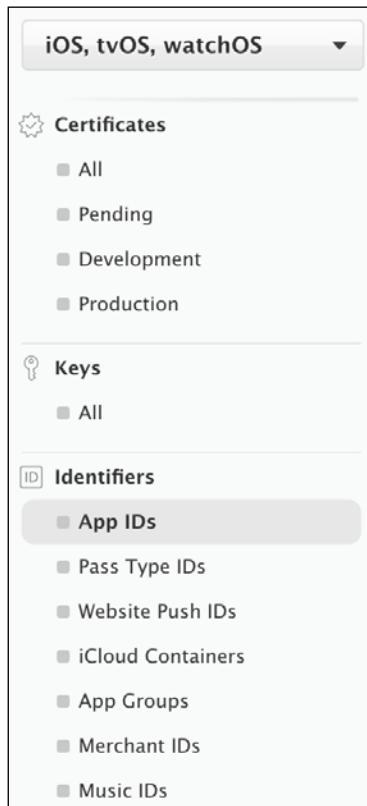
- Creating a Bundle ID for the app
- Preparing the project
- Creating the app in the `itunesconnect` portal
- Uploading the app and submitting for review

Creating the Bundle ID for the app

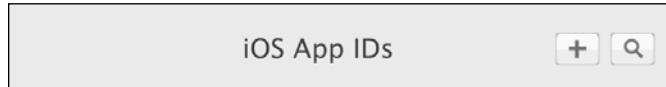
1. Go to developer.apple.com and log in to your account. Select **Certificates, Identifiers & Profiles** as shown here:



2. Under **Identifiers**, select **App IDs**:



3. In the top-right corner, select the + sign next to the search button:



The App ID is a unique word string, which is used by Apple to identify your application from all the applications that are uploaded to the App Store.

4. Once you click the + sign, you will have to fill in the **App Id Description** and explicitly provide an app ID:

The screenshot shows the 'App ID Description' configuration page. It includes fields for 'Name' (containing 'JumpyJumpAR'), 'App ID Prefix' (containing '6KJ7MQ3AE9'), and 'App ID Suffix'. A note indicates that explicit App IDs are required for services like Game Center and In-App Purchase. The 'Explicit App ID' option is selected, and the 'Bundle ID' field contains 'com.growlgamesstudio.jumpyjumpAR', with a note about using reverse-domain names.

App ID Description

Name: JumpyJumpAR
You cannot use special characters such as @, &, *, ', "

App ID Prefix

Value: 6KJ7MQ3AE9 (Team ID)

App ID Suffix

Explicit App ID
If you plan to incorporate app services such as Game Center, In-App Purchase, Data Protection, and iCloud, or want a provisioning profile unique to a single app, you must register an explicit App ID for your app.
To create an explicit App ID, enter a unique string in the Bundle ID field. This string should match the Bundle ID of your app.

Bundle ID: com.growlgamesstudio.jumpyjumpAR
We recommend using a reverse-domain name style string (i.e., com.domainname.appname). It cannot contain an asterisk (*).

For the description, provide the name of the app, and for the app ID, use the reverse website syntax followed by the app name to create a unique **Bundle ID**.

5. Once you are done, click **Done** at the bottom and click register on the next page.

Next, we will set up the app in Xcode.

Preparing the project

Open up the project in Xcode.

Under the project's **General** settings, change the **Display Name** and **Bundle Identifier** to what we set them to in the **Developer** portal:



Next, we have to change the app icon. The app icon needs to be of different sizes without an alpha. The easiest way to create different app sizes is to use the website <https://makeappicon.com/>; you just have to provide a 1024 x 1024 image, then upload it to the site, and it will email you the icon in different sizes, which you can simply drag and drop onto your project:

A screenshot of the MakeAppIcon website. It features a dark header with the logo and a sub-header 'powered by Skygear'. Below this, there's a promotional message for 'Skygear' and 'ShotBot'. The main area has a large image of a laptop displaying multiple mobile device icons. Text on the page includes 'Best icon resizer for mobile app developers.', 'Now supports iOS11 icons for Xcode9!', '30 DAY MONEY BACK GUARANTEE', 'MakeAppIcon Desktop for Mac and Windows', and a list of features: 'Save upload time to MakeAppIcon Server', 'Generate Android and iOS icons in bulk', 'Preview icons', 'Simple yet easy to use interface', and '*30-day money back guarantee'. A green button at the bottom right says 'Buy now at 9.99USD only'.

Open `Assets.xcassets` in your project in Xcode and you will see `AppIcon` already in it. Delete it and drag and drop the icon set `Appicon.appiconset` under the `iOS` folder into the downloaded icon set:



Next, in the `ViewController.swift` file, comment out the following code:

```
// Show statistics such as fps and timing information
//sceneView.showsStatistics = true

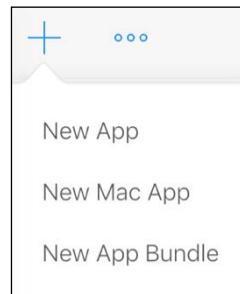
// Create a new scene
//let scene = SCNScene(named: "art.scnassets/ship.scn")!

// Set the scene to the view
//sceneView.scene = scene

//sceneView.debugOptions = [ARSCNDebugOptions.showWorldOrigin,
//                          ARSCNDebugOptions.
showFeaturePoints,
//                          SCNDebugOptions.showPhysicsShapes]
```

Create the App in the itunesconnect portal

Open up itunesconnect.apple.com and log in. Once logged in, click on **My Apps** and the + icon to create a **New App**:



Select the platform as **iOS**; for **Name**, give the name of the app; choose the **Primary Language**; choose the **Bundle ID** from the drop-down list; and provide a **SKU** number. For the **SKU**, if this is your first game, enter **001**:

The screenshot shows the 'New App' creation form. It includes fields for Platform (checkboxes for iOS and tvOS), Name (text input field), Primary Language (dropdown menu with 'Choose'), Bundle ID (dropdown menu with 'Choose'), and SKU (text input field). A note at the bottom of the form says 'Register a new bundle ID on the Developer Portal.' At the bottom right are 'Cancel' and 'Create' buttons.

New App

Platforms ?

iOS tvOS

Name ?

Primary Language ?

Choose

Bundle ID ?

Choose

Register a new bundle ID on the [Developer Portal](#).

SKU ?

Cancel Create

Once the app is created, go to the **App Information** tab. Here, we have to fill in some more information.

We have to fill in the details of the **Localizable information**, where we can add a subtitle for the game and provide a privacy policy URL:

The screenshot shows the 'Localizable Information' section. At the top right is a language dropdown set to 'English (U.S.)'. Below it are two input fields: 'Name' containing 'JumpyJumpAR' and 'Privacy Policy URL' containing 'http://example.com (optional)'. Further down is a 'Subtitle' field containing 'An AR Obstacle Avoidance game'.

When we scroll down, we can specify the **Category** of the app:

The screenshot shows the 'Category' section. It has a title 'Category' with a question mark icon. Below it are three dropdown menus: the first is set to 'Games', the second to 'Action', and the third to 'Adventure'.

For the **Main Category**, I selected **Games**, and for the Sub-Category, I selected **Action** and then **Adventure**.

That's it for this section. Next, select **Pricing and Availability** on the tab on the left.

For pricing, I chose the free option; if you would like your app to be paid, you can select the price of your choice.

For availability, I left it as the default, which is **All Territories**:

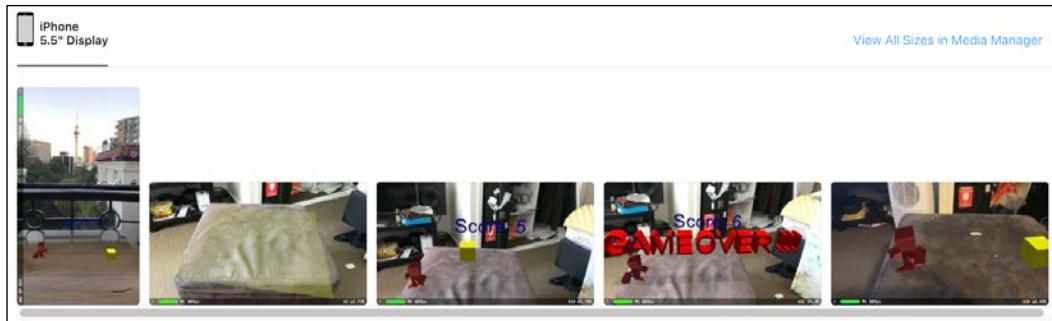
The screenshot shows the 'Pricing and Availability' section of an app's submission page. At the top right is a 'Save' button. Below it is a 'Price Schedule' table with columns for Price, Start Date, and End Date. The price is set to NZD 0 (Free) with a dropdown for other currencies, and the start date is Feb 3, 2018, with no end date. A link 'All Prices and Currencies' is at the top right of the table. Below the table is a 'Plan a Price Change' button. The next section is 'Pre-Orders', which explains what they are and how customers can order before release. It includes a checkbox 'Make available for pre-order'. The final section is 'Availability', showing 'All territories selected' with an 'Edit' link.

That's it for this section. Next, select the **Prepare for submission** tab on the left.

The first thing we have to do is upload screenshot images. There are two **iPhone** screenshot tabs and an **iPad** tab. It's good enough if we upload for the 5.5" display as Apple will copy the same screenshots for other devices as well.

We have to upload at least five images. The images also need to have a resolution of 2208 × 1242 for landscape mode and shouldn't have alpha, so it is better if the image is a JPEG.

I have uploaded five images for the screenshot, as follows. You can simply drag and drop the images to the screenshot section:



Next, we have to fill in the **Promotion Text** and **Description** for the game, as shown here:

Promotional Text ?

An Augmented Reality Obstacle Avoidance game developed for Learning iOS game development book for Packt Publishing

56

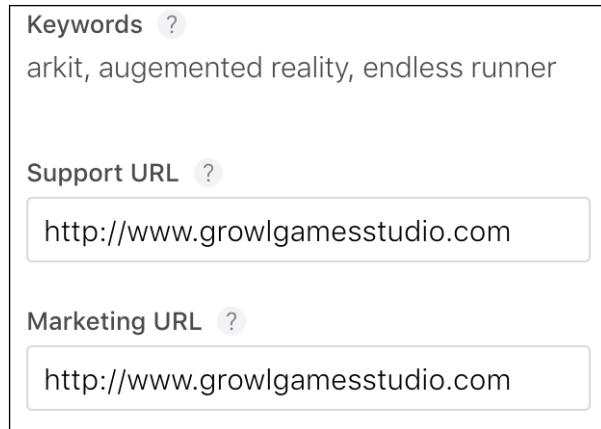
🔒 Description ?

An Augmented Reality Obstacle Avoidance game developed for Learning iOS game development book for Packt Publishing.

Instruction to play the game
=====

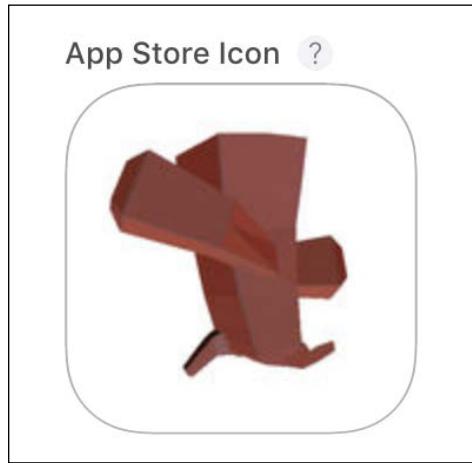
1. When the app launches, select a flat plane to play the game on.
2. Once the application detects the plane, it will be highlighted in yellow.
3. Tap on the screen about 1/3rd from the left of the screen. You will see the main character fall on to the surface
4. Now tap on the screen to make the character jump and avoid the obstacle.

We also need to provide keywords so that our app is easy to search for on the App Store, and we need to provide a **Support URL** and **Marketing URL** just in case a customer wants to reach out to us. Therefore, I have provided a bunch of keywords and my company website as required:



Next, under the **General App Information** heading, we have to provide the icon, rating, copyright information, and trade representative contact information.

First, provide the icon for the app. This needs to be an image with a resolution of 1024 x 1024 and without alpha. You can drag and drop the icon right on the area to upload it:



For the copyright information, provide your company web address.

For the trade representative contact information, provide your full name, office address, phone number, and email address.

Next, we have to provide the rating. The rating determines which age group the game is suitable for. Click on the **Edit** button next to **Rating**.

For the rating, select the appropriate one for your game. For the AR game we have developed, I chose the following options:

Apple Content Description	None	Infrequent/Mild	Frequent/Intense
Cartoon or Fantasy Violence	<input checked="" type="radio"/>	<input type="radio"/>	<input type="radio"/>
Realistic Violence	<input checked="" type="radio"/>	<input type="radio"/>	<input type="radio"/>
Prolonged Graphic or Sadistic Realistic Violence	<input checked="" type="radio"/>	<input type="radio"/>	<input type="radio"/>
Profanity or Crude Humor	<input checked="" type="radio"/>	<input type="radio"/>	<input type="radio"/>
Mature/Suggestive Themes	<input checked="" type="radio"/>	<input type="radio"/>	<input type="radio"/>
Horror/Fear Themes	<input checked="" type="radio"/>	<input type="radio"/>	<input type="radio"/>
Medical/Treatment Information	<input checked="" type="radio"/>	<input type="radio"/>	<input type="radio"/>
Alcohol, Tobacco, or Drug Use or References	<input checked="" type="radio"/>	<input type="radio"/>	<input type="radio"/>
Simulated Gambling	<input checked="" type="radio"/>	<input type="radio"/>	<input type="radio"/>
Sexual Content or Nudity	<input checked="" type="radio"/>	<input type="radio"/>	<input type="radio"/>
Graphic Sexual Content and Nudity	<input checked="" type="radio"/>	<input type="radio"/>	<input type="radio"/>
	No	Yes	
Unrestricted Web Access	<input checked="" type="radio"/>	<input type="radio"/>	
Gambling and Contests	<input checked="" type="radio"/>	<input type="radio"/>	
<input type="checkbox"/> Made for Kids			

i Your selected app ratings is **Ages 4+**.

Based on my selection, a rating of 4+ has been selected for my app.

Finally, under the **Version Release** heading, we can specify when we want to release the game. Here, I have chosen to automatically release it when it's ready for sale:

Version Release

Pre-orders are an easy way to generate sales and exposure before release. To make your app available for pre-order, go to [Pricing and Availability](#).

After your app has been approved, we can release it for you immediately. If you want to release the app yourself, choose a date or manually release it at any point after the approval. While your app is in the "Pending Developer Release" state, you can give out promotional codes, continue TestFlight Beta Testing, or reject the release and submit a new build. Whichever of these you choose, we have to process your app before it's made available on the App Store. While your app is in the "Processing for App Store" state, you can't get new promotional codes, invite new testers, or reject your app.

Manually release this version

Automatically release this version ?

Automatically release this version after App Review, no earlier than ?

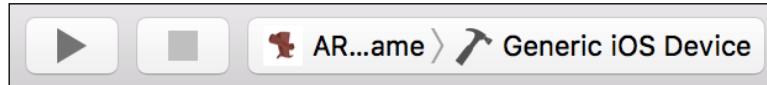
Your local date and time.

 Feb 3, 2018  12:00 AM

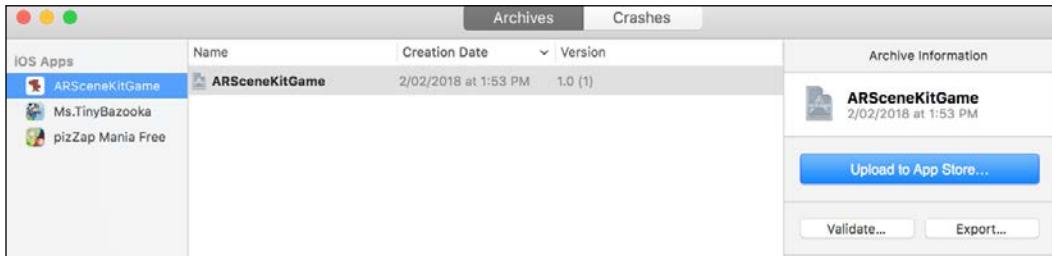
Upload the App and submit for review

Now, let's upload the app from Xcode. Go back to the project.

In the **Project**, in the simulator, select **Generic iOS Device**. Now go to **Product** on the top bar and select **Archive**. This will create a build that can be uploaded to the App Store. This will take a while:

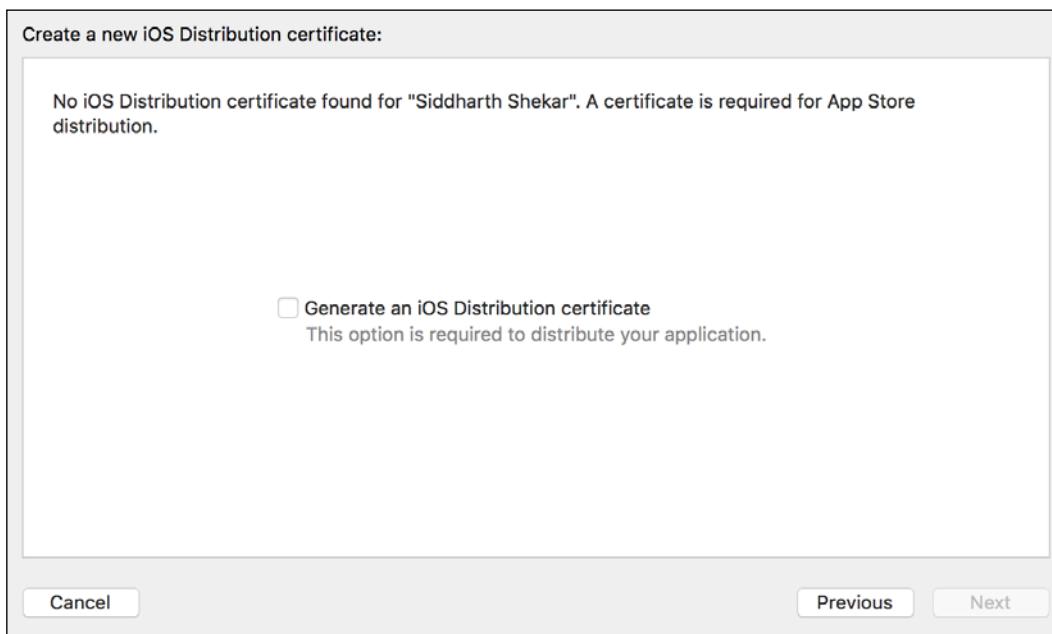


Once done, under **Window** on the top bar, select **Organizer**:



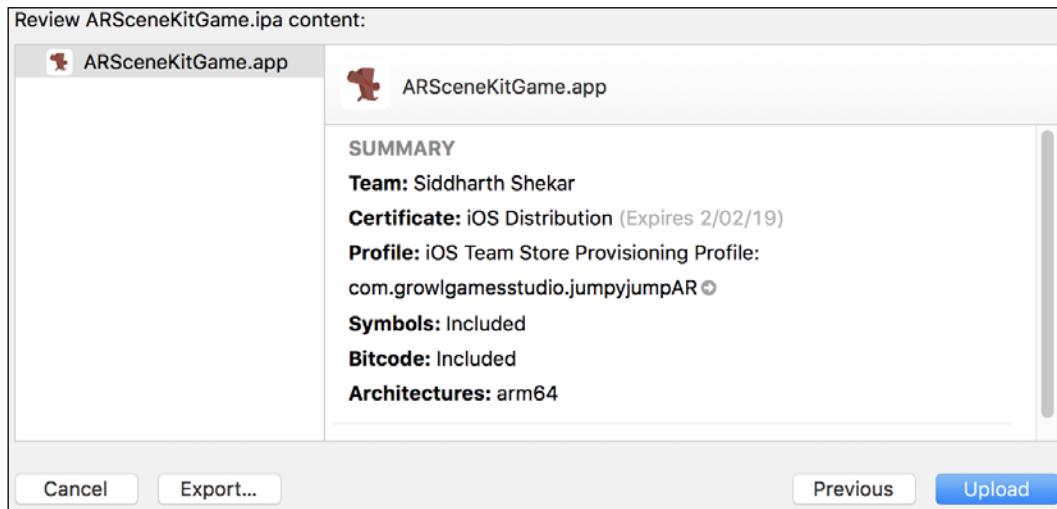
Click on the **Upload to App Store...** button... on the right. This will start the process of uploading the app to the store.

We didn't create a distribution certificate, so it will ask if you want to create one. Check the **Generate an iOS Distribution Certificate** checkbox and click **Next**. Once it is created, you can click **Next**:

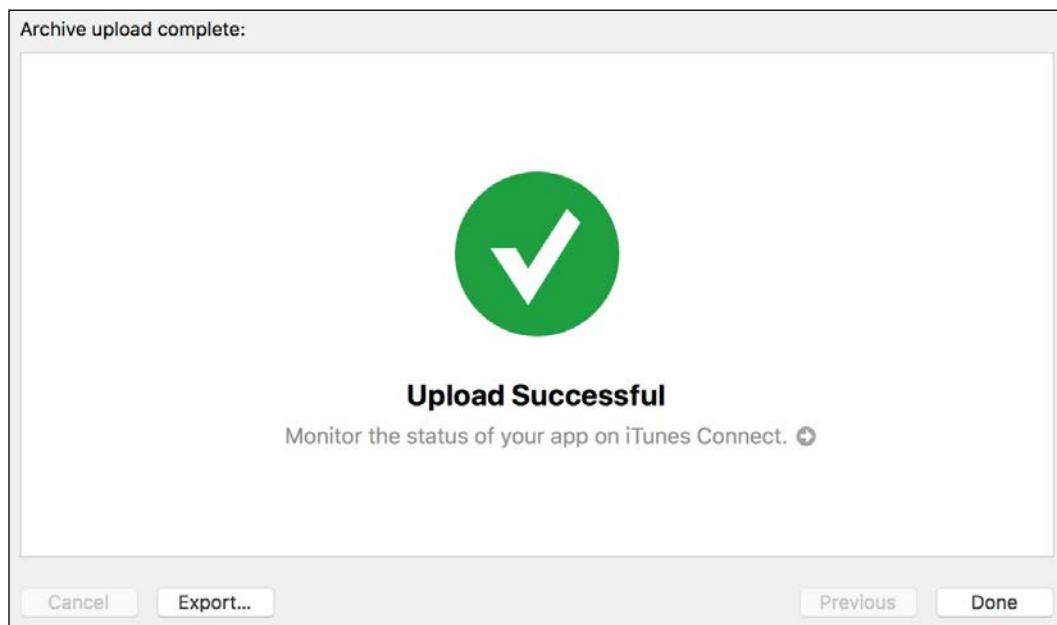


Publishing the Game on the App Store

Once it has created the archive, you can upload the content to the App Store:



Click on the **Upload** button to start uploading the app. Once it has uploaded, you will see the following screen confirming that the upload was successful:



Click Done.

Once done, we can go back to the `itunesconnect` app portal.

In **Prepare for Submission**, under the **Build** heading, you can now select the build:

Build	Upload Date
<input type="radio"/>	1.0 (1) Feb 2, 2018 at 2:19 PM

Cancel **Done**

Check build **1.0 (1)** and click **Done**.

Click on **Submit for Review** at the top right of the screen.

On the next screen, you will be asked to choose **Yes** or **No** for **Export Compliance**, **Content Rights**, and **Advertising Identifiers**. Choose appropriately. If you don't, then your binary will be deleted and a new one will have to be uploaded:

Export Compliance

Is your app designed to use cryptography or does it contain or incorporate cryptography? (Select Yes even if your app is only utilizing the encryption available in iOS or macOS.)

Yes No

i If you are making use of ATS or making a call to HTTPS please note that you are required to submit a year-end self classification report to the US government. [Learn more](#)

Content Rights

Does your app contain, display, or access third-party content?

Yes No

Advertising Identifier

Does this app use the Advertising Identifier (IDFA)?

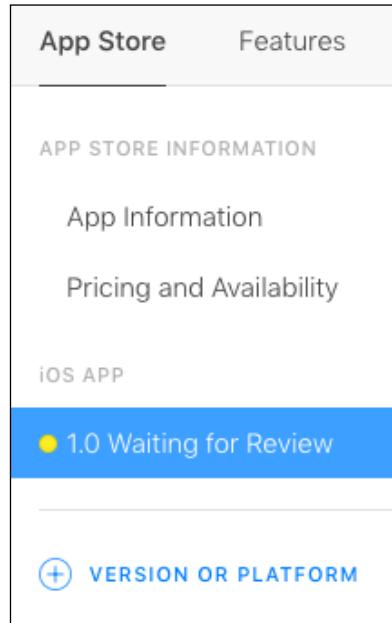
The [Advertising Identifier \(IDFA\)](#) is a unique ID for each iOS device and is the only way to offer targeted ads. Users can choose to limit ad targeting on their iOS device.

Ensure that you select the correct answer for Advertising Identifier (IDFA) usage. If your app does contain the IDFA and you select No, the binary will be permanently rejected and you will have to submit a different binary.

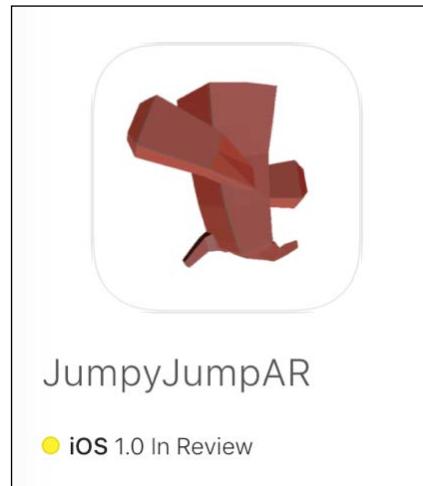
Cancel **Submit**

Once done, click **Submit**.

Congratulations! Now your app will be added to the queue of apps waiting for review:



You will get email notifications when the app is moved to **In Review** (as shown in the following screenshot), and then when the app is available for sale:



Summary

In this chapter, we saw how to prepare the app for upload to the App Store and how to create the app itself on the app store, upload the app to the app store, and make it available for distribution. You can try uploading other apps to the App Store as well.

You can add audio and a `scene manager` to the game as the implementation is exactly the same as how you would do it in a `SpriteKit` game. You can try submitting the game created in *Chapter 11, Introduction to SceneKit*, as well.

In the next chapter, we will see how to create a Multiplayer AR project.

17

Multipeer Augmented Reality

In this chapter, we will see how to create a multi-peer augmented reality app. Until now, we have only made applications that run on a single device and we didn't have to worry about how to connect with another device and also share data with the other device.

Apple already has a framework to connect with other devices and manage data transfer between the devices. This framework for connecting between devices, called Multipeer Connectivity, has been around since iOS 7. In iOS 12, Apple has extended the functionality of the framework for developers to create Multipeer applications.

In this chapter, we will cover the following topics:

- Multipeer connectivity framework overview
- Creating a multipeer session class
- Creating a UI for the app
- Setting outlets and adding variables
- Initializing the view
- Updating session and tracking
- Hosting and joining the session
- Sending and receiving data
- Initializing a multipeer session
- Testing the application

Multipeer connectivity framework overview

The framework for creating Multipeer apps is called the Multipeer Connectivity Framework.

The framework lets you discover the nearby devices. This is called the Discovery Phase. Once you connect to the users, you invite them to join a session. This enables the next phase, called the Session Phase, where the data is exchanged between the devices.

The device initiating the connection with the other devices is called the Advertiser and the device choosing to connect to the Advertiser is called the Browser. This is because the browser looks out for a device to connect to.

The framework uses Wi-Fi networks, peer-to-peer Wi-Fi, and Bluetooth personal area networks for the underlying transport of the data between the devices.

The Multipeer Connectivity Framework is not limited to iOS devices. macOS and tvOS have the same functionality. For these devices, connection can also be established using Ethernet.

With the Multipeer Connectivity Framework, the application will interact with a couple of types of framework-specific objects:

- Peers IDs (`MCPeerID`): This is a unique identifier that identifies the devices connected.
- Session objects (`MCSession`): A session object should be created first. This enables the connection between the host and the peers and manages the connection itself. The session objects maintain a set of peer IDs that represent the peers connected to the session.
- Advertiser objects (`MCNearbyServiceAdvertiser`): An advertiser object tells the nearby peers that the app is advertising a session with a specific service. This provides information to other devices with the current devices' users device ID to connect.
- Advertiser assistant objects (`MCAdvertiserAssistant`): These provide the same functionality as advertiser objects but, in addition, also give more control over which invitations are displayed. For the app we will be developing, we will use this object type to connect with other peers.
- Browser objects (`MCNearbyServiceBrowser`): These let your app search for nearby advertisers that support sessions of a certain service type.

- Browser view controller objects (`MCBrowserViewController`): These give you a standard interface to additionally choose between different nearby peers to connect to a session. We will be using browser view controller object to choose the device to connect to.

Creating the multipeer session class

First, we will create an `MCSession` object to create a session. The `init` takes three parameters:

1. The `peerID` of the current device.
2. The security identity to secure the connection (in this case, we are not going to, as we are aware who we are connecting to).
3. The encryption preference. We can specify the required, preferred, or undesirable encryption. We will enable encryption by stating the required encryption.

Next, we will add a button for the Advertiser and the Browser.

If the device is an Advertiser, then the `AdvertiseSelf` function will be called. This will start the `MCAdvertiserAssistant` to get ready to connect to other devices.

If the device is a Browser, then the user will click the browse button to look for advertisers and get a list of Advertisers to connect to.

We will create functions to send information to all the peers and will send information regarding the current surroundings, any anchors that were added, and any other data sent.

We will also create a function that returns all the current peers connected to the session.

A session also has a `MCSessionDelegate` function that handles session-related events. This delegate keeps track of the devices that are connected to the session, checking if data was received from a peer or if a stream/resource was received from a peer.

With this information, let's create a new Xcode Augmented Reality app project. For game technology, select **SceneKit**, and for Language, select **Swift**. Don't forget to add a **Product Name**, **Organization Name**, and **Identifier**.

In the new project created, make a Swift file called `MultipeerSession`.

At the top of the class, import the `MultipeerConnectivity` framework.

Create a new class called `MultipeerSession` and inherit from `NSObject`, as shown in the following code. We will add code in the class:

```
class MultipeerSession: NSObject{  
}
```

First, we will add all the properties that are required for the class. We will need a service type for the advertiser and the broadcaster. This will make sure that we are connected to the same service type:

```
// Limited to 15 ASCII characters  
static let serviceType = "ar-multi-sample"
```

Next, we will create a private variable that will store the current device's `peerID`. For this, we will use `MCPeerID` and get the current device's name using `UIDevice`:

```
// Name of current device  
private let myPeerID = MCPeerID(displayName: UIDevice.current.  
name)
```

We then have to create `MCCBrowserViewController`, `MCAAdvertiserAssistant`, and `MCSession` instances:

```
var browser:MCCBrowserViewController!  
var advertiser:MCAAdvertiserAssistant? = nil  
var session: MCSession!
```

We will also create a handler function for convenience. This function will be called when we receive data from our peers. This function is linked to another function that we will create in the `ViewController` class. This function will be called automatically when the handler function is called. These functions are linked using a closure.

Closures are self-contained blocks of code that can be passed around. Closures in Swift are similar to lambdas in other programming languages and blocks in C and Objective C.

Create the handler function, as shown in the following code:

```
let receivedDataHandler: (Data, MCPeerID) -> Void
```

These are all the properties that are required for this class. Let's next start adding the methods, starting with the `init`:

```
init(receivedDataHandler: @escaping (Data, MCPeerID) -> Void) {  
    self.receivedDataHandler = receivedDataHandler  
    super.init()  
  
    session = MCSession(peer: myPeerID, securityIdentity: nil,  
    encryptionPreference: .required)  
    session.delegate = self  
}
```

In `init`, we get the closure method passed in and assign into the `receivedDataHandler` function in the current class.

We then initialize the parent class and create a new session. In `init` for `MCSession`, we specify the current device name, set the security to nil, and set the encryption to required.

Next, we create the `advertiseSelf` function. This will be called when the advertiser wants to host a new session:

```
func advertiseSelf() {  
  
    advertiser = MCAdvertiserAssistant(serviceType:  
    MultipeerSession.serviceType, discoveryInfo: nil, session: session)  
  
    advertiser!.start()  
}
```

In the function, we create a new advertiser. The `MCAdvertiserAssistant` takes in a service type, which we have specified as a property. We set the `discoveryInfo` to nil and pass in the current session as the third parameter.

We create a function called `setupBrowser`. This will be called when the client wants to join an advertiser by browsing through the advertisers:

```
func setupBrowser() {  
  
    browser = MCBrowserViewController(serviceType:  
    MultipeerSession.serviceType, session: session )  
    browser.maximumNumberOfPeers = 1  
    browser.minimumNumberOfPeers = 1  
}
```

In it, we create a new `MCVBrowserViewController` by passing the service type and session as parameters. We also set the maximum and minimum number of peers to 1, as we want to connect to just one advertiser.

We also create two new functions to send information to the peers and get the total number of peers connected to the device:

```
func sendToAllPeers(_ data: Data) {  
    do {  
        try session.send(data, toPeers: session.connectedPeers,  
        with: .reliable) // TCP / UDP modes  
    } catch {  
        print("error sending data to peers: \(error.  
localizedDescription)")  
    }  
}
```

In a try catch statement, we use the `send` property of the session to send information to the connected peers. This takes three parameters. The first parameter is the data itself. The second parameter is the peers to whom the data needs to be sent, which in this case are all the connected peers. The third parameter specifies whether the delivery of data should be guaranteed or not. This can be `.reliable` or `.unreliable`. Since we want to make sure the data should be received by the peer, we use the `.reliable` keyword.

If there is an error when sending the data, in the `catch` statement, we print out the error.

We also add the function that will get the total number of peers:

```
var connectedPeers: [MCPeerID] {  
    return session.connectedPeers  
}
```

Finally, we will add `MCSessionDelegate`. This will give us information as to whether the peers are connected and if we received any data from other devices. `MCSessionDelegate` is an extension so it sits outside the `MutipeerSession` class:

```
extension MultipeerSession: MCSessionDelegate {  
}  
}
```

We will add the following functions to it. The first one is the session `DidChange` function, which is called when the state of a peer changes:

```
// When a user connects or disconnects from our session, the  
method  
func session(_ session: MCSession, peer peerID: MCPeerID,  
didChange state: MCSessionState) {  
  
    switch state {  
        case MCSessionState.connected:  
            print("Connected: \(peerID.displayName)")  
  
        case MCSessionState.connecting:  
            print("Connecting: \(peerID.displayName)")  
  
        case MCSessionState.notConnected:  
            print("Not Connected: \(peerID.displayName)")  
    }  
}
```

This gives information if the peer is connected, connecting, or not connected. We use the `MCSessionState` property to get the connection status of the peers and print it out:

```
func session(_ session: MCSession, didReceive data: Data, fromPeer  
peerID: MCPeerID) {  
  
    // - Received data  
  
    receivedDataHandler(data, peerID)  
}
```

Next is the session `didReceieve` function. This will be called when we receive data from other devices. In this, we call the `receieveDataHandlder` function. This will get the type of data that we have received.

Now there are three more functions that are required. Even if we will not be using these functions, we still have to add them; otherwise, we will get errors:

```
func session(_ session: MCSession, didReceive stream: InputStream,  
withName streamName: String, fromPeer peerID: MCPeerID) {  
    fatalError("This service does not send/receive streams.")  
}
```

The `didReceieveInputStream` function opens a byte stream connection to the local stream:

```
func session(_ session: MCSession,  
didStartReceivingResourceWithName resourceName: String, fromPeer  
peerID: MCPeerID, with progress: Progress) {  
    fatalError("This service does not send/receive resources.")  
}
```

Next is the `didStartReceivingResourceWithName` function. This is to indicate that we are receiving a resource from a nearby peer:

```
func session(_ session: MCSession,  
didFinishReceivingResourceWithName resourceName: String, fromPeer  
peerID: MCPeerID, at localURL: URL?, withError error: Error?) {  
    fatalError("This service does not send/receive resources.")  
}
```

Finally, we have the `didFinishReceivingResourceWithName` function, which will signal when we have finished receiving resource from the peer.

That's all for the `MultipeerSession` file.

Creating a UI for the app

We will now create some UI-like buttons for hosting and joining a session and a **Map** button to send the map once we have connected to a peer. We will also create a couple of text labels to tell us what the app is doing currently.

For sending the map, we will create a custom class so that the button lights up when the map is ready to be sent.

Create a new Swift file called `RoundedButton`. In it, create a new class of the same name and inherit from `UIButton`:

```
import UIKit

@IBDesignable
class RoundedButton: UIButton {

}
```

In this class, we will add the `init` function, which will, in turn, call the `setup()` function in which we will set the button parameters.

Also, override the `isEnabled` function, which, when set, will change `backgroundColor`; otherwise, it will be gray in color.

Add the following `init` function:

```
override init(frame: CGRect) {
    super.init(frame: frame)
    setup()
}
```

We also need the required `init` function, so add it, as shown in the following code:

```
required init?(coder aDecoder: NSCoder) {
    super.init(coder: aDecoder)
    setup()
}
```

Next, we create the `setup` function:

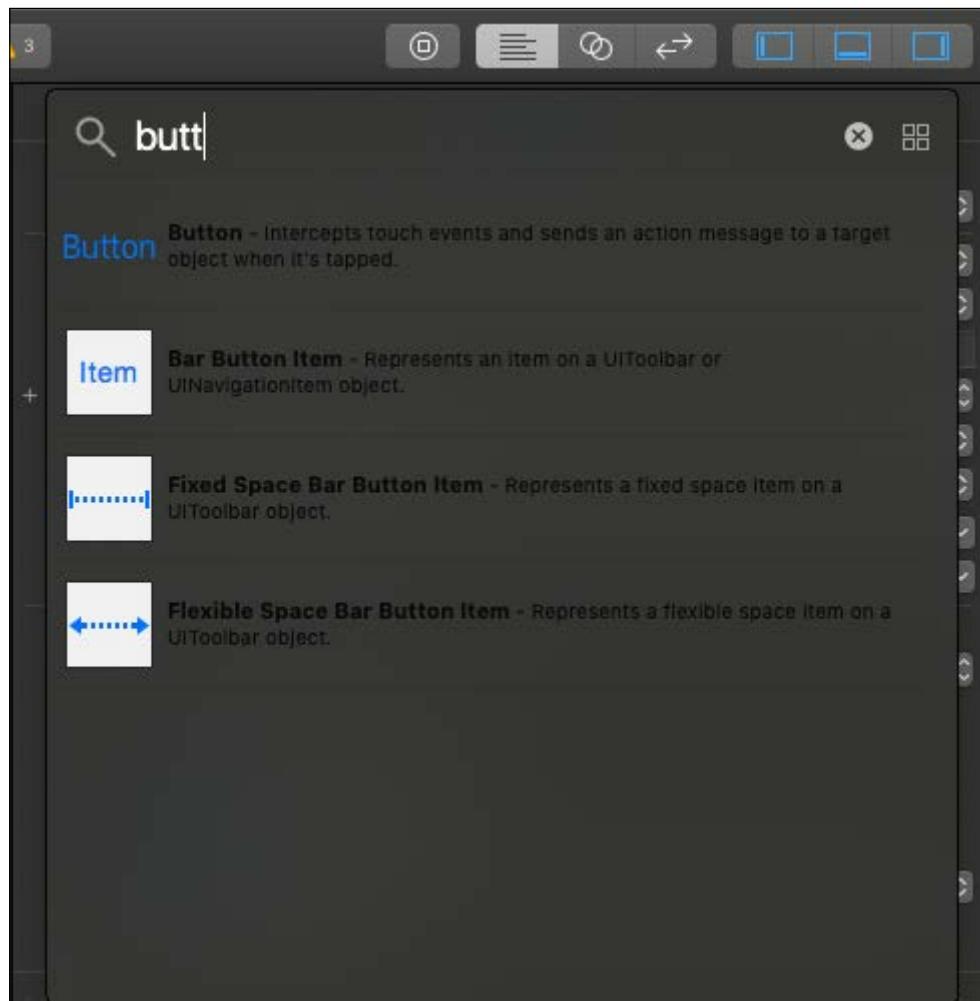
```
func setup() {
    backgroundColor = tintColor
    layer.cornerRadius = 8
    clipsToBounds = true
    setTitleColor(.white, for: [])
    titleLabel?.font = UIFont.boldSystemFont(ofSize: 17)
}
```

In this, we set the background color, set the radius, set the title color to white, and set the font size.

Next, we override the `isEnabled` variable. If enabled, we set the background to the `tintColor`; otherwise, we set it to gray:

```
override var isEnabled: Bool {  
    didSet {  
        backgroundColor = isEnabled ? tintColor : .gray  
    }  
}
```

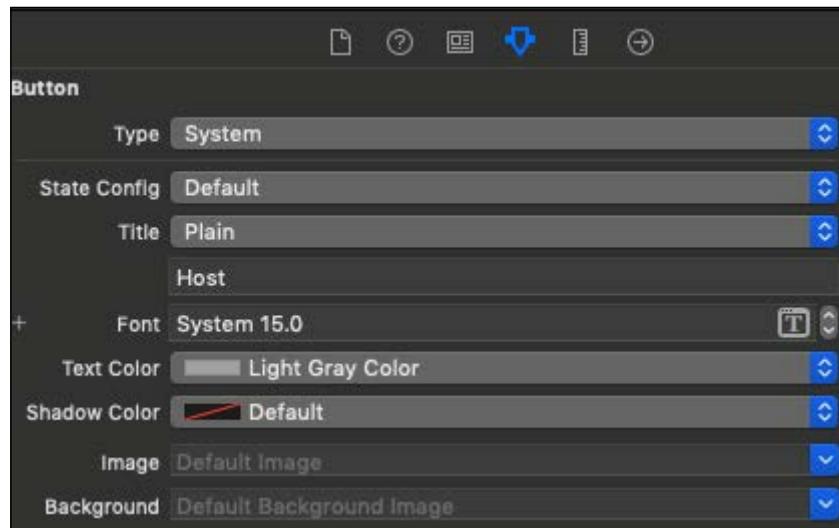
Next, to create the **Host** and **Join** buttons, click on the `Main.Storyboard` file in the project. In the Object icon on the top right of the Xcode window, select the button object. You can type in `button` in the search field to look for it faster:



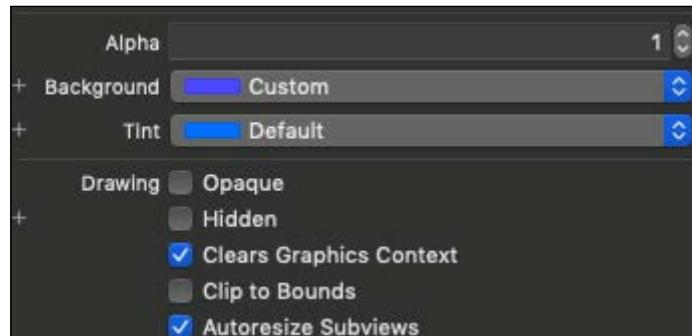
Now, click and drag the button into the view. Next, click it and drag it to a place where you want it and use the white handles on the edges to resize the button:



Next, under the attribute inspector, set the Host tile, as shown in the following screenshot:

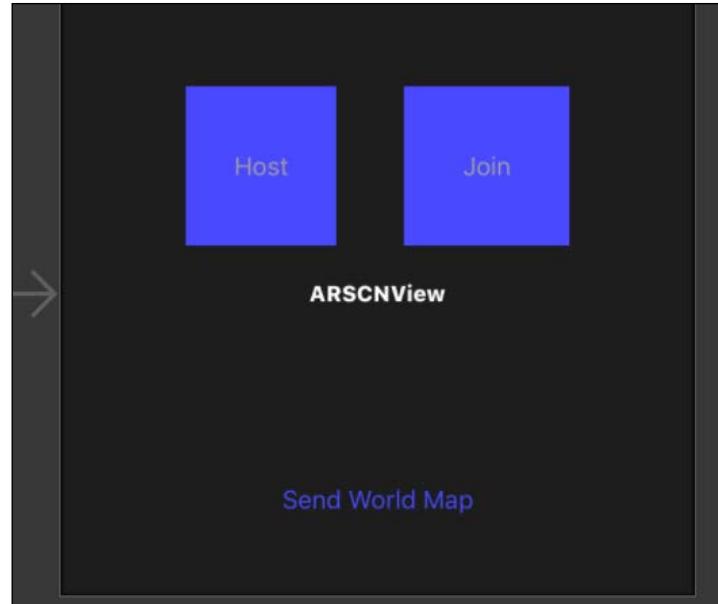


Change the color of the background under the **Background** section, as in the following screenshot:

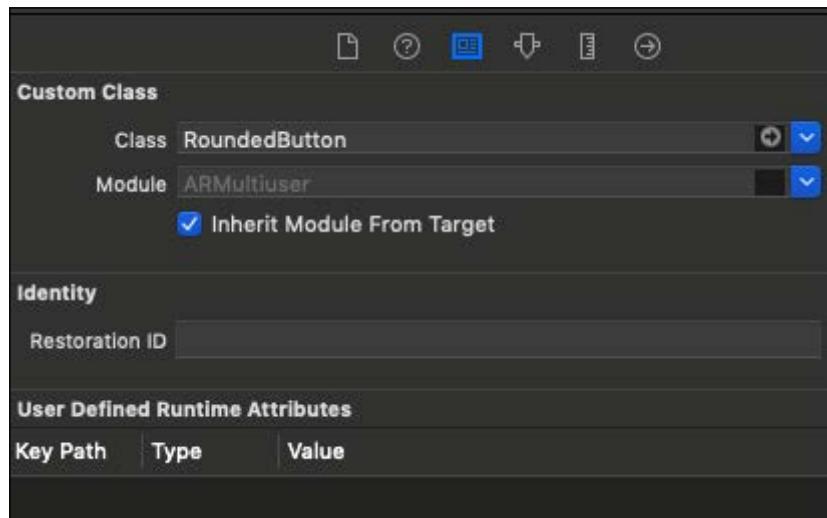


Now do the same for adding the Join button as well.

For the `sendmap` button, add a button similar to the Host and Join button but, in this case, make the background default so that it is transparent, as shown in the following screenshot:



Also in the identity inspector, select the class to be the `RoundedButton` class we created earlier and, for the module, check the **Inherit Module from Target** checkbox:

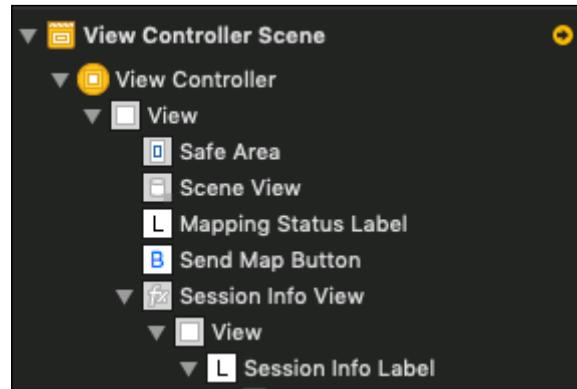


Next, we will add labels. Similar to how you added buttons, search for Label and place it above the **Send World Map** button. Rename it to **Mapping Status Label**. We don't have to initialize anything since we will do it in code.

To get a better idea of what the app is doing, we will add a Session Info text label that will sit on the top left of the window, which will give us further info about the app.

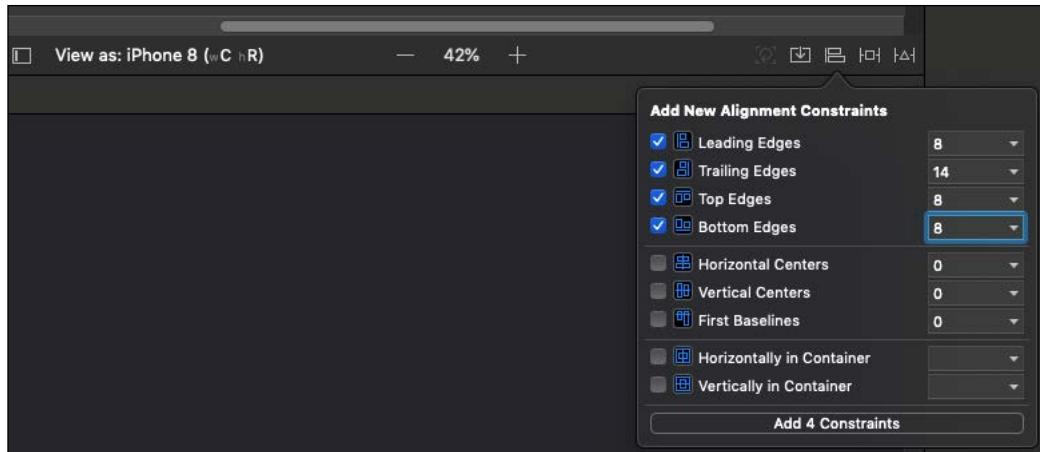
Click on the Objects button and add a new Visual Effects with Blur and drag and drop it into the window on the top left of the screen. Rename it to **Session Info View**. This already has a view. Next, create a new label inside the view. Rename it to **Session Info Label**. Go to the attributes panel and add the text `Initialize AR Session`.

The **View Controller Scene** hierarchy should look like the following screenshot:

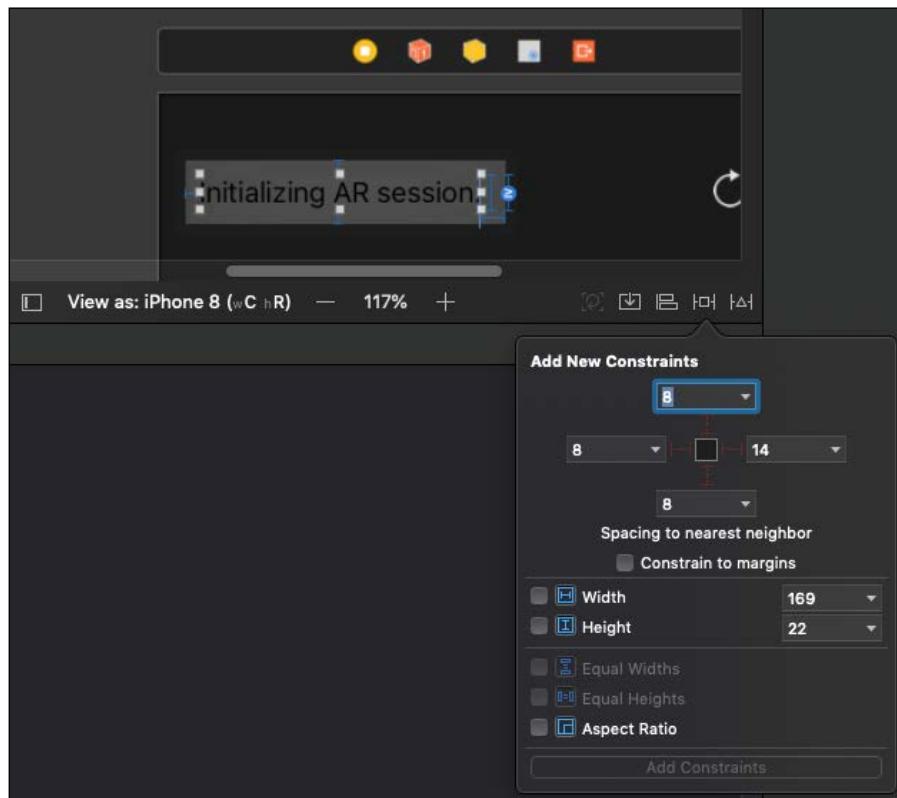


Next, we will add some constraints to the label so that it stays there if we go from portrait to landscape mode.

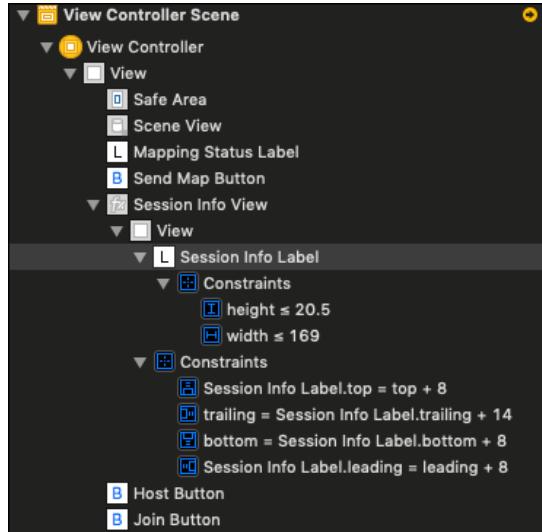
To do this, first we constrain the label within the view. Select the view and the label and click the align button and set the values for the leading, trailing, top, and bottom edges. Click on **Add 4 Constraints**:



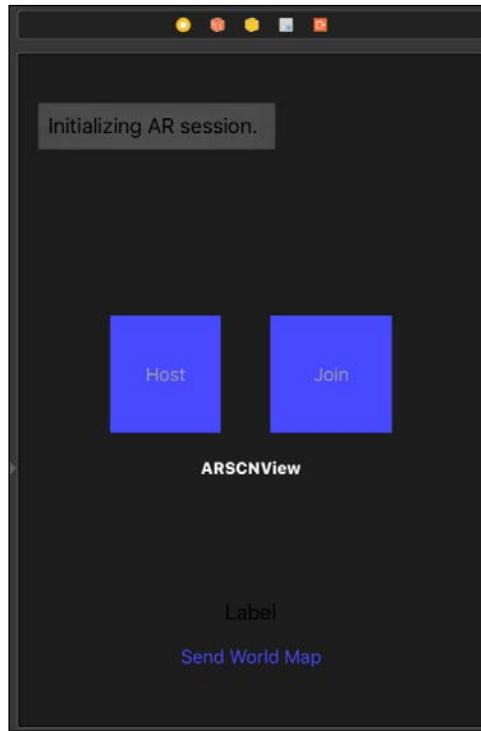
Next, select the label to constrain the width and height of the label itself. Click on the **Add New Constraints** button and type in the values. Select the Width and Height and click **Add 2 Constraints**:



The View Controller hierarchy should look like following screenshot:



The scene itself should look like following screenshot:



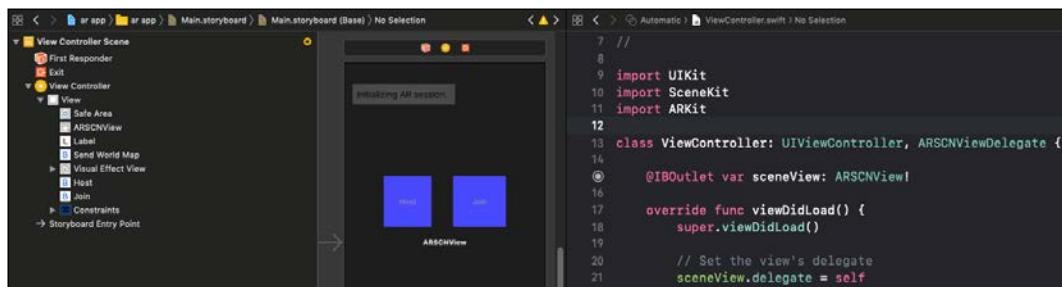
Setting outlets and adding variables

In the ViewController class, we will create the outlets for the buttons and labels and add the required variables for our program.

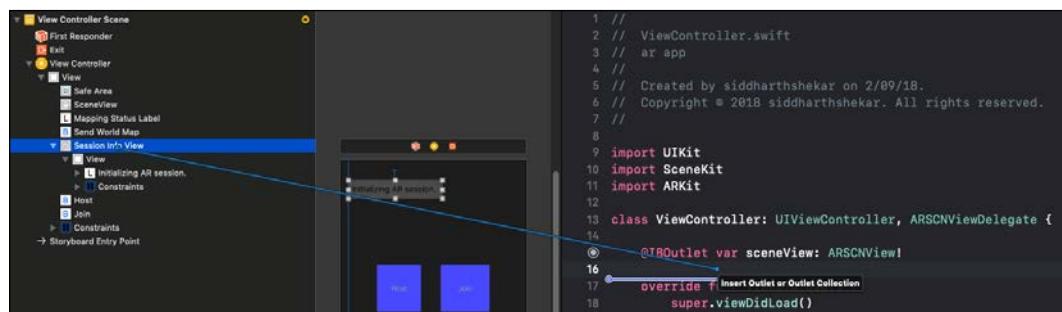
To get access to the buttons, views, and other objects we created in Main.

Storyboard, click on the **Assistant Editor** (middle of the three buttons on the top right with the two circles). This will open a second window. Now select Main.

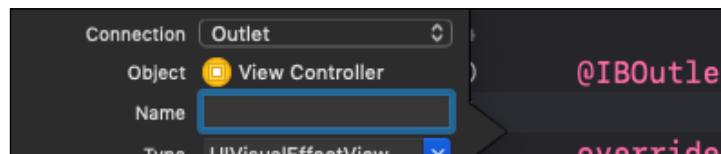
Storyboard and make sure the ViewController class is also visible to the right of the **Assistant Editor**, as shown in the following screenshot:



Now select SessionInfoView. While holding the Control key on the keyboard, left-mouse click and drag it into the ViewController.swift file, as shown in the following screenshot. Drag it to where you would like to create the variable in the file and release the left mouse and control buttons:



Next, a window will open up asking you to name the variable. Name it sessionInfoView (we will use this name to access the view in the code later on):

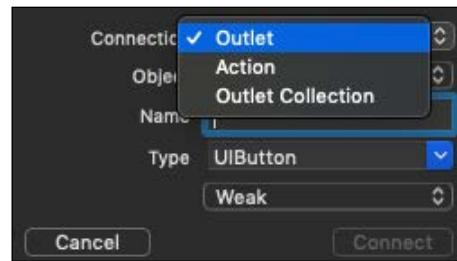


Similarly, create variables to access the two labels we created. One is the Mapping Status Label that is above the Send Map button, and the other is `SessionInfoLabel` in `SessionInfoView`.

Next, we will create outlets for the three buttons we have in the scene: the Host, Join, and Send Map buttons.

For labels and views, you can only get outlets, but for buttons, we can create actions. Actions are functions that are called when the buttons are pressed. For now, we will only create `IBOutlets` variables for the buttons.

After you click and drag to create the outlets in `ViewController.swift`, make sure to select **Outlet** from the **Connection** drop-down list:



After adding the buttons, we will need to add one last variable to access `TapRecogniser`, so that we can control when the Tap Gesture can be enabled.

In the `ViewController` hierarchy, create a new Tap Recognizer object and add an outlet for it in `ViewController.swift`.

That is it for button and labels.

Finally, we will add two variables. In the first, we will create an instance of the `MutipeerSession` class called `multiPeerSession`. The second is a Boolean called `isTrackingEnabled` and is set to false.

So, after adding all the outlets and the variables, the top of the `ViewController` class should look like this:

```
// session info view at the top left of the screen
@IBOutlet weak var sessionInfoView: UIView!

// scene view
@IBOutlet weak var sceneView: ARSCNView!
```

```
//text label inside the info view on top left
@IBOutlet weak var sessionInfoLabel: UILabel!

// text label at the bottom middle above the button
@IBOutlet weak var mappingStatusLabel: UILabel!

@IBOutlet weak var sendMapButton: UIButton!

@IBOutlet weak var hostButton: UIButton!

@IBOutlet weak var joinButton: UIButton!

@IBOutlet var tapRecogniser: UITapGestureRecognizer!

var mltipeerSession: MultipeerSession!

var isTrackingEnabled = false;
```

Initializing the view

Let's add some code now.

So, at the top, you have to import ARKit and MultipeerConnectivity as well, apart from UIKit and SceneKit:

```
import UIKit
import SceneKit
import ARKit
import MultipeerConnectivity
```

Next, delete all the content of the viewDidLoad function. We will add more functionality to it later.

Override the viewWillAppear function, as shown here:

```
override func viewWillAppear(_ animated: Bool) {
}
```

We will add the following code in the viewWillAppear function.

First, we will call the viewWillAppear of the parent class:

```
super.viewWillAppear(animated)
```

Next, we will set a guard to check if world tracking is supported by the current device. Otherwise, it will give an error saying that ARKit is not supported in the current device:

```
guard ARWorldTrackingConfiguration.isSupported else {
    fatalError("""
        ARKit is not available on this device.
    """)
}
```

We will disable the screen from getting dimmed, as the user will be taking some time to scan the world:

```
UIApplication.shared.isIdleTimerDisabled = true
```

We will round out the corners of the Host and Join buttons, disable tapRecognizer, and hide sendMapButton:

```
hostButton.layer.cornerRadius = 0.125 * hostButton.bounds.size.width
joinButton.layer.cornerRadius = 0.125 * joinButton.bounds.size.width

tapRecogniser.isEnabled = false

sendMapButton.isHidden = true;
```

That is it for the `viewDidAppear` function.

Next, we override the `viewDidDisappear` function, in which we just pause the session:

```
override func viewWillDisappear(_ animated: Bool) {
    super.viewWillDisappear(animated)

    // Pause the view's AR session.
    sceneView.session.pause()
}
```

Update session and tracking

Next, we will add the session delegate functions. There are two delegate functions in total:

`didUpdate`: This is the update function that gets called every frame and checks and tells us if any of the properties of any of the anchors got changed and updates them if required.

We will check if the world is mapped or not. If it is mapped, we will enable the `sendMap` button so that the user can send the map to the peers. We will also update `mappingStatusLabel` and `SessionInfoLabel`.

- `cameraDidChangeTrackingState`: This function gets called when the tracking of the game camera sees a change in the state. There are different values that the tracking state can have:
 - `notAvailable`: This means tracking is not available.
 - `limited`: This means that it is hard to track. It could be because the camera is moved very fast or there are fewer features/less details to track the feature points.
 - `normal`: This means it is a suitable condition to track.

Let's add these delegates to the class. First, we add `sessionDidUpdate`, as shown in the following code:

```
func session(_ session: ARSession, didUpdate frame: ARFrame) {  
  
    // Checks mapping status depending on whether  
    // if the environment is mapped or not  
    // Greys out or enables the button.  
  
    if isTrackingEnabled {  
  
        switch frame.worldMappingStatus {  
  
            case .notAvailable, .limited:  
                sendMapButton.isEnabled = false  
            case .extending:  
                sendMapButton.isEnabled = false  
  
            case .mapped:  
  
                if (!multipeerSession.connectedPeers.isEmpty) {  
  
                    sendMapButton.isEnabled = true;  
  
                    tapRecogniser.isEnabled = true  
                }  
        }  
    }  
}
```

```
        isTrackingEnabled = false
    }
}

//-- Mapping status label above the mapping button
//-- Description is in utilities

mappingStatusLabel.text = frame.worldMappingStatus.
description

//-- Session info label
//-- On the top left of the screen
updateSessionInfoLabel(for: frame, trackingState: frame.
camera.trackingState)

}

}
```

Then we check if tracking is enabled. Only the host will be tracking so that the map data can be sent to the client.

With the `worldMappingStatus` property of the frame, we check if the area has been mapped properly. We don't do anything if the mapping is limited or the current map is being extended.

Once the area has been mapped, we check if there are connected peers. If there are, then we enable `sendMapButton` and `tapRecognizer` on the host and disable tracking.

We also update `mappingStatusLabel` and `sessionInfoLabel`.

For updating the string value of `mappingStatusLabel`, we create an extension of `WorldMappingStatus` of `ARFrame` to convert the description to a string. So add the following code for extending `WorldMappingStatus` outside of the `ViewController` class but add it to the bottom of the `ViewController.swift` file:

```
extension ARFrame.WorldMappingStatus: CustomStringConvertible {
    public var description: String {
        switch self {
        case .notAvailable:
            return "Not Available"
        case .limited:
            return "Limited"
        case .extending:
            return "Extending"
        }
    }
}
```

```

        return "Extending"
    case .mapped:
        return "Mapped"

    }
}
}

```

Next, we add a convenience/helper function at the bottom of the `ViewController` class called `updateSessionInfoLabel`. This takes in an `ARFrame` and tracking state, as shown in the following code:

```

// MARK: - Helper Functions

private func updateSessionInfoLabel(for frame: ARFrame,
trackingState: ARCamera.TrackingState) {

    // tracking status

    // Update the UI to provide feedback on the state of the AR
experience.

    let message: String

    switch trackingState {

        case .normal where frame.anchors.isEmpty && multipeerSession.
connectedPeers.isEmpty:
            // No planes detected; provide instructions for this app's
AR interactions.
            message = "Move around to map the environment, or wait to
join a shared session."

        case .normal where !multipeerSession.connectedPeers.isEmpty &&
mapProvider == nil:
            let peerNames = multipeerSession.connectedPeers.map({
$0.displayName }).joined(separator: ", ")
            message = "Connected with \(peerNames)."

        case .notAvailable:
            message = "Tracking unavailable."

        case .limited(.excessiveMotion):
            message = "Tracking limited - Move the device more
slowly."
    }
}

```

```
        case .limited(.insufficientFeatures):
            message = "Tracking limited - Point the device at an area
with visible surface detail, or improve lighting conditions."

        case .limited(.initializing) where mapProvider != nil,
            .limited(.relocalizing) where mapProvider != nil:
            message = "Received map from \$(mapProvider!.displayName)."

        case .limited(.relocalizing):
            message = "Resuming session - move to where you were when
the session was interrupted."

        case .limited(.initializing):
            message = "Initializing AR session."

        default:
            // No feedback needed when tracking is normal and planes
are visible.
            // (Nor when in unreachable limited-tracking states.)
            message = ""

    }

    sessionInfoLabel.text = message
    sessionInfoView.isHidden = message.isEmpty
}
```

This, as mentioned, will update `SessionInfoLabel` with the current tracking detail, ascertaining whether tracking is normal, not available, or limited. Note that if the status is normal and there are peers connected, it will show the name of the connected peer.

At the end of the function, it will show the message to the label. If the message is empty, then it will hide `sessionInfoView` itself.

We will call this function in `cameraDidChangeTrackingState` to update `sessionInfoLabel` when the tracking changes:

```
func session(_ session: ARSession, cameraDidChangeTrackingState  
camera: ARCamera) {  
  
    updateSessionInfoLabel(for: session.currentFrame!,  
    trackingState: camera.trackingState)  
  
}
```

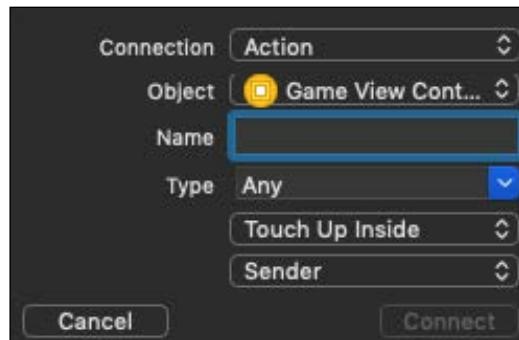
There are also session observers that track the health of the session itself to check if the session was interrupted, ended, failed, or relocalized. We won't be adding any functionality here for this app but the functions are added for your reference:

```
// MARK: - ARSessionObserver  
  
func sessionWasInterrupted(_ session: ARSession) {  
    // Inform the user that the session has been interrupted, for  
    // example, by presenting an overlay.  
    sessionInfoLabel.text = "Session was interrupted"  
}  
  
func sessionInterruptionEnded(_ session: ARSession) {  
    // Reset tracking and/or remove existing anchors if consistent  
    // tracking is required.  
    sessionInfoLabel.text = "Session interruption ended"  
}  
  
func session(_ session: ARSession, didFailWithError error: Error)  
{  
    // Present an error message to the user.  
    sessionInfoLabel.text = "Session failed: \(error.  
localizedDescription)"  
}  
  
func sessionShouldAttemptRelocalization(_ session: ARSession) ->  
Bool {  
    return true  
}
```

Hosting and joining the session

Next, we will add functions to the host and join the session.

For buttons, adding the functions or actions is similar to adding the outlets. Open the **Assistant Editor** once again, click the Control button on the keyboard, left-mouse click, and drag the button to the ViewController class. After selecting where in the code you want the function to be created, release the keyboard and mouse button. Next, in the window, instead of selecting **Outlet** from the connection dropdown, select **Action**:



Now you can name the function. For the Host button, I have named the function `hostSession`. For the Join button, it is `joinSession`, and the `sendMapButton` is called `shareSession`.

Let's add functionality to the `hostSession` function, as shown in the following code:

```
// MARK: - Buttons and Touches

@IBAction func hostSession(_ sender: Any) {

    // Start the view's AR session.
    let configuration = ARWorldTrackingConfiguration()
    configuration.planeDetection = .horizontal
    sceneView.session.run(configuration)

    // Set a delegate to track the number of plane anchors for providing
    // UI feedback.

    sceneView.session.delegate = self

    sceneView.debugOptions = [ARSCNDebugOptions.showFeaturePoints]
```

```
        multipeerSession.advertiseSelf()

        hostButton.isHidden = true
        joinButton.isHidden = true;

        isTrackingEnabled = true;

        sendMapButton.isHidden = false;

    }
```

In this, we run the AR session on the host device. We first have to set the configuration. As in the previous AR projects, we create a new `ARWorldTrackingConfiguration` variable, set plane detection to horizontal, and set `sceneView.session.run` to the configuration.

Next, we have to set the session delegate to the current `sceneView`.

We set `DebugOptions` for `sceneView` so that we can see the feature points while getting the data for the world map.

Make sure the `ViewController` class is inheriting from `ARSessionDelegate`:

```
class ViewController: UIViewController, ARSCNViewDelegate,
ARSessionDelegate{
```

Next, call the `advertiseSelf` function from the `multipeerSession` class. Hide the Host and Join buttons so that they are not visible. Enable tracking and unhide the sendmap button.

Next, we will add the functionality for the client/browser to join the session:

```
@IBAction func joinSession(_ sender: Any) {

    if multipeerSession.session != nil{

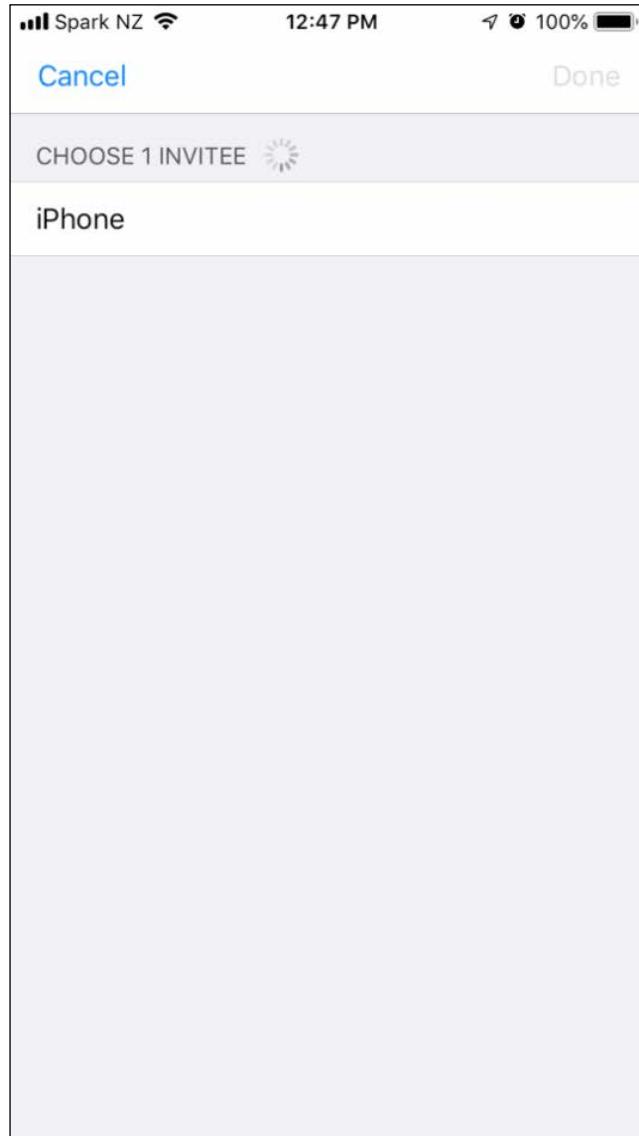
        multipeerSession.setupBrowser()
        multipeerSession.browser.delegate = self

        self.present(multipeerSession.browser, animated: true,
completion: nil)
    }
}
```

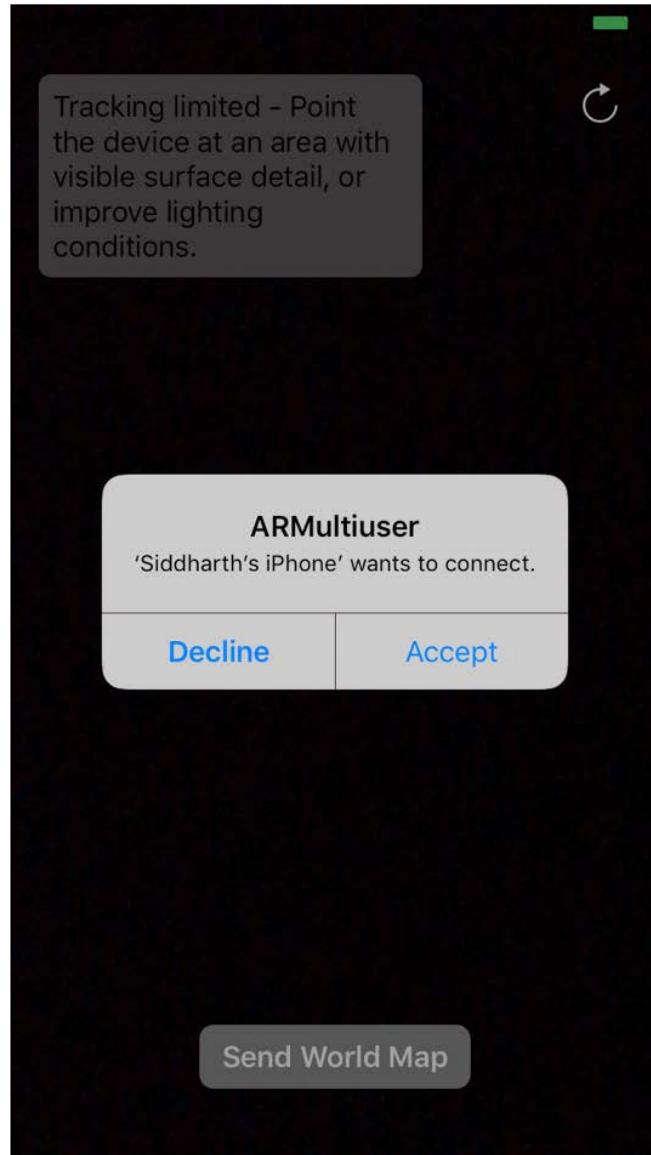
Multipeer Augmented Reality

If sessions are nil, then we call the `setupBrowser` function in the `multipeerSession` class and set `browserDelegate` to the current class and present `multipeerSessionView`.

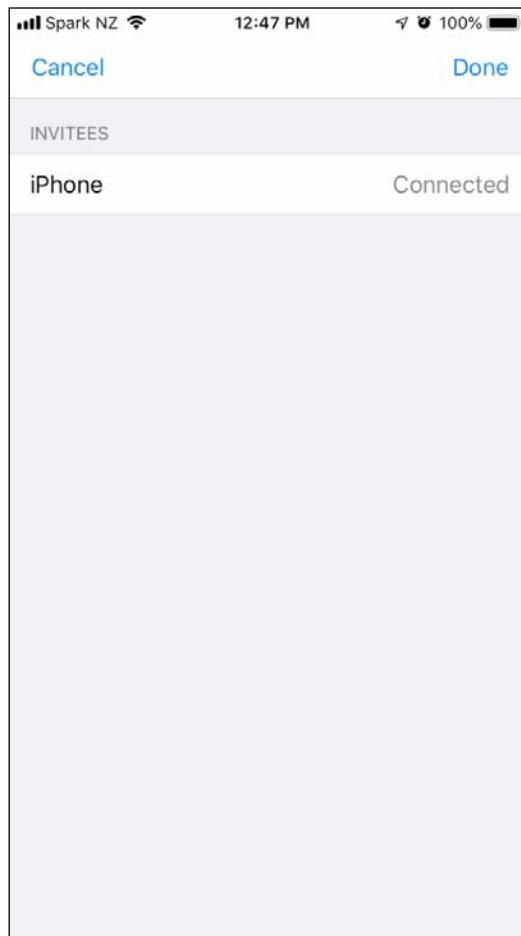
Once the browser clicks the Join button, a new view will open up showing all the hosts that are available to join. The `present` function shows this view, as shown in the following screenshot:



On the advertiser's side, the host phone will show up. When the advertiser selects the device to join by tapping on the name of the phone, on the host device a popup will open and he will have the option to connect or not with the device:



Once the host accepts the confirmation, then on the client's/browser's side, it will show it is connected to the device, and you can press the **Done** button on the top-right of the view:



The browser delegate lets us handle what happens if we press the **Done** or **Cancel** buttons at the top of the browser view.

We want the Host and Join buttons to be hidden when the user presses the **Done** button, so we first of all make sure `ViewController` inherits from `MCBrowserViewControllerDelegate`, as shown in the following code:

```
class ViewController: UIViewController, ARSCNViewDelegate,  
ARSessionDelegate, MCBrowserViewControllerDelegate {
```

And add the delegate functions that are required, as shown in the following code:

```
func browserViewControllerDidFinish(_ browserViewController:  
MCBrowserViewController) {  
  
    multipeerSession.browser.dismiss(animated: true, completion: {  
() -> Void in  
  
        print(" pressed done ")  
  
        self.hostButton.isHidden = true  
        self.joinButton.isHidden = true;  
  
    })  
}  
  
func browserViewControllerWasCancelled(_ browserViewController:  
MCBrowserViewController) {  
    multipeerSession.browser.dismiss(animated: true, completion:  
nil)  
}
```

We don't want to do anything when **Cancel** is pressed, but we want the Host and Join buttons to hide when **Done** is pressed. So, we add the lines of code in, as shown in the preceding code.

Sending and receiving data

Next, let's send the map to the client when the `SendMap` button is pressed. Create an `Action` function and add the following code:

```
@IBAction func shareSession(_ button: UIButton) {  
  
    Attempt To Get The World Map From Our ARSession  
  
    sceneView.session.getCurrentWorldMap { worldMap, error in  
  
        guard let map = worldMap  
        else { print("Error: \(error!.localizedDescription)");  
return }  
    }
```

```
We Have A Valid ARWorldMap So Send It To Any Peers
guard let data = try? NSKeyedArchiver.
archivedData(withRootObject: map, requiringSecureCoding: true)
else { fatalError("can't encode map") }

self.multipeerSession.sendToAllPeers(data)

self.sendMapButton.isHidden = true

}// get world map function
}
```

When the button is pressed, we get the world map using the `getCurrentWorldMap` function of the session. We make sure that we have the world map and then we encode the map data using the `archivedData` function of `NSKeyedArchiver` and store it in a variable called `data`.

Then, using the `sendToAllPeers` function in the `MultipeerSession` class, we send the data to all the peers. Once the data is sent, we can hide `sendMapButton`.

Once the map is sent, the host will have access to `tapGesture`. With it, we will place the player anchor in the scene. Create an action for the `TapRecognizer` as well, as shown in the following code:

```
@IBAction func handleSceneTap(_ sender: UITapGestureRecognizer) {

// -- Hit test to find a place for a virtual object.

guard let hitTestResult = sceneView
.hitTest(sender.location(in: sceneView), types:
[.existingPlaneUsingGeometry, .estimatedHorizontalPlane])
.first
else { return }

// -- Create Our Anchor & Add It To The Scene

let anchor = ARAnchor(name: "hero", transform: hitTestResult.
worldTransform)
sceneView.session.add(anchor: anchor)

// -- Send the anchor info to peers

guard let data = try? NSKeyedArchiver.
archivedData(withRootObject: anchor, requiringSecureCoding: true)
else { fatalError("can't encode anchor") }
self.multipeerSession.sendToAllPeers(data)
}
```

This is similar to how we have placed anchors in the previous ARKit project. We call `hitTest` on the current scene and get the location. We check against the horizontal plane and store the result in `hitTestResult`.

Then we create a new anchor called `hero` and then the transforms. The new anchor is added to the scene.

We also encode the anchor data and send it to the peers. Take note here that we are sending a different kind of data to the peers. Previously we sent the world data, and here we are sending the anchor data.

However, doing this alone won't add the anchor to the scene. We have to use the `rendererDidAdd` function of `ARSCNViewDelegate` to add it to the scene. So add the function next:

```
// MARK: - ARSCNViewDelegate

func renderer(_ renderer: SCNSceneRenderer, didAdd node: SCNNode,
for anchor: ARAnchor) {

    if let name = anchor.name, name.hasPrefix("hero") {

        node.addChildNode(loadPlayerModel())
    }
}
```

In the function, it checks the name of the anchor added. If it is `hero`, then the `loadPlayerModel` function is called, which returns the animated hero.

This is similar to how we loaded the hero node in the previous ARKit and SceneKit projects, except here, instead of loading it from a class, we are just loading it through a function in the current class:

```
// MARK: - Load player model

static func time(atFrame frame:Int, fps:Double = 30) ->
TimeInterval {
    return TimeInterval(frame) / fps
}
static func timeRange(forStartingAtFrame start:Int,
endingAtFrame end:Int, fps:Double = 30) -> (offset:TimeInterval,
duration:TimeInterval) {
    let startTime    = self.time(atFrame: start, fps: fps)
    let endTime     = self.time(atFrame: end, fps: fps)
```

```
        return (offset:startTime, duration:endTime - startTime)
    }

    static func animation(from full:CAAnimation, startingAtFrame
start:Int, endingAtFrame end:Int, fps:Double = 30) -> CAAnimation {
        let range = self.timeRange(forStartingAtFrame: start,
endingAtFrame: end, fps: fps)
        let animation = CAAnimationGroup()
        let sub = full.copy() as! CAAnimation
        sub.timeOffset = range.offset
        animation.animations = [sub]
        animation.duration = range.duration
        return animation
    }

    // load monster model
    private func loadPlayerModel() -> SCNNNode {

        // -- Load the monster from the collada scene

        let tempNode:SCNNNode = SCNNNode()
        let monsterScene:SCNScene = SCNScene(named: "Assets.scnassets/
theDude.DAE")!
        let referenceNode = monsterScene.rootNode.childNode(withName:
"CATRigHub001", recursively: false)! //CATRigHub001
        tempNode.addChildNode(referenceNode)

        // -- Set the anchor point to the center of the character
        let (minVec, maxVec) = tempNode.boundingBox
        let bound = SCNVector3(x: maxVec.x - minVec.x, y: maxVec.y -
minVec.y, z: maxVec.z - minVec.z)
        tempNode.pivot = SCNMatrix4MakeTranslation(bound.x * 1.1, 0 ,
0)

        // -- Set the scale and name of the current class
        tempNode.scale = SCNVector3(0.1/100.0, 0.1/100.0, 0.1/100.0)

        // -- Get the animation keys and store it in the anims
        let animKeys = referenceNode.animationKeys.first
        let animPlayer = referenceNode.animationPlayer(forKey:
animKeys!)
```

```
let anims = CAAnimation(scnAnimation: (animPlayer?.  
animation)!)  
  
        // -- Get the run animation from the animations  
        let runAnimation = ViewController.animation(from: anims,  
startingAtFrame: 31, endingAtFrame: 50)  
        runAnimation.repeatCount = .greatestFiniteMagnitude  
        runAnimation.fadeInDuration = 0.0  
        runAnimation.fadeOutDuration = 0.0  
  
        // -- Remove all the animations from the character  
        referenceNode.removeAllAnimations()  
  
        // -- Set the run animation to the player  
        let runPlayer = SCNAccelerationPlayer(animation:  
SCNAcceleration(caAnimation: runAnimation))  
        tempNode.addAnimationPlayer(runPlayer, forKey: "run")  
  
        // -- Play the run animation at start  
        tempNode.animationPlayer(forKey: "run")?.play()  
  
        return tempNode  
    }  

```

Don't forget to add the `theDude.dae` and `theDude.png` files in the `Assets.scnassets` folder.

Now we are sending the map and the hero to the peer, but we still haven't added the functionality to receive the data from others. We will finally create the `receivedData` function, which itself will be received when we initialize the `MultipeerSession` class:

```
// MARK: - Received Data Handler Functions  
  
var mapProvider: MCPeerID?  
  
func receivedData(_ data: Data, from peer: MCPeerID) {  
  
    if let unarchivedMap = try? NSKeyedUnarchiver.  
unarchivedObject(ofClasses: [ARWorldMap.classForKeyedUnarchiver()],  
from: data),
```

```
// -- 1. Try To UnArchive Our Data As An ARWorldMap

    let worldMap = unarchivedMap as? ARWorldMap {

        // Run the session with the received world map.
        let configuration = ARWorldTrackingConfiguration()
        configuration.planeDetection = .horizontal
        configuration.initialWorldMap = worldMap
        sceneView.session.run(configuration, options:
            [.resetTracking, .removeExistingAnchors])

        // Remember who provided the map for showing UI
        feedback.
        mapProvider = peer

        if(!multipeerSession.connectedPeers.isEmpty) {

            // Enable tap recognizer on client
            tapRecogniser.isEnabled = true
        }

    }

// -- 2. Try To Unarchive Our Data As An ARAnchor

else if let unarchivedAnchor = try? NSKeyedUnarchiver.unarchivedObject(ofClasses: [ARAnchor.classForKeyedUnarchiver()], from: data),

    let anchor = unarchivedAnchor as? ARAnchor {

        // Add anchor to the session, ARSCNView delegate adds
        visible content.
        sceneView.session.add(anchor: anchor)

    }

// -- 3. Unknown data type or cant decode data
else {
    print("unknown data received from \\\(peer) ")
}

}
```

When the peer receives the data, it has to be first deciphered to the kind that is being received. Is it world map information or is it anchor information that is being received?

If the type is of `ARWorldMap`, then we get the world map and set the configuration. We set the world map to the one received and run the session. We store the peer information from whom we received the information. Since the map is received by the client, we also enable `tapRecognizer` for the client as well.

Otherwise, if the type of data sent is the anchor type, we unarchive that and add it to the current scene.

If the data type is unknown, then we just print out saying that the data is of an unknown type.

Initializing a multipeer session

Finally, in the `viewDidLoad` function, initialize `MutipeerSession`:

```
override func viewDidLoad() {  
    super.viewDidLoad()  
  
    multipeerSession = MultipeerSession(receivedDataHandler:  
    receivedData)  
  
}
```

Testing the application

That's it! Now we can test the application.

If you have two devices, then you can test this out. Host the app on one device. On the other device, click **Join**, and this will open a new view in which you can select the **Host**. Click on it. The host will get a notification telling you to **Accept** or **Decline** the connection. When they are connected, it will show **Connected** on the client device. Click **Done**.

The client now waits for the host to send the map information. The client screen will be black until it receives the map information, so don't worry.

Now the host can map the surrounding. When the **Send Map** button is hit, it can send the data to the client.

The client device will take a second to receive the data, and the camera will have to refocus.

Now the host can tap the screen to create an anchor on his device and the anchor should show up on the client device. Also, the client can tap on the screen to create an anchor on his device and this will show up automatically on the host device.

Make sure that the room is well lit and the surface on which you want to place the anchor is not smooth and is textured to have more feature points for better tracking.

Summary

In this chapter, we saw how to create a Multipeer augmented reality app. We saw how to create a session, connect a host with a client, and send and receive data between them. With this information, you should be able to create a basic Multipeer AR application. With the previous knowledge of how to create AR applications using ARKit, you could also try making a Multipeer AR game.

Over the course of the book, we have learned how to use SpriteKit and SceneKit to create 2D and 3D games. We also saw how to integrate ARKit to create 2D and 3D ARKit games. We also added achievements and created in-app purchases and ads to add monetization to the app. We also saw how to publish the app on the store and tell the world about our newest creation.

I hope you learned a lot over the course of the book and found it useful. Most of all, I hope you enjoyed learning what was covered.

I wish you all the very best!

Other Books You May Enjoy

If you enjoyed this book, you may be interested in these other books by Packt:



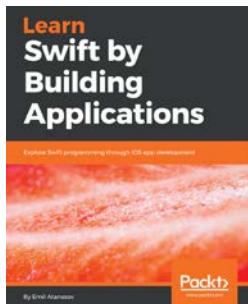
Hands-On Full-Stack Development with Swift

Ankur Patel

ISBN: 978-1-78862-524-1

- Get accustomed to server-side programming as well as the Vapor framework
- Learn how to build a RESTful API
- Make network requests from your app and handle error states when a network request fails
- Deploy your app to Heroku using the CLI command
- Write a test for the Vapor backend
- Create a tvOS version of your shopping list app and explore code-sharing with an iOS platform
- Add registration and authentication so that users can have their own shopping lists

Other Books You May Enjoy —————



Learn Swift by Building Applications

Emil Atanasov

ISBN: 978-1-78646-392-0

- Become a pro at iOS development by creating simple-to-complex iOS mobile applications
- Master Playgrounds, a unique and intuitive approach to teaching Xcode
- Tackle the basics, including variables, if clauses, functions, loops and structures, classes, and inheritance
- Model real-world objects in Swift and have an in-depth understanding of the data structures used, along with OOP concepts and protocols
- Use CocoaPods, an open source Swift package manager to ease your everyday developer requirements
- Develop a wide range of apps, from a simple weather app to an Instagram-like social app
- Get ahead in the industry by learning how to use third-party libraries efficiently in your apps

Leave a review – let other readers know what you think

Please share your thoughts on this book with others by leaving a review on the site that you bought it from. If you purchased the book from Amazon, please leave us an honest review on this book's Amazon page. This is vital so that other potential readers can see and use your unbiased opinion to make purchasing decisions, we can understand what our customers think about our products, and our authors can see your feedback on the title that they have worked with Packt to create. It will only take a few minutes of your time, but is valuable to other potential customers, our authors, and Packt. Thank you!

Index

A

achievement

creating, in iTunes Connect 283-285
updating, from code 285-287

AdMob

reference 242

AdMob Ad

adding, in app 242-249

ads, in games

downsides 242
upsides 241

anchor points

aligning with 27, 28

animations

damage animation 127-129
game over animation 129, 130
preparing 25
sequencing 26

app

Bundle ID, creating for 348, 349
creating, in itunesconnect portal 352-358
uploading, from Xcode 358-360

App Store 1

ARKit 2, 289

ARKit project

basics 311-315

AR Spritekit project

anchors, adding at random
location 299-302
creating 290-296
crosshair, adding 297-299
custom sprite, adding 302-306
requisites 289
text, adding 297-299
touch controls, registering 307, 308

art

collecting, into texture atlases 34, 35
organizing, into texture atlases 33

art assets

adding 82, 83
locating 82, 83

Assets.xcassets

exploring 33

Automatic Reference Counting (ARC) 3

AVAudio

music, adding with 165, 166

B

background music

adding, to game 166, 167

Bat class

adding 87, 88

bee image

adding, to project 30

Beekeeper 64

bee node

updating 35

Bees, bees everywhere! project

collision, creating 60, 61
GameScene class, renovating 54, 55
Ground class, adding 47
ground, solidifying 56
ground texture, adding to
Assets.xcassets 47
ground, wiring up 49
physics simulation 55
player's character, adding 50-53
texture atlas, adding for ground art 46

bitmasks 120

Blade class
adding 88-90

bronze coins
turning, to gold 114

Bundle ID
creating, for app 348, 349

buttons
adding 217, 218

C

camera
adding, to scene 204, 205
centering, on sprite 37-41
improving 73-76

categories, assigning to game objects
about 120
coins 122
enemies 122
ground 121
player 121
Power-up Star 121

category masks
using, in Swift 119, 120

character animation
adding 233-235

circle particle asset
adding 147

classes
game objects, organizing into 44-46

coin classes
creating 90, 91

coin collection
implementing 131, 132

coins
adding 90

coin sound effect
adding, to Coin class 167

collision
about 117, 118
versus contact 118

collision detection
adding 212-215

console output
viewing 124

contact 118

contact code
testing 124, 125

contact events
adding, to game 120
GameScene, preparing for 122, 123

control scheme
implementing, for penguin 70, 71

Core Motion
using 66

Core Motion code
implementing 66, 67

crate contact events
wiring up 189-191

Crate particle effects
creating 182-186

crates
adding, to smash open 180-182

crowdfunding
cons 240
leveraging 240
pros 240

custom classes
using, in scene editor 99

D

damage animation 127-129

data
receiving 395-397
sending 395-397

demo code
examining 16

development environment
setting up 7

display ads
displaying, for revenue 241

E

emitter nodes
recycling, with particle pools 186-189

EncounterManager
wiring up, in GameScene class 111

EncounterManager class
updating 108

encounters
building 107, 108

building, for Pierre Penguin 99, 100
creating 100-103
endless encounters
spawning 107
enemy
adding 212-215
external 3D application
scene, importing from 206, 207

F

fixed joint 58
floor
adding, to scene 205
foreign markets
localization 268, 269

G

game
scenes, integrating into 104-106
Game Center
account, authenticating 271-276
interacting with 5
opening, in game 276-280
game code
structuring 5
game logic
level data, separating from 99
gameloop
about 19
adding 219, 220
contact, checking between hero and enemy 221-223
Game Objects
adding 324
enemy, adding 335-338
ground node, adding 326, 327
hero, adding 327-334
light source, adding 324, 325
organizing, into classes 44-46
plane detection, stopping 324
testing 92, 93
game over animation 129, 130
GameOver condition
setting 221
Gameover text
adding 339-342

GameScene
preparing, for contact events 122, 123
touches 71
touchesBegan, implementing 68
wiring up, for game over 163
GameScene class

EncounterManager, wiring up 111
informing, on player's death 163, 164

Google Mobile Ads SDK

download link 246

gravity
about 55
fine-tuning 72
ground
looping 77, 78
moving 64

H

heads-up display (HUD)
adding 136
extending 162
implementing 137-141
health logic
implementing 125-127
health power-up crate
creating 191-193
Hero class
creating 208-211
hurt sound effects
adding, to Player class 168

I

images
loading, with SKSpriteNode 30, 31
in-app purchases
adding 252-268
selling 250
strategies 251
inbetweening 25
IndieGoGo 240
iTunes Connect
achievement, creating in 283-285
itunesconnect portal
app, creating 352-358

K

Kickstarter 240

L

labels

adding 217, 218

leaderboard

updating, from code 281-283

level data

separating, from game logic 99

levels

building 5

designing, with SpriteKit scene editor 98

light sources

adding, to scene 202, 203

limit joint 58

localization

in foreign markets 268, 269

M

MadFly class

adding 86, 87

main menu

building 157

creating 158, 159

launching, on game start 160, 161

marketing

checklist 238, 239

starting 238

marketing plan

developing 238

menu nodes

creating 158, 159

menus

building 5

Menu Scene

Options, adding to 172-178

metadata

storing, in SKSpriteNodeuserData

property 109, 110

movement 25

Multipeer Connectivity Framework

advertiser assistant objects 366

advertiser objects 366

browser objects 366

browser view controller object 367

overview 366

peers ID 366

session objects 366

multipeer session

application, testing 401

initializing 401

multipeer session class

creating 367-372

music

adding, with AVAudio 165, 166

mute button

adding 169-171

N

never-ending level

creating 94, 95

nodes

moving, with physics bodies 61

O

Objective-C 1

objects

adding, to scene 201

Options

adding, to Menu Scene 172-178

outlets

setting 381, 382

P

Parallax background layers

about 141

background assets, adding 142

background class, implementing 142

backgrounds, wiring up in GameScene

class 144, 145

parallax effect

adding 224-231

particle emitter

adding, to game 154, 155

particle pools

emitter nodes, recycling with 186-189

particles

adding 232

particle system
using 146

path particle settings
acceleration 153
alpha 153
angle 152
background 151
color blend 153
color ramp 154
configuring 150
emitter 151
lifetime 151, 152
name 151
position range 152
rotation 153
scale 153
speed 152
texture 151

physics
creating 208-211

physics bodies
about 55
assigning, to player 65
dynamic 57
edge 57
nodes, moving with 61
static 57

physics body shape
creating, from texture 65

physics categories
setting up 120

physics category masks 118, 119

physics joints
about 58
reference 59

physics simulation 55

physics simulation mechanics
exploring 57-59

Physics World 58

Pierre Penguin
encounters, creating for 99, 100

pin joint 58

plane
detecting 316-322

player
farming 251
physics body, assigning to 65

Player class
retrofitting 64
updating 64

player input
reacting to 4

player's progress
tracking 77

positioning 27

power-up sound effect
adding, to Player class 168

Power-up Star
adding 84
spawning 112

Power-up Star logic 132, 133

presskit()
reference 239

project
bee image, adding to 30
completing 269
navigating 13
preparing 20-22
preparing, in Xcode 350, 351

project navigator
organizing 91, 92

protocol 44

R

restart game menu
adding 162

restart menu
touch events, implementing for 164, 165

retina screens
designing for 31
in SpriteKit 32

rotation 25

S

safety
granting, on game start 155

scaling 25

scene
camera, adding 204, 205
floor, adding 205
importing, from external 3D
application 206, 207

integrating, into game 104-106
light sources, adding 202, 203
object, adding to 201
sphere, adding 201

scene editor
custom classes, using in 99

SceneKit
about 1
basics 311-315

SCNScene
scene, creating 198, 199, 200

scope
managing 269

score
adding 339-342

session
hosting 390-394
joining 390-394

session delegate functions
adding 384-388

SKAction
sound effects, playing 168

SKAction Class Reference 26

SKSpriteNode class
building 23, 24
used, for loading images 30, 31

SKSpriteNodeuserData property
metadata, storing 109, 110

sliding joint 58

smashable crates
spawning 193-195

smash open
crates, adding to 180-182

sound assets
adding, to game 166

sound effects
playing 167
playing, with SKAction 168

sphere
adding, to scene 201

spring joint 58

sprite
about 26
camera, centering on 37-41
drawing 23

SpriteKit
about 4
collision 117

SpriteKit demo
exploring 13-16

SpriteKit overlay
adding 216, 217

SpriteKit particle file
creating 148-150

SpriteKit scene editor
levels, designing 98

sprite onTap events
wiring up 68

Star class
adding 84, 85

START GAME button
wiring up 161, 162

Submit for Review 361

Swift
about 1, 2
category masks, using 119, 120
features 2, 3
prerequisites 3

Swift 4.2
about 1
new features 6

Swift game
creating 8-12

T

texture atlases
art, collecting into 34, 35
art, organizing into 33

texture atlas frames
iterating through 35, 36

textured sprite
drawing 29

textures
physics body shape, creating from 65
tiling 49
working with 29

touches
adding 323, 324

touchesBegan

implementing, in GameScene 68

touch events

implementing, for restart menu 164, 165

touch interactivity

adding 218, 219

tweening 25**type inference** 3**U****UI**

building 5

creating, for app 372-380

V**variables**

adding 381, 382

View

initializing 383, 384

volume slider

adding 169-171

X**Xcode**

about 7

installing 7