



Game Development with Swift

Embrace the mobile gaming revolution and bring your iPhone game ideas to life with Swift

Stephen Haney

[PACKT]
PUBLISHING

Game Development with Swift

Embrace the mobile gaming revolution and bring your iPhone game ideas to life with Swift

Stephen Haney



BIRMINGHAM - MUMBAI

Game Development with Swift

Copyright © 2015 Packt Publishing

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

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

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

First published: July 2015

Production reference: 1170715

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK.

ISBN 978-1-78355-053-1

www.packtpub.com

Credits

Author

Stephen Haney

Reviewers

Antonio Bello

Vladimir Pouzanov

Kevin Smith

Anil Varghese

Commissioning Editor

Edward Bowkett

Acquisition Editor

Reshma Raman

Content Development Editors

Prachi Bisht

Mamata Walkar

Technical Editor

Saurabh Malhotra

Copy Editors

Janbal Dharmaraj

Kevin McGowan

Rashmi Sawant

Project Coordinator

Sanjeet Rao

Proofreader

Safis Editing

Indexer

Hemangini Bari

Production Coordinator

Nitesh Thakur

Cover Work

Nitesh Thakur

About the Author

Stephen Haney began his programming journey at the age of 8 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 most enjoys programming as a creative outlet. 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.

Thank you to my beautiful girlfriend, Kayla, for her patience and advice.

About the Reviewers

Antonio Bello is a veteran software developer, who started writing code when memory was measured in bytes instead of gigabytes and storage was an optional add-on. Over his professional career, he has worked with several languages and technologies, in many cases, following a "learning by using" approach.

Today, he loves developing iOS Apps and their respective backends, favoring Swift over Objective C but loving both languages.

Vladimir Pouzanov is a systems engineer and an embedded enthusiast. He has spent countless hours hacking different mobile hardware, porting Linux to various devices on which it was not supposed to be run, and toying outside the iOS sandbox. He has been a professional iOS consultant and has been developing applications based on iOS since the first Apple iPhones were available. Later on, he switched his professional interest to systems engineering and cloud computing, but he still keeps a close eye on the mobile and embedded world.

I'd like to thank my wife for her amazing support while I was working on the review, sharing my attention between her, our daughter, and the book.

Kevin Smith is a founder and mobile developer. He released his first iPhone App in 2009. After the success of his first few apps, he founded App Press to help others build mobile apps. Through App Press, he has worked on and released countless award-winning iOS and Android apps.

Anil Varghese is a software engineer from Kerala, India, with extensive experience in iOS application development. He constantly strives to learn new technologies and better and faster ways of solving problems. He always finds time to help his fellow programmers and is an active member of developer communities, such as Stack Overflow.

You can reach him at anilvarghese@icloud.com and <http://anilvarghese.strikingly.com>.

www.PacktPub.com

Support files, eBooks, discount offers, and more

For support files and downloads related to your book, please visit www.PacktPub.com.

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

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



<https://www2.packtpub.com/books/subscription/packtlib>

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can search, access, and read Packt's entire library of books.

Why subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print, and bookmark content
- On demand and accessible via a web browser

Free access for Packt account holders

If you have an account with Packt at www.PacktPub.com, you can use this to access PacktLib today and view 9 entirely free books. Simply use your login credentials for immediate access.

Table of Contents

Preface	vii
Chapter 1: Designing Games with Swift	1
Why you will love Swift	2
Beautiful syntax	2
Interoperability	2
Strong typing	2
Smart type inference	2
Automatic memory management	3
An even playing field	3
Are there any downsides to Swift?	3
Less resources	3
Operating system compatibility	3
Prerequisites	3
What you will learn in this book	4
Embracing SpriteKit	4
Reacting to player input	4
Structuring your game code	4
Building UI/menus/levels	5
Integrating with Game Center	5
Maximizing fun	5
Crossing the finish line	5
Further research	5
Marketing and monetizing your game	5
Making games specifically for the desktop on OSX	6
Setting up your development environment	6
Introducing Xcode	6

Creating our first Swift game	7
Navigating our project	9
Exploring the SpriteKit Demo	9
Examining the demo code	12
Cleaning up	12
Summary	13
Chapter 2: Sprites, Camera, Actions!	15
Sharpening our pencils	16
Checkpoint 2- A	18
Drawing your first sprite	18
Building a SKSpriteNode class	18
Adding animation to your Toolkit	20
Sequencing multiple animations	21
Recapping your first sprite	22
The story on positioning	22
Alignment with anchor points	23
Adding textures and game art	24
Downloading the free assets	24
More exceptional art	24
Drawing your first textured sprite	24
Adding the bee image to your project	25
Loading images with SKSpriteNode	26
Designing for retina	27
Organizing your assets	29
Exploring Images.xcassets	29
Collecting art into texture atlases	30
Updating our bee node to use the texture atlas	30
Iterating through texture atlas frames	31
Putting it all together	32
Centering the camera on a sprite	33
Creating a new world	33
Checkpoint 2-B	37
Summary	37
Chapter 3: Mix in the Physics	39
Laying the foundation	39
Following protocol	40
Reinventing the bee	40
The icy tundra	42
Another way to add assets	42
Adding the Ground class	44
Tiling a texture	45
Running wire to the ground	45

A wild penguin appears!	47
Renovating the GameScene class	49
Exploring the physics system	51
Dropping like flies	51
Solidifying the ground	51
Checkpoint 3-A	52
Exploring physics simulation mechanics	52
Bee meets bee	54
Impulse or force?	55
Checkpoint 3-B	56
Summary	56
Chapter 4: Adding Controls	57
Retrofitting the Player class for flight	58
The Beekeeper	58
Updating the Player class	58
Moving the ground	58
Assigning a physics body to the player	59
Creating a physics body shape from a texture	59
Polling for device movement with Core Motion	60
Implementing the Core Motion code	60
Checkpoint 4-A	62
Wiring up the sprite onTap events	62
Implementing touchesBegan in the GameScene	63
Larger than life	63
Teaching our penguin to fly	64
Listening for touches in GameScene	66
Fine-tuning gravity	67
Spreading your wings	67
Improving the camera	68
Pushing Pierre forward	70
Tracking the player's progress	71
Looping the ground	71
Checkpoint 4-B	73
Summary	73
Chapter 5: Spawning Enemies, Coins, and Power-ups	75
Introducing the cast	76
Adding the power-up star	76
Locating the art assets	76
Adding the Star class	76

Adding a new enemy – the mad fly	78
Locating the enemy assets	78
Adding the MadFly class	78
Another terror – bats!	79
Adding the Bat class	79
The spooky ghost	80
Adding the Ghost class	81
Guarding the ground – adding the blade	82
Adding the Blade class	82
Adding the coins	84
Creating the coin classes	84
Organizing the project navigator	85
Testing the new game objects	86
Checkpoint 5-A	87
Preparing for endless flight	87
Summary	89
Chapter 6: Generating a Never-Ending World	91
Designing levels with the SpriteKit scene editor	91
Separating level data from game logic	93
Using empty nodes as placeholders	93
Encounters in endless flying	93
Creating our first encounter	94
Integrating scenes into the game	98
Checkpoint 6-A	101
Spawning endless encounters	101
Building more encounters	102
Updating the EncounterManager class	103
Storing metadata in SKSpriteNode userData property	104
Wiring up EncounterManager in the GameScene class	106
Spawning the star power-up at random	107
Checkpoint 6-B	109
Summary	109
Chapter 7: Implementing Collision Events	111
Learning the SpriteKit collision vocabulary	111
Collision versus contact	112
Physics category masks	112
Using category masks in Swift	113
Adding contact events to our game	114
Setting up the physics categories	114
Assigning categories to game objects	115
The player	115
The ground	115

The star power-up	116
Enemies	116
Coins	116
Preparing GameScene for contact events	116
Viewing console output	118
Testing our contact code	119
Checkpoint 7-A	119
Player health and damage	119
Animations for damage and game over	122
The damage animation	122
The game over animation	124
Collecting coins	125
The power-up star logic	127
Checkpoint 7-B	129
Summary	129
Chapter 8: Polishing to a Shine – HUD, Parallax Backgrounds, Particles, and More	131
Adding a heads-up display	131
Parallax background layers	136
Adding the background assets	137
Implementing a background class	137
Wiring up backgrounds in the GameScene class	139
Checkpoint 8-A	141
Harnessing SpriteKit's particle system	141
Adding the circle particle asset	142
Creating a SpriteKit Particle File	142
Configuring the path particle settings	144
Adding the particle emitter to the game	145
Granting safety as the game starts	146
Checkpoint 8-B	146
Summary	146
Chapter 9: Adding Menus and Sounds	147
Building the main menu	147
Creating the menu scene and menu nodes	148
Launching the main menu when the game starts	150
Wiring up the START GAME button	151
Adding the restart game menu	152
Extending the HUD	153
Wiring up GameScene for game over	154
Informing the GameScene class when the player dies	154
Implementing touch events for the restart menu	155

Checkpoint 9-A	157
Adding music and sound	157
Adding the sound assets to the game	157
Playing background music	157
Playing sound effects	158
Adding the coin sound effect to the Coin class	158
Adding the power-up and hurt sound effects to the Player class	159
Playing a sound when the game starts	159
Checkpoint 9-B	160
Summary	160
Chapter 10: Integrating with Game Center	161
Registering an app with iTunes Connect	162
Configuring Game Center	165
Creating a test user	166
Authenticating the player's Game Center account	167
Opening Game Center in our game	170
Checkpoint 10-A	173
Adding a leaderboard of high scores	173
Creating a new leaderboard in iTunes Connect	173
Updating the leaderboard from the code	175
Adding an achievement	176
Creating a new achievement in iTunes Connect	177
Updating achievements from the code	178
Checkpoint 10-B	180
Summary	180
Chapter 11: Ship It! Preparing for the App Store and Publication	181
Finalizing assets	182
Adding app icons	182
Designing the launch screen	183
Taking screenshots for each supported device	185
Finalizing iTunes Connect information	186
Configuring pricing	188
Uploading our project from Xcode	189
Submitting for review in iTunes Connect	192
Summary	194
Index	195

Preface

There has never been a better time to be a game developer. The App Store provides a unique opportunity to distribute your ideas to a massive audience. Now, Swift has arrived to bolster our toolkit and provide a smoother development experience. Swift is new, but is already hailed as an excellent, well-designed language. Whether you are new to game development or looking to add to your expertise, I think you will enjoy making games with Swift.

My goal in writing this book is to share a fundamental knowledge of Swift and SpriteKit. We will work through a complete example game so that you learn every step of the Swift development process. Once you finish this text, you will be comfortable designing and publishing your own game ideas to the App Store, from start to finish.

Please reach out with any questions and share your game creations:

E-mail: stephen@thinkingswiftly.com

Twitter: @sdothaney

The first chapter explores some of Swift's best features. Let's get started!

What this book covers

Chapter 1, Designing Games with Swift, introduces you to best features on Swift, helps you set up your development environment, and launches your first SpriteKit project.

Chapter 2, Sprites, Camera, Actions!, 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 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 will 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 features that make every great game shine. 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, Integrating with Game Center, links our example game to the Apple Game Center for leaderboards, achievements, and friendly challenges.

Chapter 11, Ship It! Preparing for the App Store and Publication, covers the essentials of packaging your game and submitting it to the App Store.

What you need for this book

This book uses the Xcode IDE Version 6.3.2 (Swift 1.2). If you use a different version of Xcode, you will likely encounter syntax differences; Apple is constantly upgrading Swift's syntax.

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

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

Who this book is for

If you wish to create and publish fun iOS games using Swift, then this book is for you. You should be familiar with basic programming concepts such as classes, types, and functions. However, no prior game development or Apple ecosystem experience is required. Additionally, experienced game programmers will find this book useful as they transition into game development with Swift.

Conventions

In this book, you will find a number of text styles that distinguish between different kinds of information. Here are some examples of these styles and an explanation of their meaning.

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "The game invokes the `didMoveToView` function whenever it switches to this scene."


A block of code is set as follows:


```
let mySprite = SKSpriteNode(color: UIColor.blueColor(), size:
    CGSize(width: 50, height: 50))
mySprite.position = CGPoint(x: 300, y: 300)
self.addChild(mySprite)
```

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

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

New terms and **important words** are shown in bold. Words that you see on the screen, for example, in menus or dialog boxes, appear in the text like this: "Select **iOS | Application** in the left pane, and **Game** in the right pane."

 Warnings or important notes appear in a box like this.

 Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book – what you liked or disliked. Reader feedback is important for us as it helps us develop titles that you will really get the most out of.

To send us general feedback, simply e-mail feedback@packtpub.com, and mention the book's title in the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide at www.packtpub.com/authors.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files from your account at <http://www.packtpub.com> for all the Packt Publishing books you have purchased. If you purchased this book elsewhere, you can visit <http://www.packtpub.com/support> and register to have the files e-mailed directly to you.

Additionally, each chapter provides checkpoint links you can use to download the example project to that point.

Downloading the color images of this book

We also provide you with a PDF file that has color images of the screenshots/diagrams used in this book. The color images will help you better understand the changes in the output. You can download this file from http://www.packtpub.com/sites/default/files/downloads/05310T_ColorImages.pdf.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books – maybe a mistake in the text or the code – we would be grateful if you could report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting <http://www.packtpub.com/submit-errata>, selecting your book, clicking on the **Errata Submission Form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded to our website or added to any list of existing errata under the Errata section of that title.

To view the previously submitted errata, go to <https://www.packtpub.com/books/content/support> and enter the name of the book in the search field. The required information will appear under the **Errata** section.

Piracy

Piracy of copyrighted material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works in any form on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at copyright@packtpub.com with a link to the suspected pirated material.

We appreciate your help in protecting our authors and our ability to bring you valuable content.

Questions

If you have a problem with any aspect of this book, you can contact us at questions@packtpub.com, and we will do our best to address the problem.

1

Designing Games with Swift

Apple's new language has arrived at the perfect time for game developers. **Swift** has the unique chance to be something special; a revolutionary tool for app creators. Swift is the gateway for developers to create the next big game on 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, "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 game for iPhones and iPads. We will start with installing the necessary software, work through each layer of game development, and ultimately publish our new game to the App Store.

We will also have some fun along the way! We aim to create an endless flyer game featuring a magnificent flying penguin named **Pierre**. What is an endless flyer? Picture hit games like 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. My hope is that our demo project will serve as a template for your own creative works. Before you know it, you will be publishing your own game ideas using the techniques we explore together.

The topics in this chapter include:

- Why you will love Swift
- What you will learn in this book
- Setting up your development environment
- Creating your first Swift game

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 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** (you will see it referred to as **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.

Are there any downsides to Swift?

Swift is a very enjoyable language, but we should consider these two issues when starting a new project.

Less resources

Given Swift's age, it is certainly more difficult to find answers to common questions through Internet searches. Objective-C has many years' worth of discussion and answers on helpful forums like Stack Overflow. This issue improves every day as the Swift community continues to develop.

Operating system compatibility

Swift projects will run on iOS7 and higher, and OSX 10.9 and higher. Swift is the wrong choice if, in a rare case, you need to target a device running an older operating system.

Prerequisites

I will strive to make this text easy to comprehend 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 it 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 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 game design.

Embracing SpriteKit

SpriteKit is Apple's 2D game development framework and your main tool for iOS 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 – we will use it exclusively in our demo game.

We will learn to use SpriteKit to power the mechanics of our game:

- Animate our player, enemies, and power-ups
- Paint and move side scrolling environments
- Play sounds and music
- Apply physics-like gravity and impulses for movement
- Handle collisions between game objects

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 re-use 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.

Building UI/menus/levels

We will learn 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 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 a memory 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 finish feeling confident in your ability to create games with Swift and bring them to market in the App Store.

Further research

I will focus on the mechanics and programming involved in great game design for iOS. A few secondary topics are outside the scope of this book.

Marketing and monetizing your game

Successfully promoting and marketing your game is an important job, but this text focuses on game development mechanics and Swift code. If you are interested in making money from your games, I strongly advise you to research the best ways to promote yourself within the indie gaming community and to start marketing your game well before launch.

Making games specifically for the desktop on OSX

We are going to concentrate on iOS. You can use the techniques in this book for game development on OSX too, but you may need to research publishing and environmental differences.

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.

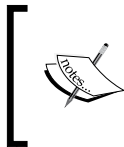
Introducing Xcode

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 game 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> and click on the **Xcode** icon. Please note the version of Xcode you are installing. At the time of writing, the current version of Xcode is 6.3.2. Swift is continually evolving and each new Xcode release brings syntax changes to Swift. For the best experience with the code in this book, use Xcode 6.3.x (with Swift version 1.2).



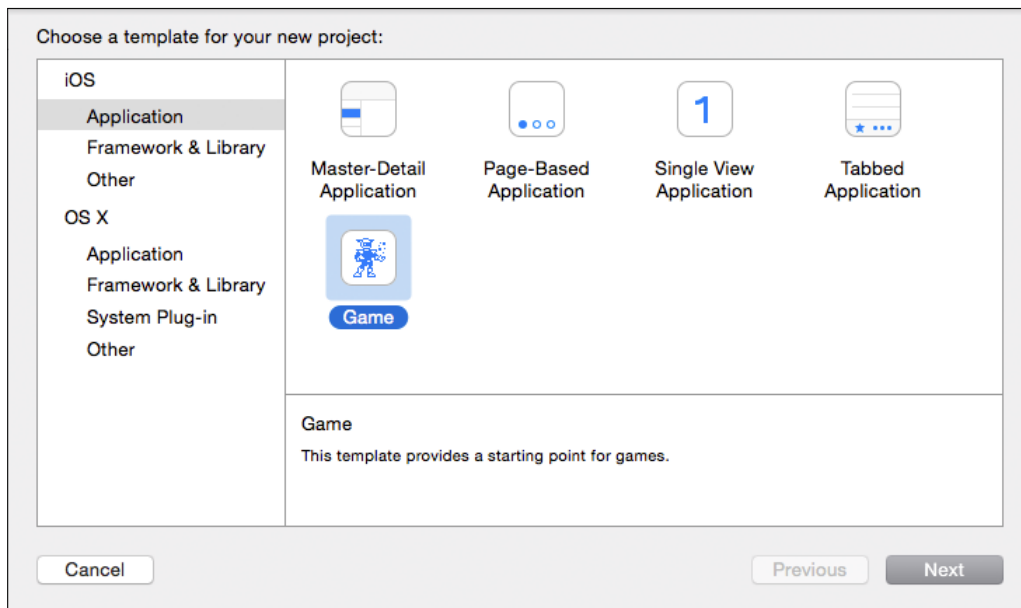
Apple announced Xcode 7 and Swift 2 at WWDC 2015, but it is still in Beta at the time of writing. It looks like there will be some minor syntax changes. The knowledge and techniques in this book will still apply.

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.

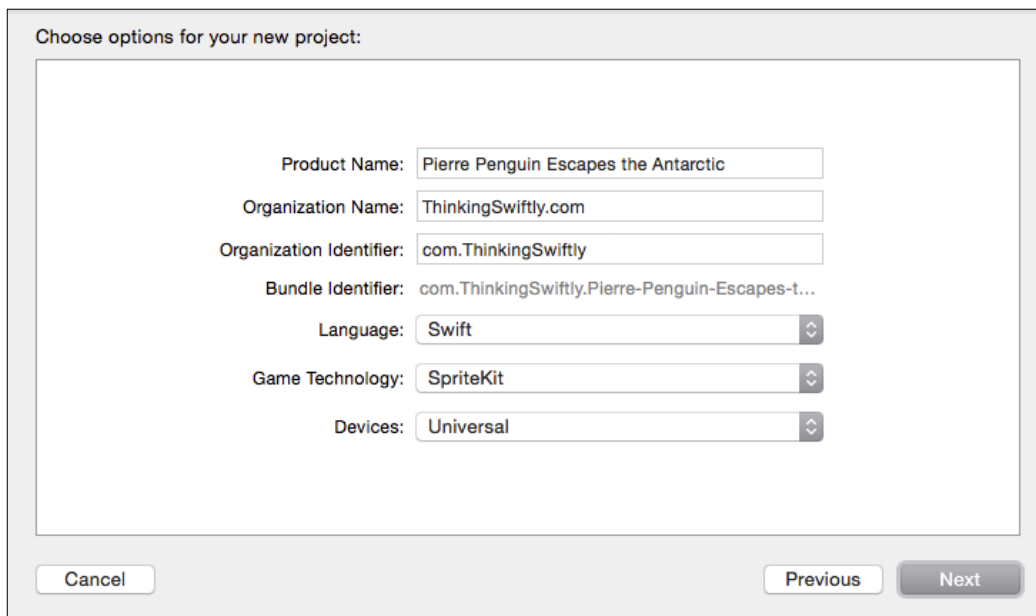
Creating our first Swift game

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

1. We will need to create a new project. Launch Xcode and navigate to **File | New | Project**. You will see a screen asking you to select a template for your new project. Select **iOS | Application** in the left pane, and **Game** in the right pane. It should look like this:



2. Once you select **Game**, click **Next**. The following screen asks us to enter some basic information about our project. Do not worry; we are almost at the fun bit. 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 game *Pierre Penguin Escapes the Antarctic*, but feel free to name your project whatever you like. For now, the names are not important. You will want to pick a meaningful **Product Name** and **Organization Identifier** when you create your own game for publication. By convention, your **Organization Identifier** should follow a reverse domain name style. I will use *com.ThinkingSwiftly*, as shown in the following screenshot.
3. After you fill out the name fields, make sure to select **Swift** for the **Language**, **SpriteKit** for **Game Technology**, and **Universal** for **Devices**. Here are my settings:



Choose options for your new project:

Product Name:

Organization Name:

Organization Identifier:

Bundle Identifier:

Language:

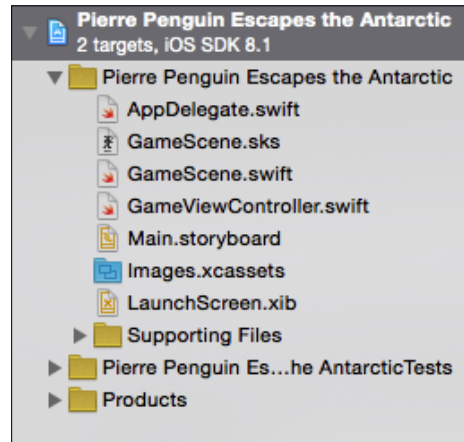
Game Technology:

Devices:

4. 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 prepopulated 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; do not feel pressure 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.

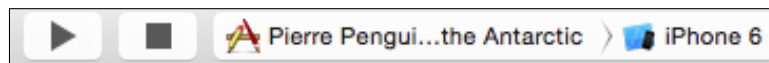
What is a scene? SpriteKit uses the concept of scenes to encapsulate each unique area of a game. Think of the scenes in a movie; we will create a scene for the main menu, a scene for the game over screen, a scene for each level in our game, and so on. If you are on the main menu of a game and you tap "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 there is already some code in this scene. The SpriteKit project template comes with a very small 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 the development environment.

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

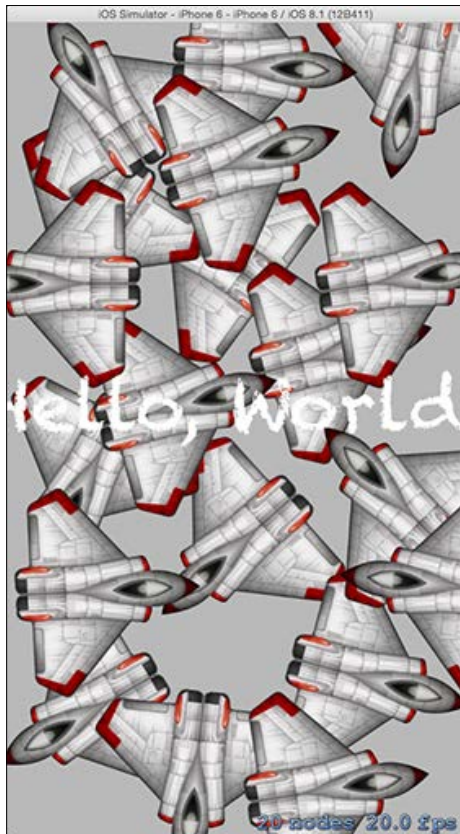


Select the iOS device of your preference to simulate using the dropdown on the far right. Which iOS device should you simulate? You are free to use the device of your choice. I will be using an iPhone 6 for the screenshots in this book, so choose **iPhone 6** if you want your results to match my images perfectly.

 Unfortunately, expect your game to play poorly in the simulator. SpriteKit suffers poor FPS 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 to test.

It is time for our first glimpse of SpriteKit in action! Press the gray play arrow (handy run 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 chalky white text: **Hello, World**. Click around on the gray background.

You will see spinning fighter jets spawning wherever you click:



I may have gone slightly overboard with the jets . . .

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 spawned a sufficient number of jets, you can close the simulator down and return to Xcode. Note: 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, after we clear away the SpriteKit demo and start on our own game.

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

The `GameScene` class implements three functions. 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.

1. The game invokes the `didMoveToView` 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.
2. The `touchesBegan` function handles the user's touch input to the iOS device screen. The SpriteKit demo uses this function to spawn the fighter jet graphic and set it spinning wherever we touch the screen.
3. 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 us 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 towards the start of our journey! Please remove lines from your `GameScene.swift` file until it looks like the following code:

```
import SpriteKit

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

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

Summary

You have already accomplished a lot. You gained 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 towards 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, Actions!*

2

Sprites, Camera, Actions!

Drawing with SpriteKit is a breeze. We are free to focus on building great gameplay experiences while SpriteKit performs the mechanical work of the **game loop**. To draw an item to the screen, we 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 to 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".

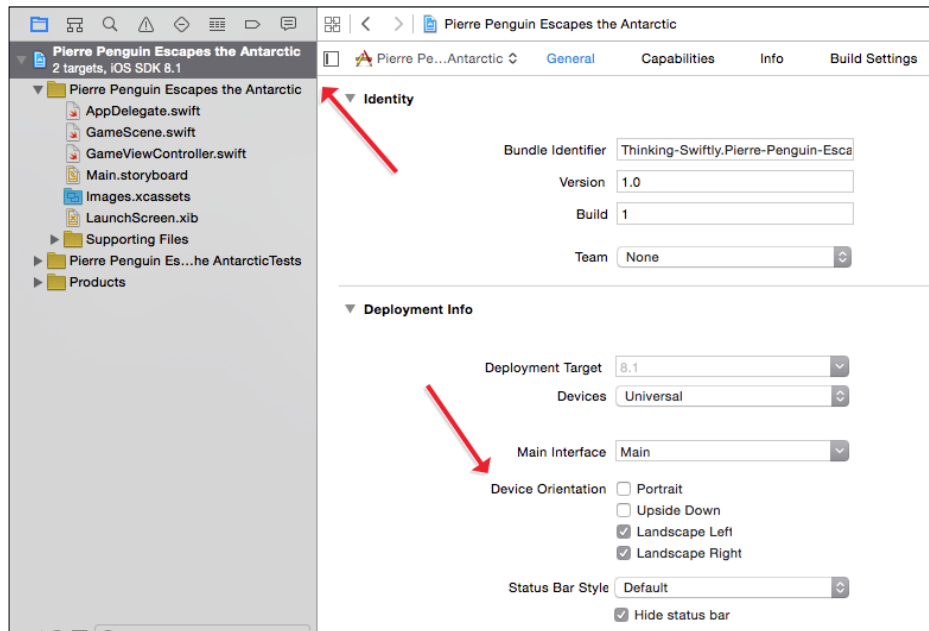
The topics in this chapter include:

- 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

Sharpening our pencils

There are four quick items to take care of before we start drawing:

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



2. The SpriteKit template generates a visual layout file for arranging sprites in our scene. We will not need it; we will use the SpriteKit visual editor later when we explore level design. To delete this extra file:
 1. Right-click on `GameScene.sks` in the project navigator and choose **delete**.
 2. Choose **Move to Trash** in the dialog window.
3. 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()
    // Create our scene:
    let scene = GameScene()
    // Configure the view:
    let skView = self.view as! SKView
    skView.showsFPS = true
    skView.showsNodeCount = true
    skView.ignoresSiblingOrder = true
    scene.scaleMode = .AspectFill
    // size our scene to fit the view exactly:
    scene.size = view.bounds.size
    // Show the new scene:
    skView.presentScene(scene)
}
```

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

```
override func supportedInterfaceOrientations() -> Int {
    return Int(
        UIInterfaceOrientationMask.Landscape.rawValue);
}
```



Downloading the example code

You can download the example code files from your account at <http://www.packtpub.com> for all the Packt Publishing books you have purchased. If you purchased this book elsewhere, you can visit <http://www.packtpub.com/support> and register to have the files e-mailed directly to you.

Additionally, each chapter provides checkpoint links you can use to download the example project to that point.

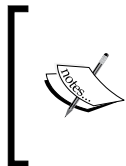
4. We should double-check that we are ready to move on. Try to run our clean project in the simulator using the toolbar play button or the *command + r* keyboard shortcut. After loading, the simulator should switch to landscape view with a blank gray background (and with the node and FPS counter in the 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*, to finish your project preparation.

Checkpoint 2- A

If you want to download my project to this point, you can do so from this URL:
<http://www.thinkingswiftly.com/game-development-with-swift/chapter-2>

Drawing your first sprite

It is time to write some game code – fantastic! Open your `GameScene.swift` file and find the `didMoveToView` function. Recall that this function fires every time the game switches to this scene. 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 developed 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 to 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 with artwork. To draw the blue square, add an instance of `SKSpriteNode` to the game:

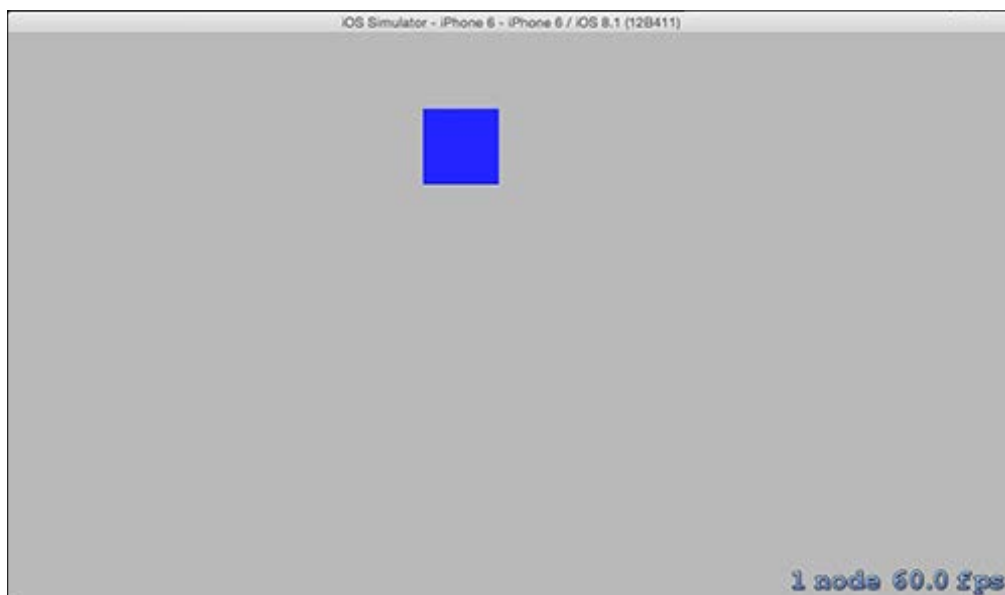
```
override func didMoveToView(view: SKView) {  
    // Instantiate a constant, mySprite, instance of SKSpriteNode  
    // The SKSpriteNode constructor can set color and size  
    // Note: UIColor is a UIKit class with built-in color presets  
    // Note: CGSize is a type we use to set node sizes
```

```
let mySprite = SKSpriteNode(color: UIColor.blueColor(), size:
    CGSize(width: 50, height: 50))

// Assign our sprite a position in points, relative to its
// parent node (in this case, the scene)
mySprite.position = CGPoint(x: 300, y: 300)

// 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 instance, 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 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.

Adding animation to your Toolkit

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 `didMoveToView` function, below the `self.addChild(mySprite)` line:

```
// Create a new constant for our action instance
// Use the moveTo 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.moveTo(CGPoint(x: 100, y: 100),
    duration: 5)
// Tell our square node to execute the action!
mySprite.runAction(demoAction)
```

Run the project. You will see our blue square slide across the screen towards the (100,100) position. This action is re-usable; any node in your scene can execute this action to move to the (100,100) 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 `moveTo` 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.moveTo` function is only one of many options. Try replacing the `demoAction` line with this code:

```
let demoAction = SKAction.scaleTo(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 each other with action groups and sequences. For instance, we can easily scale our sprite larger and spin it at the same time. Delete all of our action code so far and replace it with this code:

```
// Scale up to 4x initial scale
let demoAction1 = SKAction.scaleTo(4, duration: 5)
// Rotate 5 radians
let demoAction2 = SKAction.rotateByAngle(5, duration: 5)
// Group the actions
let actionGroup = SKAction.group([demoAction1, demoAction2])
// Execute the group!
mySprite.runAction(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.runAction(actionSequence)
```

Run the code and watch as your square first grows and then spins. Good. You 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.

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 your `didMoveToView` function matches with mine, and your sequenced animation is firing properly. Here is my code up to this point:

```
override func didMoveToView(view: SKView) {
    // Instantiate a constant, mySprite, instance of SKSpriteNode
    let mySprite = SKSpriteNode(color: UIColor.blueColor(), size:
        CGSize(width: 50, height: 50))

    // Assign our sprite a position
    mySprite.position = CGPoint(x: 300, y: 300)

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

    // Scale up to 4x initial scale
    let demoAction1 = SKAction.scaleTo(CGFloat(4), duration: 2)
    // Rotate 5 radians
    let demoAction2 = SKAction.rotateByAngle(5, duration: 2)

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

    // Execute the sequence!
    mySprite.runAction(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.

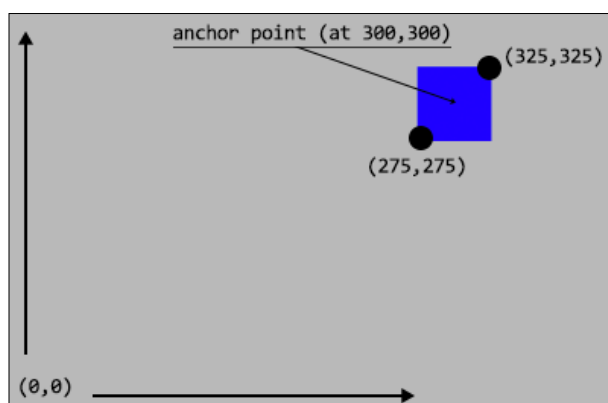
Similarly, on the individual sprite level, (0,0) refers to the bottom left corner of the sprite, while (1,1) refers to the top right corner.

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 us examine the blue square sprite we just drew on the screen. Our sprite is 50 pixels wide and 50 pixels tall, and its position is (300,300). Since we have not modified the `anchorPoint` property, its anchor point is (0.5,0.5). This means the sprite will be perfectly centered over the (300,300) position on the scene's grid. Our sprite's left edge begins at 275 and the right edge terminates at 325. Likewise, the bottom starts at 275 and the top ends at 325. The following diagram illustrates our block's position on the grid:



Why do we prefer centered sprites by default? You may think it 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 an `anchorPoint` property of (0,0) it will only expand up the y-axis and out 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 when 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.

Adding textures and game art

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 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.

These assets come from an outstanding public domain asset pack from Kenney Game Studio. I am providing a small subset of the asset pack that we will use in our game. Download the game art from this URL:

<http://www.thinkingswiftly.com/game-development-with-swift/assets>

More exceptional art

If you like the art, you can download over 16,000 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.

As CC0 assets, you can copy, modify, and distribute the art, even for commercial purposes, all without asking permission. You can read the full license here:

<https://creativecommons.org/publicdomain/zero/1.0/>

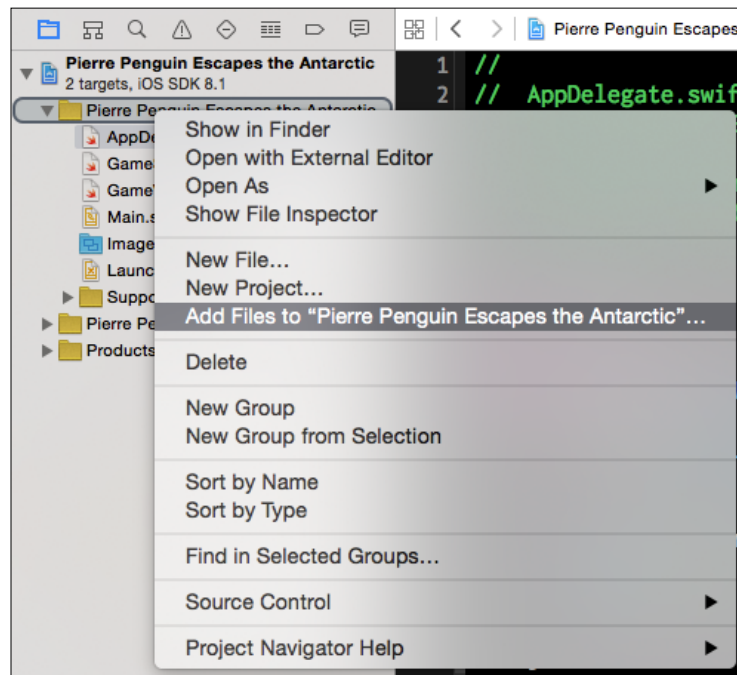
Drawing your first textured sprite

Let us 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 on **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.png` image inside the `Enemies` folder.
3. Check **Copy items if needed**, then click **Add**.

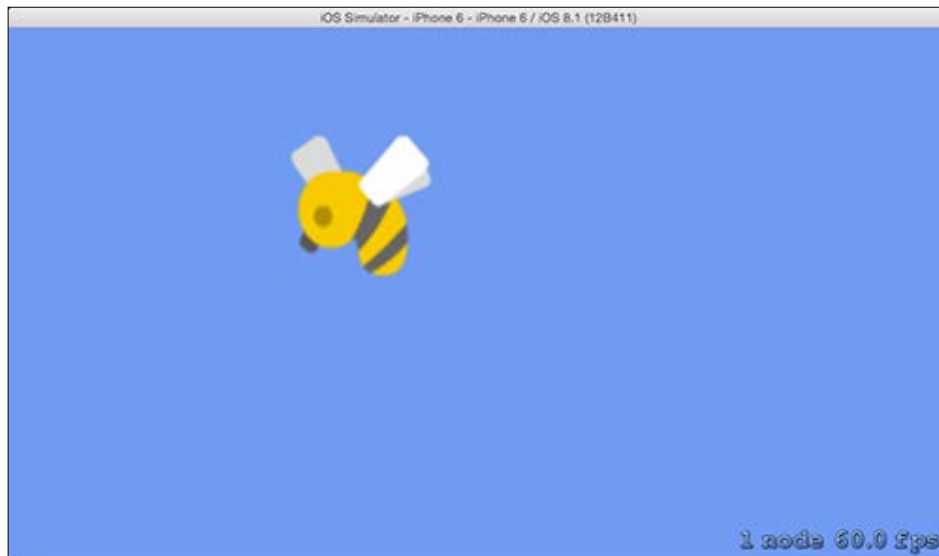
You should now see `bee.png` in your project navigator.

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 `didMoveToView` function in `GameScene.swift`. Replace `didMoveToView` with this code:

```
override func didMoveToView(view: SKView) {  
    // 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.png")  
    // 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 iPhone 6 Plus. Ignore the height for a moment; our bee node is 100 points wide but the PNG file is only 56 pixels wide. The PNG file needs to be 300 pixels wide to look sharp on the iPhone 6 Plus, 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 the 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 normally lets you provide three image files for each texture. For example, our bee node is currently 100 points wide and 100 points tall. In a perfect world, you would provide the following images to Xcode:

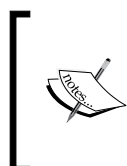
- Bee.png (100 pixels by 100 pixels)
- Bee@2x.png (200 pixels by 200 pixels)
- Bee@3x.png (300 pixels by 300 pixels)

However, there is currently an issue that prevents 3x textures from working correctly with **texture atlases**. Texture atlases group textures together and increase rendering performance dramatically (we will implement our first texture atlas in the next section). I hope that Apple will upgrade texture atlases to support 3x textures in Swift 2. For now, we need to choose between texture atlases and 3x assets for the iPhone 6 Plus.

My solution for now

In my opinion, texture atlases and their performance benefits are key features of SpriteKit. I will continue using texture atlases, thus serving 2x images to the iPhone 6 Plus (which still looks fairly sharp). This means that we will not be using any 3x assets in this book.

Further 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 both standard and 2x images for your games. Otherwise, you can safely ignore non-retina assets with Swift.



This means that we will only use double-sized images in this book. The images in the downloadable asset bundle forgo the 2x suffix, since we are only using this size. Once Apple updates texture atlases to use 3x assets, I recommend that you switch to the methodology outlined in *The ideal asset approach* section for your games.

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 sized 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 (in points) to the match texture's dimensions (in pixels). Go ahead and delete the line that sets the size for our bee node and re-run the project. SpriteKit maintains the aspect ratio automatically, but the smaller bee is still fuzzy. That is because our new node is 56 points by 48 points, matching our PNG file's pixel dimensions of 56 pixels by 48 pixels . . . yet our PNG file needs to be 112 pixels by 96 pixels for a sharp image at this node size on 2x retina screens.
- We want a smaller bee anyway, so we will resize the node rather than generate larger artwork in this case. Set the `size` property of your bee node, in points, to half the size of the texture's pixel resolution:

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

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



Great! The important concept here is to design your art files at twice the pixel resolution of your node point sizes to take advantage of 2x retina screens, or three times the point sizes to take full advantage of the iPhone 6 Plus. Now we will look at organizing and animating multiple sprite frames.

Organizing your assets

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

Exploring Images.xcassets

We can store images in an `.xcassets` file and refer to them easily from our code. This is a good place for our background images:

1. Open `Images.xcassets` from your project navigator.
2. We do not need to add any images here now but, in the future, you can drag image files directly into the image list, or right-click, then **Import**.
3. Notice that the SpriteKit demo's spaceship image is stored here. We do not need it anymore, so we can right-click on it and choose **Removed Selected Items** to delete it.

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. We need to remove our old bee texture. Right-click on `bee.png` in the project navigator and choose **Delete**, then **Move to Trash**.
2. Using Finder, browse to the asset pack you downloaded and locate the `Enemies` folder.
3. Create a new folder inside `Enemies` and name it `bee.atlas`.
4. Locate the `bee.png` and `bee_fly.png` images inside `Enemies` and copy them into your new `bee.atlas` folder. You should now have a folder named `bee.atlas` containing the two bee PNG files. This is all you need to do to create a new texture atlas – simply place your related images into a new folder with the `.atlas` suffix.
5. Add the atlas to your project. In Xcode, right-click on the project folder in the project navigator and click **Add Files...**, as we did earlier for our single bee texture.
6. Find the `bee.atlas` folder and select the folder itself.
7. Check **Copy items if needed**, then click **Add**.

The texture atlas will appear in the project navigator. Good work; we organized our bee assets into one collection and Xcode will automatically create the performance optimizations mentioned earlier.

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.png`, and a new `bee.png` exists in the texture atlas. Though we deleted the standalone `bee.png`, SpriteKit is smart enough to find the new `bee.png` in the texture atlas.

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

```
// create our bee sprite
// notice the new image name: bee_fly.png
let bee = SKSpriteNode(imageNamed: "bee_fly.png")
```

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 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 `didMoveToView` function to match mine (I removed some older comments to save space):

```
override func didMoveToView(view: SKView) {
    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:"bee.atlas")
    // 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.png"),
    beeAtlas.textureNamed("bee_fly.png")]
// Create a new SKAction to animate between the frames once
let flyAction = SKAction.animateWithTextures(beeFrames,
    timePerFrame: 0.14)
// Create an SKAction to run the flyAction repeatedly
let beeAction = SKAction.repeatActionForever(flyAction)
// Instruct our bee to run the final repeat action:
bee.runAction(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 the book. For now, pat yourself on the back. The result may seem simple, but you have unlocked a major building block towards 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 sprite to life. Let us 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 `didMoveToView` function, beneath the `bee.runAction(beeAction)` line:

```
// Set up new actions to move our bee back and forth:
let pathLeft = SKAction.moveByX(-200, y: -10, duration: 2)
let pathRight = SKAction.moveByX(200, y: 10, duration: 2)
// These two scaleXTo actions flip the texture back and forth
// We will use these to turn the bee to face left and right
let flipTextureNegative = SKAction.scaleXTo(-1, duration: 0)
let flipTexturePositive = SKAction.scaleXTo(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.repeatActionForever(flightOfTheBee)

// Tell our bee to run the flight path, and away it goes!
bee.runAction(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 that the camera follows 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. Since SpriteKit does not come with built-in camera functionality, we will create our own structure to simulate the effect we want.

One way we could accomplish this is by keeping Pierre in one position and moving every other object past him. This is effective, yet semantically confusing, and can cause errors when you are positioning game objects.

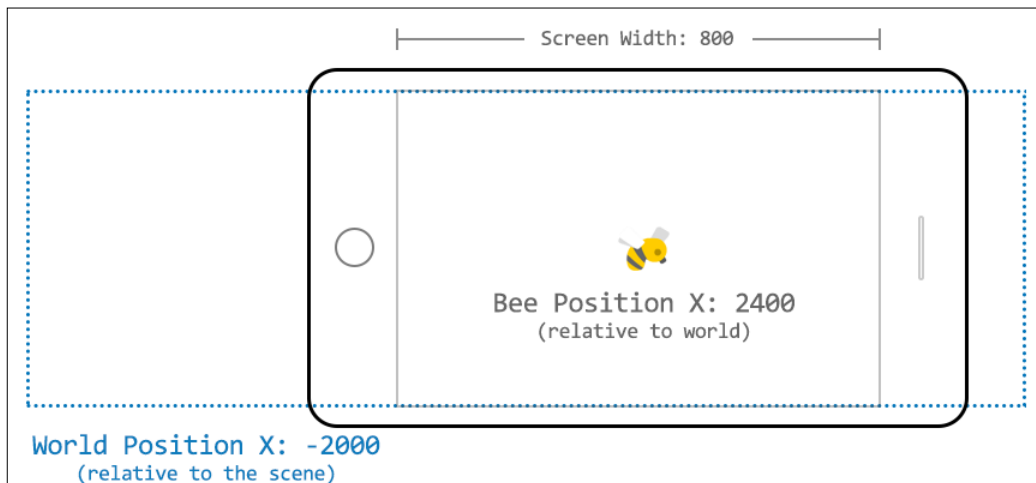
Creating a new world

I prefer to create a world node and attach all of our game nodes to it (instead of directly to the scene). We can move Pierre forward through the world and simply reposition the world node so that Pierre is always at the center of our device's viewport. All of our enemies, power-ups, and structures will be children of the world node, and will appear to move past the screen as we scroll through the world.



Each sprite node's position is always relative to its direct parent. When you change a node's position, all of its child nodes come along for the ride. This is very handy behavior for simulating our camera.

This diagram illustrates a simplified version of this technique with some made-up numbers:



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

- Our `didMoveToView` function was becoming too crowded. I 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.
- I created two new constants on the `GameScene` class: the world node and the bee node.
- I updated the `didMoveToView` function. It adds the world node to the scene's node tree, and calls the new `addTheFlyingBee` function.
- Inside the new bee function, I removed the bee constant, as `GameScene` now declares it above as its own property.
- Inside the new bee function, instead of adding the bee node to the scene, with `self.addChild(bee)`, we want to add it to the world, with `world.addChild(bee)`.
- We are implementing a new function: `didSimulatePhysics`. `SpriteKit` calls this function every frame after performing physics calculations and adjusting positions. It is a great place to update our world position. The math to change the world position resides in this new function.

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

```
import SpriteKit

class GameScene: SKScene {
    // Create the world as a generic SKNode
    let world = SKNode()
    // 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 inside the
    // didMoveToView function.)
    let bee = SKSpriteNode()

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

        // Add the world node as a child of the scene
        self.addChild(world)
        // Call the new bee function
        self.addTheFlyingBee()
    }

    // 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)
        // Notice we now attach our bee node to the world node:
        world.addChild(bee)

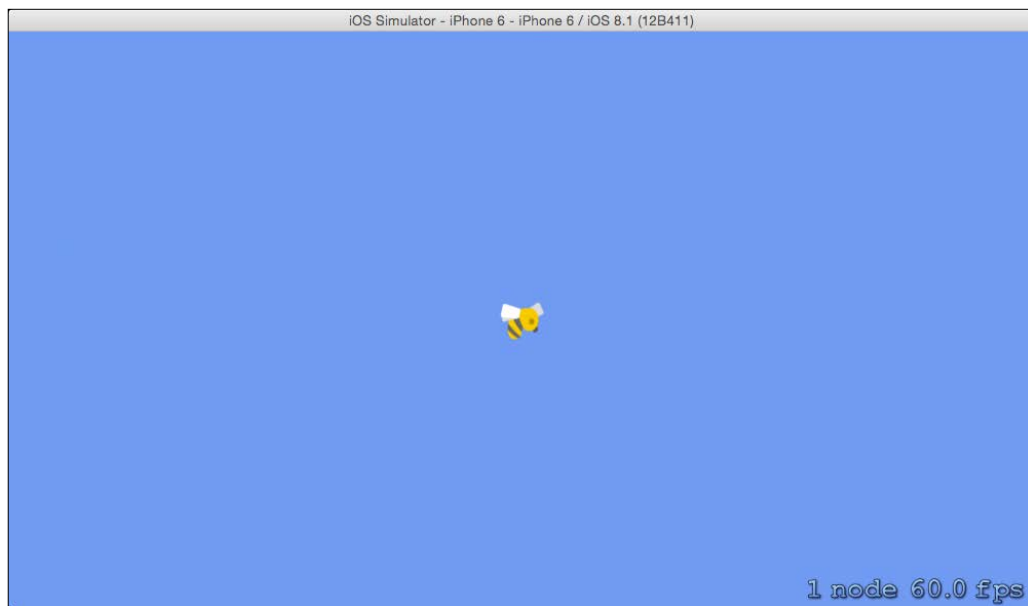
        /*
            all of the same bee animation code remains here,
            I am excluding it in this text for brevity
        */
    }

    // A new function
    override func didSimulatePhysics() {
```



```
        // To find the correct position, subtract half of the
        // scene size from the bee's position, adjusted for any
        // world scaling.
        // Multiply by -1 and you have the adjustment to keep our
        // sprite centered:
        let worldXPos = -(bee.position.x * world.xScale -
            (self.size.width / 2))
        let worldYPos = -(bee.position.y * world.yScale -
            (self.size.height / 2))
        // Move the world so that the bee is centered in the scene
        world.position = CGPoint(x: worldXPos, y: worldYPos)
    }
}
```

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 world is compensating to keep the bee centered on the screen. When we add more game objects in *Chapter 3, Mix in the Physics*, our bee will appear to fly as the entire world pans past the screen.

Checkpoint 2-B

We have made many changes to our project in this chapter. If you would like to download my project to this point, do so here:

<http://www.thinkingswiftly.com/game-development-with-swift/chapter-2>

Summary

You have gained foundational knowledge of sprites, nodes, and actions in SpriteKit and already taken huge strides towards 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. Finally, you built a world node 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; 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 world, bounce and spin colliding sprites against each other, and create realistic movement through impulses – and it does all of this before every single frame is drawn to the screen.

The topics in this chapter include:

- 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 foundation

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 re-usable classes and actively organize our new code.

Following protocol

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.swift`. Then add the following code to your new file:

```
import SpriteKit

protocol GameSprite {
    var textureAtlas: SKTextureAtlas { get set }
    func spawn(parentNode: SKNode, position: CGPoint, size:
        CGSize)
    func onTap()
}
```

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

Reinventing the bee

Our old bee is working wonderfully, but we want to spawn many bees throughout the 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.swift`. Then, add this code:

```
import SpriteKit

// Create the new class Bee, inheriting from SKSpriteNode
// and adopting the GameSprite protocol:
class Bee: SKSpriteNode, GameSprite {
    // We will store our texture atlas and bee animations as
    // class wide properties.
    var textureAtlas:SKTextureAtlas =
```

```

        SKTextureAtlas(named:"bee.atlas")
var flyAnimation = SKAction()

// The spawn function will be used to place the bee into
// the world. Note how we set a default value for the size
// parameter, since we already know the size of a bee
func spawn(parentNode:SKNode, position: CGPoint, size: CGSize
    = CGSize(width: 28, height: 24)) {
    parentNode.addChild(self)
    createAnimations()
    self.size = size
    self.position = position
    self.runAction(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.png"),
         textureAtlas.textureNamed("bee_fly.png")]
    let flyAction = SKAction.animateWithTextures(flyFrames,
        timePerFrame: 0.14)
    flyAnimation = SKAction.repeatActionForever(flyAction)
}

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

```

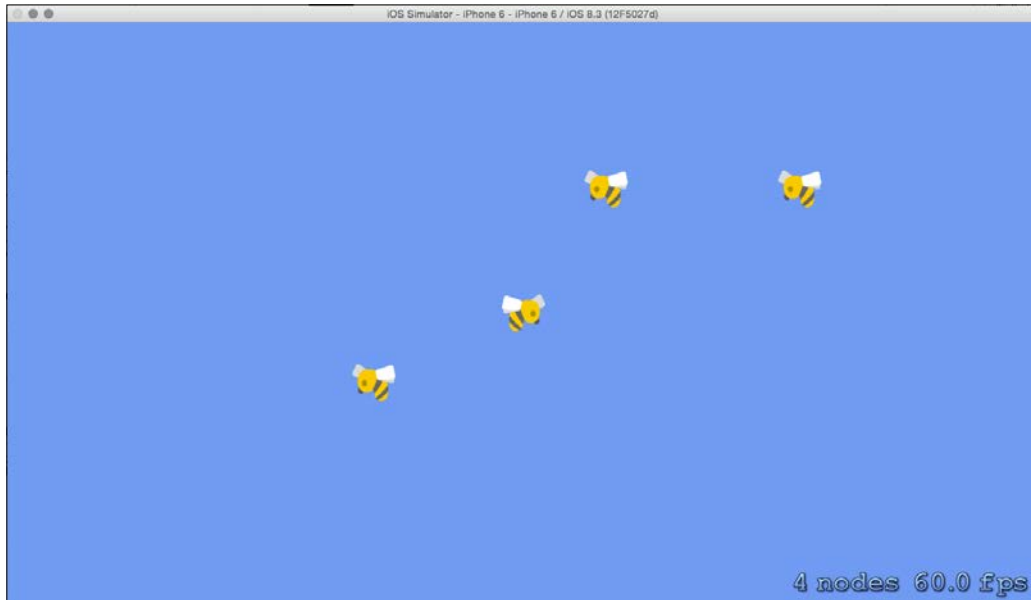
It is now easy to spawn as many bees as we like. Switch back to `GameScene.swift`, and add this code in `didMoveToView`:

```

// Create three new instances of the Bee class:
let bee2 = Bee()
let bee3 = Bee()
let bee4 = Bee()
// Use our spawn function to place the bees into the world:
bee2.spawn(world, position: CGPoint(x: 325, y: 325))
bee3.spawn(world, position: CGPoint(x: 200, y: 325))
bee4.spawn(world, position: CGPoint(x: 50, y: 200))

```

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



Depending on how you look at it, you may perceive that the new bees are moving and the original bee is still. We need to add a point of reference. Next, we will add the ground.

The icy tundra

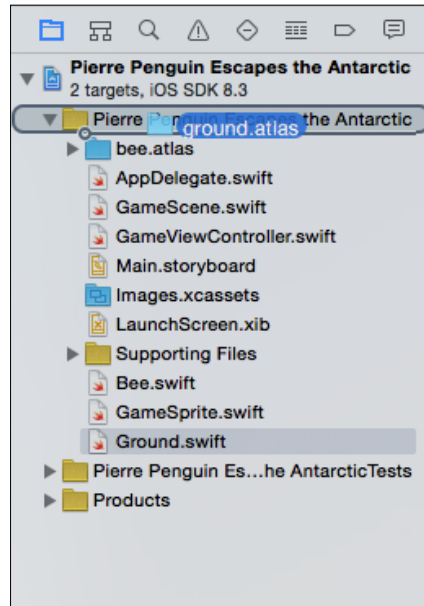
We will add some ground at the bottom of the screen to serve as a constraint for player positioning and as a reference point for movement. We will create a new class named `Ground`. First, let us add the texture atlas for the ground art to our project.

Another way to add assets

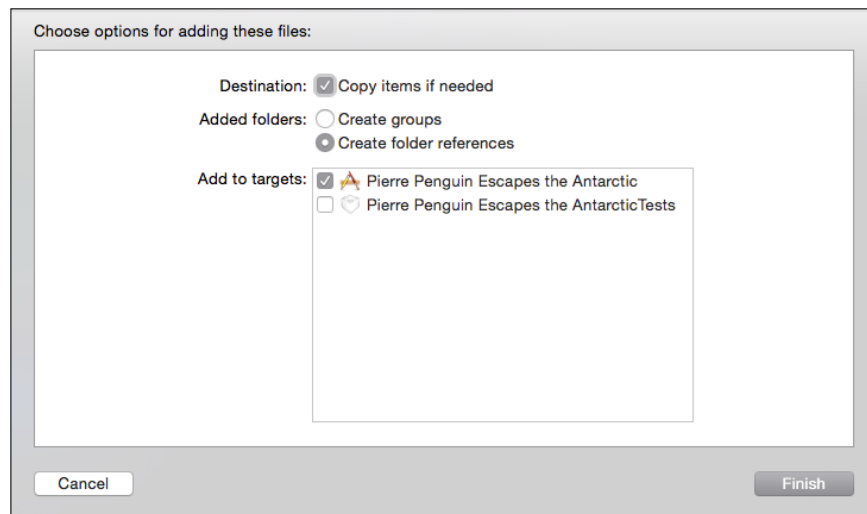
We will use a different method of adding files to Xcode. Follow these steps to add the new artwork:

1. In Finder, navigate to the asset pack you downloaded in *Chapter 2, Sprites, Camera, Actions!*, and then to the `Environment` folder.
2. You learned to create a texture atlas earlier, for our bee. I have already created texture atlases for the rest of the art we use in this game. Locate the `ground.atlas` folder.

3. Drag and drop this folder into the project manager in Xcode, under the project folder, as seen in this screenshot:



4. In the dialog box, make sure your settings match the following screenshot, and then click **Finish**:



Perfect – you should see the ground texture atlas in the project navigator.

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.swift`. 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:"ground.atlas")
    // Create an optional property named groundTexture to store
    // the current ground texture:
    var groundTexture:SKTexture?

    func spawn(parentNode:SKNode, position:CGPoint, size:CGSize) {
        parentNode.addChild(self)
        self.size = size
        self.position = position
        // This is one of those unique situations where we use
        // 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(0, 1)

        // Default to the ice texture:
        if groundTexture == nil {
            groundTexture = textureAtlas.textureNamed("ice-
                tile.png");
        }

        // We will create child nodes to repeat the texture.
        createChildren()
    }

    // Build child nodes to repeat the ground texture
    func createChildren() {
        // First, make sure we have a groundTexture value:
        if let texture = groundTexture {
            var tileCount:CGFloat = 0
            let textureSize = texture.size()
            // We will size the tiles at half the size
            // of their texture for retina sharpness:
```

```

        let tileSize = CGSize(width: textureSize.width / 2,
                               height: textureSize.height / 2)

        // 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 = CGPoint(x: 0, y: 1)
            // Add the child texture to the ground node:
            self.addChild(tileNode)

            tileCount++
        }
    }

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

```

Tiling a texture

Why do we need the `createChildren` function? SpriteKit does not support a built-in method to repeat a texture over the size of a node. Instead, we create children nodes 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 world node (the new code is in bold):

```

let world = SKNode()
let ground = Ground()

```

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

```
// size and position the ground based on the screen size.  
// Position X: Negative one screen width.  
// Position Y: 100 above the bottom (remember the ground's top  
// left anchor point).  
let groundPosition = CGPoint(x: -self.size.width, y: 100)  
// Width: 3x the width of the screen.  
// Height: 0. Our child nodes will provide the height.  
let groundSize = CGSize(width: self.size.width * 3, height:  
    0)  
// Spawn the ground!  
ground.spawn(world, position: groundPosition, size:  
    groundSize)
```

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



A wild penguin appears!

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 texture atlas for our penguin art. By now, you are familiar with adding files through the project navigator. Add the Pierre art as you did previously with the ground assets. I named Pierre's texture atlas `pierre.atlas`. You can find it in the asset pack, inside the Pierre folder.

Once you add Pierre's texture atlas to the project, 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 textureAtlas:SKTextureAtlas =
        SKTextureAtlas(named:"pierre.atlas")
    // Pierre has multiple animations. Right now we will
    // create an animation for flying up, and one for going down:
    var flyAnimation = SKAction()
    var soarAnimation = SKAction()

    func spawn(parentNode:SKNode, position: CGPoint,
        size:CSize = CSize(width: 64, height: 64)) {
        parentNode.addChild(self)
        createAnimations()
        self.size = size
        self.position = position
        // If we run an action with a key, "flapAnimation",
        // we can later reference that key to remove the action.
        self.runAction(flyAnimation, withKey: "flapAnimation")
    }

    func createAnimations() {
        let rotateUpAction = SKAction.rotateToAngle(0, duration:
            0.475)
        rotateUpAction.timingMode = .EaseOut
        let rotateDownAction = SKAction.rotateToAngle(-1,
            duration: 0.8)
        rotateDownAction.timingMode = .EaseIn
    }
}
```

```
// Create the flying animation:
let flyFrames:[SKTexture] = [
    textureAtlas.textureNamed("pierre-flying-1.png"),
    textureAtlas.textureNamed("pierre-flying-2.png"),
    textureAtlas.textureNamed("pierre-flying-3.png"),
    textureAtlas.textureNamed("pierre-flying-4.png"),
    textureAtlas.textureNamed("pierre-flying-3.png"),
    textureAtlas.textureNamed("pierre-flying-2.png")
]
let flyAction = SKAction.animateWithTextures(flyFrames,
    timePerFrame: 0.03)
// Group together the flying animation frames with a
// rotation up:
flyAnimation = SKAction.group([
    SKAction.repeatActionForever(flyAction),
    rotateUpAction
])

// Create the soaring animation, just one frame for now:
let soarFrames:[SKTexture] =
    [textureAtlas.textureNamed("pierre-flying-1.png")]
let soarAction = SKAction.animateWithTextures(soarFrames,
    timePerFrame: 1)
// Group the soaring animation with the rotation down:
soarAnimation = SKAction.group([
    SKAction.repeatActionForever(soarAction),
    rotateDownAction
])
}

func onTap() {}
}
```

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 instantiate an instance of `Player`. Add the new line: `let player = Player()`.
2. Completely delete the `addTheFlyingBee` function.
3. In `didMoveToView`, remove the line that calls `addTheFlyingBee`.

4. In `didMoveToView`, at the bottom, add a new line to spawn the player:
`player.spawn(world, position: CGPoint(x: 150, y: 250))`
5. Further down, in `didSimulatePhysics`, replace the references to the bee with references to `player`. 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 your `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 this:

```
class GameScene: SKScene {
    let world = SKNode()
    let player = Player()
    let ground = Ground()

    override func didMoveToView(view: SKView) {
        // Set a sky-blue background color:
        self.backgroundColor = UIColor(red: 0.4, green: 0.6, blue:
            0.95, alpha: 1.0)

        // Add the world node as a child of the scene:
        self.addChild(world)

        // Spawn our physics bees:
        let bee2 = Bee()
        let bee3 = Bee()
        let bee4 = Bee()
        bee2.spawn(world, position: CGPoint(x: 325, y: 325))
        bee3.spawn(world, position: CGPoint(x: 200, y: 325))
        bee4.spawn(world, position: CGPoint(x: 50, y: 200))
    }
}
```

```
// Spawn the ground:
let groundPosition = CGPoint(x: -self.size.width, y: 30)
let groundSize = CGSize(width: self.size.width * 3,
    height: 0)
ground.spawn(world, position: groundPosition, size:
    groundSize)

// Spawn the player:
player.spawn(world, position: CGPoint(x: 150, y: 250))
}

override func didSimulatePhysics() {
    let worldXPos = -(player.position.x * world.xScale -
        (self.size.width / 2))
    let worldYPos = -(player.position.y * world.yScale -
        (self.size.height / 2))
    world.position = CGPoint(x: worldXPos, y: worldYPos)
}
}
```

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:



Exploring the physics system

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 `spawn` 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 three `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 the physics simulation can stop the bees from falling through it. Open your `Ground.swift` file, locate the `spawn` function, and then 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 pointTopRight = CGPoint(x: size.width, y: 0)
self.physicsBody = SKPhysicsBody(edgeFromPoint: CGPointZero,
    toPoint: pointTopRight)
```


Run the project. The bees will now quickly drop and then stop once they collide with the ground. Notice how bees that fall farther bounce more energetically. After the bees land, your simulator will look like this:



Checkpoint 3-A

Great work so far. We have added a lot of structure to our game and started to explore the physics system. If you would like to download my project to this point, do so here:

<http://www.thinkingswiftly.com/game-development-with-swift/chapter-3>

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, but 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.

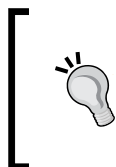
Voluminous (dynamic and static) bodies have a variety of properties that modify how they react to collisions and movement through space. This allows us to create a wide range of realistic physics effects. Each property controls one aspect of a body's physical characteristics:

- **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 measure from 0.0 to 1.0, with a default value of 0.1.
- **Mass** 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.

All right – enough with the textbook! Let us solidify our learning with some examples.

First, we want gravity to skip our bees. 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 `spawn` 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 us purposefully collide the bees.

Bee meets bee

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 `didMoveToView` function in `GameScene.swift`. At the bottom, below 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 towards 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 `-15`:

```
bee2.physicsBody?.applyImpulse(CGVector(dx: -15, 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; the lighter bee shoots away after contact. 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 the chapter. You can create many collision variations; the physics simulation offers a lot of depth with out much effort.

Impulse or force?

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: 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

We have made a number of structural changes to our project in this chapter. Feel free to download my project to this point:

<http://www.thinkingswiftly.com/game-development-with-swift/chapter-3>

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 to 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 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:

- Retrofitting the `Player` class for flight
- Polling for device movement with Core Motion
- Wiring up the sprite `onTap` 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 `didMoveToView`, and locate the bottom two lines; one sets a mass for `bee2`, the other applies an impulse to `bee2`. Remove these lines.

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: NSTimeInterval) {  
    player.update()  
}
```

Perfect. The `GameScene` class will call the `player class 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 final position since the player will soon be moving around, bringing the camera wherever they go.

In `GameScene.swift`, locate the line that defines the `groundPosition` constant and change the `y` value from 100 to 30:

```
let groundPosition = CGPoint(x: -self.size.width, 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 – they 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 easily create extremely accurate shapes for our sprites. There is a performance penalty; 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 in `Player.swift`, at the bottom of the `spawn` 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,
// and use the size from the spawn function's parameters:
let bodyTexture = textureAtlas.textureNamed("pierre-flying-3.png")
self.physicsBody = SKPhysicsBody(
    texture: bodyTexture,
    size: 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; later we will give him the tools to fly into the sky. 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 game-play 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 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 below 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` function `didMoveToView`, 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.sharedApplication().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 complete
// 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 the 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

To download my project, including the Core Motion code, visit this address:

<http://www.thinkingswiftly.com/game-development-with-swift/chapter-4>

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 you can add tap events to each sprite without building an 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 if 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 below to the `GameScene` class – I like to place it just below the `didSimulatePhysics` function:

```
override func touchesBegan(touches: Set<NSObject>, withEvent
    event: UIEvent) {
    for touch in (touches as! Set<UITouch>) {
        // Find the location of the touch:
        let location = touch.locationInNode(self)
        // Locate the node at this location:
        let nodeTouched = nodeAtPoint(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 the effect.

Larger than life

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

```
self.xScale = 4
self.yScale = 4
```

Run the project and tap on the bees. They will expand to four times their original size, as shown in the following screenshot:



Oh no – giant bees! This example shows that our `onTap` functions work. You can remove the scaling code you added to the `Bee` class. 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 `spawn` function, remove the line that runs `flyAnimation` and, instead, run `soarAnimation`:

```
self.runAction(soarAnimation, withKey: "soarAnimation")
```

3. When the player touches the screen, we apply the upward force in the `Player` class update function. Remember that `GameScene` calls this 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 gameplay:
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.removeActionForKey("soarAnimation")
        self.runAction(flyAnimation, withKey: "flapAnimation")
        self.flapping = true
    }

    // Stop the flap animation, set flapping to false:
    func stopFlapping() {
        self.removeActionForKey("flapAnimation")
        self.runAction(soarAnimation, withKey: "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 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 bottom to start the `Player` flapping when the user touches the screen:
2. Below `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:

```
player.startFlapping()

override func touchesEnded(touches: Set<NSObject>,
    withEvent event: UIEvent) {
    player.stopFlapping()
}

override func touchesCancelled(touches: Set<NSObject>!,
    withEvent 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 `didMoveToView` 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 towards 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()
```
2. In the `didMoveToView` 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() {
    var worldYPos:CGFloat = 0

    // Zoom the world as the penguin flies higher
    if (player.position.y > screenCenterY) {
        let percentOfMaxHeight = (player.position.y -
            screenCenterY) / (player.maxHeight -
            screenCenterY)
        let scaleSubtraction = (percentOfMaxHeight > 1 ?
            1 : percentOfMaxHeight) * 0.6
        let newScale = 1 - scaleSubtraction
        world.yScale = newScale
        world.xScale = newScale
        // The player is above half the screen size
        // so adjust the world on the y-axis to follow:
```

```

        worldYPos = -(player.position.y * world.yScale -
                      (self.size.height / 2))
    }

    let worldXPos = -(player.position.x * world.xScale -
                      (self.size.width / 3))

    // Move the world for our adjustment:
    world.position = CGPoint(x: worldXPos, y: worldYPos)
}

```

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 in the top screen, Pierre flies higher and the camera zooms out. In the bottom shot, the camera stops following Pierre vertically as he approaches the ground:



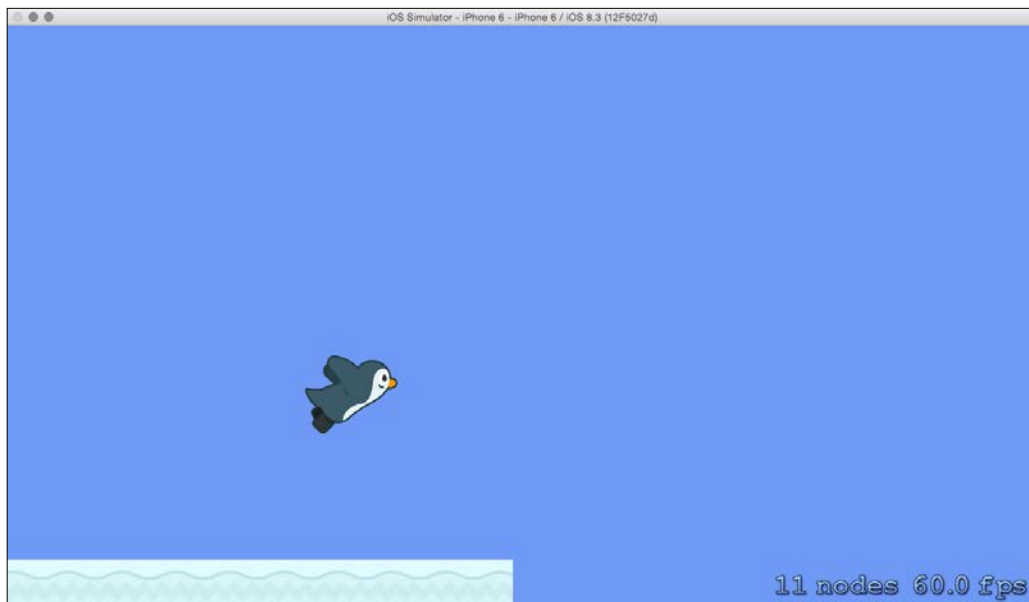
The combined effect adds a lot of polish to the game and increases the fun of flying. Our flying mechanic feels great. The next step is to move Pierre forward through the world.

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 in 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 three. Rather than extending the ground further, we will move the ground's position at well-timed intervals. Since the ground is made from repeating tiles, there are many opportunities to jump its position forward seamlessly. We simply need to figure out when the player travels 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 `didMoveToView` function, update the line that spawns the player to use the new `initialPlayerPosition` constant instead of the old hard-coded value:

```
// Spawn the player:
player.spawn(world, 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

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 travels 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: near the bottom of the function, but inside the conditional that unwraps the texture. I will show the entire function in the following example, for context, with the new line in bold:

```
func createChildren() {
    if let texture = groundTexture {
        var tileCount:CGFloat = 0
        let textureSize = texture.size()
        let tileSize = CGSize(width: textureSize.width / 2,
                               height: textureSize.height / 2)

        while tileCount * tileSize.width < self.size.width
        {
            let tileNode = SKSpriteNode(texture: texture)
            tileNode.size = tileSize
            tileNode.position.x = tileCount *
                tileSize.width
            tileNode.anchorPoint = CGPoint(x: 0, y: 1)
            self.addChild(tileNode)

            tileCount++
        }

        // Find 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, below the `createChildren` function. The scene will call this function at every frame to check if 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++
    }
}
```

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)
```

Run the project. The ground will seem to stretch on forever as Pierre flies forward. This looping ground is a big step towards 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

To download my project up to this point, visit this address:

<http://www.thinkingswiftly.com/game-development-with-swift/chapter-4>

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 in 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:

- Adding the power-up star
- A new enemy – the mad fly
- Another terror – bats!
- The spooky ghost
- Guarding the ground with the blade
- Adding coins
- Testing the new game objects

Introducing the cast

Strap on your hard hat, 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:



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 star power-up to our game. Meet our star:



Locating the art assets

You can find the art assets for power-up stars and coins inside the `goods.atlas` texture atlas in the `Coins` and `Powerups` folder of the assets bundle. Add the `goods.atlas` texture atlas to your project now.

Adding the Star class

Once 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 `SKAction`-based zany animation 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 textureAtlas:SKTextureAtlas =
        SKTextureAtlas(named:"goods.atlas")
    var pulseAnimation = SKAction()

    func spawn(parentNode:SKNode, position: CGPoint,
               size: CGSize = CGSize(width: 40, height: 38)) {
        parentNode.addChild(self)
        createAnimations()
        self.size = size
        self.position = position
        self.physicsBody = SKPhysicsBody(circleOfRadius:
            size.width / 2)
        self.physicsBody?.affectedByGravity = false
        // Since the star texture is only one frame, set it here:
        self.texture =
            textureAtlas.textureNamed("power-up-star.png")
        self.runAction(pulseAnimation)
    }

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

    func onTap() {}
}
```

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.

Adding 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:



Locating the enemy assets

You can find all of the art for our new enemies in the `Enemies` folder of the asset bundle, in the `enemies.atlas` texture atlas. Add this texture atlas to your project now.

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 textureAtlas:SKTextureAtlas =
        SKTextureAtlas(named:"enemies.atlas")
    var flyAnimation = SKAction()

    func spawn(parentNode:SKNode, position: CGPoint,
        size: CGSize = CGSize(width: 61, height: 29)) {
        parentNode.addChild(self)
        createAnimations()
        self.size = size
        self.position = position
        self.runAction(flyAnimation)
```

```

        self.physicsBody = SKPhysicsBody(circleOfRadius:
            size.width / 2)
        self.physicsBody?.affectedByGravity = false
    }

    func createAnimations() {
        let flyFrames:[SKTexture] = [
            textureAtlas.textureNamed("mad-fly-1.png"),
            textureAtlas.textureNamed("mad-fly-2.png")
        ]
        let flyAction = SKAction.animateWithTextures(flyFrames,
            timePerFrame: 0.14)
        flyAnimation = SKAction.repeatActionForever(flyAction)
    }

    func onTap() {}
}

```

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 textureAtlas:SKTextureAtlas =
        SKTextureAtlas(named:"enemies.atlas")
    var flyAnimation = SKAction()

    func spawn(parentNode:SKNode, position: CGPoint,
        size: CGSize = CGSize(width: 44, height: 24)) {
        parentNode.addChild(self)
        createAnimations()
    }
}

```

```
        self.size = size
        self.position = position
        self.runAction(flyAnimation)
        self.physicsBody = SKPhysicsBody(circleOfRadius:
            size.width / 2)
        self.physicsBody?.affectedByGravity = false
    }

    func createAnimations() {
        // The Bat has 4 animation textures:
        let flyFrames:[SKTexture] = [
            textureAtlas.textureNamed("bat-fly-1.png"),
            textureAtlas.textureNamed("bat-fly-2.png"),
            textureAtlas.textureNamed("bat-fly-3.png"),
            textureAtlas.textureNamed("bat-fly-4.png"),
            textureAtlas.textureNamed("bat-fly-3.png"),
            textureAtlas.textureNamed("bat-fly-2.png")
        ]
        let flyAction = SKAction.animateWithTextures(flyFrames,
            timePerFrame: 0.06)
        flyAnimation = SKAction.repeatActionForever(flyAction)
    }

    func onTap() {}
}
```

Now that you have created the `Bat` class, there are two more enemies to add. We will add the `Ghost` class next.

The spooky ghost

We will complement the bat with another spooky enemy: the Ghost as seen here:



Instead of animating through multiple frames, we will use actions to animate the ghost's single frame.

Adding the Ghost class

As with the other classes, create a new file in your project, `Ghost.swift`, and then add the following code:

```
import SpriteKit

class Ghost: SKSpriteNode, GameSprite {
    var textureAtlas:SKTextureAtlas =
        SKTextureAtlas(named:"enemies.atlas")
    var fadeAnimation = SKAction()

    func spawn(parentNode:SKNode, position: CGPoint,
        size: CGSize = CGSize(width: 30, height: 44)) {
        parentNode.addChild(self)
        createAnimations()
        self.size = size
        self.position = position
        self.physicsBody = SKPhysicsBody(circleOfRadius:
            size.width / 2)
        self.physicsBody?.affectedByGravity = false
        self.texture =
            textureAtlas.textureNamed("ghost-frown.png")
        self.runAction(fadeAnimation)
        // Start the ghost semi-transparent:
        self.alpha = 0.8;
    }

    func createAnimations() {
        // Create a fade out action group:
        // The ghost becomes smaller and more transparent.
        let fadeOutGroup = SKAction.group([
            SKAction.fadeAlphaTo(0.3, duration: 2),
            SKAction.scaleTo(0.8, duration: 2)
        ]);
        // Create a fade in action group:
        // The ghost returns to full size and transparency.
        let fadeInGroup = SKAction.group([
            SKAction.fadeAlphaTo(0.8, duration: 2),
            SKAction.scaleTo(1, duration: 2)
        ]);
        // Package the groups into a sequence, then a
        // repeatActionForever action:
```

```
        let fadeSequence = SKAction.sequence([fadeOutGroup,
        fadeInGroup])
        fadeAnimation = SKAction.repeatActionForever(fadeSequence)
    }

    func onTap() {}
}
```

Perfect. Our ghost is ready for action. We have added a lot of flying enemies to chase Pierre Penguin through the sky. We need a ground-based enemy that will prevent the player from taking an easy path just above the ground. Next, we will add the `Blade` class.

Guarding the ground – adding 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 much faster 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 textureAtlas:SKTextureAtlas =
    SKTextureAtlas(named:"enemies.atlas")
var spinAnimation = SKAction()

func spawn(parentNode:SKNode, position: CGPoint,
    size: CGSize = CGSize(width: 185, height: 92)) {
    parentNode.addChild(self)
    self.size = size
    self.position = position
    // Create a physics body shaped by the blade texture:
    self.physicsBody = SKPhysicsBody(
        texture: textureAtlas.textureNamed("blade-1.png"),
        size: size)
    self.physicsBody?.affectedByGravity = false
    // No dynamic body for the blade, which never moves:
    self.physicsBody?.dynamic = false
    createAnimations()
    self.runAction(spinAnimation)
}

func createAnimations() {
    let spinFrames:[SKTexture] = [
        textureAtlas.textureNamed("blade-1.png"),
        textureAtlas.textureNamed("blade-2.png")
    ]
    let spinAction = SKAction.animateWithTextures(spinFrames,
        timePerFrame: 0.07)
    spinAnimation = SKAction.repeatActionForever(spinAction)
}

func onTap() {}
}

```

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. The benefits of this structure will become apparent as your games increase in complexity.

Next, we add the class for our coins.

Adding the coins

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

- A bronze coin, worth one coin.
- A gold coin, worth five coins.

The two coins will be distinguishable by their color on the screen and the denomination text on the coin, 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 textureAtlas:SKTextureAtlas =
        SKTextureAtlas(named:"goods.atlas")
    // Store a default value for the bronze coin:
    var value = 1

    func spawn(parentNode:SKNode, position: CGPoint,
               size: CGSize = CGSize(width: 26, height: 26)) {
        parentNode.addChild(self)
        self.size = size
        self.position = position
        self.physicsBody = SKPhysicsBody(circleOfRadius:
            size.width / 2)
        self.physicsBody?.affectedByGravity = false
        self.texture =
            textureAtlas.textureNamed("coin-bronze.png")
    }
}
```

```

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

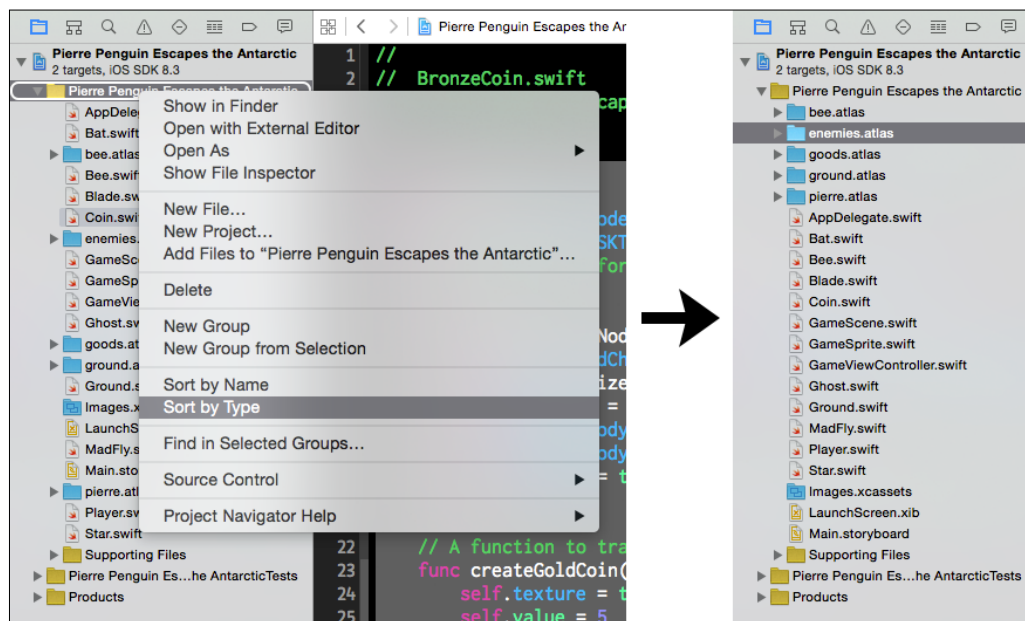
func onTap() {}
}

```

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 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 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 bees. Add this code after the bee lines:

```
// Spawn a bat:
let bat = Bat()
bat.spawn(world, position: CGPoint(x: 400, y: 200))

// A blade:
let blade = Blade()
blade.spawn(world, position: CGPoint(x: 300, y: 76))

// A mad fly:
let madFly = MadFly()
madFly.spawn(world, position: CGPoint(x: 50, y: 50))

// A bronze coin:
let bronzeCoin = Coin()
bronzeCoin.spawn(world, position: CGPoint(x: 490, y: 250))

// A gold coin:
let goldCoin = Coin()
goldCoin.spawn(world, position: CGPoint(x: 460, y: 250))
goldCoin.turnToGold()

// A ghost!
let ghost = Ghost()
ghost.spawn(world, position: CGPoint(x: 50, y: 300))

// The powerup star:
let star = Star()
star.spawn(world, position: CGPoint(x: 250, y: 250))
```

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. 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 this screenshot:



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

Checkpoint 5-A

To download my project to this point, browse to this URL:

<http://www.thinkingswiftly.com/game-development-with-swift/chapter-5>

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 spawning test code we just added to the `GameScene` class. Also, remove the six lines that we have been using to spawn the three bees from previous chapters.

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

```
override func didMoveToView(view: SKView) {
    // Set a sky-blue background color:
    self.backgroundColor = UIColor(red: 0.4, green: 0.6, blue:
        0.95, alpha: 1.0)

    // Add the world node as a child of the scene:
    self.addChild(world)

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

    // Spawn the ground:
    let groundPosition = CGPoint(x: -self.size.width, y: 30)
    let groundSize = CGSize(width: self.size.width * 3, height: 0)
    ground.spawn(world, position: groundPosition, size:
        groundSize)

    // Spawn the player:
    player.spawn(world, position: initialPlayerPosition)

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

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, a spooky ghost, the mad fly, bats, and a 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 in place in the next chapter.

We spent a lot of effort 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 to design and implement levels in SpriteKit for any genre of game.

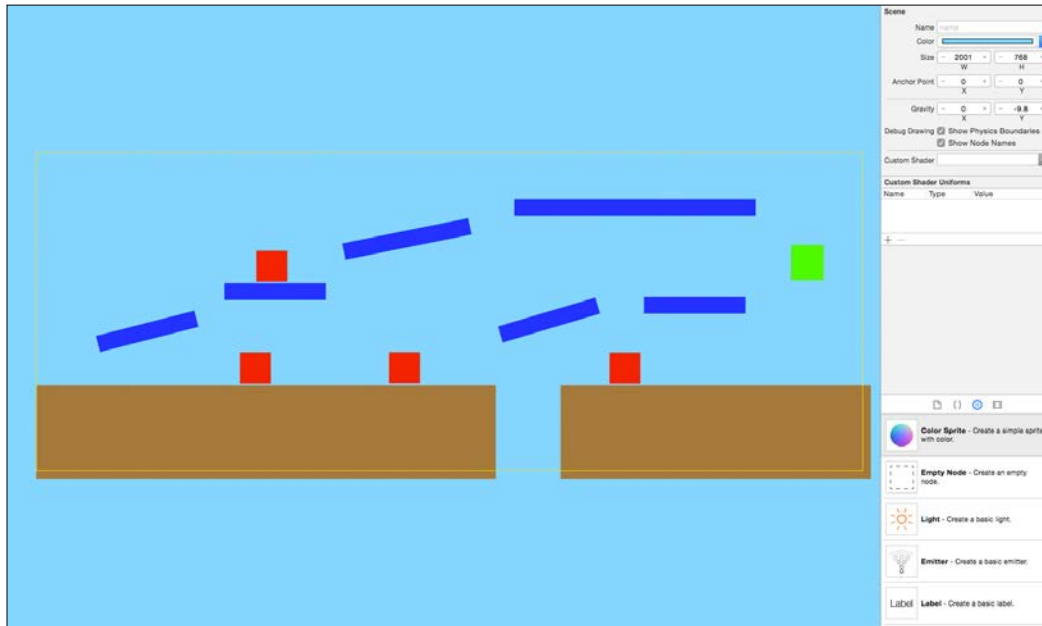
The topics in this chapter include:

- 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 star power-up at random

Designing levels with the SpriteKit scene editor

The scene editor is a valuable addition to SpriteKit. Previously, developers would be forced to hardcode positional values or rely on third party tools or custom solutions. Now, we can lay out our levels directly within Xcode. We can create nodes, attach physics bodies and constraints, create physics fields, and edit properties directly from the interface.

Feel free to experiment with the scene editor and familiarize yourself with its interface. To use the scene editor, add a new scene file to your game and then select the scene in the project navigator. Here is a simple example scene you might build for a platformer game:



In this example, I simply dragged and positioned **Color Sprite** in the scene. If you are making an unsophisticated game, you can paint nodes that do not require texture-based animation directly within the scene editor. 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 data with minimal effort.

Separating level data from game logic

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:

- 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 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 like Git.

Using empty nodes as placeholders

The scene editor lacks the ability to create reusable classes and there is no strongly typed method to link your code classes to scene editor nodes. Instead, we will use empty nodes as placeholders in the scene editor and replace them in the code with instances of our own classes. You will often see variations of this technique. For instance, the SpriteKit adventure game demo from Apple uses this technique for parts of its level design.

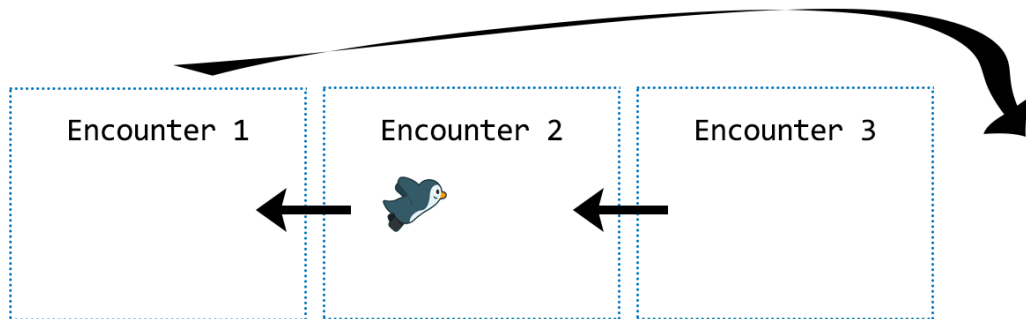
You can assign names to nodes in the scene editor and then query those names in your code. For example, you can create empty nodes named **Bat** in the scene editor, and then write code to replace every node named "Bat" with an instance of our `Bat` class in the `GameScene` class.

To illustrate this concept, we will create our first encounter for the penguin game.

Encounters in endless flying

Endless flyer games continue until the player loses. They do not feature distinct levels; instead, we will design "encounters" 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 image illustrates the basic concept:



A finished game might include 20 or more encounters to feel varied and random. We will create three encounters in this chapter 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.

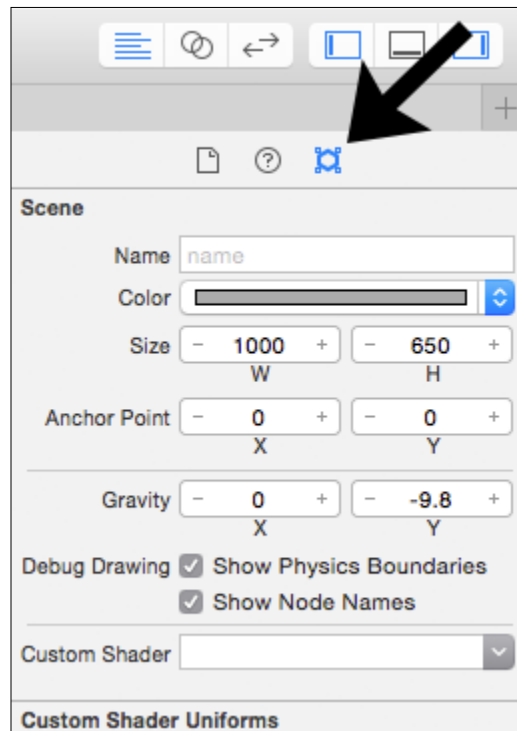
Creating our first encounter

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 SpriteKit scene file (from the **iOS | Resource** category) named `EncounterBats.sks`.

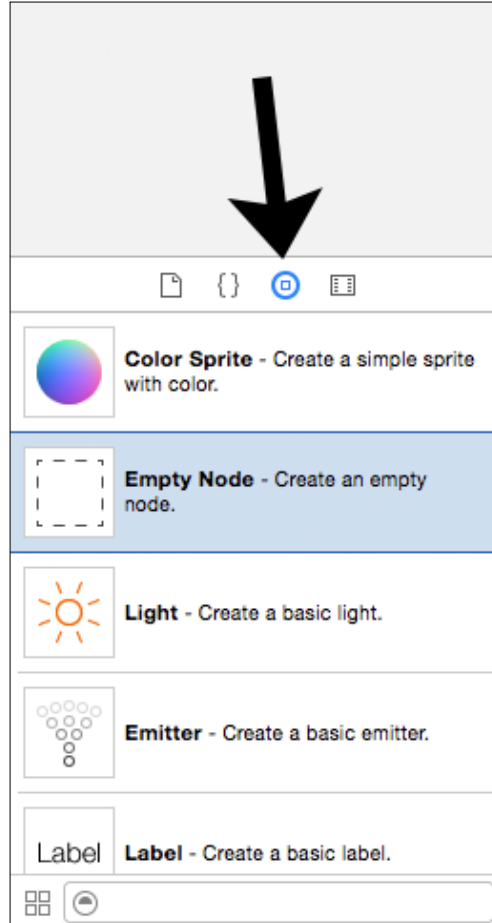
Xcode will add the new scene file to your project and open the scene editor. You should see a gray background with a yellow border, indicating the boundaries of the new scene. Scenes default to 1024 points wide by 768 points tall. We should change these values. It will be easy to chain encounters together if each encounter is 1000 pixels wide and 650 points tall.

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

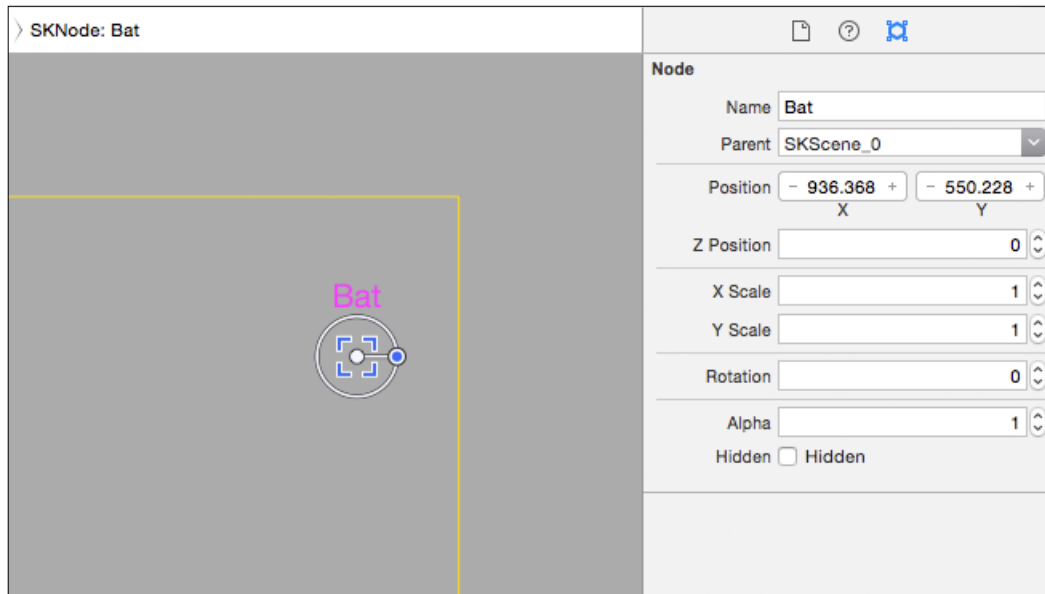


Next, we will create our first placeholder node for the `Bat` class. Follow these steps to create an **Empty Node** in the scene editor:

1. You can drag nodes from the object library. To open the object library, look towards the lower right side of the scene editor and select the circular icon, as shown in the following screenshot:



2. Drag an **Empty Node** onto your scene.
3. Using the SKNode inspector on the upper right side, name your node **Bat**, as shown in this screenshot:

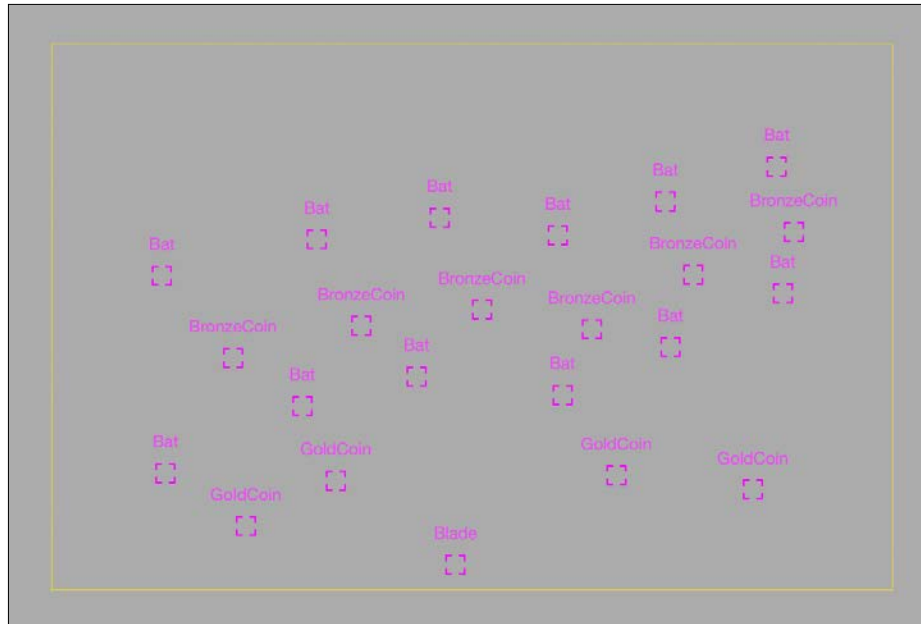


You will see **Bat** appear above the Empty Node. Great, we have created our first placeholder. We will repeat this process until we have built an entire encounter for Pierre Penguin to navigate. We can use more than just bats, but we need to first define the names we will use to label each node. If you are making games in a team, you will want to agree on labels beforehand. Here are the labels I will use for each game object:

Game object class	Scene editor node name
Bat	Bat
Bee	Bee
Blade	Blade
Coin (bronze)	BronzeCoin
Coin (gold)	GoldCoin
Ghost	Ghost
MadFly	MadFly

Feel free to build out your bat encounter. Add more empty nodes and use the labels until you are satisfied with the design. Try to picture the penguin character flying through the encounter.

In my encounter, I created an easier path through the bats, filled with bronze coins, and a more difficult path below the bats and above a blade, filled with gold coins. You can use my bat encounter, shown in the following image, 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] = [
        "EncounterBats"
    ]
}
```

```
// 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 encounter = SKNode()

        // Load this scene file into a SKScene instance:
        if let encounterScene = SKScene(fileName:
            encounterFileName) {
            // Loop through each placeholder, spawn the
            // appropriate game object:
            for placeholder in encounterScene.children {
                if let node = placeholder as? SKNode {
                    switch node.name! {
                        case "Bat":
                            let bat = Bat()
                            bat.spawn(encounter, position:
                                node.position)
                        case "Bee":
                            let bee = Bee()
                            bee.spawn(encounter, position:
                                node.position)
                        case "Blade":
                            let blade = Blade()
                            blade.spawn(encounter, position:
                                node.position)
                        case "Ghost":
                            let ghost = Ghost()
                            ghost.spawn(encounter, position:
                                node.position)
                        case "MadFly":
                            let madFly = MadFly()
                            madFly.spawn(encounter, position:
                                node.position)
                        case "GoldCoin":
                            let coin = Coin()
                            coin.spawn(encounter, position:
                                node.position)
                            coin.turnToGold()
                        case "BronzeCoin":
```



```
        let coin = Coin()
        coin.spawn(encounter, position:
                    node.position)
        default:
            println("Name error: \(node.name)")
        }
    }
}

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

// We will call this addEncountersToWorld function from
// the GameScene to append all of the encounter nodes to the
// world node from our GameScene:
func addEncountersToWorld(world:SKNode) {
    for index in 0 ... encounters.count - 1 {
        // Spawn the encounters behind the action, with
        // increasing height so they do not collide:
        encounters[index].position = CGPoint(x: -2000, y:
            index * 1000)
        world.addChild(encounters[index])
    }
}
```

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 on the `GameScene` class:
2. At the bottom of the `didMoveToView` function, call `addEncountersToWorld` to add each encounter node as a child of the `GameScene` class world node:
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 `didMoveToView` function:

```
encounterManager.encounters[0].position = CGPoint(x: 300,
    y: 0)
```

Run the project. You will see Pierre flying through your new bat encounter. 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

You can download my project to this point at this URL:

<http://www.thinkingswiftly.com/game-development-with-swift/chapter-6>

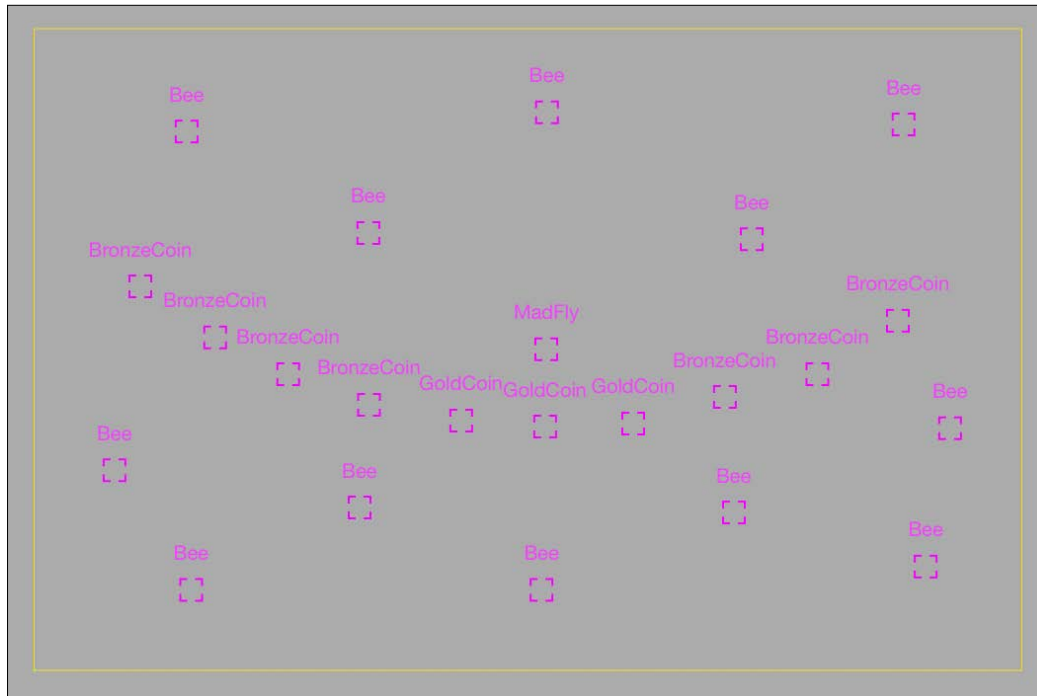
Spawning endless encounters

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 and a third positioned ahead of the player. We can track Pierre's progress and reposition the encounter nodes ahead of him.

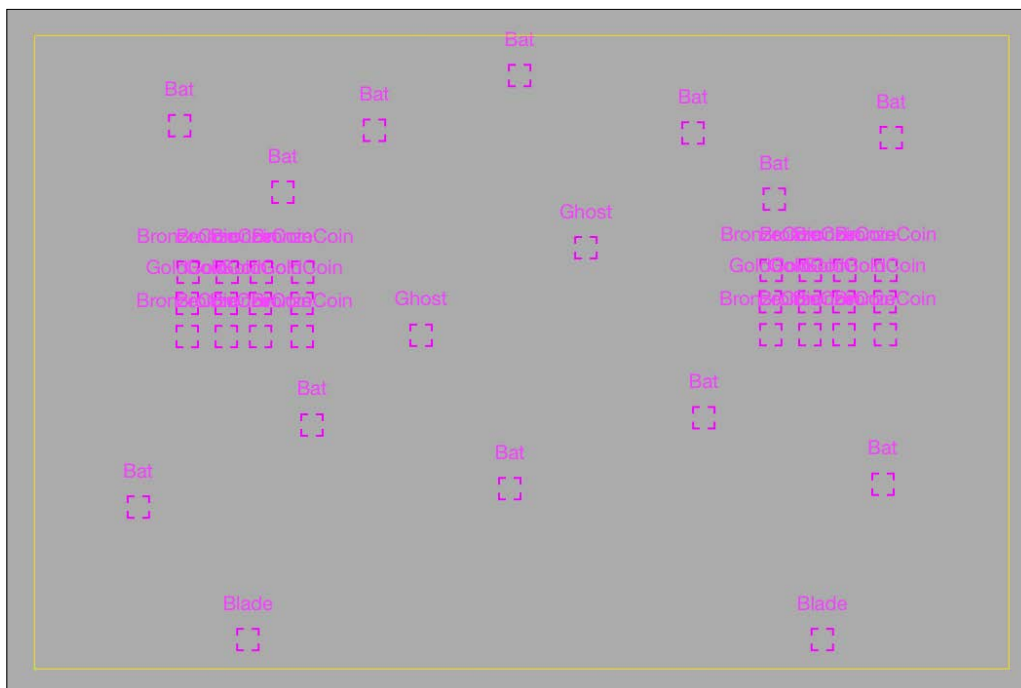
Building more encounters

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 scene files to your game: `EncounterBees.sks` and `EncounterCoins.sks`. You can completely fill these encounters with bees, ghosts, blades, coins, and bats – have fun!

For inspiration, here is my bee encounter:



Here is my coin encounter:



Updating the EncounterManager class

We have to let the `EncounterManager` class know 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] = [
    "EncounterBats",
    "EncounterBees",
    "EncounterCoins"
]
```

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 would knock nodes out of place.

The SKSpriteNode class provides a property named `userData` that we can use to store any miscellaneous data about the sprite. We will use the `userData` property to store the initial position of each sprite in the encounter so 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 initialValue = NSValue(CGPoint:
                sprite.position)
            spriteNode.userData = ["initialPosition":
                initialValue]
            // Save the positions for children of this node:
            saveSpritePositions(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 initialValue =
                spriteNode.userData?.valueForKey("initialPosition")
                as? NSValue {
                // Reset the position of the sprite:
                spriteNode.position =
```

```

        initialPositionVal.CGPointValue()
    }

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

```

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

```

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

```

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: 0)
    resetSpritePositions(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 if 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(
        nextEncounterSpawnPosition)
    nextEncounterSpawnPosition += 1400
}
```

Fantastic – we have added all the functionality we need for 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!

Spawning the star power-up at random

We still need to add the star power-up into the world. We can randomly spawn a star every 10 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. Call the star's `spawn` function, anywhere inside the `GameScene` `didMoveToView` function:

```
// Spawn the star, out of the way for now
powerUpStar.spawn(world, position: CGPoint(x: -2000,
    y: - 2000))
```

3. Inside the `GameScene` `didSimulatePhysics` function, update your new encounter code as follows:

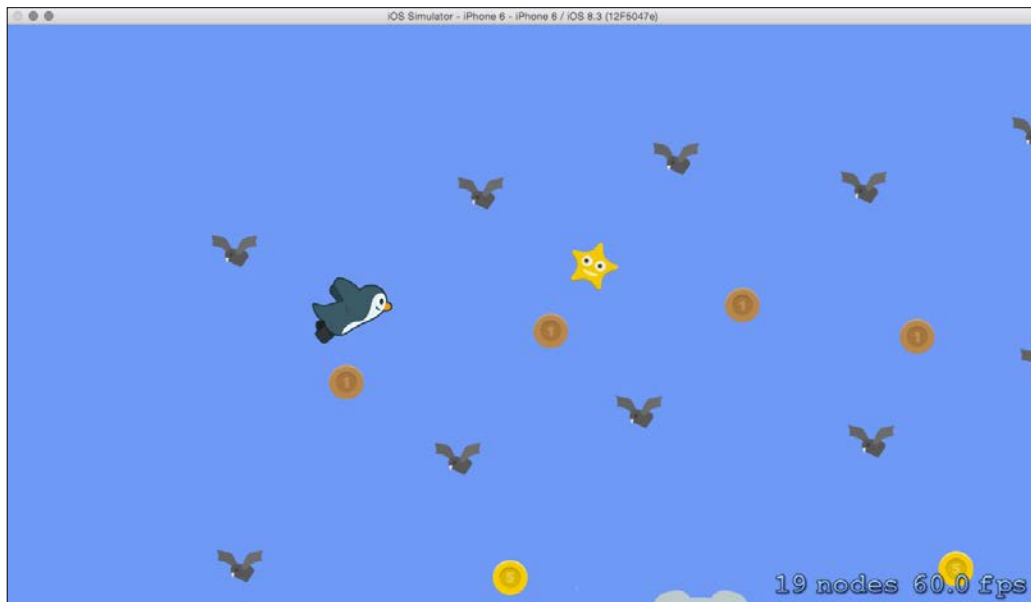
```
// Check to see if we should set a new encounter:
if player.position.x > nextEncounterSpawnPosition {
    encounterManager.placeNextEncounter(
        nextEncounterSpawnPosition)
    nextEncounterSpawnPosition += 1400

    // Each encounter has a 10% chance to spawn a star:
```



```
let starRoll = Int(arc4random_uniform(10))
if starRoll == 0 {
    if abs(player.position.x - powerUpStar.position.x)
        > 1200 {
        // Only move the star if it is off the screen.
        let randomYPos =
            CGFloat(arc4random_uniform(400))
        powerUpStar.position = CGPoint(x:
            nextEncounterSpawnPosition, y: randomYPos)
        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:



Checkpoint 6-B

To download my project to this point, visit this URL:

<http://www.thinkingswiftly.com/game-development-with-swift/chapter-6>

Summary

Great job – we have covered a lot of ground in this chapter. You learned about Xcode's new scene editor, learned to use the scene editor to lay out placeholder nodes, and interpreted the node 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:

- 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 the chapter.

Collision versus contact

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

- A **collision** 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.
- A **contact** 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 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 if 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 allow 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 which 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 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 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. 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 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 in 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 bottom, 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 `spawn` function:

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

We assigned the penguin physics category to the `Player` physics body, and used the `contactTestBitMask` property to set up contact tests with enemies, the ground, power-ups, and coins.

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 bottom of the `spawn` 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 star power-up

Open `Star.swift` and add the following code at the bottom of the `spawn` 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`, `Ghost.swift`, and `MadFly.swift`. Add the following code inside their `spawn` 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 `spawn` 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 class. At the bottom of the GameScene didMoveToView function, add this line:
`self.physicsWorld.contactDelegate = self`
3. SKPhysicsContactDelegate defines a didBeginContact function that will fire when contact occurs. We can now implement this didBeginContact function in the GameScene class. Create a new function in the GameScene class named didBeginContact, as shown in the following code:

```
func didBeginContact(contact: SKPhysicsContact) {  
    // Each contact has two bodies; we do not know which is which.  
    // We will find the penguin body, 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:  
        otherBody = contact.bodyB  
    }  
    else {  
        // bodyB is the penguin, we will test bodyA:  
        otherBody = contact.bodyA  
    }  
    // Find the type of contact:  
    switch otherBody.categoryBitMask {  
    case PhysicsCategory.ground.rawValue:  
        println("hit the ground")  
    case PhysicsCategory.enemy.rawValue:  
        println("take damage")  
    }
```

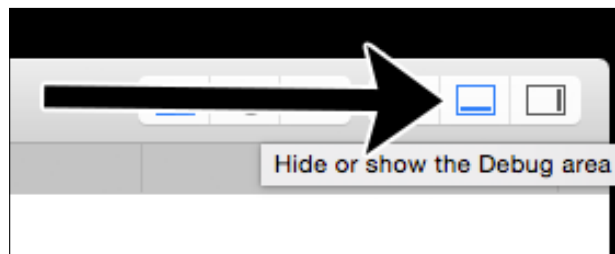
```
        case PhysicsCategory.coin.rawValue:
            println("collect a coin")
        case PhysicsCategory.powerup.rawValue:
            println("start the power-up")
        default:
            println("Contact with no game logic")
    }
}
```

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 `println` 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 the debug area is turned on, as shown in this screenshot:



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



Testing our contact code

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



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 the chapter. Next, we will add game logic for each type of contact.

Checkpoint 7-A

To download my project to this point, visit this URL:

<http://www.thinkingswiftly.com/game-development-with-swift/chapter-7>

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 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 dead:

```
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.runAction(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--
        if self.health == 0 {
            // If we are out of health, run the die function:
            die()
        }
        else {
            // Run the take damage animation:
            self.runAction(self.damageAnimation)
        }
    }
}

```

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

```

case PhysicsCategory.enemy.rawValue:
    println("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:
    println("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 in a damaged state after the player hits an enemy. 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.runBlock {
    // Allow the penguin to pass through enemies:
    self.physicsBody?.categoryBitMask =
        PhysicsCategory.damagedPenguin.rawValue
    // Use the bitwise NOT operator ~ to remove
    // enemies from the collision test:
    self.physicsBody?.collisionBitMask =
        ~PhysicsCategory.enemy.rawValue
}
// Create an opacity pulse, slow at first and fast at the end:
let slowFade = SKAction.sequence([
    SKAction.fadeAlphaTo(0.3, duration: 0.35),
    SKAction.fadeAlphaTo(0.7, duration: 0.35)
])
let fastFade = SKAction.sequence([
    SKAction.fadeAlphaTo(0.3, duration: 0.2),
    SKAction.fadeAlphaTo(0.7, duration: 0.2)
])
let fadeOutAndIn = SKAction.sequence([
    SKAction.repeatAction(slowFade, count: 2),
    SKAction.repeatAction(fastFade, count: 5),
    SKAction.fadeAlphaTo(1, duration: 0.15)
])
// Return the penguin to normal:
let damageEnd = SKAction.runBlock {
    self.physicsBody?.categoryBitMask =
        PhysicsCategory.penguin.rawValue
    // Collide with everything again:
    self.physicsBody?.collisionBitMask = 0xFFFFFFFF
    // 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

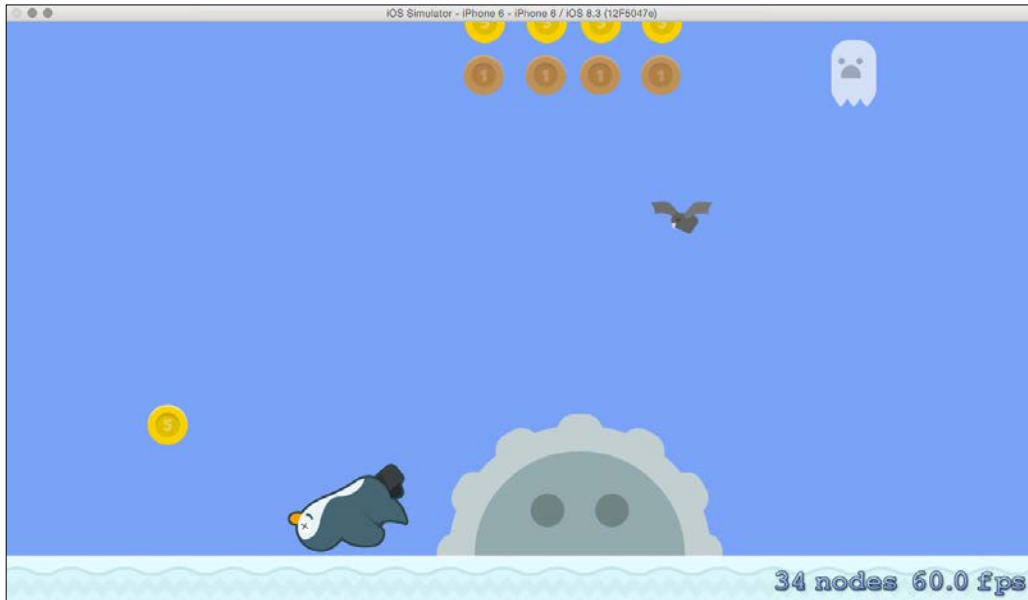
We will create a funny, over-the-top death animation when the penguin runs out of health. When Pierre loses his last hit point, he will hang in the air, scale larger, flip over on to his back, and then finally fall to the ground. To implement this animation, add the following code at the bottom of the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

```
/* --- Create the death animation --- */
let startDie = SKAction.runBlock {
    // Switch to the death texture with X eyes:
    self.texture =
        self.textureAtlas.textureNamed("pierre-dead.png")
    // Suspend the penguin in space:
    self.physicsBody?.affectedByGravity = false
    // Stop any movement:
    self.physicsBody?.velocity = CGVector(dx: 0, dy: 0)
    // Make the penguin pass through everything except the ground:
    self.physicsBody?.collisionBitMask =
        PhysicsCategory.ground.rawValue
}

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

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

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



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

Collecting coins

As a 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.fadeAlphaTo(0, duration: 0.2),
    SKAction.scaleTo(1.5, duration: 0.2),
    SKAction.moveBy(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.runBlock {
    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.runAction(collectSequence)
}
```

3. In `GameScene.swift`, call the new `collect` function from the coin contact case in the `didBeginContact` 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
        println(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 invulnerability for a short time and give the player great speed 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.removeActionForKey("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.scaleTo(1.5, duration: 0.3),
        SKAction.waitForDuration(8),
        SKAction.scaleTo(1, duration: 1),
        SKAction.runBlock {
            self.forwardVelocity = 200
            self.invulnerable = false
        }
    ])
    // Execute the sequence:
    self.runAction(starSequence, withKey: "starPower")
}
```

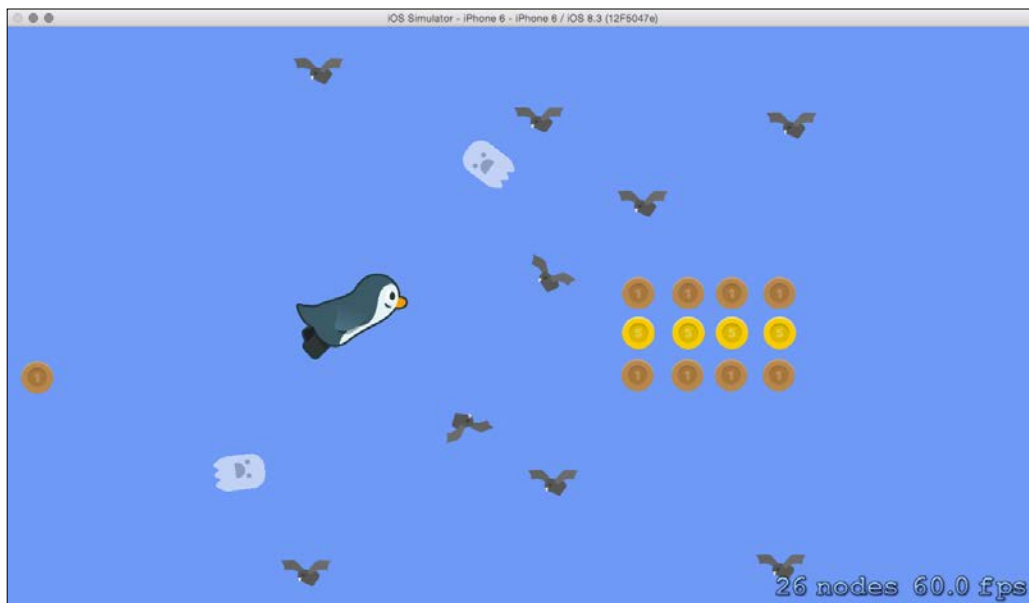
2. Invoke the new function from the `GameScene` class `didBeginContact` function, under the power-up case:

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

You may find it helpful to increase the spawn rate of the star power-up 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 star power-up. 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

We have made terrific progress in this chapter. To download my project up to this point, visit this URL:

<http://www.thinkingswiftly.com/game-development-with-swift/chapter-7>

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 star power-up.

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:

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

Adding a heads-up display

Our game needs a HUD to show the player's current health and coin score. We can use hearts to indicate health – like 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 scene itself, instead of to the `world` node, 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):



To implement the HUD, follow these steps:

1. First, we need to add the HUD art assets into the game. In the asset pack, find the `HUD.atlas` texture atlas and add it to your project.
2. Next, 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 =
        SKTextureAtlas(named:"hud.atlas")
    // An array to keep track of the hearts:
    var heartNodes:[SKSpriteNode] = []
    // An SKLabelNode to print the coin score:
    let coinCountText = SKLabelNode(text: "000000")
}
```

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

```
func createHudNodes(screenSize:CGSize) {
    // --- Create the coin counter ---
    // First, create and position a bronze coin icon:
    let coinTextureAtlas:SKTextureAtlas =
        SKTextureAtlas(named:"goods.atlas")
    let coinIcon = SKSpriteNode(texture:
        coinTextureAtlas.textureNamed("coin-bronze.png"))
    // Size and position the coin icon:
    let coinYPos = screenSize.height - 23
    coinIcon.size = CGSize(width: 26, height: 26)
    coinIcon.position = CGPoint(x: 23, y: coinYPos)
    // Configure the coin text label:
    coinCountText.fontName = "AvenirNext-HeavyItalic"
    coinCountText.position = CGPoint(x: 41, y: coinYPos)
    // 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 var index = 0; index < 3; ++index {
        let newHeartNode = SKSpriteNode(texture:
            textureAtlas.textureNamed("heart-full.png"))
        newHeartNode.size = CGSize(width: 46, height: 40)
        // Position the hearts below the coin counter:
        let xPos = CGFloat(index * 60 + 33)
        let yPos = screenSize.height - 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)
    }
}
```

4. 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 = NSNumberFormatter()  
    formatter.minimumIntegerDigits = 6  
    if let coinStr =  
        formatter.stringFromNumber(newCoinCount) {  
        // Update the label node with the new coin count:  
        coinCountText.text = coinStr  
    }  
}
```

5. 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 out any lost hearts:  
    let fadeAction = SKAction.fadeAlphaTo(0.2,  
        duration: 0.3)  
    // Loop through each heart and update its status:  
    for var index = 0; index < heartNodes.count; ++index {  
        if index < newHealth {  
            // This heart should be full red:  
            heartNodes[index].alpha = 1  
        }  
        else {  
            // This heart should be faded:  
            heartNodes[index].runAction(fadeAction)  
        }  
    }  
}
```

6. 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()
```

7. We need to place the HUD node into the scene, on top of the other game objects. Add this code at the bottom of the `GameScene` `didMoveToView` function:

```
// Create the HUD's child nodes:
hud.createHudNodes(self.size)
// Add the HUD to the scene:
self.addChild(hud)
// Position the HUD above any other game element
hud.zPosition = 50
```

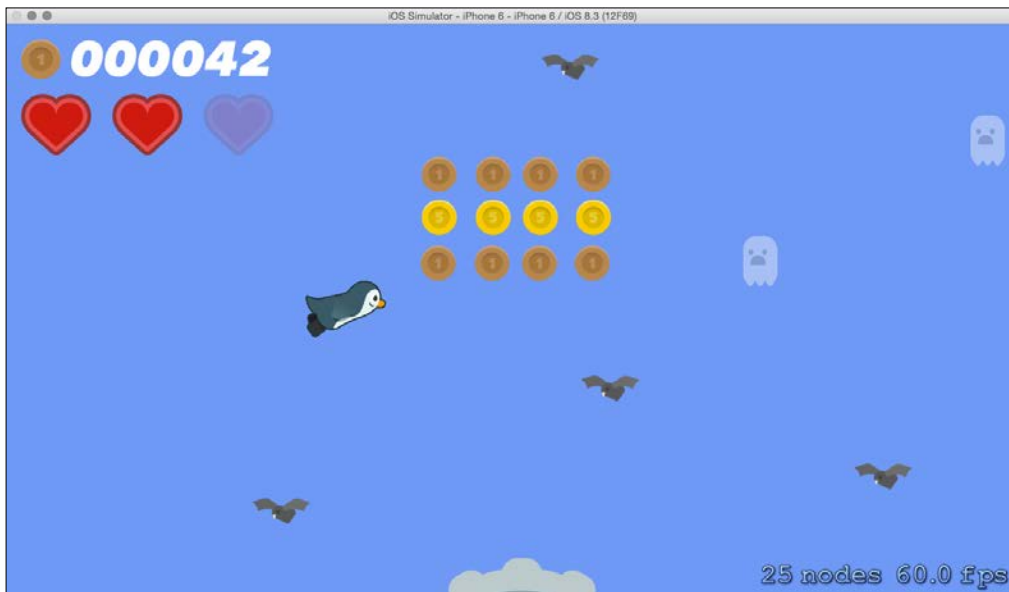
8. We are ready to send health and coin updates to the HUD. First, we will update the HUD with health updates when the player takes damage. Inside the `GameScene` `didBeginContact` function, locate the contact cases where the player takes damage – when he or she touches the ground or an enemy – and add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

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

9. Finally, we will update the HUD whenever the player collects a coin. Locate the contact case where the player contacts a coin and call the HUD `setCoinCountDisplay` function (new code 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(self.coinsCollected)
    }
```

10. Run the project and you should see your coin counter and health meter appear in the upper left hand corner, as seen in this 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 very close to the player. We can enhance the effect by painting faraway objects with increasingly desaturated colors.

In our game, we will achieve the parallax effect by attaching our backgrounds to the world, then slowly pushing the backgrounds to the right as the world moves left. As the world moves to the left (bringing the backgrounds with it), we will move the background's x position to the right so that the total movement is less than for the normal game objects. The result will be background layers that appear to move more slowly than the rest of our game, and thus appear farther away.

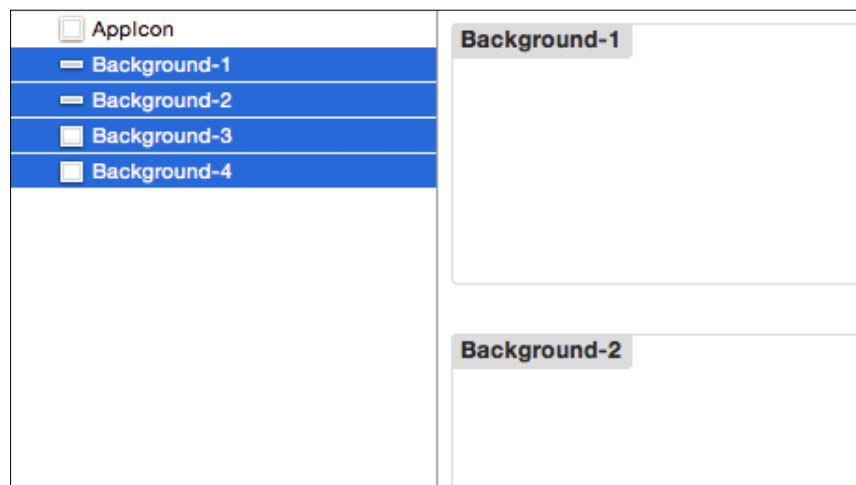
In addition, each background will only be 3000 points wide, but will jump forward at precise intervals to loop seamlessly, in a similar way to the `Ground` class.

Adding the background assets

First, add the art by following these steps:

1. Open your project's `Images.xcassets` file in Xcode.
2. In the provided game assets, locate the four background images in the `Backgrounds` folder.
3. Drag and drop the four backgrounds into the left pane of the `Images.xcassets` file.

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: 1000, height: 1000)

func spawn(parentNode:SKNode, imageName:String,
           zPosition:CGFloat, movementMultiplier:CGFloat) {
    // Position from the bottom left:
    self.anchorPoint = CGPointZero
    // 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)

    // 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(imageNamed: imageName)
        // Set the size for this node from constant:
        newBGNode.size = backgroundSize
        // Position these nodes by their lower left corner:
        newBGNode.anchorPoint = CGPointZero
        // 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 forward as the world
        // moves back 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:
2. At the bottom of the `didMoveToView` function, instantiate and spawn our four backgrounds:

```

var backgrounds:[Background] = []

// Instantiate four Backgrounds to the backgrounds array:
for i in 0...3 {
    backgrounds.append(Background())
}
// Spawn the new backgrounds:
backgrounds[0].spawn(world, imageName: "Background-1",
    zPosition: -5, movementMultiplier: 0.75)
backgrounds[1].spawn(world, imageName: "Background-2",
    zPosition: -10, movementMultiplier: 0.5)

```

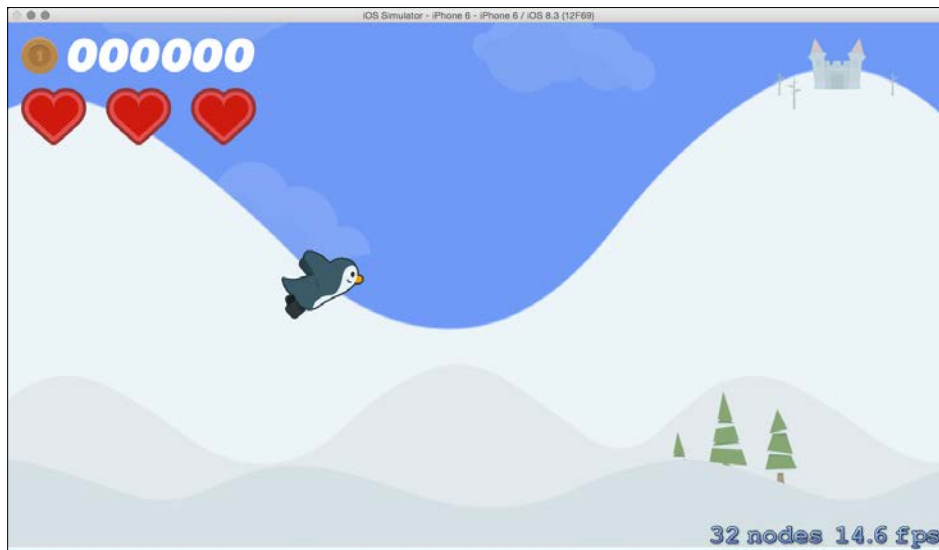


```
backgrounds[2].spawn(world, imageName: "Background-3",  
    zPosition: -15, movementMultiplier: 0.2)  
backgrounds[3].spawn(world, imageName: "Background-4",  
    zPosition: -20, movementMultiplier: 0.1)
```

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)  
}
```

4. Run the project. You should see the four 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 lowered frame rate after adding these large background textures to the game. The game will still run well on 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

To download my project to this point, visit this URL:

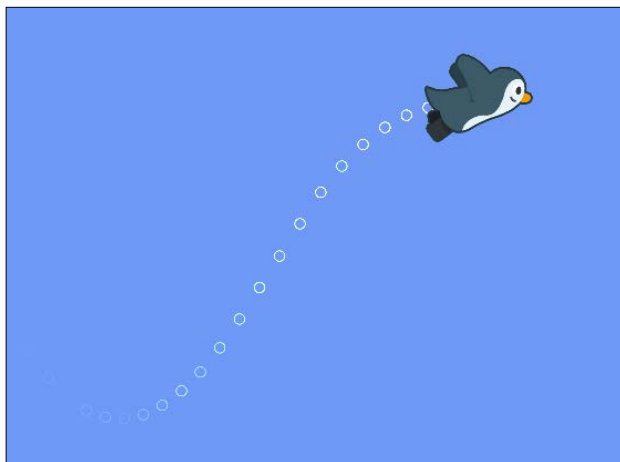
<http://www.thinkingswiftly.com/game-development-with-swift/chapter-8>

Harnessing SpriteKit's 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 combine together to create a great-looking effect. 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 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

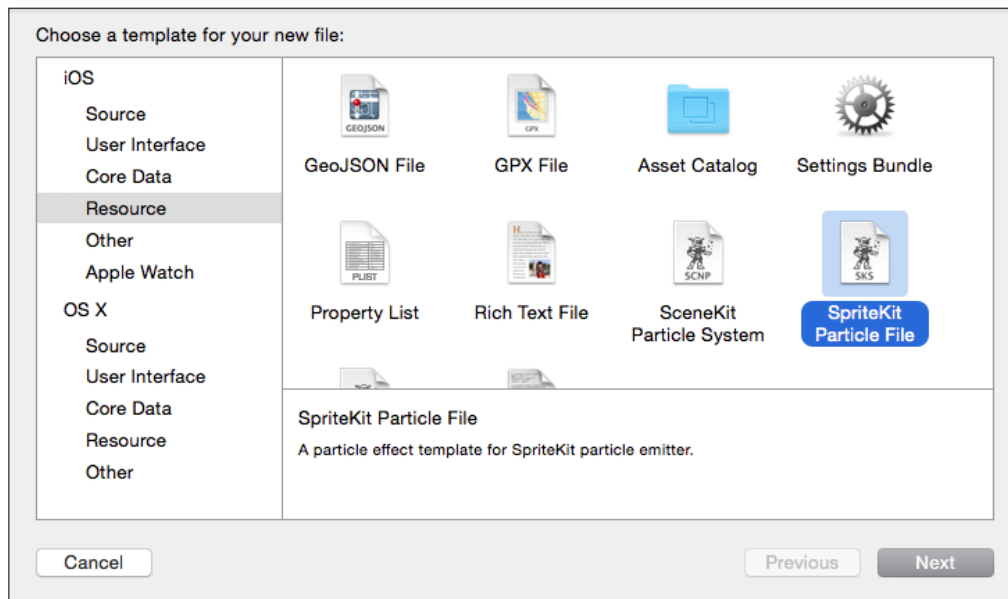
Each particle system emits multiple versions of a single image in order to create a cumulative particle effect. In our case, the image is a simple circle. To add the circle image to the game, follow these steps:

1. Open the `Images.xcassets` file in Xcode.
2. Locate the `dot.png` image in the `Particles` folder of the provided game assets.
3. Drag and drop the image file into the left pane of `Images.xcassets`.

Creating a SpriteKit Particle File

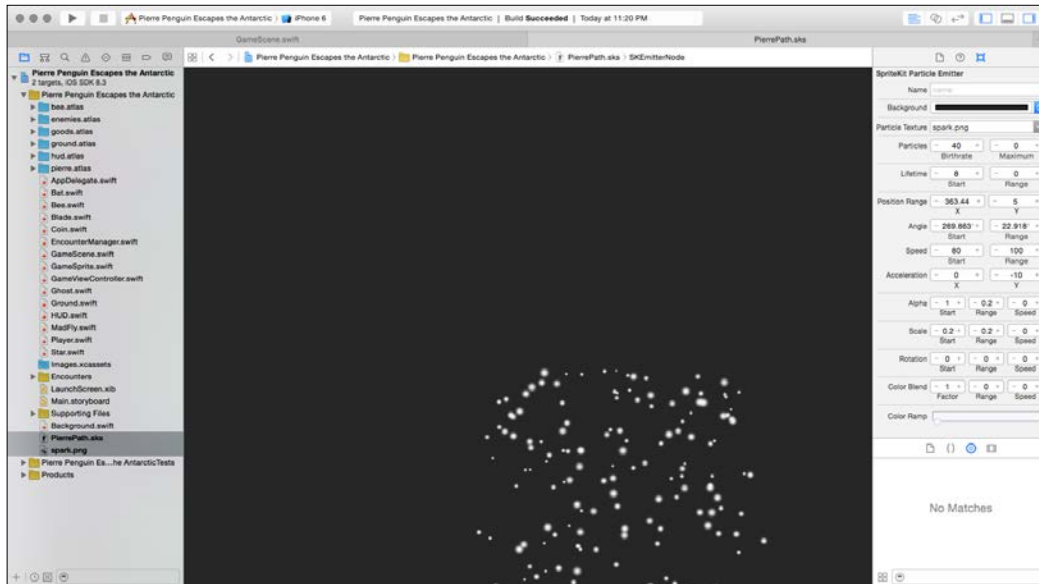
Xcode provides an excellent UI for creating and editing particle systems. To use the UI, we will add a new **SpriteKit Particle File** to our project. Follow these steps to add the new file:

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:

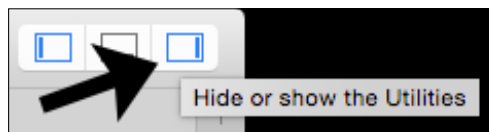


Previewing the Snow template in Xcode's particle editor



At the time of writing, 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.

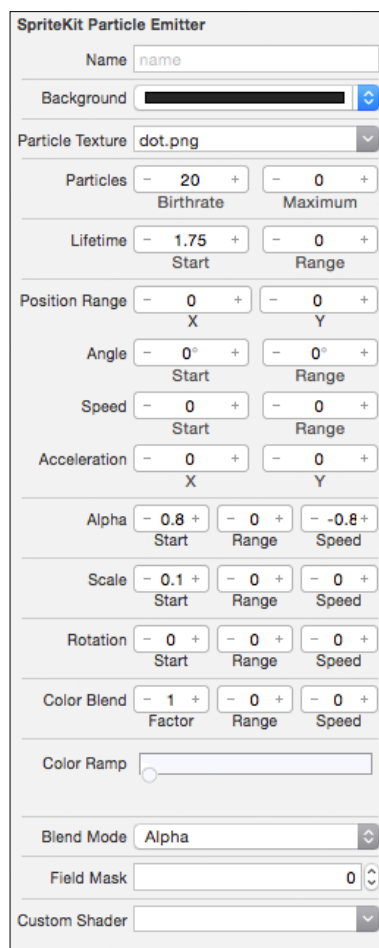
Make sure you have the right-hand sidebar turned on by lighting up the Utilities button in the upper right corner of Xcode, as shown here:



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. Feel free to experiment by changing the particle properties.

Configuring the path particle settings

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



The screenshot shows the 'SpriteKit Particle Emitter' settings panel. The settings are as follows:

- Name: name
- Background: (black bar)
- Particle Texture: dot.png
- Particles: Birthrate (20), Maximum (0)
- Lifetime: Start (1.75), Range (0)
- Position Range: X (0), Y (0)
- Angle: Start (0°), Range (0°)
- Speed: Start (0), Range (0)
- Acceleration: X (0), Y (0)
- Alpha: Start (0.8), Range (0), Speed (-0.8)
- Scale: Start (0.1), Range (0), Speed (0)
- Rotation: Start (0), Range (0), Speed (0)
- Color Blend: Factor (1), Range (0), Speed (0)
- Color Ramp: (empty)
- Blend Mode: Alpha
- Field Mask: 0
- Custom Shader: (empty)

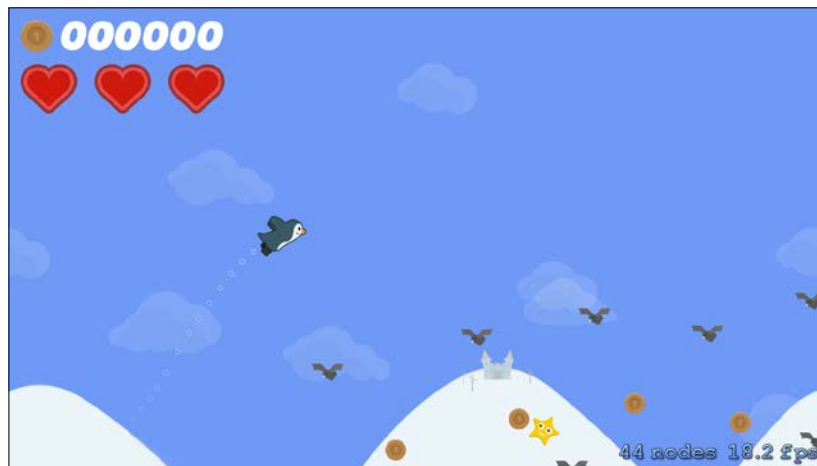
You have the correct settings when your editor shows a tiny white circle with no apparent movement.

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 `Player.swift` and add this code at the bottom of the `spawn` function:

```
// Instantiate a SKEmitterNode with the PierrePath design:
let dotEmitter = SKEmitterNode(fileName: "PierrePath.sks")
// Place the particle zPosition behind the penguin:
dotEmitter.particleZPosition = -1
// By adding the emitter node to the player, the emitter will move
// with the penguin and emit new dots wherever the player moves
self.addChild(dotEmitter)
// However, the particles themselves should attach to the world,
// so they trail behind as the player moves forward.
// (Note that self.parent refers to the world node)
dotEmitter.targetNode = self.parent
```

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, or gentle snow falling in the background.

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 `spawn` function:

```
// Grant a momentary reprieve from gravity:
self.physicsBody?.affectedByGravity = false
// Add some slight upward velocity:
self.physicsBody?.velocity.dy = 50
// Create a SKAction to start gravity after a small delay:
let startGravitySequence = SKAction.sequence([
    SKAction.waitForDuration(0.6),
    SKAction.runBlock {
        self.physicsBody?.affectedByGravity = true
    }]
)
self.runAction(startGravitySequence)
```

Checkpoint 8-B

To download my project to this point, visit this URL:

<http://www.thinkingswiftly.com/game-development-with-swift/chapter-8>

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 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 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. 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.

Topics in this chapter include:

- 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, 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. We will use a new background image for the menu. Let's add it to our project.
 1. Locate `Background-menu.png` in the `Backgrounds` folder of the asset bundle.
 2. Open `Images.xcassets` in Xcode, and then drag and drop `Background-menu.png` into `Images.xcassets` to make it available in your project.
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 texture atlas:
    let textureAtlas:SKTextureAtlas =
        SKTextureAtlas(named:"hud.atlas")
    // Instantiate a sprite node for the start button
    // (we'll use this in a moment):
    let startButton = SKSpriteNode()

    override func didMoveToView(view: SKView) {
    }
}
```

4. Next, we need to configure a few scene properties. Add this code inside the new scene's `didMoveToView` function:

```
// Position nodes from the center of the scene:
self.anchorPoint = CGPoint(x: 0.5, y: 0.5)
// Set a sky-blue background color:
self.backgroundColor = UIColor(red: 0.4, green: 0.6,
    blue: 0.95, alpha: 1.0)
// Add the background image:
let backgroundImage = SKSpriteNode(imageNamed: "Background-menu")
backgroundImage.size = CGSize(width: 1024, height: 768)
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 `didMoveToView` 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 a `SKSpriteNode` for the button graphic and a `SKLabelNode` for the "Start Game" text. Add this code at the bottom of the `didMoveToView` function to create the button:

```
// Build the start game button:
startButton.texture = textureAtlas.textureNamed("button.png")
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"
startButton.addChild(startText)
```

7. Finally, we will make the start button pulse in and out to add movement and excitement to the menu. Add this code at the bottom of the `didMoveToView` function to fade the button in and out:

```
// Pulse the start button in and out gently:
let pulseAction = SKAction.sequence([
    SKAction.fadeAlphaTo(0.7, duration: 0.9),
    SKAction.fadeAlphaTo(1, duration: 0.9),
])
startButton.runAction(
    SKAction.repeatActionForever(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

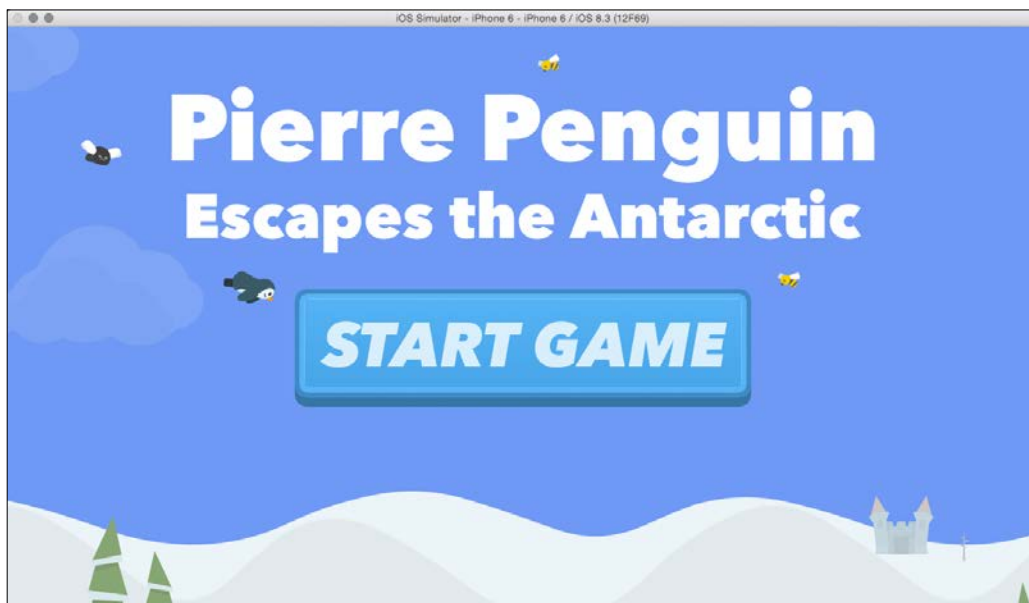
So far, our app has launched directly to the `GameScene` class whenever it starts. We will now update our view controller to start with the `MenuScene` class instead. Follow these steps to launch the menu when the game starts:

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 and 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<NSObject>, withEvent
    event: UIEvent) {
    for touch in (touches as! Set<UITouch>) {
        let location = touch.locationInNode(self)
        let nodeTouched = nodeAtPoint(location)

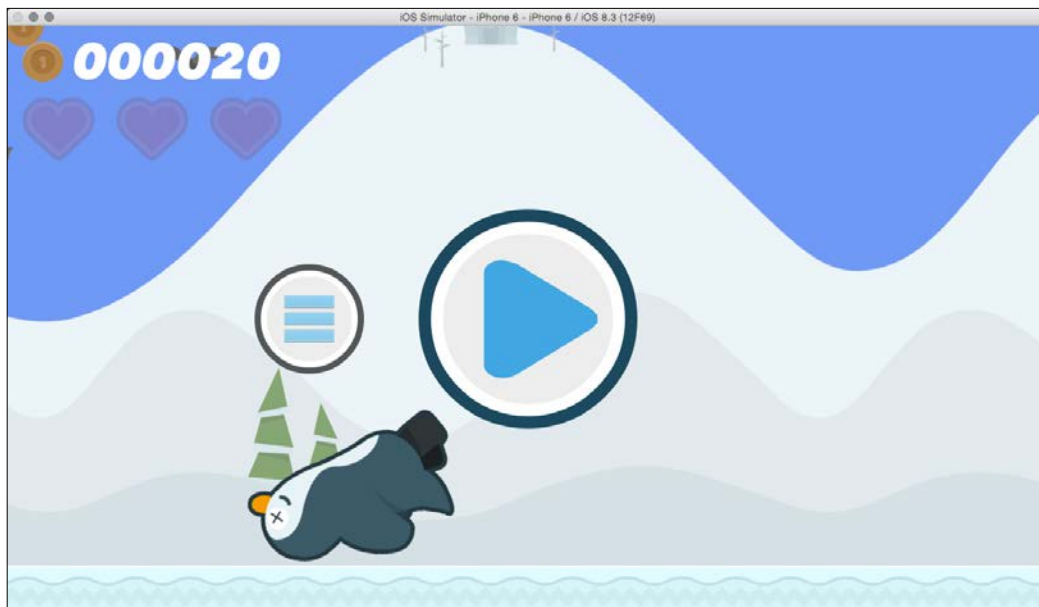
        if nodeTouched.name == "StartBtn" {
            // Player touched the start text or button node
        }
    }
}
```

```
        // 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 create 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, as in this screenshot:



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.png")
menuButton.texture =
    textureAtlas.textureNamed("button-menu.png")
// Assign node names to the buttons:
restartButton.name = "restartGame"
menuButton.name = "returnToMenu"
// Position the button node:
let centerOfHud = CGPoint(x: screenSize.width / 2,
    y: screenSize.height / 2)
restartButton.position = centerOfHud
menuButton.position =
    CGPoint(x: centerOfHud.x - 140, y: centerOfHud.y)
// Size the button nodes:
restartButton.size = CGSize(width: 140, height: 140)
menuButton.size = CGSize(width: 70, height: 70)
```

3. 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.fadeAlphaTo(1, duration: 0.4)
        restartButton.runAction(fadeAnimation)
        menuButton.runAction(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.

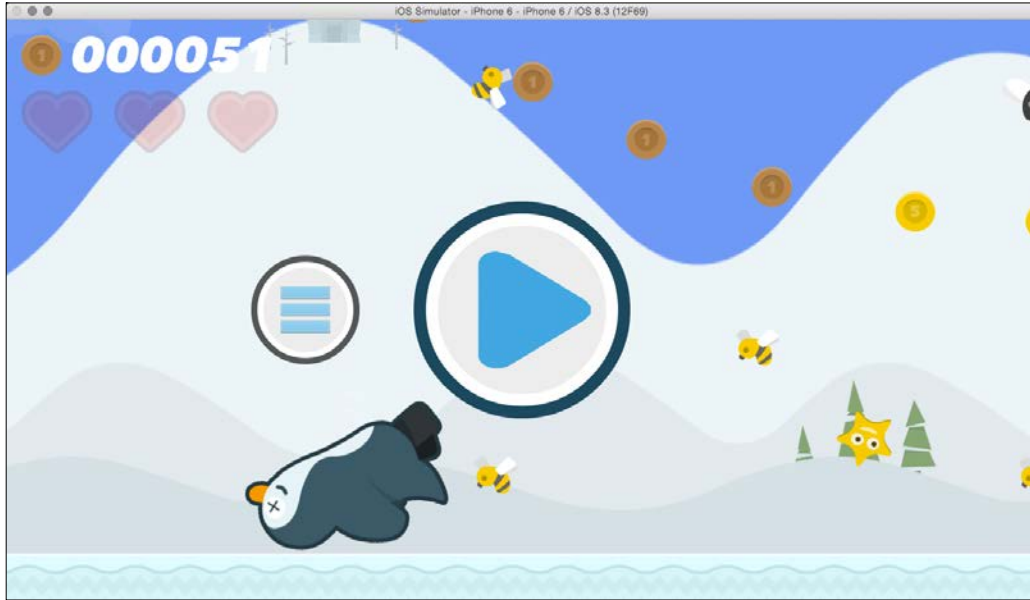
Informing the GameScene class when the player dies

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?.parent as? GameScene {
    gameScene.gameOver()
}
```

We access `GameScene` by traveling up the node tree. The `Player` node's parent is the `world` node. The `world` 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<NSObject>, withEvent
    event: UIEvent) {
    player.startFlapping()

    for touch in (touches as! Set<UITouch>) {
        let location = touch.locationInNode(self)
        let nodeTouched = nodeAtPoint(location)

        if let gameSprite = nodeTouched as? GameSprite {
            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: .crossFadeWithDuration(0.6))
        }
        else if nodeTouched.name == "returnToMenu" {
            // Transition to the main menu scene:
            self.view?.presentScene(
                MenuScene(size: self.size),
                transition: .crossFadeWithDuration(0.6))
        }
    }
}
```

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 towards 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

Download my project to this point at this URL:

<http://www.thinkingswiftly.com/game-development-with-swift/chapter-9>

Adding music and sound

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 `SKAction playSoundFileNamed`.

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 the 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. You should see the `Sound` folder show up in your project just like any other asset.

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 `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:  
let musicUrl = NSBundle.mainBundle().URLForResource(  
    "Sound/BackgroundMusic.m4a", withExtension: nil)
```

```
if let url = musicUrl {
    musicPlayer =
        AVAudioPlayer(contentsOfURL: url, error: nil)
    musicPlayer.numberOfLoops = -1
    musicPlayer.prepareToPlay()
    musicPlayer.play()
}
```

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.

Playing sound effects

Playing simple sounds is even easier. We will use `SKAction` objects to play sounds on specific events, such as 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:

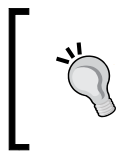
1. Open `Coin.swift` and add a new property to the `Coin` class to cache a coin sound action:

```
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.runAction(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 a `SKAction` object for each playback.

Adding the power-up and hurt sound effects to the Player class

We will play an exciting 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.runAction(hurtSound)
```

3. Find the `starPower` function and add this line at the bottom:

```
// Play the powerup sound:
self.runAction(powerupSound)
```

Playing a sound when the game starts

Lastly, we will play a sound when the game starts. Follow these steps to play this sound:

1. Open `GameScene.swift`. We will play this sound effect from the `didMoveToView` 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` `didMoveToView` function:

```
// Play the start sound:
self.runAction(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.

Checkpoint 9-B

Download my project to this point at this URL:

<http://www.thinkingswiftly.com/game-development-with-swift/chapter-9>

Summary

We have taken large steps towards finishing the game in this chapter. We learned 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. The game now feels finished; we are very nearly ready to publish our game.

One final step remains: we will explore integration with the Apple Game Center to track high scores and achievements in *Chapter 10, Integrating with Game Center*. Game Center integration encourages your players to keep playing and improving. They will be able to see their own best score, view the top scores from around the world, and challenge their friends to beat their best effort.

10

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.



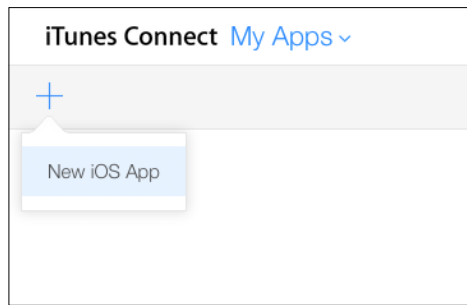
The topics in this chapter include:

- Registering an app with iTunes Connect
- Authenticating the player's Game Center account in our app
- Opening Game Center from the `MenuScene` class
- Adding a leaderboard
- Creating and awarding achievements

Registering an app with iTunes Connect

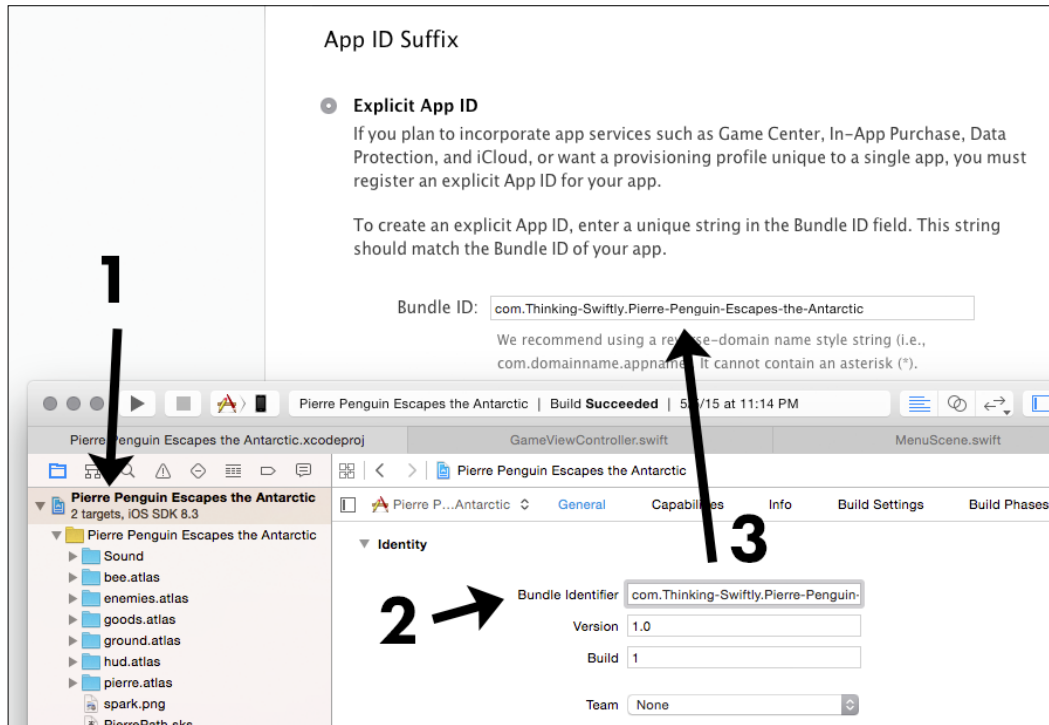
Since Apple will be storing our high scores and achievements on their centralized servers, we need to communicate to Apple that we need Game Center for our app. The first step is to create a record for our app on the iTunes Connect website. Follow these steps to create an iTunes Connect record:

1. In a web browser, navigate to <http://itunesconnect.apple.com>.
2. Sign in with your Apple developer account information.
3. When you reach the **iTunes Connect** dashboard, click on the **My Apps** icon.
4. Towards the upper left, click on the **+** symbol and select **New iOS App**, as shown here:




5. In the subsequent dialogue, locate the link at the bottom that says **Register a new bundle ID on the Developer Portal**. Click this link to create a bundle ID for your app.
6. You will arrive on a page titled **Registering an App ID**. This page may appear overwhelming at first, but you only need to fill out two fields. First, enter the name of your app in the **App Description** section.

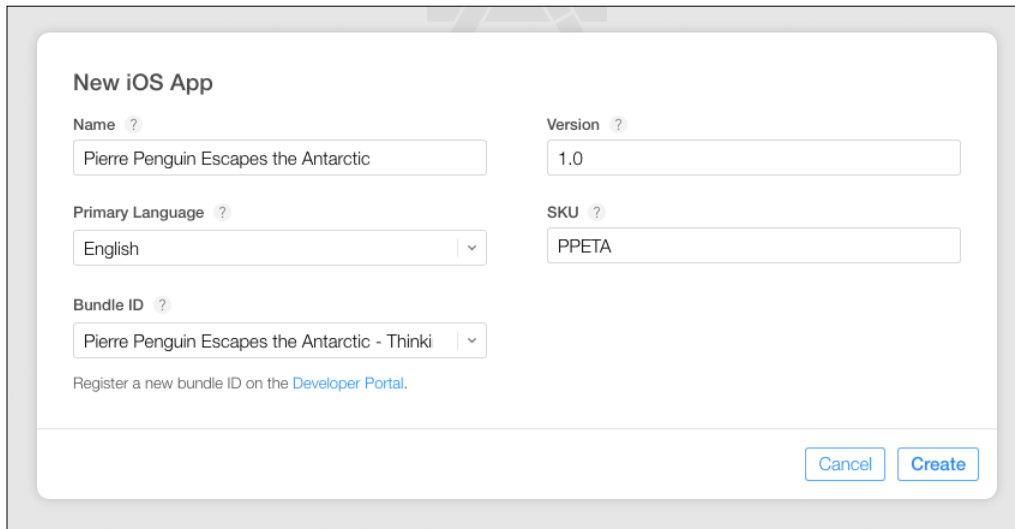
7. Scroll down to the **App ID Suffix** section. Make sure to select **Explicit App ID** and then enter the **Bundle ID** field from your Xcode project settings, as shown here:



8. Scroll down to the **App Services** section and double-check that the Game Center option is already checked.
9. At the bottom of the page, click **Continue**. Then click **Submit** on the subsequent confirmation page.
10. You can now close this tab and return to iTunes Connect, picking up where you left off on the new iOS app screen.

 It can take some time before the bundle ID you just created shows up in iTunes Connect. If this happens, take a break and try again after a few moments.

11. Enter the **Name** of your app, the **Primary Language**, the **Version**, and **SKU** (which is not visible to the public). Then select the **Bundle ID** you just created, as shown in this screenshot:



New iOS App

Name ?
Pierre Penguin Escapes the Antarctic

Version ?
1.0

Primary Language ?
English

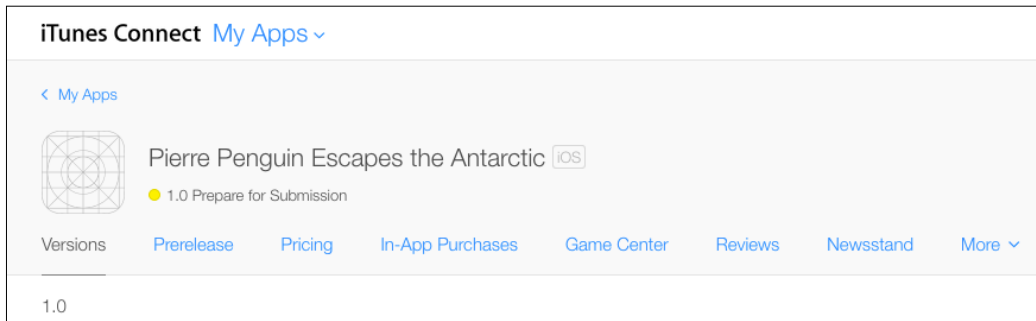
SKU ?
PPETA

Bundle ID ?
Pierre Penguin Escapes the Antarctic - Thinki

Register a new bundle ID on the [Developer Portal](#).



Cancel Create

12. Click **Create** in the lower right. You should now see an overview for your app in iTunes Connect, which will look something like this screenshot:



iTunes Connect My Apps

< My Apps

 Pierre Penguin Escapes the Antarctic 

● 1.0 Prepare for Submission

Versions Prerelease Pricing In-App Purchases Game Center Reviews Newsstand More

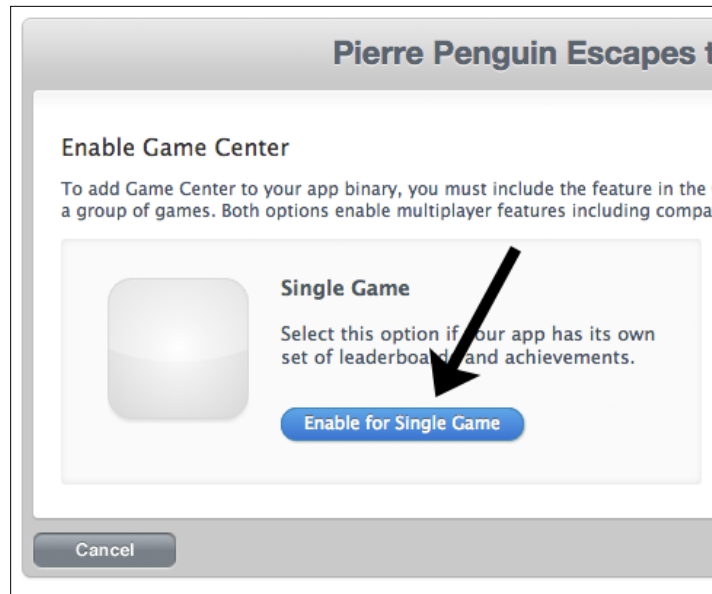
1.0

Congratulations, not only are we closer to configuring Game Center, we have also taken the first step towards preparing our app for submission to the app store!

Configuring Game Center

Now that we have an iTunes Connect app record, we can tell Apple more about how we want to use Game Center in our game. Follow these steps to configure Game Center:

1. On your app page, click the link for Game Center in the top navigation.
2. Choose **Enable for Single Game**, as shown here:



3. You will see a screen allowing you to create new leaderboards and achievements for your game. Perfect! We will use this page later in the chapter.

We have informed Apple that we want to use Game Center in our game. Next, we need to create a sandbox user account for testing purposes.

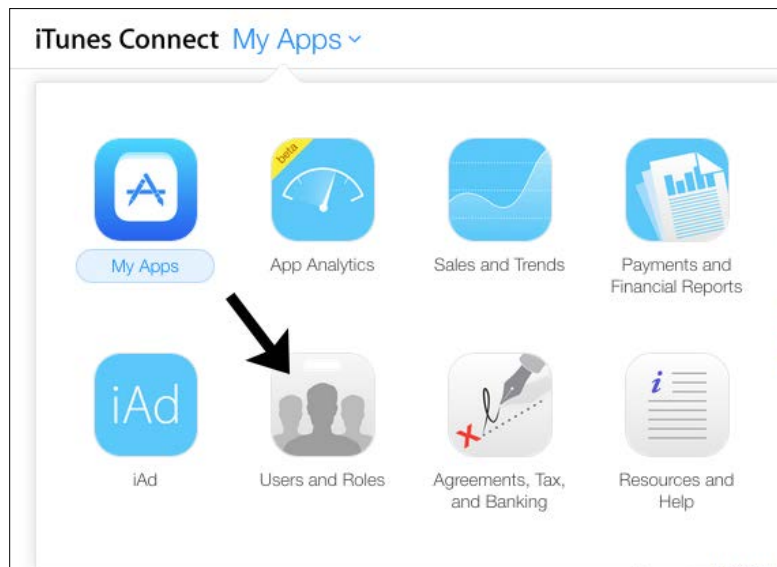
Creating a test user

Game Center uses separate test servers during app development, so we will not be able to use our real Apple ID to log in to Game Center while we are testing. Instead, we will create a sandbox account in iTunes Connect.

The website of iOS Developer Library states "Always create new test accounts to test your game in Game Center. Never use an existing Apple ID."

Follow these steps to create a Game Center sandbox account for testing:

1. In **iTunes Connect**, use the dropdown menu in the upper left to select **Users and Roles**, as shown here:



2. Once you are on the **Users and Roles** page, click on **Sandbox Testers** in the navigation bar at the top of the screen.
3. As directed on the **Sandbox Testers** page, click the + icon to add a new user.

4. Fill out the test user's information to your liking. Here is how I filled out my test user's information:

Tester Information

First Name

Joe

Last Name

McTest

Email

joe@mctest.com

Password

••••••••

Confirm Password

••••••••

Secret Question

Are flying penguins amazing?

Secret Answer

so much

Date of Birth

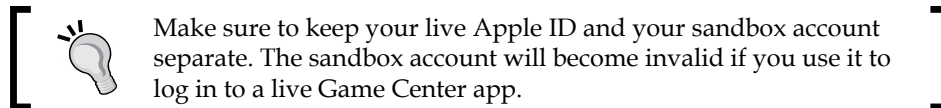
March

6

App Store Territory

United States

5. Click the **Save** button to create the new user.



Great! That is all we need to begin implementing Game Center into our game. The next step is to integrate Game Center with our game code. We will start by authenticating the player's Game Center account when they open our app.

Authenticating the player's Game Center account

As soon as our app starts, we will check if 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. We will be working in the `GameViewController` class, so open `GameViewController.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. 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 : UIViewController!,
         error : NSError!) -> Void in
        if viewController != nil {
            // They are not logged in, show the log in:
            self.presentViewController(viewController,
                                       animated: true, completion: nil)
        }
        else if localPlayer.authenticated {
            // They authenticated successfully!
            // We will be back later to create a
            // leaderboard button in the MenuScene
        }
        else {
            // Not able to authenticate, skip Game Center
        }
    }
}
```

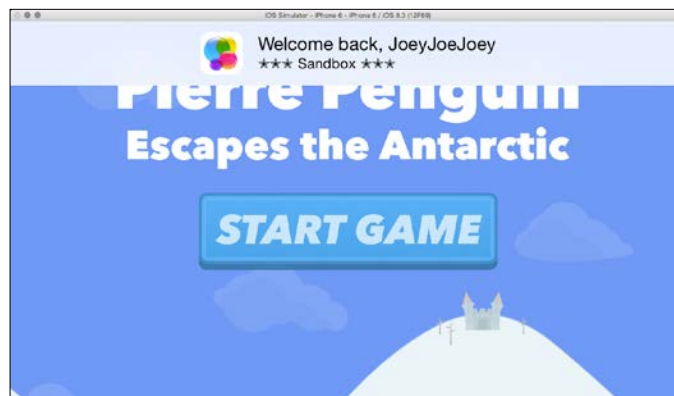
4. At the bottom of the `GameViewController` class `viewWillLayoutSubviews` function, add a call to the new `authenticateLocalPlayer` function you just created:

```
authenticateLocalPlayer(menuScene)
```

Run your project. You should see Game Center animate in, asking for your credentials, as seen here:



Great! Remember to use your sandbox account. The first time you log in, Game Center will ask a few extra questions to set up your account. Once you finish with the Game Center form, you should return to the main menu, with a small banner animating in and out from 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 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 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 `MenuScene.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 `MenuScene` 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 MenuScene: SKScene, GKGameCenterControllerDelegate {
```

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 `MenuScene` class named `createLeaderboardButton` as shown here:

```
func createLeaderboardButton() {  
    // Add some text to open the leaderboard  
    let leaderboardText = SKLabelNode(fontNamed:  
        "AvenirNext")  
    leaderboardText.text = "Leaderboard"  
    leaderboardText.name = "LeaderboardBtn"  
    leaderboardText.position = CGPoint(x: 0, y: -100)  
    leaderboardText.fontSize = 20  
    self.addChild(leaderboardText)  
}
```

5. We will call our `createLeaderboardButton` function from the `didMoveToView` function if the player is already authenticated with Game Center. This creates the button for players who return to the main menu after playing a game. Add the following code to the bottom of the `didMoveToView` function:

```
// If they're logged in, create the leaderboard button  
// (This will only apply to players returning to the menu)  
if GKLocalPlayer.localPlayer().authenticated {  
    createLeaderboardButton()  
}
```

6. Next, we will create the function that actually opens the Game Center. Add a new function named `showLeaderboard`, as shown here:

```
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.showViewController(gameCenter,
            sender: self)
        gameViewController.navigationController?
            .pushViewController(gameCenter, animated: true)
    }
}
```

7. We need to add another function to adhere to the `GKGameCenterControllerDelegate` protocol. This function is named `gameCenterViewDidFinish`, and the Game Center will invoke it when the player clicks the **Done** button in Game Center. Add the function to the `MenuScene` class, as shown here:

```
// This hides the game center when the user taps 'done'
func gameCenterViewControllerDidFinish
    (gameCenterViewController:
        GKGameCenterViewController!) {
    gameCenterViewController.dismissViewControllerAnimated(
        true, completion: nil)
}
```


8. To wrap up the MenuScene code, we need to check for taps on our leaderboard button in the touchesBegan function to invoke showLeaderboard. Update the touchesBegan function if block as shown in the following (new code in bold):

```
if nodeTouched.name == "StartBtn" {
    self.view?.presentScene(GameScene(size: self.size))
}
else if nodeTouched.name == "LeaderboardBtn" {
    showLeaderboard()
}
```

9. Next, open GameViewController.swift and locate the authenticateLocalPlayer function.
10. Update the block where the player authenticated successfully to call our new createLeaderboardButton function in the MenuScene class. This creates the leaderboard button for newly authenticated people as they start the app. The code is shown here (new code in bold):

```
else if localPlayer.authenticated {
    // They authenticated successfully
    menuScene.createLeaderboardButton()
}
```

Good work. Run the project and you should see a leaderboard button appear in the menu after Game Center authenticates, as shown here:



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.

Checkpoint 10-A

To download my project to this point, visit this URL:

<http://www.thinkingswiftly.com/game-development-with-swift/chapter-10>

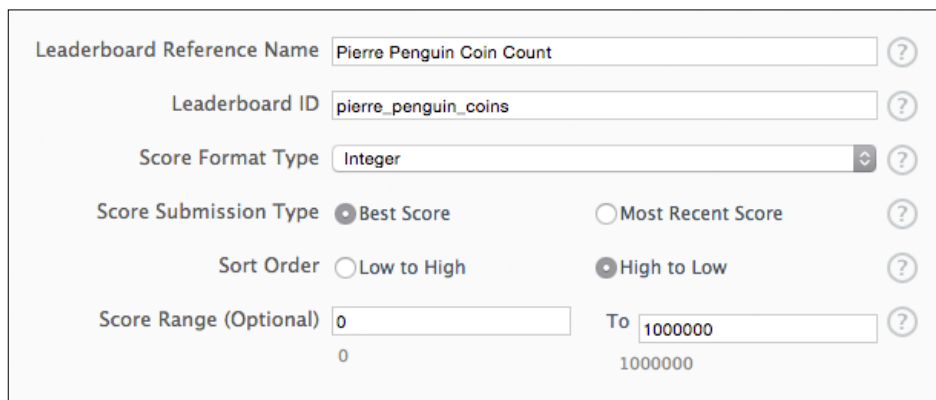
Adding a leaderboard of high scores

We will submit the player's scores to the Game Center servers every time they finish a game. The first step is to register a new leaderboard on iTunes Connect.

Creating a new leaderboard in iTunes Connect

First, we will create our leaderboard in iTunes Connect. We can then connect to this leaderboard from our code and send new scores. Follow these steps to create the leaderboard record in iTunes Connect:

1. Log back in to iTunes Connect and navigate into the Game Center page for your app.
2. Locate and click the button that says **Add Leaderboard**.
3. The next page asks you what type of leaderboard you want to create. Choose **Single Leaderboard**.
4. Fill out the information for your leaderboard. You can reference my example here:

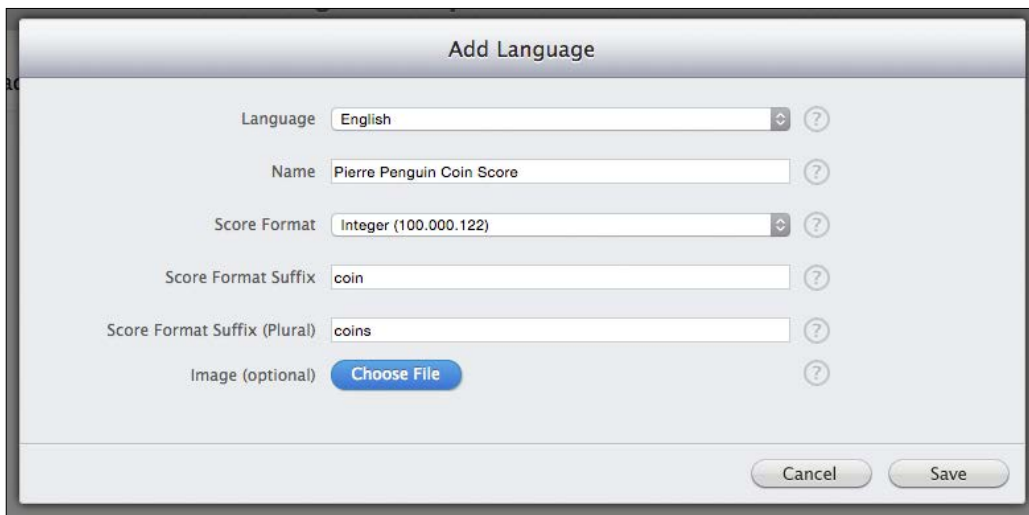


The screenshot shows the 'Add Leaderboard' form in iTunes Connect. The form is titled 'Leaderboard Reference Name' and contains the following fields and options:

- Leaderboard Reference Name:** Pierre Penguin Coin Count
- Leaderboard ID:** pierre_penguin_coins
- Score Format Type:** Integer
- Score Submission Type:** ☒ Best Score, ☐ Most Recent Score
- Sort Order:** ☐ Low to High, ☒ High to Low
- Score Range (Optional):** From 0 to 1000000

Let's take a look at each field:

- **Reference Name** is an internal use name for leaderboard listings in iTunes Connect
 - **Leaderboard ID** is a unique identifier we will reference from our code
 - **Score Format Type** describes the type of data you will be passing in (most commonly integer data for high scores)
 - Normal leaderboards use a **Score Submission Type** of **Best Score**, with a **Sort Order** of **High to Low**
 - **Score Range** is an anti-cheating measure you can use to block obviously false scores from showing up on the leaderboard
5. Next, click the **Add Language** button. You will choose a name and score formatting for your leaderboard on this screen. These fields are largely self-explanatory, but you can reference my example here:



6. Click **Save** twice (once for the language dialogue and once on the leaderboard screen).

You should be back on the Game Center page with your new leaderboard listed in the leaderboards section. Next, we will push new scores into the leaderboard from our game code.

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 `GameScene.swift`.
2. Add an `import` statement at the top so we can use the GameKit framework in this file:

```
import GameKit
```

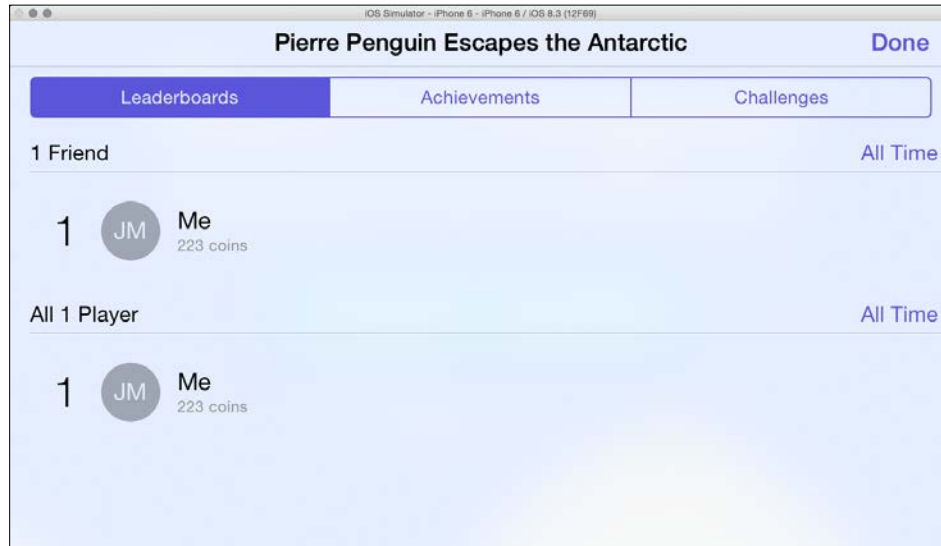
3. Add a new function in the `GameScene` class named `updateLeaderboard`, as shown here:

```
func updateLeaderboard() {
    if GKLocalPlayer.localPlayer().authenticated {
        // Create a new score object, with our leaderboard:
        let score = GKScore(leaderboardIdentifier:
            "pierre_penguin_coins")
        // Set the score value to our coin score:
        score.value = Int64(self.coinsCollected)
        // Report the score (wrap the score in an array)
        GKScore.reportScores([score],
            withCompletionHandler:
            {(error : NSError!) -> Void in
                // The error handler was used more in previous
                // versions of iOS, it would be unusual to
                // receive an error now:
                if error != nil {
                    println(error)
                }
            })
    }
}
```

4. In the `GameScene` class `GameOver` function, call the new `updateLeaderboard` function:

```
// Push their score to the leaderboard:
updateLeaderboard()
```

Run the project and play through a game to send a test coin score to the leaderboard. Then, tap back to the menu scene and click the **Leaderboard** button to open Game Center within your game. You should see your first score appear in the leaderboard! It will look something like this:



Great work – you have implemented your first Game Center leaderboard. Next, we will follow a similar series of steps to create an achievement for collecting 500 coins in one game.

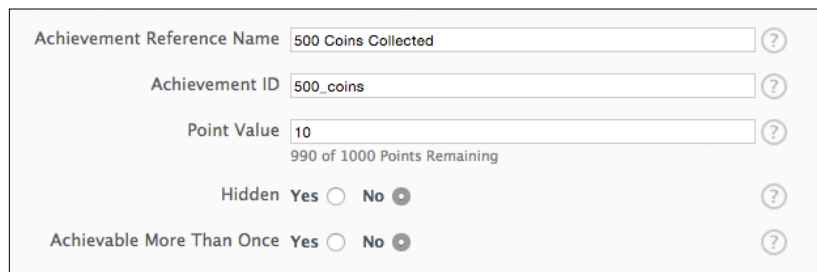
Adding an achievement

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 collecting 500 coins without dying.

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 the record:

1. Log into iTunes Connect and navigate to the Game Center page for your app.
2. Underneath the leaderboards list, locate and click the **Add Achievement** button.
3. Fill out the information for your achievement. Here are my values:



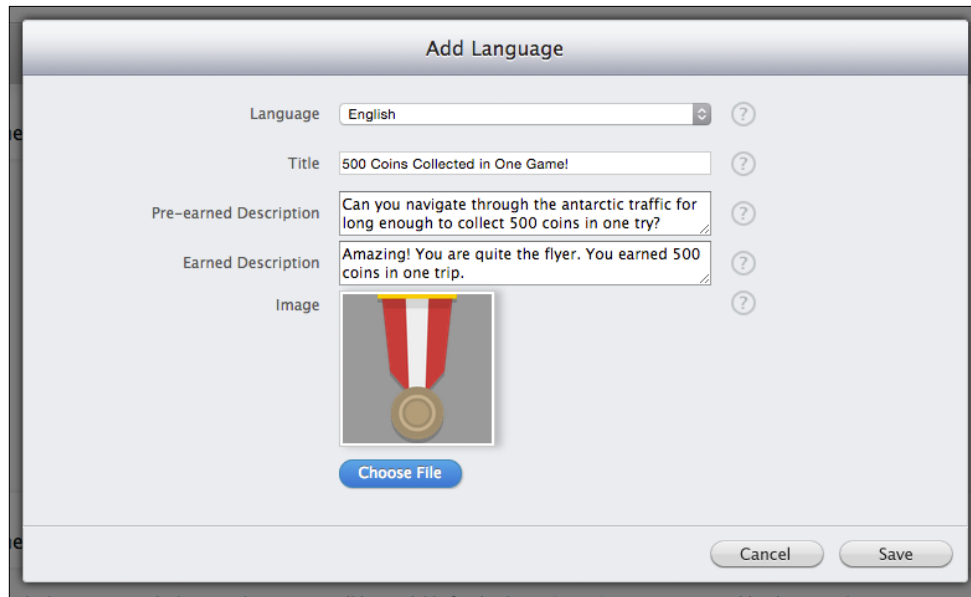
A screenshot of the iTunes Connect 'Add Achievement' form. The form is light gray with white input fields. It contains the following fields and controls:

- Achievement Reference Name:** A text input field containing '500 Coins Collected' and a question mark icon to its right.
- Achievement ID:** A text input field containing '500_coins' and a question mark icon to its right.
- Point Value:** A text input field containing '10' and a question mark icon to its right. Below this field, it says '990 of 1000 Points Remaining'.
- Hidden:** A label followed by 'Yes' with an unselected radio button and 'No' with a selected radio button, and a question mark icon to the right.
- Achievable More Than Once:** A label followed by 'Yes' with an unselected radio button and 'No' with a selected radio button, and a question mark icon to the right.

Let's take a look at each field:

- **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 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

4. 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 512x512 or 1024x1024. You can find the one I used in our **Assets** bundle download, in the **Extras** folder, `gold-medal.png`. Here are my values:



5. Click **Save** twice (once for the language dialogue 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. 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 500 coin achievement:

1. Open `GameScene.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 we can use GameKit. If you have already implemented the leaderboard, you can skip this:
3. Add a new function to the GameScene class named `checkForAchievements`, as shown here:

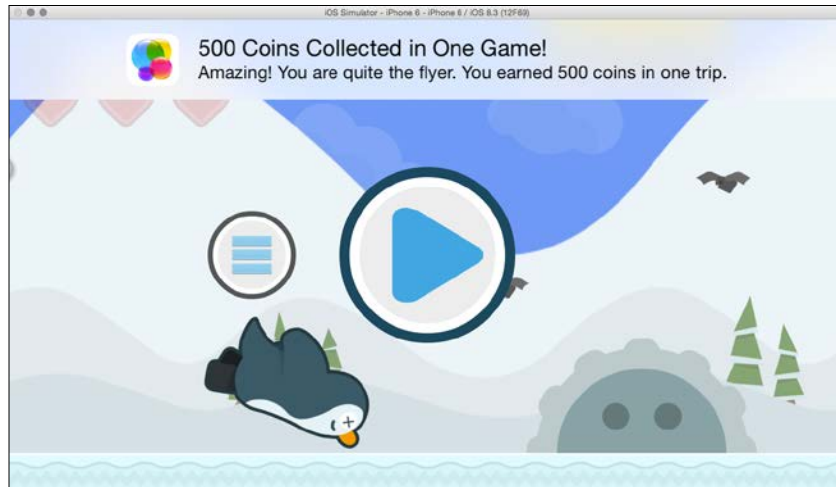
```
import GameKit

func checkForAchievements() {
    if GKLocalPlayer.localPlayer().authenticated {
        // Check if they earned 500 coins in this game:
        if self.coinsCollected >= 500 {
            let achieve = GKAchievement(identifier:
                "500_coins")
            // Show a notification that they earned it:
            achieve.showsCompletionBanner = true
            achieve.percentComplete = 100
            // Report the achievement!
            GKAchievement.reportAchievements([achieve],
                completionHandler:
                    {(error: NSError!) -> Void in
                        if error != nil {
                            println(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, if you dare, complete a 500 coin fly through. When your game ends, you should see a banner proclaiming your new achievement conquest, as shown here:



Great work! You have implemented Game Center leaderboards and achievements into your game.

Checkpoint 10-B

To download my project to this point, visit this URL:

<http://www.thinkingswiftly.com/game-development-with-swift/chapter-10>

Summary

Integrating with Game Center is a great feature for your players. In this chapter, we learned how to create an iTunes Connect record for our app, 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!

We are officially finished working on the game itself. In the next chapter, we will prepare our app for publication, upload the code for Apple to review, and revisit what we have learned while creating our great game. Everything is coming together and we are ready to take the final step to publish our game. Congratulations!

11

Ship It! Preparing for the App Store and Publication

What a grand journey! We have stepped through each component of the game development process in Swift and we are finally ready to share our hard work with the world. We need to prepare our project for publication by finishing the assets associated with it: the assorted app icons, the launch screen, and the screenshots for the App Store. Then, we will fill out the description and information for our app in iTunes Connect. Finally, we will use Xcode to upload a production archive build and submit it to the Apple review process. We are very close to seeing our game in the App Store!

While I can show you the general path you can use to submit your app, this process is constantly changing as Apple updates iTunes Connect. In addition, every app has unique aspects that may require a variation on the path I demonstrate in this chapter. I encourage you to browse Apple's official documentation in the iOS Developer Library and refer to Stack Overflow for updated answers. You can locate the iOS Developer Library by browsing to <https://developer.apple.com/library/ios>.

Topics in this chapter include:

- Finalizing assets: app icons and the launch screen
- Finalizing iTunes Connect information
- Configuring pricing
- Uploading our project from Xcode
- Submitting for review in iTunes Connect

Finalizing assets

There are several peripheral assets we need before we can publish our game. We will create a set of app icons, redesign the launch screen, and take screenshots for each device we support for the App Store previews.

Adding app icons

Our app requires multiple sizes of our app icon to display correctly in the App Store and the various iOS devices we support. You can find a sample icon set in the provided assets bundle, in the `Icon` folder.

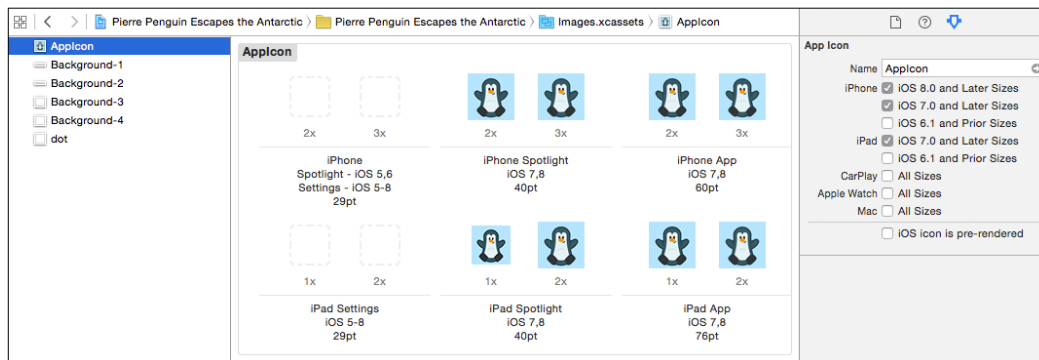


You should design your icon to be 1024 pixels wide by 1024 pixels tall and then resize down for the other variations. Make sure to check each variation to ensure it looks good after resizing. You will upload this large size directly to iTunes Connect later in the chapter.

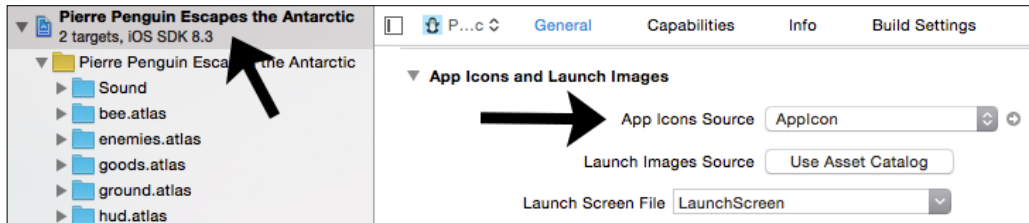
The best way to integrate your icons into your project is to use the `Images.xcassets` asset bundle, preconfigured for app icons, that comes along with new projects. We will drag and drop our icons into this file to bring them into the project.

Follow these steps to add our icons to the project:

1. In Xcode, open the `Images.xcassets` file and locate the **AppIcon** image set in the left pane.
2. Drag and drop the images from the assets bundle into the corresponding icon slots. You can drag your files in as a group and Xcode will process them into the correct slots. You can ignore the icon slots for Settings icons, since our app does not integrate with iOS settings. When you are finished, your icon image set will look something like this:



- Go into your general project settings by clicking on your project in the project navigator. Locate the **App Icons Source** setting and make sure it is set to **AppIcon** to use the image bundle, as shown here:



We are finished adding our icons in Xcode. We will need to upload a few more icon sizes to iTunes Connect later. You can run your project on a real device to see your new icons in action.

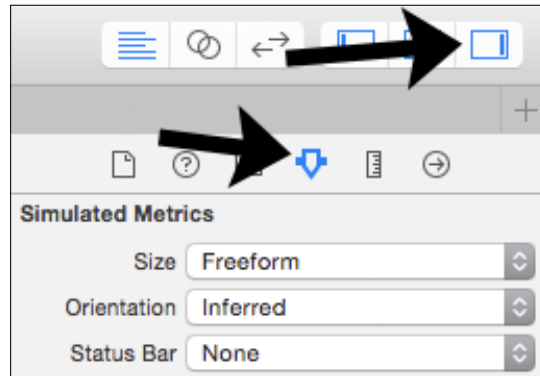
Designing the launch screen

When a user taps your icon on their device, iOS shows your app's launch screen as an extremely fast-loading simple preview. This creates the illusion that your app loads almost instantly. The player gets immediate feedback from their tap while your app actually loads in the background. This is not the place to add logos, branding, or information of any kind. The goal is to create a very simple screen that looks like your app before the content is in place. For Pierre Penguin, we will implement a simple blank sky blue background that looks like the main menu before it has any content.

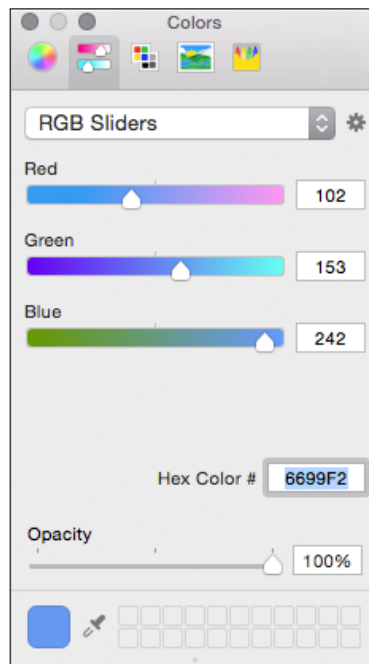
Follow these steps to set up your sky blue launch screen:

- Open the `LaunchScreen.xib` file in Xcode. You will see the launch screen open in the interface builder.
- Select each preexisting text element and delete each one with the *Delete* key on your keyboard.
- Select the entire frame by clicking anywhere in the white space of the launch screen.

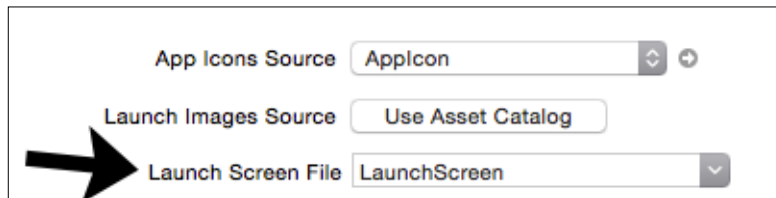
4. Make sure you have your Utilities bar open on the right hand side of Xcode, and open the **Attributes Inspector**, as demonstrated here:



5. Locate the background color setting in the right bar, then click on the existing white color option to open a color selection window.
6. Choose the color sliders tab, and enter the RGB value 102, 153, 242, as shown here:



7. You should see the entire frame turn the sky blue color from our game.
8. Next, enter your general settings by clicking the project name in the project navigator. As you did before for the app icons, make sure the **Launch Screen File** setting is **LaunchScreen**:



Perfect! When we run our app, we will see the sky blue color immediately, providing a smoother transition between the home screen and our fully loaded app.

Taking screenshots for each supported device

Fun screenshots will make your game stand out in the App Store. I created some sample screenshots for Pierre Penguin in the assets bundle's `Screenshots` folder. You will need to create separate screenshots for each iOS device you want to support.

Screenshots must be JPG or PNG files. You can use my example screenshots as templates for each size of screenshot you need, or follow this table:

Device size	Screenshot size for a full screen game
3.5" (required)	960x640 pixels
4" (required)	1136x640 pixels
4.7"	1334x750 pixels
5.5"	2208x1242 pixels
iPad (all versions)	2048x1536 pixels

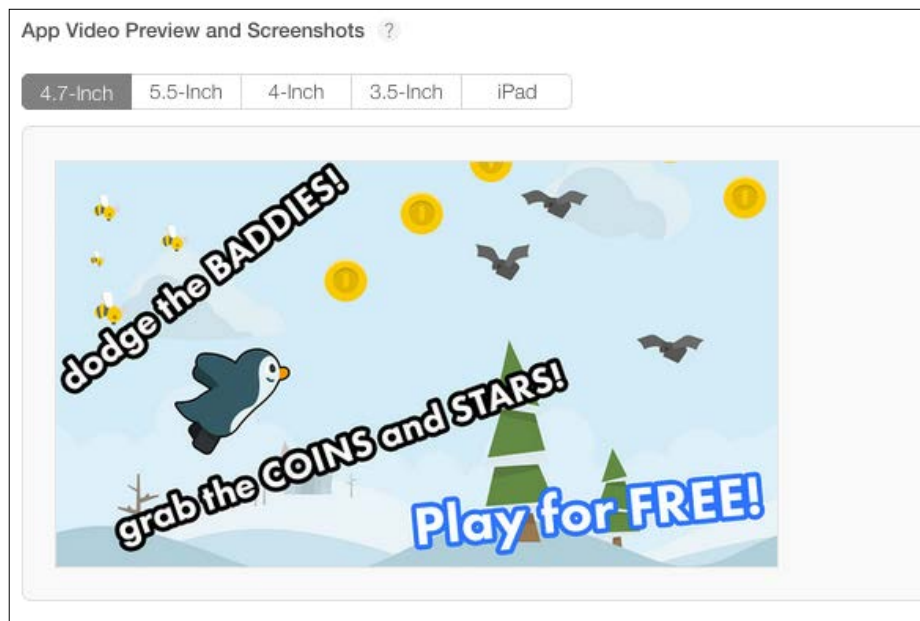
Once your screenshots are prepared, you are ready to finalize your game settings in iTunes Connect. We will complete the iTunes Connect details next.

Finalizing iTunes Connect information

iTunes Connect controls our app's details in the App Store. We will use iTunes Connect to create a description for our game, add the screenshots we want to display in the App Store, and configure our pricing information and project settings.

Follow these steps to fill out your iTunes Connect information:

1. Open the iTunes Connect website in your Web browser. Browse to the **My Apps** section, and then click on your game. iTunes Connect will take you to the **Versions** tab of your game's page.
2. We will start with the screenshots. Drag and drop each device screenshot into the corresponding slot in the **App Video Preview and Screenshots** section, as seen here:



3. Scroll down and fill out the information in the next section: **Name**, **Description**, **Keywords**, and associated URLs. These fields are self-explanatory, but you can always click on the small gray question mark circles for detailed information from Apple.



A word on keywords: users will find your app more easily if you have strong, accurate keywords. Try to use phrases you think people may type in to the App Store that should lead them to your game. You are limited to 100 characters, so omit spaces between keywords.

4. Next, scroll down to the **General App Information** section. Here you will upload your app icon, enter a version number (1.0), pick the App Store category for your app (**Games**), and provide your address information. Again, click on the gray question mark circles if you need further information on any of these fields.
5. Scroll down and locate **Game Center**, then flip the slider to the on position. You will need to add your leaderboard and achievements by clicking the blue plus icons, as shown here:

Game Center ☒

Leaderboards [+](#)

Reference Name	Leaderboard ID	Type
Pierre Penguin Coin Count	pierre_penguin_coins	Single

Achievements [+](#)

Reference Name	Achievement ID	Points
500 Coins Collected	69970772	10

6. Finally, scroll to the **App Review Information** section and fill out your contact information again. This is for the Apple employee reviewing your app in the event they need more information. You can also select whether you want your game to release to the App Store automatically after approval, or wait for you to release it manually later to coincide with your marketing efforts.
7. Click on **Save** in the upper right corner.

Configuring pricing

Pierre Penguin is going to be free for all to play, but you can choose from many pricing strategies for your games.



Apple is constantly updating iTunes Connect and I expect that the Pricing section will soon receive an overhaul. Your experience may not match these steps exactly.

Follow these steps to set the price for your game:

1. On the iTunes Connect page for your game, click the **Pricing** tab in the top navigation bar.
2. Choose an **Availability Date**, **Price Tier**, and educational discount. Here are my settings for reference:

Select the availability date and price tier for your app.

Availability Date ?

Price Tier ?
[View Pricing Matrix ►](#)

Price Tier Effective Date ?

Price Tier End Date ?

Price Tier Schedule		
Price Tier	Price Effective Date	Price End Date
Free	Existing	None

3. Click on **Save** in the lower right.

Perfect! Our iTunes Connect information is complete and ready to submit to the App Store review process. Now we just need to finalize and upload our build in Xcode.

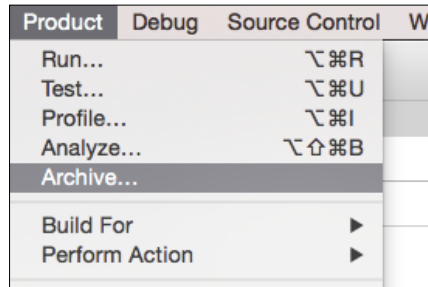


If you want to charge for your game then you will need to fill out the contracts and banking information found in the **Agreements, Tax, and Banking** section of iTunes Connect.

Uploading our project from Xcode

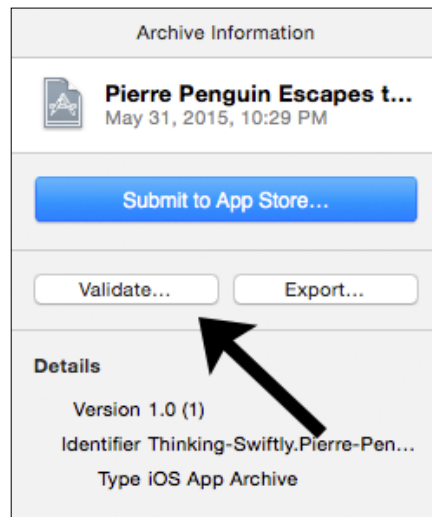
Next, we will create a final build of our game, validate that it contains everything it needs for the App Store, and upload the bundle to iTunes Connect.

First, we will create the deployment archive for our game. When you are happy with your project, use the **Product** menu and select **Archive...**, as shown here:

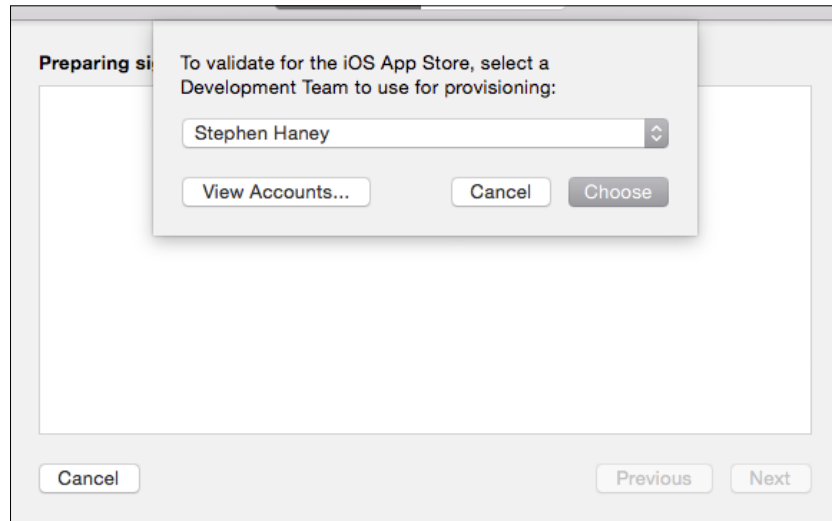


Once the process finishes, Xcode will open your archive list. From here, you can validate your app to make sure it includes all the requisite assets and profiles it needs to be on the App Store. Follow these steps to validate your app and upload it to iTunes Connect:

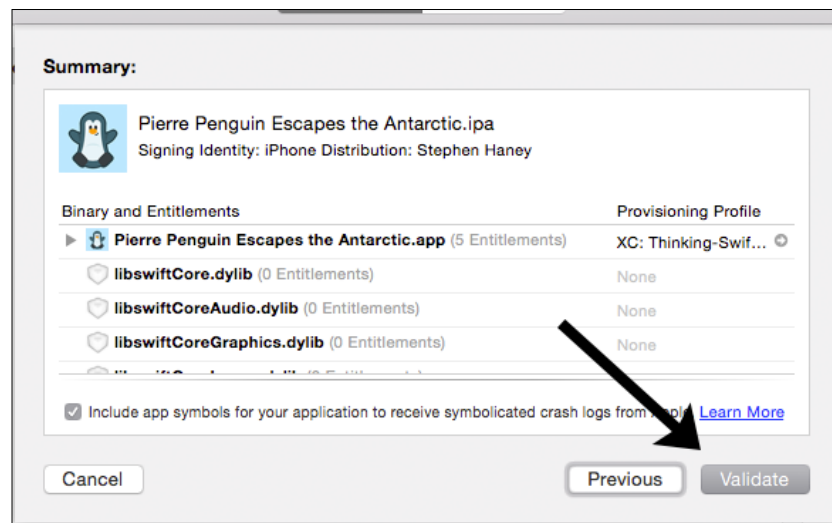
1. Click on the **Validate** button, as shown in the following screenshot, to validate your app.



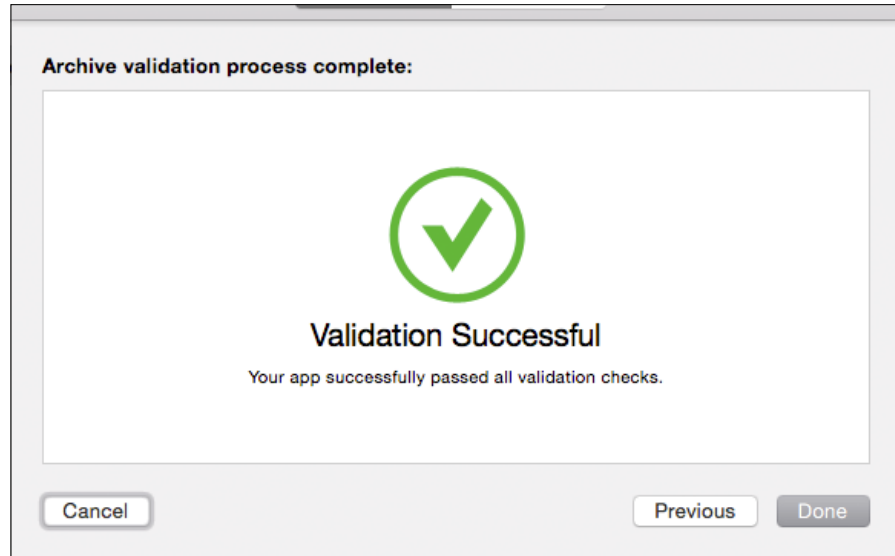
- The following screen will ask you to choose a development team for your app. If you are a solo developer, you will simply select your own name, as shown here:



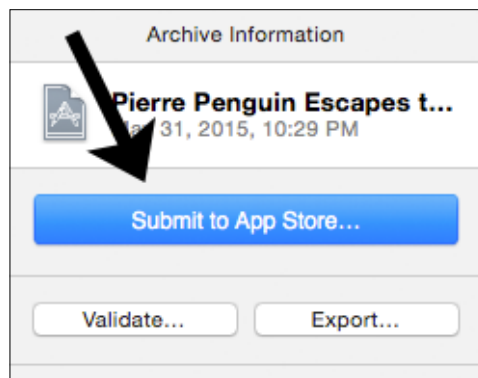
- Xcode will create a distribution provisioning profile for you, and then take you to a summary screen. Simply click the **Validate** button:



4. Xcode will proceed to validate that everything is ready for the App Store, which may take a few moments. After it completes, you should see a success message, as shown in the following screenshot. If you receive any errors, you may be missing an asset or profile that the App Store requires. Read and respond to the error message, and refer to the iOS Developer Library, Internet searches, or Stack Overflow for further assistance.



5. Click **Done**, then click the blue **Submit to App Store** button to upload the archive to iTunes Connect, as shown here:



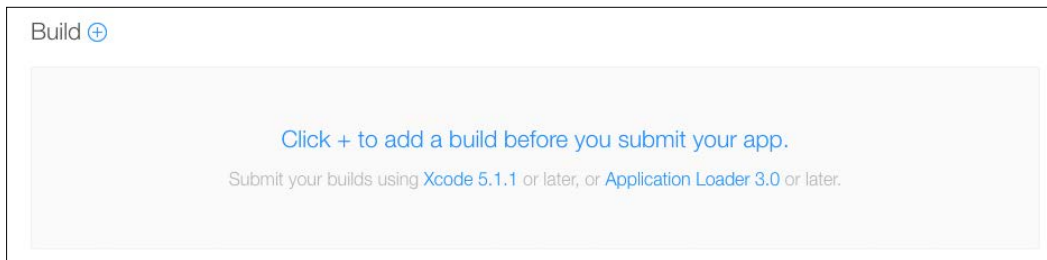
6. You will need to click through the validate steps again, and then finally click **Submit**. Xcode will then upload your app to iTunes Connect and display another success message.

Congratulations! You have successfully uploaded your app to Apple. We are almost finished submitting our app. Next, we will return to iTunes Connect to push our app into the review and approval process.

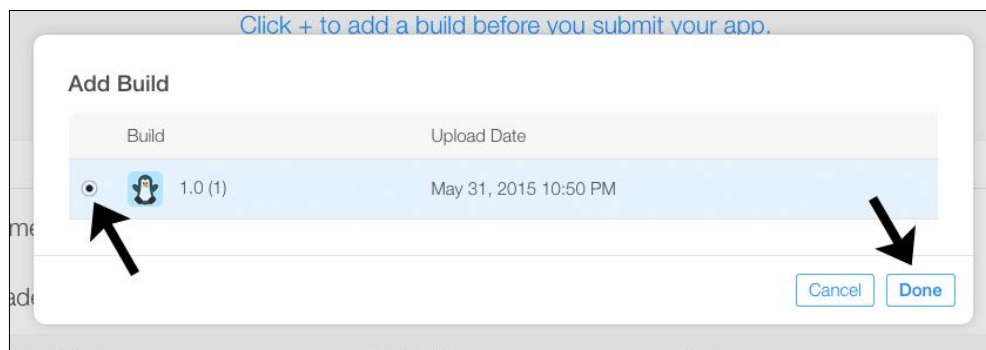
Submitting for review in iTunes Connect

We are finished prepping our project and we are ready to push our hard work into the Apple review process. Follow these steps to submit your app to Apple:

1. Return to the iTunes Connect website and browse to your game's page (on the **versions** tab).
2. Scroll down to the **Build** section, and select **Click + to add a build before you submit your app**.



3. Use the radio button to select the archive you just uploaded, then click **Done**, as shown here. It can take a few minutes (or sometimes hours) for the uploaded build to show in this list:



4. Click **Save** in the upper right corner and the **Submit for Review** button should light up in blue:



5. Click **Submit for Review** and iTunes Connect will show the **Submit for Review** page with three final questions about your game. Apple wants to know if your app uses cryptography, third party content, or advertising. I answered no to all three questions for Pierre Penguin. It is important to answer these questions accurately, so use the question mark icons in iTunes Connect for more information if you are unsure how to proceed.
6. After you answer the **Submit for Review** questions, click on **Submit** in the upper right. This is the final step of the submission process.

If your app submits successfully, iTunes Connect will return to the versions tab of your app's page. You will see the app status change to **Waiting For Review**, as shown here:



Terrific! We have submitted our game to Apple. It is typical for the review process to take 7-14 days. Do not be discouraged if your game comes back without approval, Apple commonly requires developers to correct small issues and resubmit their apps. You are on your way to seeing your game in the App Store!

Summary

Many indie developers struggle with the final steps of publishing their games. If you are ready to publish a game, you are doing a great job! In this chapter, we created app icons and our launch screen, finalized our App Store marketing information in iTunes Connect, used Xcode to archive and upload our game, and submitted our game to Apple for review. You should now be confident in your ability to publish your games to the App Store.

We accomplished a great deal in the course of this book: we assembled a complete Swift game from a new project template to publication. As we go our separate ways, I wish you tremendous luck in your future game development endeavors. My hope is that you are now confident in starting your own game projects with Swift. I look forward to seeing your creations in the App Store!

Index

A

achievement

- about 176
- adding 176
- creating, in iTunes Connect 177, 178
- updating, from code 178-180

anchorPoint property 23

app

- registering, with iTunes Connect 162-164

artwork, adding to sprite

- bee image, adding 25
- designing, for retina 27
- exceptional art 24
- free assets, downloading 24
- images, loading with SKSpriteNode 26

assets

- app icons, adding 182, 183
- art, collecting into texture atlases 30
- finalizing 182
- Images.xcassets, exploring 29
- launch screen, designing 183-185
- organizing 29
- screenshots, capturing for supported device 185

Automatic Reference Counting (ARC) 3

B

bats

- adding 79

Bat class

- adding 79, 80

bee

- collision, creating 54, 55

Bee class

- creating 40-42

bee texture atlas

- bee node, updating 30
- building 30
- iterating, through texture atlas frames 31, 32

blade

- adding 82

Blade class

- adding 82, 83

C

camera

- centering, on sprite 33
- enhancing 68-70

cast

- about 76
- bats, adding 79
- blade, adding 82
- coins, adding 84
- mad fly, creating 78
- power-up star, adding 76
- spooky ghost, adding 80

categories, assigning to game objects

- coins 116
- enemies 116
- ground 115
- player 115
- star power-up 116

categoryBitMask property 113

coins

- adding 84
- collecting 125-127

- Coin class**
 - creating 84, 85
- collision 112**
- collisionBitMask property 113**
- collision events**
 - implementing 111
- constant velocity, for Pierre**
 - endless ground loop, creating 71, 72
 - player progress, tracking 71
 - setting 70
- contact events**
 - about 112
 - adding, to game 114
 - categories, assigning to game objects 115
 - GameScene, preparing for 116-118
 - physics categories, setting up 114
- contactTestBitMask property 113**
- control scheme, for penguin**
 - gravity, fine-tuning 67
 - implementing 64-66
 - touchesBegan, in GameScene 66
 - wings, spreading 67
- Core Motion**
 - about 60
 - code, implementing 60-62
 - device movement, polling with 60

D

- damage animation 122, 123**
- damping 53**
- density 53**
- development environment**
 - setting up 6
 - Xcode 6
- device movement**
 - polling, with Core Motion 60
- dynamic physics bodies 52**

E

- edge physics bodies 53**
- encounters, in endless flying**
 - building 93, 94
 - first encounter, creating 94-98

- endless encounters**
 - building 102
 - EncounterManager class, updating 103
 - EncounterManager, wiring up in GameScene 106
 - metadata, storing in SKSpriteNode userData property 104, 105
 - spawning 101

F

- force 56**
- friction 53**

G

- game**
 - enhancements 5
 - finish line, crossing 5
 - marketing 5
 - monetizing 5
- game art**
 - adding, to sprite 24
- Game Center**
 - about 5
 - configuring 165
 - integrating with 5
 - opening, in game 170-172
 - player account, authenticating 167-169
 - test user, creating 166, 167
- game code**
 - structuring 4
- game objects**
 - testing 86, 87
- game over animation 124, 125**
- GameScene class**
 - didMoveToView function 12
 - renovating 49, 50
 - touchesBegan function 12
 - update function 12
- GameScene, preparing for contact events**
 - about 116-118
 - console output, viewing 118
 - contact code, testing 119
- Ghost class**
 - adding 81, 82
- Ground class**
 - adding 44, 45

H

heads-up display (HUD)

- about 131
- adding 131, 132
- implementing 132-136

I

Images.xcassets

- exploring 29

impulse

- using 54, 55

Integrated Development

Environment (IDE) 6

iTunes Connect

- app, registering with 162-164
- URL 162

iTunes Connect information

- finalizing 186, 187
- pricing, configuring 188
- project, uploading from Xcode 189-191
- review, submitting 192, 193

L

leaderboard, of high scores

- adding 173
- creating, in iTunes Connect 173, 174
- updating, from code 175, 176

levels

- building 5

levels, never-ending world

- designing, with SpriteKit scene editor 91, 92
- empty nodes, using as placeholders 93
- level data, separating from game logic 93

limitations, Swift

- less resources 3
- operating system compatibility 3

M

mad fly

- adding 78
- enemy assets, locating 78

MadFly class

- adding 78, 79

main menu

- building 147
- launching, on game start 150, 151
- menu nodes, creating 148-150
- menu scene, creating 148-150
- START GAME button, wiring up 151, 152

mass 53

menus

- building 5

music

- adding 157
- background music, playing 157, 158

N

never-ending level

- preparing 87, 88

never-ending world

- encounters, in endless flying 93, 94
- endless encounters, spawning 101
- generating 91
- levels, designing with SpriteKit scene editor 91, 92
- scenes, integrating into game 98-101
- star power-up, spawning 107

nodes

- moving, with physics bodies 55, 56

O

Objective-C 1

P

parallax background layers

- about 136
- background assets, adding 137
- background class, implementing 137-139
- backgrounds, wiring up in GameScene class 139-141

particle system, SpriteKit

- circle particle asset, adding 142
- harnessing 141

- particle emitter, adding to game 145
- path particle settings, configuring 144
- SpriteKit Particle File, creating 142, 143
- penguin texture atlas**
 - adding 47, 48
- physics bodies**
 - about 51
 - dynamic physics 52
 - edge physics 53
 - static physics 53
- physics simulation mechanics 52, 53**
- physics system**
 - about 51
 - dropping like flies 51
 - ground, solidifying 51
- player account, Game Center**
 - authenticating 167-169
- Player class**
 - Beekeeper 58
 - ground, moving 58
 - physics body, assigning to player 59
 - physics body shape, creating from texture 59
 - retrofitting 58
 - updating 58
- player damage 119**
- player health**
 - about 119
 - implementing 119-121
- player input**
 - reacting to 4
- players control**
 - about 57
 - camera, enhancing 68-70
- positioning**
 - about 22
 - alignment, with anchor points 23, 24
- power-up star**
 - adding 76
 - art assets, locating 76
 - implementing 127, 128
- project navigator**
 - organizing 85
- protocol**
 - following 40

R

- restart game menu**
 - about 152
 - adding 152
 - GameScene, informing when player dies 154
 - GameScene, wiring up for game over 154
 - HUD, extending 153
 - touch events, implementing 155
- restitution 53**
- retina, designing for**
 - about 27
 - ideal asset approach 27
 - implementing 28
 - texture atlases 27

S

- safety**
 - granting, on game start 146
- SKScene class 10**
- SKSpriteNode class**
 - building 18, 19
- sound**
 - adding 157
 - playing, on game start 159
 - sound assets, adding to game 157
- sound effects**
 - coin sound effect, adding to Coin class 158
 - playing 158
 - power-up and hurt sound effects, adding to Player class 159
- spooky ghost 80**
- sprite**
 - about 18
 - animation, adding to toolkit 20
 - artwork, adding 24
 - camera, centering on 33
 - centered behavior benefits 23
 - drawing 18
 - implementing 32
 - multiple animations, sequencing 21
 - recapping 22
 - SKSpriteNode class, building 18, 19

- SpriteKit**
 - about 4
 - particle system, harnessing 141
- SpriteKit collision vocabulary**
 - about 111
 - category masks, using in Swift 113
 - collision, versus contact 112
 - physics category masks 112, 113
- sprite onTap events**
 - touchesBegan, implementing in
GameScene 63
 - wiring up 62
- Star class**
 - adding 76-78
- star power-up**
 - spawning 107
- static physics bodies 53**
- Swift**
 - about 1
 - automatic memory management 3
 - features 2, 3
 - interoperability 2
 - limitations 3
 - prerequisites 3
 - strong typing 2
 - syntax 2
 - type inference 2
- Swift game**
 - creating 7, 8
 - demo code, clearing 12
 - demo code, examining 12
 - project, navigating 9
 - SpriteKit demo, exploring 9-11

T

- texture**
 - adding, to sprite 24
 - tiling 45
- texture atlases 27**
- touchesBegan**
 - implementing, in GameScene 63, 64

U

- UI**
 - building 5

W

- wire**
 - running, to ground 45, 46
- world node**
 - creating 33-36

X

- Xcode**
 - about 6
 - installing 6



**Thank you for buying
Game Development with Swift**

About Packt Publishing

Packt, pronounced 'packed', published its first book, *Mastering phpMyAdmin for Effective MySQL Management*, in April 2004, and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

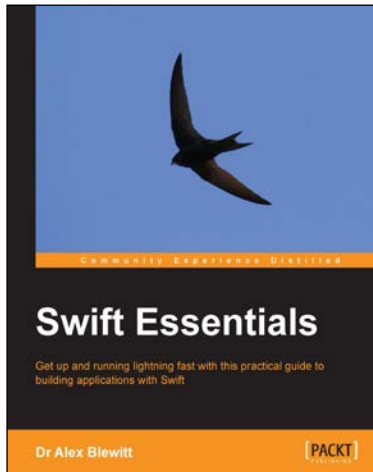
Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution-based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern yet unique publishing company that focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website at www.packtpub.com.

Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to author@packtpub.com. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, then please contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.



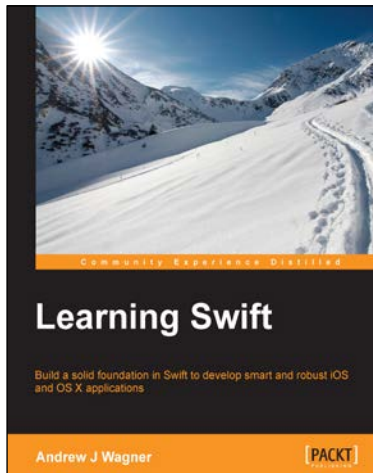
Swift Essentials

ISBN: 978-1-78439-670-1

Paperback: 228 pages

Get up and running lightning fast with this practical guide to building applications with Swift

1. Rapidly learn how to program Apple's newest programming language, Swift, from the basics through to working applications.
2. Create graphical iOS applications using Xcode and storyboard.
3. Build a network client for GitHub repositories, with full source code on GitHub.



Learning Swift

ISBN: 978-1-78439-250-5

Paperback: 266 pages

Build a solid foundation in Swift to develop smart and robust iOS and OS X applications

1. Practically write expressive, understandable, and maintainable Swift code.
2. Discover and optimize the features of Swift to write cleaner and better code.
3. This is a step-by-step guide full of practical examples to create efficient iOS applications.

Please check www.PacktPub.com for information on our titles



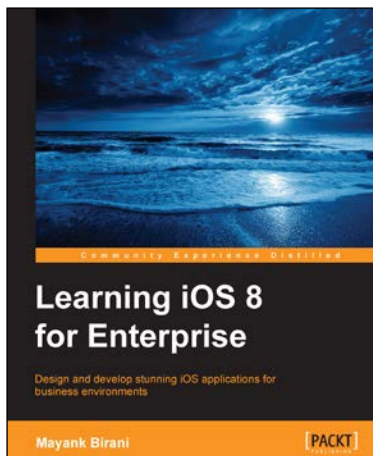
iOS Game Programming Cookbook

ISBN: 978-1-78439-825-5

Paperback: 300 pages

Over 45 interesting game recipes that will help you create your next enthralling game

1. Learn to create 2D graphics with Sprite Kit, game physics, AI behaviours, 3D game programming, and multiplayer gaming.
2. Use native iOS frameworks for OpenGL to create 3D textures, allowing you to explore 3D animations and game programming.
3. Explore powerful iOS game features through detailed step-by-step recipes.



Learning iOS 8 for Enterprise

ISBN: 978-1-78439-182-9

Paperback: 220 pages

Design and develop stunning iOS applications for business environments

1. Learn how to develop iPhone apps in an easier, step-by-step manner using real-world solutions.
2. Save time, learn faster, and gather knowledge of new technologies.
3. Work with powerful tools like Xcode and Simulator.

Please check www.PacktPub.com for information on our titles