



Community Experience Distilled

# iOS 7 Game Development

Develop powerful, engaging games with ready-to-use utilities from Sprite Kit

Dmitry Volevodz

**PACKT**  
PUBLISHING

# Table of Contents

[iOS 7 Game Development](#)

[Credits](#)

[About the Author](#)

[About the Reviewers](#)

[www.PacktPub.com](#)

[Support files, eBooks, discount offers and more](#)

[Why Subscribe?](#)

[Free Access for Packt account holders](#)

[Preface](#)

[What this book covers](#)

[What you need for this book](#)

[Who this book is for](#)

[Conventions](#)

[Reader feedback](#)

[Customer support](#)

[Downloading the example code](#)

[Errata](#)

[Piracy](#)

[Questions](#)

[1. Updates on iOS 7](#)

[Redesigning the iOS](#)

[New APIs](#)

[Developing games for iOS 7](#)

[Framework for game development](#)

[Knowing about Sprite Kit](#)

[Benefits of Sprite Kit](#)

[Game controller support](#)

[Game center renovations](#)

[Summary](#)

[2. Our First Sprite Kit Project](#)

[Sprite Kit basics](#)

[Anatomy of a Sprite Kit project](#)

[Scenes](#)

[Nodes](#)

[Node types](#)

[Actions](#)

[Game loop](#)

[Adding a background image to our game](#)

[Moving the character with actions](#)

[Adding infinite scrolling](#)

[Adding a score label](#)

[Summary](#)

[3. Interacting with Our Game](#)

[Handling touches](#)

[Using gesture recognizers](#)

[Accelerometer](#)

[Physics engine](#)

[Physics simulation basics](#)

[Implementing the physics engine](#)

[Summary](#)

#### [4. Animating Sprites](#)

[What is animation?](#)

[What is a texture atlas?](#)

[Adding animations to our project](#)

[Character states](#)

[Adding shield animations](#)

[Adding a parallax background](#)

[Summary](#)

#### [5. Particle Effects](#)

[Particle emitters](#)

[First particle effect](#)

[Advanced physics](#)

[Scene transitions](#)

[Summary](#)

#### [6. Adding Game Controllers](#)

[Native game controllers](#)

[Game controller basics](#)

[Using a controller in our game](#)

[Handling controller notifications](#)

[Adding sound and music](#)

[Summary](#)

#### [7. Publishing to the iTunes App Store](#)

[Registering as a developer](#)

[Bundle ID](#)

[Provisioning profiles](#)

[Preparing our application for the App Store](#)

[Managing applications in iTunes Connect](#)

[Life after uploading](#)

[Summary](#)

[Index](#)

# iOS 7 Game Development

---

# iOS 7 Game Development

Copyright © 2014 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: January 2014

Production Reference: 1140114

Published by Packt Publishing Ltd.

Livery Place

35 Livery Street

Birmingham B3 2PB, UK.

ISBN 978-1-78355-157-6

[www.packtpub.com](http://www.packtpub.com)

Cover Image by Aniket Sawant

(<[aniket\\_sawant\\_photography@hotmail.com](mailto:aniket_sawant_photography@hotmail.com)>)

# Credits

## **Author**

Dmitry Volevodz

## **Reviewers**

Jayant C Varma

Dave Jewell

## **Acquisition Editors**

Meeta Rajani

Rebecca Youe

## **Commissioning Editor**

Manasi Pandire

## **Technical Editors**

Kunal Anil Gaikwad

Krishnaveni Haridas

Manal Pednekar

## **Copy Editors**

Alfida Paiva

Sayanee Mukherjee

## **Project Coordinators**

Sherin Padayatty

Akash Poojary

## **Proofreader**

Paul Hindle

## **Indexer**

Mariammal Chettiyar

**Graphics**

Yuvraj Mannari

**Production Coordinator**

Kyle Albuquerque

**Cover Work**

Kyle Albuquerque

# About the Author

**Dmitry Volevodz** is an iOS developer. He has been doing freelance software development for a few years and has finally settled in a small company. He does enterprise iOS development by day and game development is his hobby.

**I would like to thank my beloved wife Olesya for her patience and support in everything I do. I would also like to thank Gennady Evstratov for believing in my programming abilities. Without him, this book would have never happened. I would also like to thank Alex Kuster for the artwork he provided for this book.**

# About the Reviewers

**Jayant C Varma** is an Australian author, developer, and trainer who has gained experience from several other countries. He is the author of *Learn Lua for iOS Game Development* and is the founder of OZ Apps, a development consultancy specializing in mobile development. He has managed the IT operations for BMW dealerships since the mid 90s and has been an adopter of new technologies. He has also been an academic with James Cook University, and is actively involved in training and conducting workshops with AUC and ACS. He has previously created a text-based adventure game engine which was used in Z-Day Survival Simulator for Mongadillo Studios. He has been a reviewer for Packt Publishing on numerous iOS-related books and technologies including iOS development, such as *MonoTouch Cookbook*, *Corona SDK Mobile Game Development*, and *Instant New iPad Features in iOS 6 How-to*.

**Dave Jewell** has been working with microprocessors since you could count Bill Gates' bank balance. He has developed apps for Windows 1.0 (and still wakes up screaming!), the original 128K Apple Mac, and many other refugees from the science museum. Current interests include cross-platform mobile app development, CMS systems, and designing of programming languages and compilers. He is currently working as a freelance software developer, specializing in the creation of bespoke apps for iOS and Android. In the past, Dave has written thousands of technical articles as a contributing editor, and is a regular writer for many programming magazines including *Program Now*, *.EXE*, *Delphi Magazine*, *Developer's Review*, *PC Plus*, and *PC Answers*. He has also authored and co-authored a number of books including *Instant Delphi* (Wrox Press) and *Polishing Windows* (Addison-Wesley). Most of his books are now, like their author, long past their sell-by date.

# **www.PacktPub.com**

## **Support files, eBooks, discount offers and more**

You might want to visit [www.PacktPub.com](http://www.PacktPub.com) for support files and downloads related to your book.

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](http://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>](mailto:service@packtpub.com) for more details.

At [www.PacktPub.com](http://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.



<http://PacktLib.PacktPub.com>

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

### **Free Access for Packt account holders**

If you have an account with Packt at [www.PacktPub.com](http://www.PacktPub.com), you can use this to access PacktLib today and view nine entirely free books. Simply use your login credentials for immediate access.

# Preface

Sprite Kit is a new framework from Apple for developing 2D games for iOS devices. It is new, fresh, and exciting.

Developers have been waiting long for a native library for games, but Apple did not deliver it until Version 7.0 of their operating system. Developers had to use unreliable third-party libraries, work on fixing bugs in these libraries, and experiencing headaches when suddenly your project just stops compiling under new versions of the operating system.

All these problems can be forgotten with the new Sprite Kit framework. It allows for easy and fast game development. It mimics many methods and the API of the Cocos2d library, which is a wildly popular library for game development. If you have ever checked out Cocos2d, you will feel right at home with Sprite Kit.

iOS 7 Game Development will take you on a journey to build a game from scratch using a hands-on approach. We will start with the basics and continue with advanced topics. We will explain every bit of code for maximum understanding.

We will build an endless runner game, an amazingly popular genre on the App Store, and will explain all the systems that have to be built in order to provide the best user experience.

## What this book covers

[Chapter 1](#), *Updates on iOS 7*, provides you with a short coverage of what features iOS 7 brings to the table—operating system redesign, new frameworks, and game controller support.

[Chapter 2](#), *Our First Sprite Kit Project*, explains you Sprite Kit basics, how to show a sprite on the screen, how to move it, and what properties and methods are available on sprite nodes. We will also discuss game loops and actions.

[Chapter 3](#), *Interacting with Our Game*, shows you the way to control our character sprite, either by using gesture recognizers or with raw touch processing.

[Chapter 4](#), *Animating Sprites*, walks you through the process of creating a texture atlas, animating our character, and creating actions to handle starting and finishing animations. We will also add nice parallax scrolling to our game.

[Chapter 5](#), *Particle Effects*, explains how to create cool-looking particle effects, how to store and edit them and their properties, and ways to improve your game performance when using particle effects.

[Chapter 6](#), *Adding Game Controllers*, walks you through the process of adding native game controller support to your game. We will check different controllers, their layouts, and ways to handle thumbstick, direction pad, and button inputs.

[Chapter 7](#), *Publishing to the iTunes App Store*, explains how to post your application to the iTunes App Store. We will learn about different application icons, categories, certificates, provisioning profiles, new Xcode publishing features, and the review process.

## What you need for this book

You will need a Mac running OS X 10.9 and Xcode Version 5.0 or higher.  
You are expected to have familiarity with Objective-C.

## Who this book is for

This book is intended for those who have great ideas for games and who want to learn about iOS game development. You should know and understand Objective-C. Being familiar with iOS development is helpful, but is not required. This book will make you familiar with the new Sprite Kit framework in no time.

# Conventions

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

Code words in text are shown as follows: “The thing that might have caught your attention is the format specifier `@“run%.3d”.`”

A block of code is set as follows:

```
- (void) stopRunningAnimation
{
    [self removeActionForKey:@"running"];
}
```

**New terms** and **important words** are shown in bold.

## Note

Warnings or important notes appear in a box like this.

## Tip

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 may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply send an e-mail to [<feedback@packtpub.com>](mailto:feedback@packtpub.com), and mention the book title through 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 on [www.packtpub.com/authors](http://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 for all Packt books you have purchased from your account at <http://www.packtpub.com>. 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.

## 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 would 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/support>, 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.

## Piracy

Piracy of copyright 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>](mailto: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

You can contact us at [<questions@packtpub.com>](mailto:questions@packtpub.com) if you are having a problem with any aspect of the book, and we will do our best to address it.

# Chapter 1. Updates on iOS 7

In this chapter, we will find out what's new in iOS 7, starting from new designs, which new APIs and SDKs were presented by Apple with iOS 7, and why you should pick Sprite Kit for game development.

## Redesigning the iOS

The new operating system from Apple features overhauled design in almost every element. All buttons, pickers, labels, navigation bars—everything looks and feels different. The concepts that Apple has chosen are simplicity and minimalism. Apple designers have chosen to get rid of the rich textures and gradients that we grew to love for six versions of their mobile operating system.

The new interface is unobtrusive; everything seems to be in its place. Everything that used to draw your attention is now gone, and your content is now in the center of the new design. For example, the following is the screenshot of the iOS 6 calendar followed by its iOS 7 version.

The change to "flat" design was met with enthusiasm by some and not so by others, but all we know is that it is here to stay.

When you are working on your game, you should probably check out the best practices by Apple designers, as they are thought-out, thoroughly tested, and well implemented.

Everything that you need to know about interface design on iOS devices can be found in *Human interface guidelines* by Apple at <https://developer.apple.com/library/ios/documentation/userexperience/conceptual/mobilehig/>.

On comparing the following two screenshots, you can see the old and new look of the Calendar application. Apple has focused on the user-generated content; you can see that the space for user data is much larger in the iOS 7 version; however, in the old Calendar application, we can see only two lines of our content.



*The Calendar application in iOS 6*

Buttons have transformed into simple lines of text, without any background or frame, while gradients on the navigation and bottom bar are gone and are replaced by the simple gray background.



*The Calendar application on iOS 7*

# New APIs

The new operating system from Apple features numerous new APIs. Some of them are long overdue and expected, and some are quite surprising. Some of them that are worth mentioning are as follows:

- **Text Kit:** This is an API for laying out text and for fine typography. Text Kit helps you lay out text in the way you like without any headache.
- **Dynamic behaviors for views:** With this, you can now assign physics effects to your views so they can follow each other, or gravity effects can be applied to views with ease.
- **Multitasking enhancements:** With this, applications have the ability to fetch data in the background, something that was available only to certain applications such as Newsstand apps. Now, your game can fetch some amount of data while the user is not playing it, such as daily missions or news.
- **Sprite Kit framework:** This is a new framework for developing 2D games, and features hardware sprites acceleration, simple sound playback support, and physics simulation.
- **Game controller support:** This is a new framework that provides common ground for hardware controllers.

# Developing games for iOS 7

In June 2013, Apple announced that the App Store has hit the next milestone—50 billion downloads with more than 14 billion dollars paid to developers all over the world. The ecosystem that only started to exist a few years ago already raked in more money for developers than any other platform.

A major share of this revenue is taken by game developers, ranging from large companies such as EA, Disney, and Rovio to small indie developers that manage to create best-selling applications with small budgets—everyone can find their place under the sun.

The most profitable and most downloaded titles on the App Store are 2D games—Angry Birds, Cut The Rope, and Doodle Jump. Rovio managed to create an empire out of a single title, and now it is selling merchandise, soft drinks, and toys, and all of this came out of a single mobile game (not their first one though, as Angry Birds was their 52nd title!).

# Framework for game development

Before iOS 7 (and Sprite Kit), there were various options for frameworks that could be used for game development. Each of them has its own advantages and disadvantages.

If you wanted to make a game before iOS 7, you had only so many options. They are as follows:

- OpenGL ES
- UIKit controls and views
- Third-party libraries (Unity, Unreal 3D, and Cocos2d)

Let us see each of them in detail:

- **OpenGL** is very customizable and open-ended, but it is hard to learn, and you need to know a lot of things just to get an image on screen. It is good if you are an experienced programmer or a company, and you want to write cross-platform solutions. OpenGL offers good performance, but you have to pay with code complexity.
- Next up is **UIKit**, which is the default iOS programming framework. Every element that you see in a regular iOS application, such as buttons, pickers, views, and navigation bars, comes from here. But there are only so many games that can look good with the default interface—some trivia games, maybe some manager games, but that's it. There are benefits to this—your user already knows everything he can do with the interface, gesture controls, and back buttons, and this makes it easier to actually present your idea, but at the same time, UIKit fails to immerse your user into the game; you get the same interface as almost every other application in the App Store. Another big problem with UIKit is performance, or the lack of it. After all, it was not designed for dynamic games, and if you decide to make something complicated in it, you will find the bottleneck pretty fast.
- Another option to consider is third-party libraries. There are dozens of them, and few are very popular among the developers. Unity3D is good, as it offers a cross-platform solution as well as massive numbers of tutorials. The same can be said about Unreal 3D. But these libraries often require you to know completely different programming languages such as C#, C++, or even Lua. It might not be a good choice if you know Objective-C and want to write native applications for the platform, not to mention that the level of complexity of these frameworks is high. You need to learn a lot just before you can have simple sprites moving on screen.
- Another option that you have is the Cocos2d framework. It is somewhat easy, can get you going fast, is open source, and works with Objective-C. But as with any third-party library, it has its disadvantages. It does not support ARC out of the box. It has

problems when Apple releases new versions of iOS—so far, every OS release had left Cocos2d code broken in one way or another. You could have the rotation feature stop working altogether, or suddenly some methods may fail to compile with errors. This doesn't really work if all you want is a simple framework for your games.

# Knowing about Sprite Kit

Apple presented iOS 7 in September 2013, featuring numerous new features for users and developers. One of them is Sprite Kit—a new framework for game developers.

Sprite Kit is a native SDK from Apple that allows simple entry into game development, without unnecessary overhead and a long learning process. If you are familiar with Cocos2d, you will have an easy time with Sprite Kit, as it is heavily inspired by the former. Sprite Kit provides native rendering and animation infrastructure to work with sprites as well as animations, particle systems, scenes, actions, physics simulation, and video effects.

## Note

A sprite is a two-dimensional image or animation integrated into a larger scene. Any image can be a sprite—a character, a tree, or a bullet.

It allows easy work with sprites, the core component of all 2D games. Almost everything you can see on the screen of a 2D game is a sprite.

# Benefits of Sprite Kit

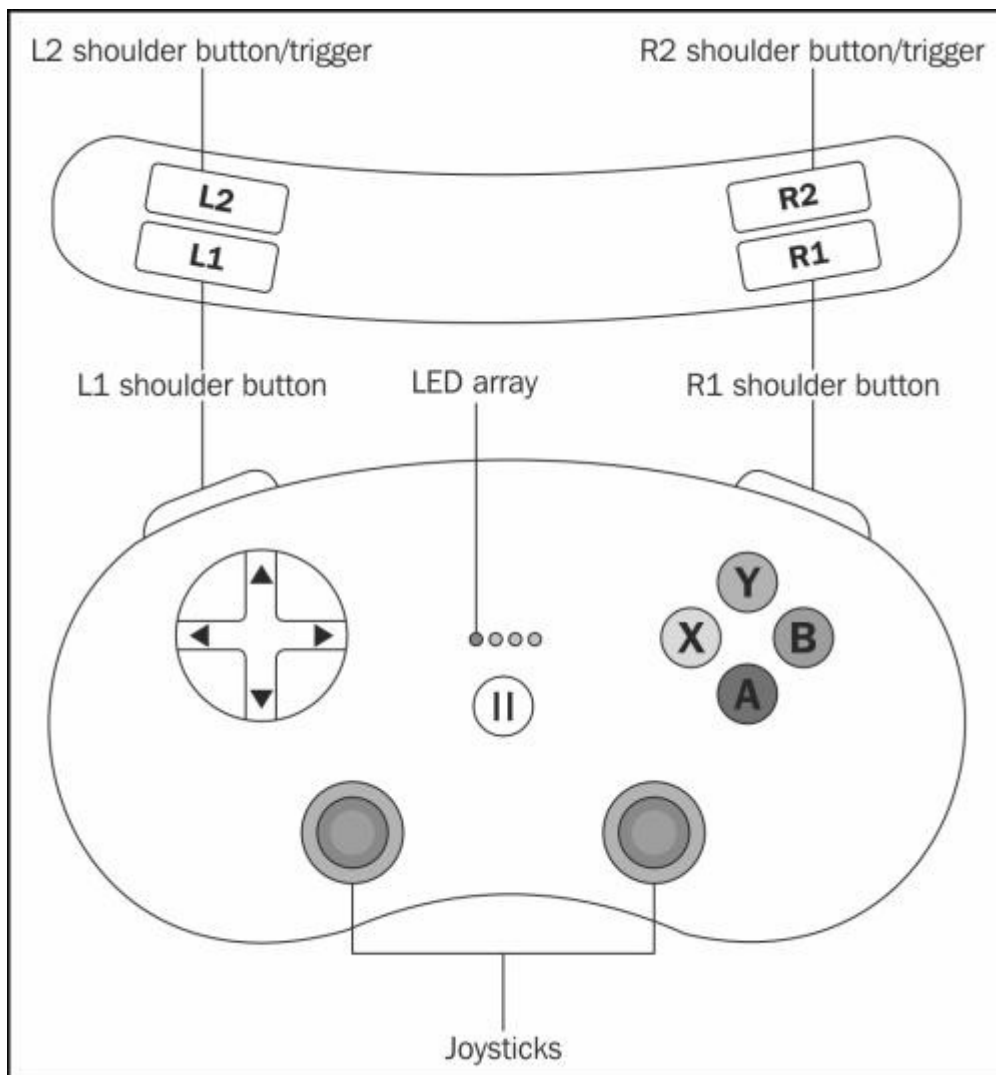
Sprite Kit has certain advantages that will help you determine if you want to base your game on it. They are as follows:

- **Sprite Kit is part of the iOS SDK:** This means that it will be supported by Apple, and everything you write is likely to be future-proof. Your code will not magically stop working (or even compiling!), and things you are getting on screen are guaranteed to stay the same. Everyone who works with third-party libraries is aware of the issues that come with using a non-native SDK. With Sprite Kit, you can forget about installation problems and compatibility problems.
- **Easy-to-use API:** This is developed by some of the best engineers in the world. Everything is logical and works as expected. Clear methods and properties work just as you would expect them to.
- **Built-in tools:** With this, you no longer have to use third-party software to make your texture atlases, assets, or fonts. You just drop in your images and Xcode does everything for you.
- **Built-in physics engine:** This makes your life as a developer much easier. You do not have to pick out one of the third-party physics engines or work on integration of that code into your project—it just works out of the box.
- **Your game will work both on iOS and Mac without much effort:** Sprite Kit supports both Mac and iOS, and all you need to change is controls. You will have touch controls for your iPhone and iPad versions and the mouse and keyboard controls for Mac.

## Game controller support

One of the most interesting features of iOS 7 is the native controller support. Some companies such as iCade and others tried working on their own controllers, but this effort has not seen much success. Surely, some games supported it, but the majority of games were left unsupported.

Developers did not feel the need to support such devices, as their install base is small, and return on investment was just not available. But everything changed when Apple decided to roll out support for such controllers. Now we have a native API to work with controllers and all future controllers by different vendors that will work with this API. In the following diagram, you can see an Apple-proposed design for one of the game controllers. As you can see, it offers all the features of a modern controller—two thumb sticks, shoulder buttons, and LEDs.



*Game controller for iOS 7*

There have been rumors that vendors such as Logitech are already working on such controllers, and you as a developer should probably work on implementing them in your game, as the effort required to make them work is really small, and the satisfaction that your player gets when the game works with his controller is enormous.

The new Game Controller framework allows discovering and connecting compatible game controllers to your iOS device.

# Game center renovations

Game center have several new features that will help you with your games. Some of them have been listed as follows:

- **Increased limit of leaderboards per application:** Now you can have up to 100 leaderboards in your game.
- **New feature—exchanges:** This allows your turn-based player to initialize actions even if it is not their turn. Previously, you had to wait for your turn to even chat, and now you can do that on other players' turns if the game supports that.
- **Improved features to prevent cheating:** Cheating, obviously, is never good, especially if your game is competitive and has leaderboards. We all know how such games are infested with hackers, and these new features will certainly help with that.

## Summary

In this chapter, we have learned what new exciting features and APIs iOS 7 has to offer us. We have found out what Sprite Kit is and why we should use it for game development and its advantages. We have found out that Apple unified game controllers, and new ones will be available shortly. If you are reading this book, chances are that you are planning to make games for iOS, and Sprite Kit is an excellent choice.

In the next chapter, we will start working on our first Sprite Kit project, a fully featured endless runner game.

# Chapter 2. Our First Sprite Kit Project

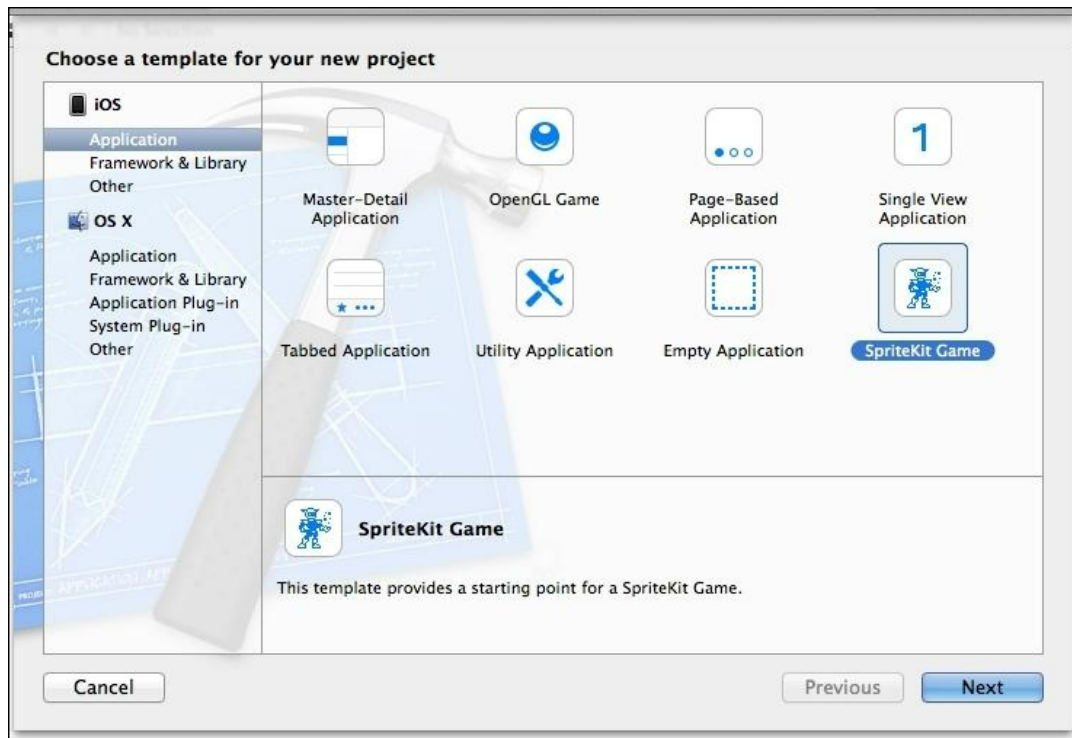
In this chapter, we will look into Sprite Kit basics. Sprite Kit consists of a lot of small elements and we need to have a clear understanding of what they do to have an overview of a typical Sprite Kit project to see how we might plan and implement a full-featured game.

We will explore everything that we might need when creating our own project, as the main goal of the book is to provide an understanding of Sprite Kit as a tool for game development. We will start with the project that we will be creating in this book—an endless runner game.

This style of game is really popular on mobile devices, as it allows for quick gameplay when you get a minute and has this "just one more" feeling to it. A player wants to beat/better his own score, maybe even beat his friend's scores. Our game will feature jumping and you will want to evade dangerous things that may appear on screen. You get a higher score the longer you run without failing.

## Sprite Kit basics

First of all, we need to create a basic project from the template to see how everything works from the inside. Create a new project in Xcode by navigating to **File | New | Project**. Choose **SpriteKit Game** after navigating to **iOS | Application**, as shown in the following screenshot:



*Creating a Sprite Kit project*

On the next screen, enter the following details:

- **Product Name:** `Endless Runner`.
- **Organization name:** Enter your name.
- **Company Identifier:** This is a unique Identifier that identifies you as a developer. The value should be similar to what you entered while registering your Apple developer account or `com.your_domain_name`.
- **Class Prefix:** This is used to prefix your classes due to poor Objective-C namespace. People often ignore this (it makes sense when you make small projects, but if you use any third-party libraries or want to follow best practices, use three-lettered prefixes). We will use `ERG` (denoting Endless Runner Game) for this. Apple reserves using two-lettered prefixes for internal use.
- **Devices:** We are making the game for iPhones, so ensure **iPhone** is selected.

Now, save the project and click on **Create**.

Once you have created the project, build and run it by clicking on the play button at the top-left corner of the window.

You will see the **Hello, World!** label at the center of the screen, and if you tap the screen, you will get a rotating spaceship at that point. At the bottom-right corner of the screen, you can see the current **FPS (frames per second)** and number of nodes in the scene.

# Anatomy of a Sprite Kit project

A Sprite Kit project consists of things usual to any iOS project. It has the `AppDelegate`, `Storyboard`, and `ViewController` classes. It has the usual structure of any iOS application. However, there are differences in `ViewController.view`, which has the `SKView` class in `Storyboard`.

You will handle everything that is related to Sprite Kit in `SKView`. This class will render your gameplay elements such as sprites, nodes, backgrounds, and everything else. You can't draw Sprite Kit elements on other views.

It's important to understand that Sprite Kit introduces its own coordinate system. In UIKit, the origin (0,0) is located at the top-left corner, whereas Sprite Kit locates the origin at the bottom-left corner. The reason why this is important to understand is because of the fact that all elements will be positioned relative to the new coordinate system. This system originates from OpenGL, which Sprite Kit uses in implementation.

# Scenes

An object where you place all of your other objects is the `SKScene` object. It represents a single collection of objects such as a level in your game. It is like a canvas where you position your Sprite Kit elements. Only one scene at a time is present on `SKView`. A view knows how to transition between scenes and you can have nice animated transitions. You may have one scene for menus, one for the gameplay scene, and another for the scene that features after the game ends.

If you open your `ViewController.m` file, you will see how the `SKScene` object is created in the `viewDidLoad` method.

Each `SKView` should have a scene, as all other objects are added to it. The scene and its object form the node tree, which is a hierarchy of all nodes present in the scene.

Open the `ERGMyScene.m` file. Here, you can find the method where scene initialization and setup take place:

```
- (id)initWithSize:(CGSize)size
```

The scene holds the view hierarchy of its nodes and draws all of them. Nodes are very much like `UIView`s; you can add nodes as children to other nodes, and the parent node will apply its effects to all of its children, effects such as rotation or scaling, which makes working with complex nodes so much easier.

Each node has a position property that is represented by `CGPoint` in scene coordinates, which is used to set coordinates of the node. Changing this position property also changes the node's position on the screen.

After you have created a node and set its position, you need to add it to your scene node hierarchy. You can add it either to the scene or to any existing node by calling the `addChild:` method. You can see this in your test project with the following line:

```
[self addChild:myLabel];
```

After the node has been added to a visible node or scene, it will be drawn by the scene in the next iteration of the run loop.

# Nodes

The methods that create `SKLabelNode` are self-explanatory and it is used to represent text in a scene.

The main building block of every scene is `SKNode`. Most things you can see on the screen of any given Sprite Kit game is probably a result of `SKNode`.

## Node types

There are different node types that serve different purposes:

- `SKNode`: This is a parent class for different types of nodes
- `SKSpriteNode`: This is a node that draws textured sprites
- `SKLabelNode`: This is a node that displays text
- `SKShapeNode`: This is a node that renders a shape based on a Core Graphics path
- `SKEmitterNode`: This is a node that creates and renders particles
- `SKEffectNode`: This is a node that applies a Core Image filter to its children

Each of them has their own initializer methods—you create one, add it to your scene, and it does the job it was assigned to do.

Some node properties and methods that you might find useful are:

- `position`: This is a `CGPoint` representing the position of a node in its parent coordinate system.
- `zPosition`: This is a `CGFloat` that represents the position of a node on an imaginary Z axis. Nodes with higher `zPosition` will be over the nodes that have lower `zPosition`. If nodes have the same `zPosition`, the ones that were created later will appear on top of those that were created before.
- `xScale` and `yScale`: This is a `CGFloat` that allows you to change the size of any node. The default is 1.0 and setting it to any other value will change the sprite size. It is not recommended, but if you have an image of a certain resolution, scaling it will upscale the image and it will look distorted. Making nodes smaller can lead to other visual artifacts. But if you need quick and easy ways to change the size of your nodes, this property is there.
- `name`: This is the name of the node that is used to locate it in the node hierarchy. This allows you to be flexible in your scenes as you no longer need to store pointers to your nodes, and also saves you a lot of headache.

- `childNodesWithName:(NSString *)name`: This finds a child node with the specified name. If there are many nodes with the same name, the first one is returned.
- `enumerateChildNodesWithName:usingBlock:`: This allows you to run custom code on your nodes. This method is heavily used throughout any Sprite Kit game and can be used for movement, state changing, and other tasks.

# Actions

Actions are one of the ways to add life to your game and make things interactive. Actions allow you to perform different things such as moving, rotating, or scaling nodes, playing sounds, and even running your custom code. When the scene processes its nodes, actions that are linked to these nodes are executed.

To create a node, you run a class method on an action that you need, set its properties, and call the `runAction:` method on your node with action as a parameter.

You may find some actions in the `touchesBegan:` method in `ERGMyScene.m`. In this method, you can see that a new node (of the type `SKSpriteNode`) is created, and then a new action is created and attached to it. This action is embedded into another action that makes it repeat forever, and then a sprite runs the action and you see a rotating sprite on the screen.

To complete the preceding process, it took only five lines, and it is very intuitive. This is one of the Sprite Kit strengths—simplicity and self-documenting code. As you might have noticed, Apple names methods in a simpler way so that you can understand what it does just by reading the method. Try to adhere to the same practice and name your variables and methods so that their function can be understood immediately. Avoid naming objects `a` or `b`, use `characterSprite` or `enemyEmitter` instead.

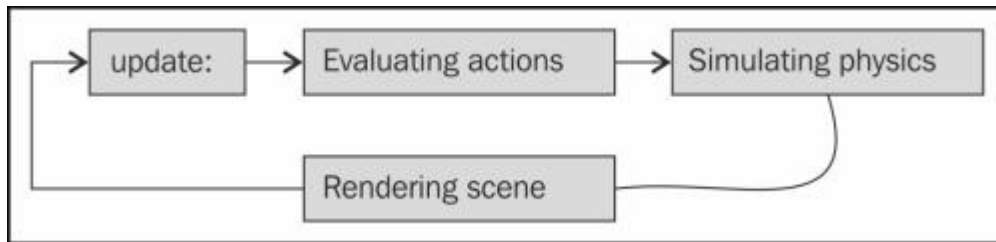
There are different action types; here we will list some that you may need in your first project:

- Move actions (`moveTo:duration:`, `moveBy`, `followPath`) are actions that move the node by a specified distance in points
- Rotate actions are actions that rotate your nodes by a certain angle (`rotateByAngle:duration:`)
- Actions that change node scale over time (`scaleBy:duration`)
- Actions that combine other actions (`sequence:` to play actions one after another, and `repeatAction:` to play an action a certain amount of times or forever)

There are many other actions and you might look up the `SKAction` class reference if you want to learn more about actions.

# Game loop

Unlike UIKit, which is based on events and waits for user input before performing any drawing or interactions, Sprite Kit evaluates all nodes, their interactions, and physics as fast as it can (capped at 60 times per second) and produces results on screen. In the following figure, you can see the way a game loop operates:



*The update loop*

First, the scene calls the `update:(CfTimeInterval)currentTime` method and sends it the time at which this method was invoked. The usual practice is to save the time of the last update and calculate the time that it took from the last update (delta) to the current update to move sprites by a given number of points, by multiplying the velocity of a sprite by delta, so you will get the same movement regardless of FPS. For example, if you want a sprite to move 100 pixels every second, regardless of your game performance, you multiply delta by 100. This way, if it took long to process the scene, your sprite will move slightly further for this frame; if it is processed fast, it will move just a short distance. Either way you get expected results without complex calculations.

After the update is done, the scene evaluates actions, simulates physics, and renders itself on screen. This process repeats itself as soon as it's finished. This allows for smooth movement and interactions.

You will write the most essential code in the `update:` method, since it is getting called many times per second and everything on screen happens with the code we write in this method.

You will usually iterate over all objects in your scene and dispatch some job for each to do, such as character moving and bullets disappearing off screen. The `update:` method is not used in a template project, but it is there if you want to customize it. Let's see how we can use it to move the **Hello, World!** label off the screen.

First, find where the label is created in the scene `init` method, and find this line:

```
myLabel.text = @"Hello, World!";
```

Add this code right after it:

```
myLabel.name = @"theLabel";
```

Find the `update:` method; it looks like this:

```
- (void)update:(CFTimeInterval)currentTime
```

Insert the following code into it:

```
[self enumerateChildNodesWithName:@"theLabel"
usingBlock:^(SKNode *node, BOOL *stop) {

    node.position =
CGPointMake(node.position.x - 2, node.position.y);

    if (node.position.x < -
node.frame.size.width) {

        node.position =
CGPointMake(self.frame.size.width,
node.position.y);
    }
}];
```

This method first finds the child node with the name `"theLabel"`, and as we named our label the same, it finds it and gives control to the block inside. The child that it found is a node. If it finds other nodes with the name `"theLabel"`, it will call the same block on all of them in the order they were found. Inside the block, we change the label position by 2 pixels to the left, keeping the vertical position the same. Then, we do a check, if the position of the label from the left border of the screen is further than the length of the label, we move the label to the right-hand side of the screen. This way, we create a seamless movement that should appear to be coming out of the right-hand side as soon as the label moves off screen.

But if you run the project again, you will notice that the label does not disappear. The label takes a bit longer to disappear and blinks on screen instead of moving gracefully.

There are two problems with our code. The first issue is that the frame is not changing when you rotate the screen, it stays the same even if you rotate the screen. This happens because the scene size is incorrectly calculated at startup. But we will fix that using the following steps:

1. Locate the Endless Runner project root label in the left pane with our files. It says **Endless Runner, 2 targets, iOS SDK 7.0**. Select it and select the **General** pane on the main screen.

There, find the device orientation and the checkboxes near it. Remove everything but **Landscape Left** and **Landscape Right**. We will be making our game in landscape and we don't need the **Portrait** mode.

2. Next, locate your `ERGViewController.m` file. Find the `viewDidLoad` method. Copy everything after the `[super viewDidLoad]` call.
3. Make a new method and add the following code:

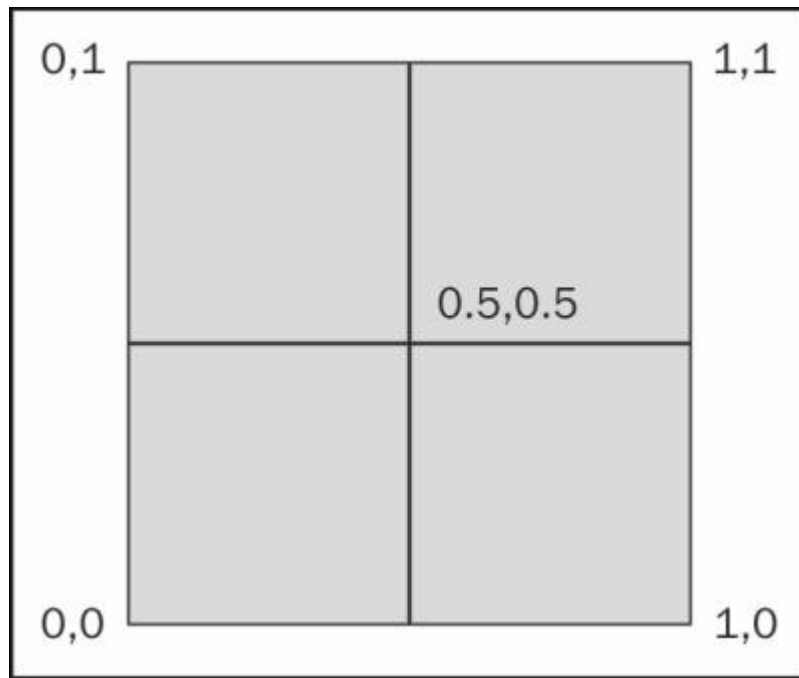
```
- (void) viewWillLayoutSubviews
{
    // Configure the view.
    [super viewWillLayoutSubviews];
    SKView * skView = (SKView *)self.view;
    skView.showsFPS = YES;
    skView.showsNodeCount = YES;

    // Create and configure the scene.
    SKScene * scene = [ERGMyScene
sceneWithSize:skView.bounds.size];
    scene.scaleMode =
    SKSceneScaleModeAspectFill;

    // Present the scene.
    [skView presentScene:scene];
}
```

4. Let's see why calculations of frame size are incorrect by default. When the view has finished its load, the `viewDidLoad` method is getting called, but the view still doesn't have the correct frame. It is only set to the correct dimensions sometime later and it returns a portrait frame before that time. We fix this issue by setting up the scene after we get the correct frame.

The second problem is the anchoring of the nodes. Unlike `UIViews`, which are placed on screen using their top-left corner coordinates, `SKNodes` are getting placed on the screen based on their `anchorPoint` property. The following figure explains what anchor points are. By default, the anchor point is set at (0.5, 0.5), which means that the sprite position is its center point. This comes in handy when you need to rotate the sprite, as this way it rotates around its center axis.



*Anchor point positions*

Imagine that the square in the preceding figure is your sprite. Different anchor points mean that you use these points as the position of the sprite. The anchor point at (0, 0) means that the left-bottom corner of our sprite will be on the position of the sprite itself. If it is at (0.5, 0.5), the center of the sprite will be on the position point. Anchor points go from 0 to 1 and represent the size of the sprite. So, if you make your anchor point (0.5, 0.5), it will be exactly on sprite center.

We might want to use the (0,0) anchor point for our text label.

### **Tip**

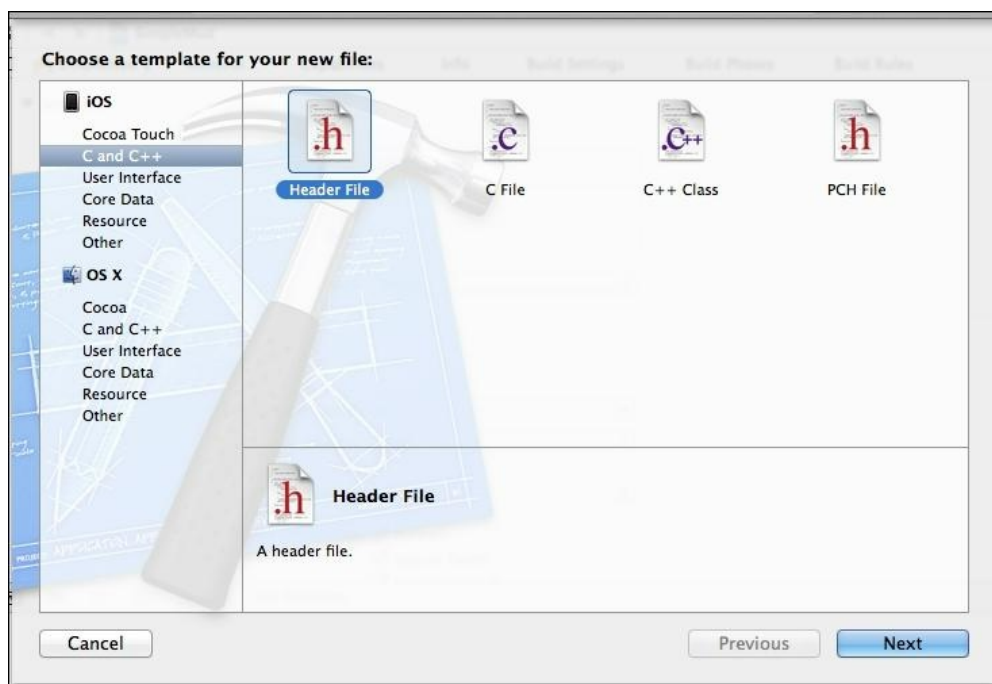
The problem is that we can't set an anchor point for `SKLabelNode`. There are several ways to overcome this problem, such as adding an empty `SKSpriteNode`, attaching `SKLabelNode` to it, and setting the anchor point of the first node to (0,0). We will leave this as an exercise for you.

# Adding a background image to our game

First we need to add the background file to our project. Xcode 5 offers new ways to handle your assets. Find `background.png` in resource files for this chapter. Find `images.xcassets` in the project navigator on the left-hand side of the screen. Click on it and you will see a list of resources currently in your application. Drag-and-drop `background.png` into the left-hand list near **Appicon** and **LaunchImage**. Background will be added to the list of available resources. Select it and drag it from the **1x** box to the **2x** box, as this is a high-resolution image to use on retina devices.

Next, we should utilize this image somehow. The first thing that comes to mind is to make `SKSpriteNode` out of the image and move it in the `update:` method. Sounds good, but if we add everything into the scene, it will be too bloated and unmanageable. Let's make a separate class file that will handle background images for us:

1. We will need some place to store all our constants and variables that we might need throughout our game. Common header file looks like a good place to store them. Navigate to **File | New File**, click on **C and C++** on the left-hand side under the **iOS** category and select **Header File** there. Name it `Common.h`.



2. Locate the file `Endless Runner-Prefix.pch`. The contents of this file are added to every source file in our project. Add the `#import "Common.h"` line right after `Foundation.h`. This way, our header file

will be automatically imported into everything. You can find this file inside the [Supporting Files](#) folder in the list of project files on the left-hand side of the Xcode window.

3. Add the name of our background node, `static NSString *backgroundName = @"background";` to the `Common.h` file so we can reference it from anywhere.
4. Again, create a new file, an Objective-C class, name it `ERGBackground`, and set `SKSpriteNode` as its parent class when asked.

We will be handling everything background related in the `ERGBackground` class. Let's make the class method return the preset background so we can use it in our game.

Add this method to the implementation file and its prototype to the header file:

```
+ (ERGBackground *)generateNewBackground
{
    ERGBackground *background = [[ERGBackground
    alloc] initWithImageNamed:@"background.png"];
    background.anchorPoint = CGPointMake(0, 0);
    background.name = backgroundName;
    background.position = CGPointMake(0, 0);
    return background;
}
```

This method starts by creating a new background node from the file that we have added before. We set the anchor point to (0,0) to help with scrolling. This way, the position of the node will be at the left-bottom corner of the image so that we don't have to calculate the starting position of the node. By default, the anchor point is (0.5,0.5), and if we want to set two sprites back-to-back, we have to calculate halves of those nodes. If we use (0,0) as the anchor point, we just add a new sprite on the position where the last sprite ended and that's it.

Why did we assign static `NSString` as the name of the background and not just type some arbitrary string? Compilers offer no error handling for the names of files or names, so you can miss small mistakes in filenames, and this mistake won't be easy to find. By using static `NSString`, we let compilers handle errors for us. Next is the node position on screen—we want it to start from the left edge of screen, so we set it there.

After we have created the background, we need to use it somewhere. In the `ERGMyScene.m` file, import `ERGBackground.h`, and inside the header file, add the `@class ERGBackground` line before the `@interface` declaration, and also add a new property:

```
@property (strong, nonatomic) ERGBackground
*currentBackground
```

We will hold the currently shown background in it. Next up, remove everything inside the brackets of the `initWithSize:` method, create a new background there, and add it as a child node to the scene:

```
-(id)initWithSize:(CGSize)size {
    if (self = [super initWithSize:size]) {

        self.currentBackground = [ERGBackground
generateNewBackground];
        [self addChild:self.currentBackground];
    }
    return self;
}
```

Build and run the project now and you will see the background there. Everything looks pretty good. But how do we get it to move?

We may go into the `update:` method and do it in the same quick and dirty way as we did with the label—just move it by some amount of pixels each time it updates. But we don't want a different speed of scrolling on different devices. That's why we will implement the moving speed based on time and not on iterations per second.

To do this, we need to add a new property to `ERGMyScene.h`:

```
@property (assign) CFTimeInterval
lastUpdateTimeInterval;
```

This will hold the time of the last update, and having this and `currentTime` (which we get from the `update:` method), we can find the delta (difference) since the last update.

In your `update:` method, remove any old code and add this code:

```
CFTimeInterval timeSinceLast = currentTime -
self.lastUpdateTimeInterval;
self.lastUpdateTimeInterval = currentTime;
if (timeSinceLast > 1) { // more than a
second since last update
    timeSinceLast = 1.0 / 60.0;
}
```

The preceding code is taken from Apple and it looks good enough for us. It calculates the delta we need in `timeSinceLast` and handles this value if too

much time has passed since the last update (for example, you exit the application and came back to it or took a call).

Having done this, we can set up a much more precise movement. Let's add a new constant to the `Common.h` file:

```
static NSInteger backgroundMoveSpeed = 30;
```

We will use this to handle background scrolling, new code to scroll background that uses timing, and add it after handling time in the `update:` method:

```
[self
enumerateChildNodesWithName:backgroundName
usingBlock:^(SKNode *node, BOOL *stop) {

    node.position =
CGPointMake(node.position.x - backgroundMoveSpeed
* timeSinceLast, node.position.y);
}];
```

We multiply time passed by speed to get the amount of pixels that the background needs to be shifted in this frame (by frame, we mean the picture that gets drawn 60 times per second on screen, not the sprite bounding box). On the next frame, the calculation proceeds, and if for some reason the frame rate drops, the user won't notice it as the background moves at the same speed regardless of the frame rate. There is a problem when the background scrolls too far to the left; we are left with an empty screen. We will fix this in [Chapter 4](#), *Animating Sprites*.

The next thing that we need to do is to add a character. Add the `character.png` image to the project in the same way as you did with the background (find `Images.xcassets`, select it, drag-and-drop the file there). Don't forget to set this image as a **2x** image, since it is high resolution.

We will need a separate class for our player. First, let's add a new string to `Common.h` to identify the player node:

```
static NSString *playerName = @"player";
```

After this, create a new class called `ERGPlayer` and make it inherit from `SKSpriteNode`. Import the header file (`ERGPlayer`) into `ERGScene.m` so that our program can access methods and properties from it.

We need to redefine its `init` method so that we always get the same character for the player. Add the following code to `ERGPlayer.m`:

```
-(instancetype)init
```

```

{
    self = [super
initWithImageNamed:@"character.png"];
    {
        self.name = playerName;
    }
    return self;
}

```

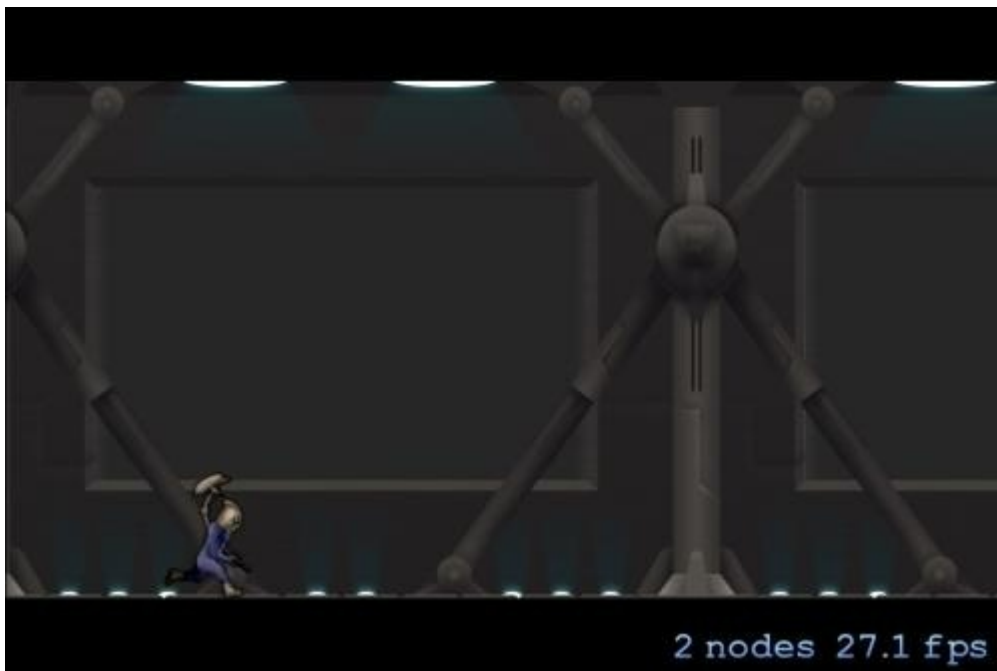
This method calls the parent implementation of the designated initializer. Next, add the player object to `ERGMyScene.m` by adding this code to the scene `init` method:

```

ERGPlayer *player = [[ERGPlayer alloc] init];
player.position = CGPointMake(100, 68);
[self addChild:player];

```

If you run the project now, you will see the moving background and character sprite on it, as shown in the following screenshot:



*Background and character sprites on screen*

# Moving the character with actions

Let's discover how we can add simple player movements to our game, for example, jumping. One of the ways to handle this could be by creating a new action that will move the character up by a certain amount of pixels and then move the character down. Let's try this out.

Remove everything from the `touchesBegan:` method in `ERGMyScene.m`. It should look like this:

```
-(void)touchesBegan:
(NSSet *)touches withEvent:(UIEvent *)event {

    // we are creating action to move the node
    that runs it by vector
    // of x and y components with duration in
    seconds
    SKAction *moveUp = [SKAction
moveBy:CGVectorMake(0, 100) duration:0.8];

    // same as before, but it is opposite vector
    to go down, and it is a bit
    // faster, since gravity accelerates you
    SKAction *moveDown = [SKAction
moveBy:CGVectorMake(0, -100) duration:0.6];

    // sequence action allows you to compound few
    actions into one
    SKAction *seq = [SKAction sequence:@[moveUp,
moveDown]];

    // childNodeWithName: method allows you to
    find any node in hierarchy with
    // certain name. This is useful if you don't
    want to store things
    // as instance variables or properties
    ERGPlayer *player = (ERGPlayer *)[self
childNodeWithName:playerName];

    // after creating all actions we tell
    character to execute them
    [player runAction:seq];
}
```

If you tap now, the character sprite will jump (that is, move up and then go down). Notice that we didn't write anything regarding the player in the update loop and it still works. The actions system is separate from the update loop. If we set some player movement in the `update:` method, they would work simultaneously, running the action for a duration and updating every frame. Also, notice that if you jump a couple of times, these will all

stack up and work awkwardly. We could fix this by adding a variable that tells us whether the character is in the middle of a jump, and further jump and other actions should be ignored.

But this is unnecessarily complicated. What if we wanted to add platforms, how would we handle that?

Generally speaking, the action system is useful if you know you need to do a certain thing with your objects and nothing changes while executing them. This way, actions are useful and very helpful, since doing something like that in the update loop can be daunting.

Another powerful feature of actions is that they can be sequenced or repeated, and you can make very complex movements and effects with them.

# Adding infinite scrolling

Since having the background disappear is not the way it is meant to be, we will add an infinite background. One way to do it is to load as large a background as the memory allows and hope the player loses before reaching the end of the background image. But there is a much better way. We will have relatively small background segments, and when the background segment is going to end, we will create a new one and attach it to the end of the current segment. When the old one goes off the screen, it is removed and destroyed.

This way, we will have only two backgrounds in memory at any time, and this makes it easier and simpler to manage.

First we need to adjust the `zPosition` property of the player and the `Background` object. If we make a new background when the player is on screen, eventually it will cover the player, as nodes that were made later are rendered on top of earlier ones. Go to `ERGPlayer.m`, and in the `init` method, add `self.zPosition = 10`. Do the same for `ERGBackground.m` in the class method that generates the background by setting `backg.zPosition = 1`.

Nodes with a predetermined `zPosition` will always be rendered in the correct order. Nodes with a higher `zPosition` will be rendered on top of nodes with a lower `zPosition`.

Here is the new `update:` method that gives us infinite background:

```
-(void)update:(CFTimeInterval)currentTime {

    CFTimeInterval timeSinceLast = currentTime -
self.lastUpdateTimeInterval;
    self.lastUpdateTimeInterval = currentTime;
    if (timeSinceLast > 1) { // more than a
second since last update
        timeSinceLast = 1.0 / 60.0;
        self.lastUpdateTimeInterval = currentTime;
    }
    [self
enumerateChildNodesWithName:backgroundName
usingBlock:^(SKNode *node, BOOL *stop) {
        node.position =
CGPointMake(node.position.x - backgroundMoveSpeed
* timeSinceLast, node.position.y);
        if (node.position.x < -
(node.frame.size.width + 100)) {
            // if the node went completely off
screen (with some extra pixels)
            // remove it
        }
    }];
}
```

```

        [node removeFromParent];
    }]];
    if (self.currentBackground.position.x < -500)
    {
        // we create new background node and set
        it as current node
        ERGBBackground *temp = [ERGBBackground
generateNewBackground];
        temp.position =
CGPointMake(self.currentBackground.position.x +
self.currentBackground.frame.size.width, 0);
        [self addChild:temp];
        self.currentBackground = temp;
    }
}

```

You can adjust the speed in [Common.h](#) to check if the scrolling is really infinite. You can do this by modifying [backgroundMoveSpeed](#). There are some optimizations that can be done about this. Instead of creating new nodes each time, we could just keep two backgrounds in memory at all times and just reposition them every time one went off the screen. But this way, you are limited to the same background image.

## Adding a score label

The next thing we need to have in our game is a label that shows how far we have gone. First, add two new properties to `ERGMyScene.h`:

```
@property (strong, nonatomic) SKLabelNode
*scoreLabel;
@property (assign) double score;
```

After this, add a new label and actions for it in the scene `initWithSize:` method:

```
self.score = 0;
self.scoreLabel = [[SKLabelNode alloc]
initWithFontNamed:@"Chalkduster"];
self.scoreLabel.fontSize = 15;
self.scoreLabel.color = [UIColor
whiteColor];
self.scoreLabel.position =
CGPointMake(20, 300);
self.scoreLabel.zPosition = 100;
[self addChild:self.scoreLabel];

SKAction *tempAction = [SKAction
runBlock:^(
    self.scoreLabel.text = [NSString
stringWithFormat:@"%3.0f", self.score];
)];

SKAction *waitAction = [SKAction
waitForDuration:0.2];
[self.scoreLabel runAction:[SKAction
repeatActionForever:
[SKAction sequence:@[tempAction, waitAction]]];
```

Why would we need so many actions? We could go the easier way of refreshing a label in the `update:` method, but rendering text on labels is an expensive task, and calling it for every update is not a good idea. That's why we create an action that updates the label every 0.2 seconds.

The label is created the way you expect; we set the properties that we need and add it as a child to the scene.

Next, add the following code to the end of the `update:` method:

```
self.score = self.score + (backgroundMoveSpeed *
timeSinceLast / 100);
```

Now the label updates in real time. As our character moves, the score increases.

# Summary

In this chapter, we have learned the basics of Sprite Kit. There are many things that have to be finished in our game, but we have a robust background now. Our project is relatively small, but we can already see the basic gameplay elements in action. Things that we have learned are:

- The properties and hierarchy of SKNodes
- SKSpriteNode, anchoring, and drawing
- How to draw an infinite scrolling background
- How to draw a text label
- How to move a sprite on screen (actions and update method)
- How a game loop operates

In the next chapter, we will discuss interacting with our game, handling touches and gesture recognizers, and moving sprites.

# Chapter 3. Interacting with Our Game

In this chapter, we will discuss the ways in which we can get input from the player. As there are many different ways to handle user input, we will see the advantages and disadvantages of each of them. Some of the ways to interact with the game are as follows:

- Touching the screen (taps)
- Gesture recognizers
- Accelerometer and gyroscope
- Game controller events

## Handling touches

When the user touches the iOS device screen, our current running scene will receive a call to one of the following methods:

- `-(void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event`
- `-(void)touchesMoved:(NSSet *)touches withEvent:(UIEvent *)event`
- `-(void)touchesEnded:(NSSet *)touches withEvent:(UIEvent *)event`
- `-(void) touchesCancelled:(NSSet *)touches withEvent:(UIEvent *)event`

Each method gets an `NSSet` class of a `UITouch` object (which holds the coordinates of our touch), and the event holds information about the window and view in which the touch was triggered as well as information about the touch. Let's see what each of them does:

- `touchesBegan:` This method gets triggered each time the user touches the screen, unrelated to whether this touch continues as a long press or not.
- `touchesMoved:` This method provides you with events that occurred if the user moved a finger while on screen, and provides new touches.
- `touchesEnded:` This method gets called when the user takes their finger off the screen. This can be useful, for example, if you want some animation to play after the user removes their finger, or to re-enable physics calculations on the node.
- `touchesCancelled:` This method is rarely needed, but you should implement it anyway. It is called when some event cancels the touch, for example, an on-screen alert, phone call, notifications (such as a

push notification or calendar notification), and others. You will likely want to trigger the same code as in `touchesEnded:`.

Sprite Kit offers a few helper methods to assist us with touches and detecting nodes in which touches happened. We will list some of them here. These are all methods of `SKScene`:

- `[touch locationInNode:(SKNode*)node]`: This is an example of one of the methods to convert location from `UIView` coordinates to coordinates in `SKNode`. This is useful since touch objects in those methods have `UIView` coordinates and we operate with Sprite Kit coordinates.
- `[self nodeAtPoint:(CGPoint)p]`: This method returns the node in the scene's hierarchy that intersects the provided point. This can be useful to detect touches on certain nodes without tedious calculations.

Now that we have learned the basics, let's implement the basic functionality of touch handling. We will move the character sprite by dragging it around. We will need to perform a few steps to accomplish this, which are as follows:

1. Add `@property (assign) BOOL selected;` to your player class in `ERGPlayer.h`—we will use this to handle the state of the player sprite; whether we should drag it when the user drags a finger on the screen or disregard such touches.
2. Add the following code to the beginning of the `touchesBegan:` method in `ERGScene`:

```
UITouch *touch = [touches anyObject];
SKSpriteNode *touchedNode = (SKSpriteNode
*)
[self nodeAtPoint:[touch
locationInNode:self]];

if (touchedNode.name == playerName) {
    ERGPlayer *player = (ERGPlayer *)
touchedNode;
    player.selected = YES;
    return;
}
```

On the first line, we get the `touch` object from the `touches` set and then try to find if it intersects any node. To do that, we call the `nodeAtPoint:` method and provide the location of the touch by converting `touch` to the Sprite Kit coordinates in the `locationInNode:` method. We want to ignore taps on the background itself, so this is why we check to make sure the player was tapped.

If we tap the player, we set `selected` to `YES` so that the `touchesMoved:` method knows that we are dragging the character sprite. After that, we call `return` to exit the method, as after this return statement, we have other code that we don't want to trigger.

The next thing that we need to handle is the `touchesMoved:` method. We haven't used that before, so type it as follows:

```
-(void)touchesMoved:
(NSSet *)touches withEvent:(UIEvent *)event
{
    UITouch *touch = [touches anyObject];

    ERGPlayer *player = (ERGPlayer *)[self
childNodeWithName:playerName];
    if (player.selected) {

        player.position = [touch locationInNode:self];
    }
}
```

This method (`touchesMoved:`) checks if the player is actually selected, and if it is, the coordinates are changed to the touch location. We only want to drag the character sprite if touches began on it.

And the thing that we need to do last is remove the selected property after the touch has ended:

```
-(void) touchesEnded:(NSSet *)touches
withEvent:(UIEvent *)event
{
    ERGPlayer *player = (ERGPlayer *)[self
childNodeWithName:playerName];
    if (player.selected) {
        player.selected = NO;
    }
}
```

Now, if you build and run the project, you will be able to drag the character sprite around. Another option to consider is to move the score label around the screen to the position that you would like.

You might have noticed that we are checking the view hierarchy each time we want to grab a pointer to the player. This is not the optimal way to do it. Another option to consider is using a player as a property of the scene. This looks like a fast and easy way to do it, but consider that you might have a few objects on screen, or even dozens of them that need interaction. Having 20 properties for this reason does not seem feasible.

You should also know that traversing a node tree is not a very fast or efficient operation. Imagine that you have 1,000 nodes. If you need to find a node by name and it is not there, the code will traverse each of these nodes as it needs to make sure that there is none with your name.

This traversal might happen a few times during just one frame calculation, and since this happens many times per second, you might get a very real slowdown from this. You might want to cache certain nodes in weak pointers in case you need them later. Usually, you might want to avoid many node tree traversals.

# Using gesture recognizers

Gesture recognizers allow us to not bother with the low-level code that was explained earlier in this chapter. Without gesture recognizers, certain things might be extremely hard to implement, such as pinching in and out or rotating. Thankfully, Apple has handled all of this for us.

We might want to increase and decrease the speed of scrolling for testing reasons, so we will implement this feature.

The first thing that comes to mind is adding a gesture recognizer to our scene, pointing it to some method, and be done with it. But unfortunately, this won't work—SKNodes and SKScenes do not support adding gesture recognizers. But there is a way to make this work.

SKScene has a handy method, `-(void)didMoveToView:(SKView *)view`, which we can use, and it gets called each time a scene is about to be attached to some view. When we run our game, this happens right after the scene creation, and thus it is a useful place to set up our gesture recognizers. The following is the code for this method:

```
- (void)didMoveToView:(SKView *)view
{
    UISwipeGestureRecognizer *swiper =
    [[UISwipeGestureRecognizer alloc]
    initWithTarget:self
    action:@selector(handleSwipeRight:)];
    swiper.direction =
    UISwipeGestureRecognizerDirectionRight;
    [view addGestureRecognizer:swiper];

    UISwipeGestureRecognizer *swiperTwo =
    [[UISwipeGestureRecognizer alloc]
    initWithTarget:self
    action:@selector(handleSwipeLeft:)];
    swiperTwo.direction =
    UISwipeGestureRecognizerDirectionLeft;
    [view addGestureRecognizer:swiperTwo];
}
```

We create two gesture recognizers, set the methods to be called, and specifically to `UISwipeGestureRecognizer`, set the swipe direction. Thus, if we swipe to the right, one method is called, and if we swipe to the left, another one is triggered. These methods are fairly straightforward, as they only increase or decrease speed:

```
- (void) handleSwipeRight:(UIGestureRecognizer
*)recognizer
{
```

```

        if (recognizer.state ==
    UIGestureRecognizerStateRecognized) {
            backgroundMoveSpeed += 50;
        }
    }

    - (void) handleSwipeLeft:(UIGestureRecognizer
    *)recognizer
    {
        if (recognizer.state ==
    UIGestureRecognizerStateRecognized &&
    backgroundMoveSpeed > 50) {
            backgroundMoveSpeed -= 50;
        }
    }
}

```

The interesting part here is a second check for background speed in the `handleSwipeLeft:` method. We don't want to go into negative or zero speed, since our background generator works only with positive scrolling (from right to left), and if we have negative scrolling, the background will end.

Another thing that we need to remember is to remove the gesture recognizer once the scene gets removed from the view, as we might get another scene in the same view that doesn't know how to handle these methods. Thus, your application will crash at this point. To prevent this, add this method:

```

- (void)willMoveFromView:(SKView *)view
{
    for (UIGestureRecognizer *recognizer in
    view.gestureRecognizers) {
        [view removeGestureRecognizer:recognizer];
    }
}

```

This gets called when our scene is about to be removed from the view. It iterates over the gesture recognizers in the view, removing each of them.

# Accelerometer

Accelerometers and gyroscopes in modern devices are probably the things that have contributed to much of the popularity of iOS devices. Many people still play Doodle Jump, a game where you control a character that jumps on platforms, and is controlled by tilting your device.

Accelerometer controls have many advantages, which are as follows:

- You don't clutter the screen with game controls
- You don't need to cover the (already small) screen space with fingers while playing your game
- The accelerometer controls can imitate real-life interactions—such as a car steering wheel
- With new hardware (gyroscope) available on the latest devices (starting with iPhone 4), you get precise data about the device's position, which leads to smooth controls

Some games benefit greatly from accelerometer support, and we are going to implement this in our game.

## Note

Accelerometer data is available only on devices. Thus, if you test it on the simulator, this code won't do anything, as the accelerometer will return 0 all the time. Consider getting an Apple developer license if you have not got one already, as it offers the option of testing on a real device. You can get it at <https://developer.apple.com/programs/ios/>. It is especially useful for developing games, as the frame rate in Sprite Kit in the simulator is far from the one that you get on real devices. The simulator uses a much faster CPU and much faster memory, but the emulated rendering pipeline is much slower, and you can get very low fps in most trivial situations. Don't trust the frame rate in the simulator!

The accelerometer is handled by the Core Motion library, so you will need to include that in your project. To do that, simply add `@import CoreMotion;`, where all imports are at the top of the `ERGMyScene.h` file.

## Note

`@import` is the new way to import frameworks into your code, and it was introduced with iOS 7 and Xcode 5. Earlier, you had to add a framework to your project and use `#import` everywhere you needed it. Now you can use `@import` and forget about adding a framework in the project's

settings. The only drawback to this is that you can only use the Apple frameworks this way. Let's hope it will change to all libraries some day.

The next thing that you need to do is add the Core Motion manager as a property to your scene.

To do that, add the following line to `ERGMyScene.h`:

```
@property (strong, nonatomic) CMMotionManager
*manager;
```

The manager handles all accelerometer data, and we need it to use the accelerometer's input. In order to start getting accelerometer data, we need to call a few more methods.

In the `initWithSize:` method of `ERGMyScene.m`, add the following lines:

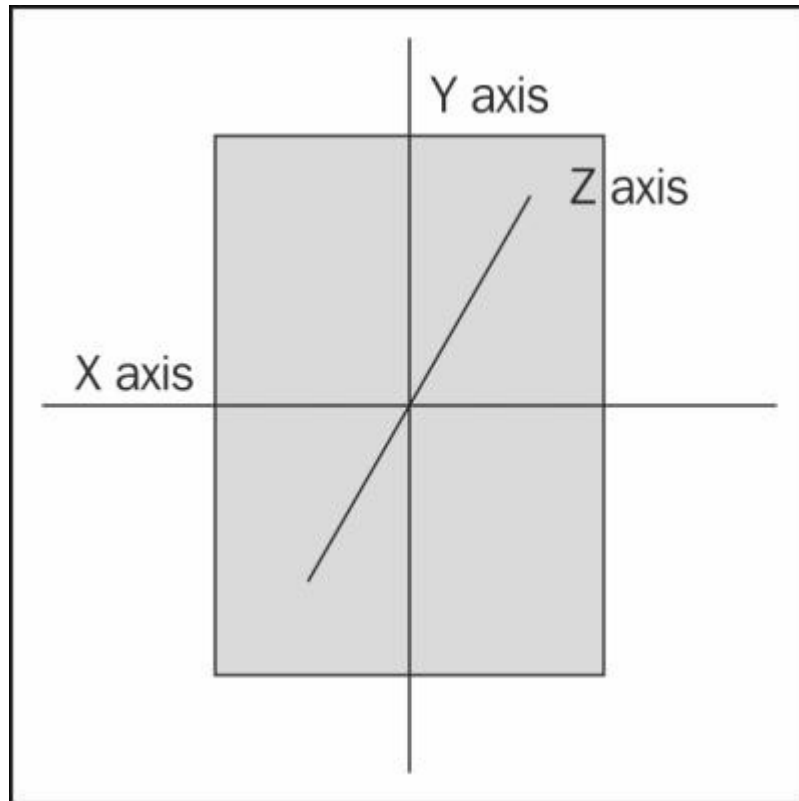
```
self.manager = [[CMMotionManager alloc] init];
self.manager.accelerometerUpdateInterval = 0.1;
[self.manager startAccelerometerUpdates];
```

In the first line, we instantiate the manager, and in the second one, we set the update interval—how often the values in the motion manager will be updated. On the third line, we tell the manager to start updating, as it won't do anything before that.

## Note

Using the accelerometer increases the power use of the device. Heavy use will drain the battery faster, thus making your game unlikely to be played further. Only use the accelerometer when you actually need to, and stop polling it if something stops the application or if you don't need to parse the accelerometer data at this time. You can use `[self.manager stopAccelerometerUpdates];` to stop the accelerometer.

Accelerometer data can be read from `manager.accelerometerData`, and it consists of three fields—X, Y, and Z axis angle. Each of them represents a value from -1 to 1.



*Accelerometer data axes*

As you can see in the previous diagram, there are three axes, and rotating the device around each of them changes only that axis' values. You can see this if you add the following line to the `update:` method of `ERGScene`:

```
NSLog(@"%@", self.manager.accelerometerData);
```

In order to see what is going on, run the application and start rotating the device along a different axis, and you will see the value of that go from -1 to 1. We will use the accelerometer data to move the character sprite. Remember that the accelerometer does not work on the simulator.

How can we do that? We may read the accelerometer data and set the sprite position to certain values based on that. But how do we set these positions? We need some baseline value so that our character stands on the ground.

As you rotate the device, you can see that values change very fast, and different device positions yield different results. For every person, that baseline will be different, and we need to acknowledge that. To compensate that baseline, we will read the starting value on the launch of an application and base our movement on it.

For our landscape mode game, reading the X axis' coordinates will be most useful. To get it, we poll

```
self.manager.accelerometerData.acceleration.x.
```

In order to get the data that we need, we have to save that baseline value. This is where it gets interesting. Accelerometer data is not available straight away when you ask it to start updating. It takes some time to poll hardware. First, add `@property (assign) float baseline;` to `ERGMyScene.h`—we will store the offset for accelerometer data here.

In the `init` method, right after the `startAccelerometerUpdates` method, add the `performSelector` call to execute the method that sets the baseline:

```
[self performSelector:@selector(adjustBaseline)
withObject:nil afterDelay:0.1];
```

After this, we need to add the `adjustBaseline` method:

```
- (void) adjustBaseline
{
    self.baseline =
self.manager.accelerometerData.acceleration.x;
}
```

We will also need some value to be multiplied with the accelerometer data, and having it as a magic number is never good, so let's add a new line to the `Common.h` file:

```
static NSInteger accelerometerMultiplier = 15;
```

Add the following code to the bottom of your update method in `ERGMyScene.m`:

```
ERGPlayer *player = (ERGPlayer *) [self
childNodesWithName:playerName];
player.position =
CGPointMake(player.position.x, player.position.y
- (self.manager.accelerometerData.acceleration.x
- self.baseline) * accelerometerMultiplier);

if (player.position.y < 68) {
    player.position =
CGPointMake(player.position.x, 68);
}
if (player.position.y > 252) {
    player.position =
CGPointMake(player.position.x, 252);
}
```

First, we get the pointer to our player sprite. Then, we calculate the new position by multiplying the accelerometer data and the out multiplier to get the sprite position on the Y axis. As we don't want to have our character fly off screen, we add a check; if it goes below or over a certain point, its vertical position is reset to the minimum value. Build and run the project to see if everything works.

# Physics engine

The next big topic that we will discover is physics in our application. Physics-based games are very popular on the App Store. Angry Birds, Tiny Wings, and Cut the Rope are all physics-based. They offer lifelike interactions that are fun and appealing.

Another reason why we might want to have a physics engine is that it offers a lot of functionality "for free". You no longer need to calculate different complicated collisions, forces, and all things that may affect your nodes. Most of these things are handled for you by a physics engine.

## Physics simulation basics

If you want your node to participate in physics simulation, you have to add a physics body to it. There are many available methods to generate physics bodies, and they are as follows:

- `BodyWithCircleOfRadius`: This method creates a circular physics body. It is the fastest physics body, and if you have many objects that need to be simulated, consider setting their physics body to a circle.
- `BodyWithRectangleOfSize`: This is the same as the previous one, but with a rectangle. Usually, passing a frame of a node is sufficient enough for most games.
- `BodyWithPolygonFromPath`: This creates a physics body from `CGPath`. This is useful if you have a very complex sprite and you want it to be simulated as real as possible.
- `BodyWithEdgeFromPoint:toPoint`: This is useful to create edges such as ground level.

There are more methods available; you can check them in the [SKPhysicsBody](#) class reference.

In order to simulate physics on many bodies in real time, there are many different optimizations and simplifications. Let's cover some of them:

- All movements, just like in real life, are handled by impulses and forces. The impulse is applied instantaneously; for example, if a moving ball collides with a standing ball, the standing ball gets an impulse in one moment. Force is a continuous effect that moves the body in some direction. For example, launching a rocket is slow and steady as force gets applied that pushes it up. All forces and impulses add to each other; thus, you can have very complex movements with little hassle.
- All bodies are considered rigid—they can't deform as a result of physics simulation. Of course, you can write your system over existing

physics simulation to do that for you, but it is not provided out of the box.

Each physics body has many properties, some of which are as follows:

- **Mass**: This is the mass of the body in kilograms. The more massive the body is, the harder it is to move by impulses or forces.
- **Density**: This indicates how much mass the body has in a fixed amount of volume. The denser the body is, the more mass it has for the same volume.
- **Dynamic**: This is the property that allows you to apply impulses or forces to the body. Sometimes it is not needed; for example, ground will always be static and not dynamic. You might also pick dynamic if you want to move your body yourself, without the physics engine touching it.
- **Restitution**: This is the property that determines how "jumpy" or "bouncy" the body is. The higher this is, the higher the knockback is. This ranges from 0 to 1, but you can set this higher than 1, and in this case, the body will accelerate each time after the collision, which may eventually lead to a crash.
- **usesPreciseCollisionDetection**: This is used to handle very fast moving objects. There are certain situations in which collision might not get detected, for example, if in one frame, the body is in front of another body, and in the next frame it is completely behind it without touching it; in this case, the collision will never be detected, and you don't want this. This method is very CPU-intensive, so use it only if absolutely necessary.
- **affectedByGravity**: This property is set to `NO` if you don't want some objects such as balloons to be affected by gravity.
- **categoryBitMask**: This is the bitmask that identifies the body. You might choose different bitmasks for different objects.
- **collisionBitMask**: This is the bitmask that identifies what bodies this body collides with. You might want some objects to collide with others, and other objects to pass through others. You will need to set the same bitmasks for objects that you want to collide.
- **contactBitMask**: This is the bitmask that identifies the body for handling collisions manually. You might want to have special effects when things collide; that's why we have this. When bodies have the same bitmask, a special method will get called for you to handle such collisions manually.

## Implementing the physics engine

We might as well start implementing physics in our engine in order to see what the physics engine gives us. First, we need to set the physics body to our player sprite. In order to do that, we need more constants in `Common.h`:

```
static NSInteger playerMass = 80;
```

```
static NSInteger playerCollisionBitmask = 1;
```

Next, change your `init` method in `ERGPlayer.m` to look as follows:

```
-(instancetype)init
{
    self = [super
initWithImageNamed:@"character.png"];
    {
        self.name = playerName;
        self.zPosition = 10;
        self.physicsBody = [SKPhysicsBody
bodyWithRectangleOfSize:CGSizeMake(self.size.width,
self.size.height)];
        self.physicsBody.dynamic = YES;
        self.physicsBody.mass = playerMass;
        self.physicsBody.collisionBitMask =
playerCollisionBitmask;
        self.physicsBody.allowsRotation = NO;
    }
    return self;
}
```

After creating the sprite and setting the name and z order, we start with the physics body. We create a physics body with the same size as that of our sprite—it is accurate enough for our game. We set it as dynamic, as we want to use forces and impulses on our character. Next up, we set the mass, collision bitmask, and disallow rotation, as we expect our character sprite to never rotate.

This looks sufficient for our character sprite. What can we do with backgrounds?

Obviously, we need the player to collide with the top and bottom of the screen. This is when other methods come in handy. Add this code right before the return statement in the `generateNewBackground` method in `ERGBackground.m`:

```
background.physicsBody = [SKPhysicsBody
bodyWithEdgeFromPoint:CGPointMake(0, 30)
toPoint:CGPointMake(background.size.width, 30)];
background.physicsBody.collisionBitMask =
playerCollisionBitmask;
```

We set the bitmask to be the same as we want the player to collide with the background. But here comes the problem. Since all physics bodies need to be convex, we can't create the physics body that accommodates both the top and bottom collision surfaces. We might think of adding a second collision body to the same node, but it is not supported.

There is a small hack around this—we will just create a new node, attach it as a child to the background node, and attach the top collision body to it. Add the following code after the previous code and right before the return statement:

```
SKNode *topCollider = [SKNode node];
topCollider.position = CGPointMake(0, 0);
topCollider.physicsBody = [SKPhysicsBody
bodyWithEdgeFromPoint:CGPointMake(0,
background.size.height - 30)
toPoint:CGPointMake(background.size.width,
background.size.height - 30)];
topCollider.physicsBody.collisionBitMask = 1;
[background addChild:topCollider];
```

Here, we create a new physics body with `EdgeFromPoint`, from the left edge to the right edge. This way, we effectively have two colliding bodies on one background.

As we now want to have physics-based controls, you may remove all the code that handles movement. These tapping and actions methods are as follows:

- `touchesBegan:`
- `touchesMoved:`
- `touchesEnded:`
- `swipeLeft`
- `swipeRight`

Or, you may just follow us. Change the `didMoveToView` method to the following:

```
- (void) didMoveToView:(SKView *)view
{
    UILongPressGestureRecognizer *tapper =
    [[UILongPressGestureRecognizer alloc]
    initWithTarget:self
    action:@selector(tappedScreen:)];
    tapper.minimumPressDuration = 0.1;
    [view addGestureRecognizer:tapper];
}
```

Here, we initialize and use the new gesture recognizer that will set the player's state to `accelerating`, meaning that the player is moving up now. To handle this, add a new property to `ERGPlayer.h`—`@property (assign) BOOL accelerating;`

After this, add the following code to `ERGMyScene.m`:

```
- (void) tappedScreen:(UITapGestureRecognizer
*)recognizer
{
    ERGPlayer *player = (ERGPlayer *)[self
childNodesWithName:@"player"];
    if (recognizer.state ==
    UIGestureRecognizerStateBegan) {
        player.accelerating = YES;
    }

    if (recognizer.state ==
    UIGestureRecognizerStateEnded) {
        player.accelerating = NO;
    }
}
```

As said before, we need this method to set an accelerated property on the player—if it is active, we will apply a certain force on each frame of the player. To do this, add the following to your update method:

```
[self enumerateChildNodesWithName:@"player"
usingBlock:^(SKNode *node, BOOL *stop) {
    ERGPlayer *player = (ERGPlayer *)node;
    if (player.accelerating) {
        [player.physicsBody
        applyForce:CGVectorMake(0, playerJumpForce *
        timeSinceLast)];
    }
}];
```

This code uses the `playerJumpForce` variable, which sets it in `Common.h`, as well as the gravity vector:

```
static NSInteger playerJumpForce = 8000000;
static NSInteger globalGravity = -4.8;
```

We set gravity to custom as the real world's gravity is too high, and this strips away the fun and precise controls of our game.

To affect gravity, add the following statement in `ERGMyScene.m` in the `initWithSize:` method:

```
self.physicsWorld.gravity = CGVectorMake(0,
globalGravity);
```

In order to make everything work, we need to comment out or delete the old methods that are in the way. Comment out or delete the following methods:

`touchesBegan`, `touchesMoved`, `touchesEnded`, and `touchesCancelled`. If you still have other gesture recognizers such as the long press gesture recognizer in the `didMoveToView:` method, remove them too.

After these changes, we have a really precise and fun way to control our character. We can control the jump of the player sprite just by tapping on the screen for a certain amount of time. This mechanism is used in many endless runners, and is both fun and addictive. You have to not only react fast but also predict what can happen, since applying force to the character does not change your position instantaneously as your movement carries some momentum, so you have to plan accordingly.

## Summary

In this chapter, we have learned how to use the touch controls and gesture recognizers to control characters on the screen. We have found out how to move sprites with touch and learned the basics of the physics engine in Sprite Kit. Integrating the physics engine in your game is a great way to handle collision detection and add many interesting behaviors without bloating your code.

In the next chapter, we will learn about animations, how to implement them, and related details, such as textures and texture atlases.

# Chapter 4. Animating Sprites

In the last few chapters, we managed to get the basics of our game out of the way—we already have an infinite scrolling background, we handle collision detection with the physics engine, and we already have established controls. How can we improve our game? Of course, it can be enhanced with animation.

Our character does not move his legs while running, and that is clearly not the way it should be. In this chapter, we are going to add animations to our character—he will animate when he is running and jumping. We will also learn about texture atlases and how to efficiently animate our game.

## What is animation?

In order to achieve the desired effect of animating our character, we need to understand a little more about animation. Behind the scenes, we will be showing a rapid succession of images, which gives the illusion of movement. This is exactly what we are going to do.

Animation is an easy way to add life to your game, to make it look nice and engaging. We are going to use animation actions in order to run animations on our character. An easy way to do this is by simply adding animation frames to some array, feeding it to an action, and instructing the character sprite to run it as shown in the following figure:



*Changing the animation frames creates an illusion of movement*

But there are some problems with it. Rendering individual frames from separate textures is not a fast task. It requires a lot of so-called *draw calls*. Each time you draw something on screen, you transfer that data to the video chip, and it performs the drawing. But each of these calls is expensive as it bears some overhead. If you have many animated objects on screen, you might experience slowdowns (lags) and the frame rate may drop. To optimize our sprite drawing, we will use texture atlases.

# What is a texture atlas?

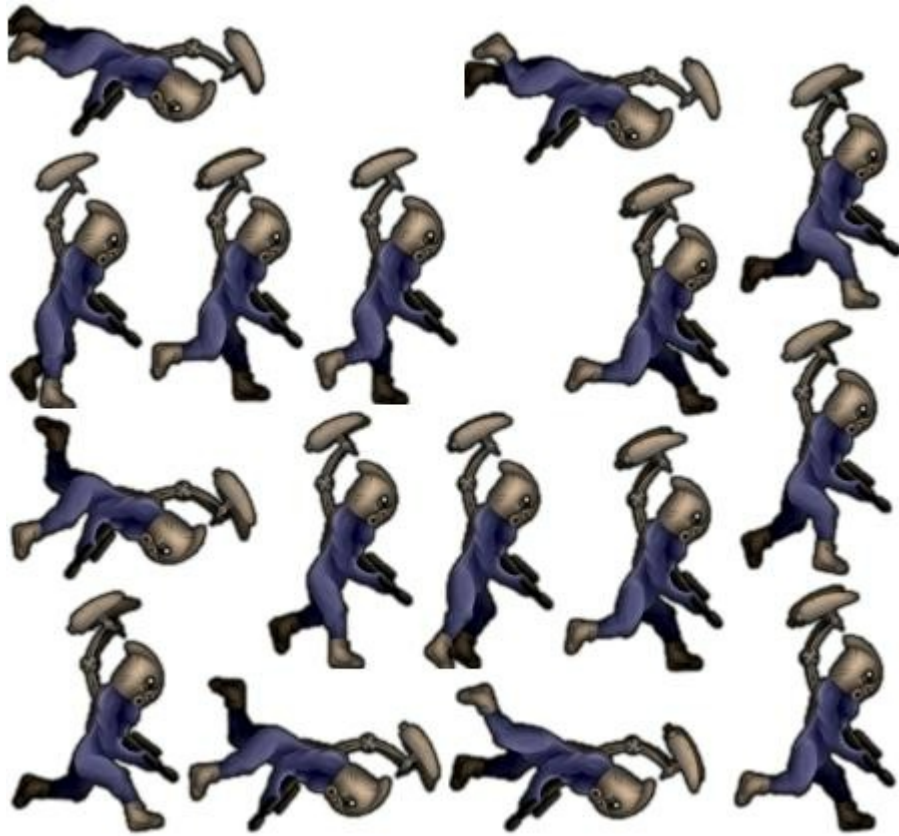
To understand what a texture atlas is, check the figure following this section. As you can see, an atlas is an image that contains many subimages. Our game is able to access certain images in a texture atlas due to a special configuration file that keeps the coordinates of each image in a texture atlas.

Before Xcode 5 and Sprite Kit, you had to use third-party tools in order to create texture atlases, but now, all you need to do is create a folder named `name.atlas`, where the name can be anything; add images to it and add that into your project in Xcode. Xcode will handle everything for you transparently, and you won't have to worry about plists, coordinates, efficiency, and everything else.

Benefits that texture atlases provide are as follows:

- All drawing from one atlas can be processed in one draw call, thereby increasing performance dramatically.
- If your image has empty space, it will be cropped, and when you need the image, it will be restored. This way, you save memory, and your applications are smaller, which is a good thing.

However, you should remember that a texture atlas may not exceed 2000 x 2000 pixels, and if you have images that are larger than that, there is no point putting them into atlases. If the sum of all images exceeds this value, the second image of the atlas will be created in order to fit all images, as you can see in the following figure:



*Images in a texture atlas*

Texture atlases are also smart. If you run on a retina device and you have two versions of the atlas—one for the retina and one for regular resolution—it will use the correct images. All you have to do is have one atlas with all of the images, but retina images should have a `@2x` suffix, for example, `spaceship.atlas`, and images such as `spaceship1.png` and `spaceship1@2x.png`. Xcode will create different texture atlases for different devices, so you don't have to worry about memory and other limitations.

# Adding animations to our project

In order to add animations to our endless runner game, we need to perform the following steps:

1. Find `run.atlas`, `shield.atlas`, `deplete.atlas`, and `jump.atlas` in the resources for this chapter. Drag-and-drop them into the project and be sure to check **Copy items into destination group's folder**.
2. Add the following property to `ERGPlayer.h`. We will use it to store animation frames:

```
@property (strong, nonatomic) NSMutableArray
*runFrames;
```

3. Add the following code at the end of the `ERGPlayer.m` `init` method:

```
[self setupAnimations];

[self runAction:[SKAction
repeatActionForever:
[SKAction animateWithTextures:self.runFrames
timePerFrame:0.05 resize:YES restore:NO]]
withKey:@"running"];
```

First, we will call the `setupAnimations` function, where we will create an animation from the atlas. On the second line, we will create an action that repeats forever, takes an array of animation frames, and animates the sprite, showing each frame for 0.05 seconds. The `resize` property is needed to adjust the size of the sprite if it is smaller or larger in a new frame of animation. The `restore` property changes the sprite back to the texture it had before animation. If we add a key to the animation, we will be able to find it later and remove it if needed.

4. The next thing to add is the method that makes that animation; add it to `ERGPlayer.m`:

```
- (void) setupAnimations
{
    self.runFrames = [[NSMutableArray alloc]
init];
    SKTextureAtlas *runAtlas =
[SKTextureAtlas atlasNamed:@"run"];

    for (int i = 0; i <
[runAtlas.textureNames count]; i++) {
        NSString *tempName = [NSString
stringWithFormat:@"run%.3d", i];
        SKTexture *tempTexture = [runAtlas
textureNamed:tempName];
        if (tempTexture) {
```

```

        [self.runFrames
         addObject:tempTexture];
    }
}

```

On the third line of this method, we create a new array for animation frames. Then, we load the texture atlas named `run`. In the `for` loop, we create a new name for the texture and load it with that name into the frames array. We need to check for nil, as adding nil objects to an array raises an exception, and we don't want that. The format specifier, `@\"run%.3d\"`, might have caught your attention. It means that we want a string starting with `run` and ending with a decimal number `.3d` means that the number has to be at least 3 digits long; if it is smaller, replace the missing digits with zeroes. The names that this code generates will look like `run000`, `run001`, and so on. If you are using a retina device, they will automatically be adjusted to `run000@2x` and `run001@2x`. You don't need to specify the file extension. You can read more about format specifiers in the [NSString](#) class reference.

Build and run the project in a simulator. As you can see, our character has gained nice animation. Try adjusting the scroll speed and animation speed to fit each other better. There are some other adjustments we can do, which are as follows:

- If the player is already running, we don't want to start the animation again
- We may want a way to stop an animation

To accomplish these tasks, add the following two methods to `ERGPlayer.m`:

```

- (void) startRunningAnimation
{
    if (![self actionForKey:@"running"]) {
        [self runAction:[SKAction
repeatActionForever:
[SKAction animateWithTextures:self.runFrames
timePerFrame:0.05 resize:YES restore:NO]]
withKey:@"running"];
    }
}

- (void) stopRunningAnimation
{
    [self removeActionForKey:@"running"];
}

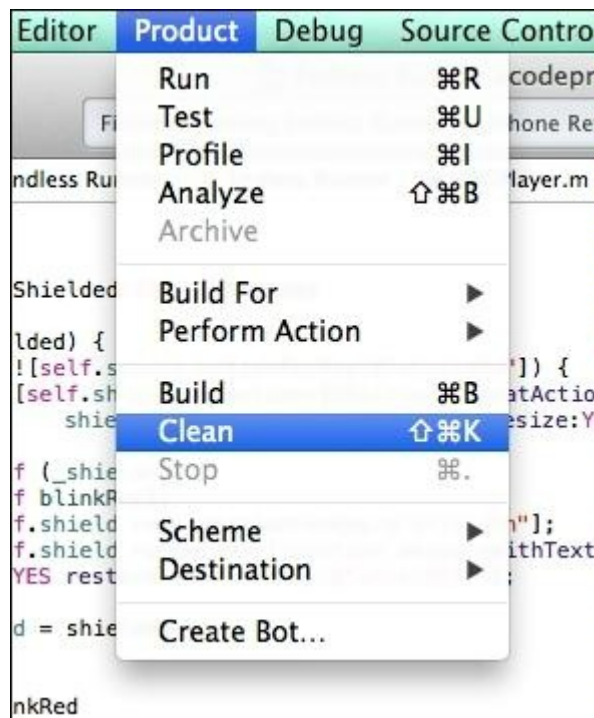
```

In the `startRunningAnimation` method, we do the same action that we did before in the `init` block, but we check if the player already has an

animation with this key, and if he does, we do nothing here. In the second method, we find the animation and remove it, effectively stopping it.

## Tip

If you change the images in your project or add new ones, Xcode doesn't make new atlases. In order to do that, execute the **Run** and **Clean** commands from the **Product** menu on the Xcode file menu. This forces Xcode to recreate texture atlases. You can see this in the following screenshot:



Next up is the jump animation. We will add it in the same way as before. The following things need to be done:

1. Check if you have `jump.atlas` in your project. If you don't, add `jump.atlas` to your project in the same way as `run.atlas`.
2. Add the following property to `ERGPlayer.h`:

```
@property (strong, nonatomic) NSMutableArray  
*jumpFrames;
```

3. Add the following code to the `setupAnimations` method in `ERGPlayer.m`. It does the same, in that we create an array and add frames to it:

```
self.jumpFrames = [[NSMutableArray alloc]  
init];
```

```

        SKTextureAtlas *jumpAtlas =
[SKTextureAtlas atlasNamed:@"jump"];

        for (int i = 0; i <
[runAtlas.textureNames count]; i++) {
            NSString *tempName = [NSString
stringWithFormat:@"jump%.3d", i];
            SKTexture *tempTexture = [jumpAtlas
textureNamed:tempName];
            if (tempTexture) {
                [self.jumpFrames
addObject:tempTexture];
            }
        }
    }
}

```

#### 4. Add a new method to handle jumping:

```

- (void) startJumpingAnimation
{
    if (![self actionForKey:@"jumping"]) {
        [self runAction:[SKAction
sequence:@[[SKAction
animateWithTextures:self.jumpFrames
timePerFrame:0.03 resize:YES
restore:NO], [SKAction runBlock:^(
    self.animationState =
playerStateInAir;
        )]]] withKey:@"jumping"];
    }
}

```

The only interesting thing in this method is the new action—the `runBlock` action. It lets you run any code in action, and here we set the player's state to `playerStateInAir`, which means that the player is neither running nor jumping.

This is pretty much it. But how can we determine when to execute the jump animation or run animation? We don't want our character to run while they are in the air. This is why we need a state system for a player sprite.

# Character states

In order to correctly handle states, we will expand our character code to handle different states that can occur. Firstly, we need to identify the states, which are as follows:

- **Running state:** This is the default state when the character is running on the ground
- **Jumping state:** This is the state when we press a button to jump, but it should be limited so that we don't continue the jumping state when we are in the air
- **In air state:** This is the state when the character is still in the air following a jump

In order to use these states, let's define them in `ERGPlayer.h`:

```
typedef enum playerState {  
  
    playerStateRunning = 0,  
    playerStateJumping,  
    playerStateInAir  
  
} playerState;
```

This code creates a new type of variable that is internally a usual integer, and we use that to identify the state of the character. We could use integer, character, or even string, but these can lead to problems. We will have to remember what state corresponds to what integer, and here we just write the state as it is.

Now, we need to store this state in our character data. Add the following line to `ERGPlayer.h`:

```
@property (assign, nonatomic)playerState  
animationState;
```

In this case, we want to execute some custom code when the `animationState` property is changed, so we implement our own `setAnimationState:` method. It is called each time we change the `animationState` property:

```
- (void)  
setAnimationState:(playerState)animationState  
{  
    switch (animationState) {  
        case playerStateJumping:
```

```

        if (_animationState ==
playerStateRunning) {
            [self stopRunningAnimation];
            [self startJumpingAnimation];
        }
        break;
    case playerStateInAir:
        [self stopRunningAnimation];
        break;
    case playerStateRunning:
        [self startRunningAnimation];
        break;
    default:
        break;
}

_animationState = animationState;
}

```

This method is triggered every time we try to set `self.animationState`, and instead of the default implementation, it goes through this. We have a switch here that looks for what kind of `animationState` we received. If it is the jumping state, and the previous state was running, we stop the running animation and start the jumping animation. As you may remember, after we have finished the jumping animation, it changes its state to `playerStateInAir`. If we get to the running state, we start the running animation. After everything has been handled, we set our instance variable to the new value that we received. We do this only now as we need to know the last known state, and we can get it from `_animationState`.

## Tip

### Properties and instance variables

There are two ways to store data in your classes—by using properties or instance variables. Instance variables are easy and fast to access, and you might use them if you need high-performance code. Properties usually add some overhead to access, but offer much more. Properties are essentially instance variables that are accessed with special methods. If you have `@property NSString *myString`, it will be stored internally in `_myString`, even if you don't add anything else. There will also be two methods that access the said variable—`(NSString *)myString` and `(void) setMyString: (NSString *)myString`. These methods are hidden and generated by the compiler. Each time you access properties, these methods are called. All of this is generated for us by Xcode behind the scenes. However, if you need to change the way they are accessed—like we do in this chapter—you can redefine these setter and getter methods. You might want to inform someone or call some other methods from there. This functionality adds a little bit of

overhead, which in most situations is negligible. But should you need to perform some very complicated and CPU-intensive calculations, make sure to check the performance difference. There are also other technologies that are only possible with properties such as key-value observing.

The next thing that needs to be done is the location where we change states. The place where everything changes is the `update:` method in `ERGMyScene.m`; find the line where we enumerate child nodes with the name `player` and replace the current implementation with this:

```
[self enumerateChildNodesWithName:@"player"
usingBlock:^(SKNode *node, BOOL *stop) {
    ERGPlayer *player = (ERGPlayer *)node;
    if (player.accelerating) {
        [player.physicsBody
        applyForce:CGVectorMake(0, playerJumpForce *
        timeSinceLast)];
        player.animationState =
        playerStateJumping;
    } else if (player.position.y < 75) {
        player.animationState =
        playerStateRunning;
    }
}];
```

The preceding code searches for a `player` node and applies force to it—this is all old code. The thing that was added here is the state change. Right after we apply force, we know that we are probably jumping. Handling further jumping continues in the previously discussed method. If the position of the sprite on the y axis is less than `75`, the state is probably running, since even the smallest impulse will get us out of that position. The actual position for the default sprite is `68`, but as the frames of animations change, this can fluctuate up to `75`. Build and run the project to see the animation in action.

# Adding shield animations

Other animations and methods that we might want to add are the shield animations. A shield is something that protects the player from different hazards. It has on and off animations. You can check how the effect looks in the following screenshot:



*The background looks dull*

Let's check the course of the action:

1. Add the following properties to `ERGPlayer.h`:

```
@property (strong, nonatomic) NSMutableArray
*shieldOnFrames;
@property (strong, nonatomic) NSMutableArray
*shieldOffFrames;
@property (strong, nonatomic) SKSpriteNode
*shield;
@property (assign, nonatomic) BOOL shielded;
```

2. The first two properties are arrays for animation frames, the `shield` node is added to a character sprite in order to show shield animations (we can't show it on the character itself as it will disappear), and `shielded` is a state variable so that other nodes can find out if we are shielded or not.
3. The next step is to create a `shield` node in the `init` method of `EGPlayer`:

```
self.shield = [[SKSpriteNode alloc]
init];
self.shield.blendMode =
SKBlendModeAdd;
```

```
[self addChild:self.shield];
```

Here, we change `blendMode` to `add` since it results in a better visual effect.

## Tip

### Blending modes

A blending mode is the way how different pixels are added together. By default, `SKBlendModeAlpha` is used. It uses an alpha value of the pixel to determine which pixels and what percent of color of each pixel are visible, and which are not. Other blending modes are used if you need to stimulate light (the additive blending mode) or increase the brightness and color, or completely overlay one layer by another. The list of available blending modes can be found in the `SKBlendMode` class reference.

4. The next thing to do is add another custom setter for the `shielded` variable, where we will handle all animations:

```
- (void) setShielded:(BOOL)shielded
{
    if (shielded) {
        if (![self.shield
            actionForKey:@"shieldOn"]) {
            [self.shield runAction:[SKAction
                repeatActionForever:
                [SKAction
                    animateWithTextures:self.shieldOnFrames
                    timePerFrame:0.1 resize:YES restore:NO]
                withKey:@"shieldOn"];
            }
        } else if (!_shielded) {
            [self blinkRed];
            [self.shield
                removeActionForKey:@"shieldOn"];
            [self.shield runAction:[SKAction
                animateWithTextures:self.shieldOffFrames
                timePerFrame:0.15 resize:YES restore:NO]
                withKey:@"shieldOff"];
        }
        _shielded = shielded;
    }
}
```

5. First, we look for a new value that we are presented with. If it is `YES` and we don't have the animation running, we start the shield on animation. It repeats itself since we want the shield to animate while it is on. If we are presented with `NO`, the other portion of this method

gets triggered. It checks if the player was shielded before, and if it was, the character sprite blinks red, and it is time to run the shield off the animation. If there was no shield before, we don't want to play the shield dismissing animation. We want it to run only once. We also don't want to have the shield on animation repeating itself, so we remove the old action. After that, we set the `shielded` variable to the provided value.

6. Add the `blinkRed` method:

```
- (void) blinkRed
{
    SKAction *blinkRed = [SKAction sequence:@[

        [SKAction colorizeWithColor:[SKColor
redColor] colorBlendFactor:1.0 duration:0.2],

        [SKAction waitForDuration:0.1],

        [SKAction colorizeWithColorBlendFactor:0.0
duration:0.2]]];
    [self runAction:blinkRed];
}
```

7. This method makes our player sprite red for a short while and then returns it back to its regular state. To do this, it uses an action named `colorizeWithColor:`, waits a little, and proceeds to colorize with the default value, which is `0.0`, meaning no colorization. The blend factor specifies the amount of target color to be added to the existing texture. The higher the blend factor is, the more intense is the color, and on `1.0`, all pixels of the texture will be of the chosen color.

8. Next is the `setupAnimations` method; we need to add new textures to it:

```
self.shieldOnFrames = [[NSMutableArray
alloc] init];
SKTextureAtlas *shieldOnAtlas =
[SKTextureAtlas atlasNamed:@"shield"];

for (int i = 0; i <
[shieldOnAtlas.textureNames count]; i++) {
    NSString *tempName = [NSString
stringWithFormat:@"shield%.3d", i];
    SKTexture *tempTexture =
[shieldOnAtlas textureNamed:tempName];
    if (tempTexture) {
        [self.shieldOnFrames
addObject:tempTexture];
    }
}

self.shieldOffFrames = [[NSMutableArray
alloc] init];
```

```

        SKTextureAtlas *shieldOffAtlas =
        [SKTextureAtlas atlasNamed:@"deplete"];

        for (int i = 0; i <
        [shieldOffAtlas.textureNames count]; i++) {
            NSString *tempName = [NSString
            stringWithFormat:@"deplete%.3d", i];
            SKTexture *tempTexture =
            [shieldOffAtlas textureNamed:tempName];
            if (tempTexture) {
                [self.shieldOffFrames
                addObject:tempTexture];
            }
        }
    }
}

```

9. Same as before, we create arrays and populate them with textures extracted from texture atlases.
10. In order to test the shielding functionality, you can add `self.shielded = NO` to the `startRunningAnimation` method, and `self.shielded = YES` to the `startJumpingAnimation` method. This way, while you are in the air, the shield will work, but if you touch the ground, it will fade away.

Build and run the project now to see if everything runs as expected.

Now that we have most of our animations out of the way, we can recall the process of adding animations to the node:

- Create an array to hold animation frames
- Extract textures from the texture atlas by iterating over it in a loop
- Create a method to run the animation action on the needed node
- Remove the action from the node if needed

# Adding a parallax background

As you may have noticed, we have huge grey windows in our background (see the screenshot in the *Adding shield animations* section), and it would be great to add a parallax background over it. We will do it the same way we did with the original background. We will add new sprite nodes and move them every frame.

The things that need to be done to add a parallax background are as follows:

1. Add `parallax.png` to `images.xcassets` in the Xcode project, either by drag-and-drop or by clicking on the plus button in Xcassets. Select `parallax` in the left Xcassets pane and move it from **1x** to **2x** by dragging, as we have high-resolution artwork (see the following screenshot).
2. The next thing we need is some variables in `Common.h`:

```
static NSString *parallaxName = @"parallax";
static NSInteger parallaxMoveSpeed = 10;
```

3. We also need to have `parallax` as a property in `ERGMyScene.h` in order to replace it with a new one. Add the following line:

```
@property (strong, nonatomic) ERGBackground
*currentParallax;
```

4. The next thing that we need to do is create that background layer. To do that, go to the scene's `initWithSize:` method, and right after the creation of `self.background`, add the code for the parallax layer's creation:

```
self.currentParallax = [ERGBackground
generateNewParallax];
[self addChild:self.currentParallax];
```

5. In order to generate a new parallax, we need to expand `ERGBackground.h`. Add the following method definition there:

```
+ (ERGBackground *)generateNewParallax;
```

6. In `ERGBackground.m`, write the method itself:

```
+ (ERGBackground *)generateNewParallax
{
    ERGBackground *background =
    [[ERGBackground alloc]
    initWithImageNamed:@"parallax.png"];
    background.anchorPoint = CGPointMake(0,
    0);
    background.name = parallaxName;
```

```

        background.position = CGPointMake(0, 0);
        background.zPosition = 4;
        return background;
    }

```

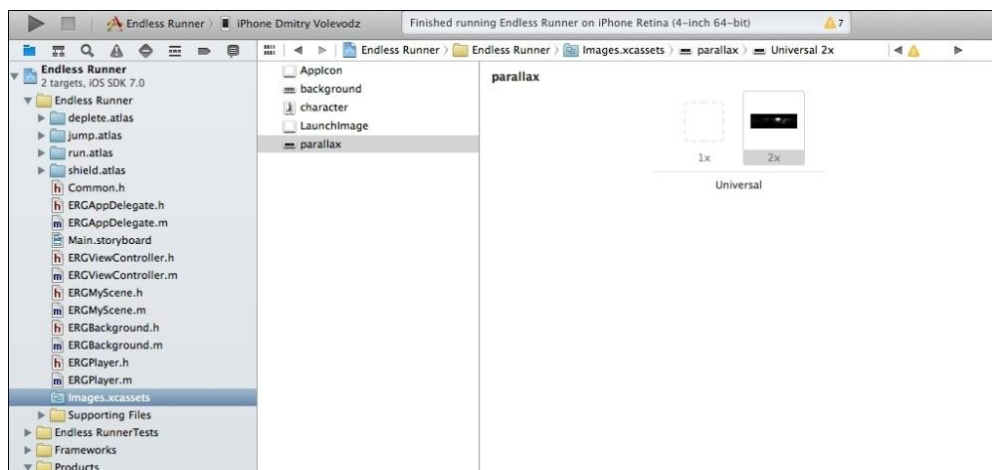
7. Here, we create the new sprite node, set its position and anchor point for our convenience, and adjust its position on the z axis. Within the same file, in the `generateNewBackground` method, change the assigned `zPosition` value from 1 to 5 so that it appears above the parallax layer.
8. And finally, in the `update:` method in `ERGMyScene.m`, add code to handle the moving and creating of new parallax layers:

```

    [self
    enumerateChildNodesWithName:parallaxName
    usingBlock:^(SKNode *node, BOOL *stop) {
        node.position =
        CGPointMake(node.position.x -
        parallaxMoveSpeed * timeSinceLast,
        node.position.y);
        if (node.position.x < -
        (node.frame.size.width + 100)) {
            // if the node went completely
            off screen (with some extra pixels)
            // remove it
            [node removeFromParent];
        }
    }];
    if (self.currentParallax.position.x <
    -500) {
        // we create new background node and
        set it as current node
        ERGBackground *temp = [ERGBackground
        generateNewParallax];
        temp.position =
        CGPointMake(self.currentParallax.position.x +
        self.currentParallax.frame.size.width, 0);
        [self addChild:temp];
        self.currentParallax = temp;
    }

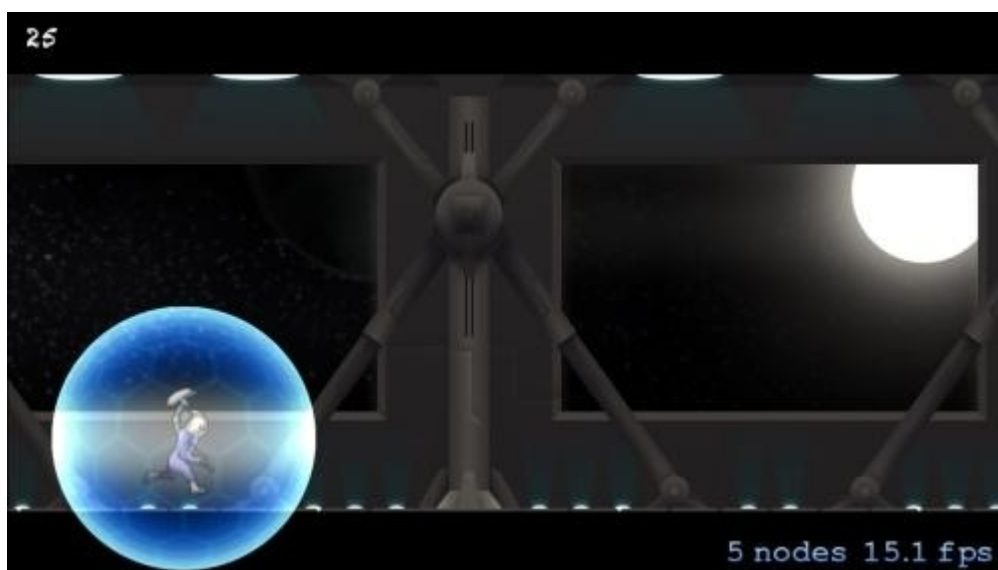
```

9. Here, we find a node with `parallaxName`, and we remove it if it is off the screen. If the current parallax layer is halfway done or so, we create a new parallax layer and set its position to where the last one ends, so that they interconnect flawlessly. The following screenshot shows the addition of `parallax.png` to `Images.xcassets`:



*Adding parallax.png to Images.xcassets*

Build and run the project, and you will see that the parallax background layer adds to the atmosphere and look of the project. Suddenly, our game has gained depth and looks great, as shown in the following screenshot:



*The parallax layer adds depth to our game*

## Tip

### What is parallax?

Parallax is an effect that allows us to simulate depth on a two-dimensional screen. This effect can be seen when you travel by train or by car. Trees that are near you move fast, those that are at a distance move slowly, and things on the horizon barely move. When some objects move faster and other objects move slower, we get a feeling

that there is depth to the scene. We use this effect to add polish to our game.

## Summary

In this chapter, we have learned that animation is a powerful way to add life to your game. It helps to engage the player, as smooth and nice-looking animations are always an eye candy. Learn to appreciate your animations and players will enjoy your games. We have also learned about animations; how to create them, and how to run and cancel them. We also found out what texture atlases are and how they increase performance and save memory. We also learned how to colorize sprites, how to perform parallax scrolling, and how to add depth to our game.

In the next chapter, we will learn about particle effects, and how to create and use them to add more eye candy to your game.

# Chapter 5. Particle Effects

In this chapter, we will be discussing particle effects. These are the effects that are created by utilizing particles, which are very small graphical entities. They are not sprites and are rendered very efficiently, and you can have thousands of particles on screen without any slowdown.

Particle effects are used for various effects that could be hard to implement otherwise.

This is why particle effects are often used to add polish and engagement to your game. There are different kinds of effects that can be implemented by particle effects, such as fire, explosions, rain, smoke, snow, fireworks, stars, and so on.

A particle is just a small sprite that appears on the screen for a very short amount of time and disappears. Its speed, velocity, and other properties affect the outcome that you will get on screen. There are hundreds and even thousands of those particles on screen at any time particle effect is used. This is possible due to the fact that graphics chips are optimized to show many objects with the same texture.

Particles are generated and destroyed by a special object called the particle emitter. It knows when to make a new particle, when to kill off old particles, and the properties of particles.

## Particle emitters

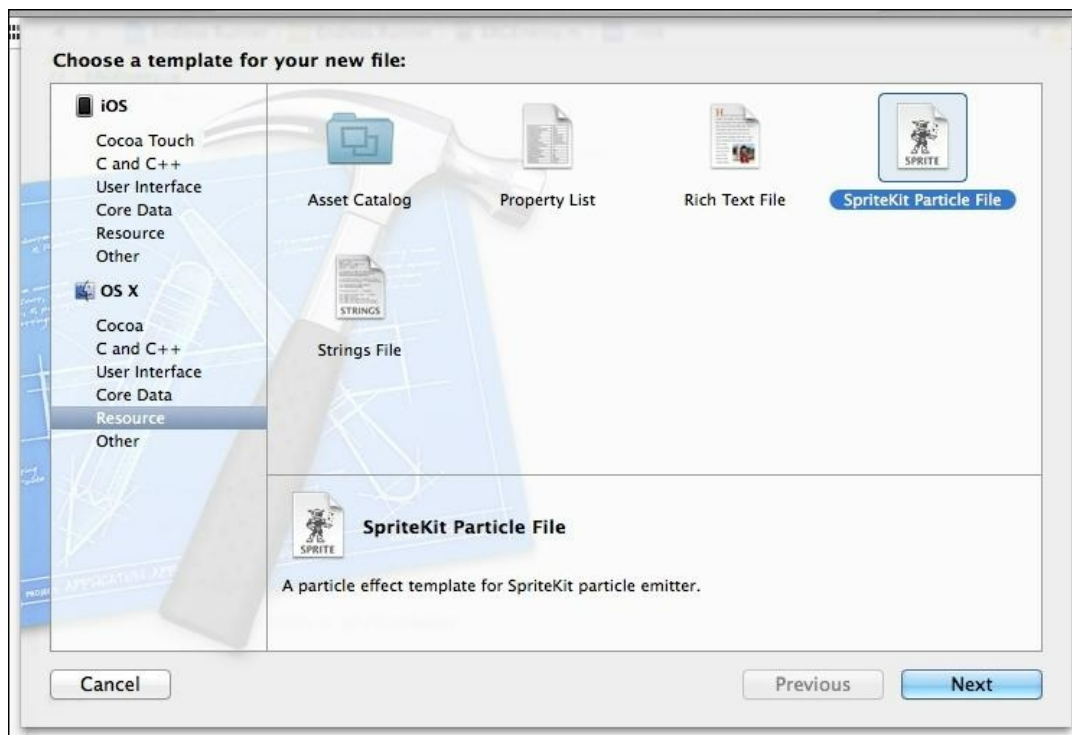
Particle effects are generated by the `SKEmitterNode` particle emitter that generates and manages particles. Generally, particles originate from their emitter. Each emitter has a plethora of properties. There are more than twenty properties that affect the emitter itself and the particles it generates.

Particle emitters are smart and they re-use particles like many other classes in iOS ecosystem (for example `UITableView` and `UICollectionView`). Rather than creating a new particle every time, it re-uses particles that have gone off-screen instead. This is one of the reasons why particle effects are so efficient.

There are two ways to create the particle emitter, either from file or programmatically. We will look at both the methods.

# First particle effect

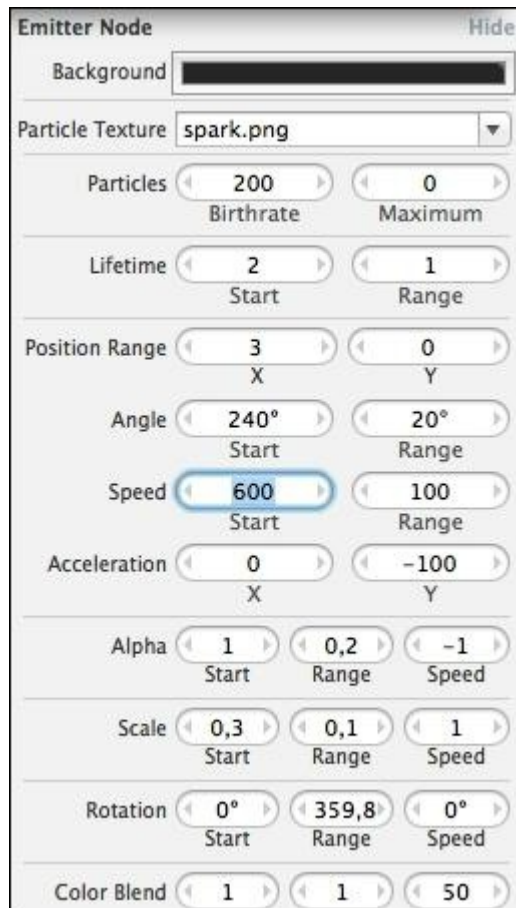
How can we apply particle effects to our project? The first thing that comes to mind is to add nice flames to our jetpack. In order to do that, we need to create a new particle effects file. To do this, we need to navigate to the **File** menu and then to **New | New File**. On the left-hand side table, click on **Resource** under the **iOS** section and from the right-hand side of the screen, select **SpriteKit Particle File**, as shown in the following screenshot:



*Creating a new particle effects file*

On the next screen, you will be asked what preset type of particle effect you want to create. Pick a **Spark** type. Name it as `jet`.

After you have done that, two new files will appear in your project, `jet.sks` and `spark.png`. Click on the first one and you will see the particle effect in action, then select **SKNode Inspector** on the right-hand side of the screen. Make sure the **Utilities** pane is visible on the right-hand side (the top-right icon in the **Xcode** window). Next, click on the rightmost icon below the toolbar, which looks like a small cube. You will see **Particle Editor**, which is a built-in tool to customize particle effects.



*Particle effects editor in Xcode*

Now you will see the **Emitter Node** properties. Many of them are cryptic, but try playing with them and you will see that they are not that scary. Let's discuss some of them:

- **Birthrate:** This is the rate at which the emitter generates particles. The greater the number here, the more intense the effect feels. Try to keep this as low as possible to achieve the same effect for a good frame rate.
- **Lifetime:** This defines how long a particle will live. This indirectly affects how far away the particles will fly from the emitter. Range here means that the emitter will use a random value from the first value +/- range.
- **Position Range:** This defines how far from the center of [SKEmitterNode](#) the particles will appear. In practice, this affects how big you want your emitter to be.
- **Angle:** This is the direction in which the particles will fly.
- **Speed:** This is the starting speed of the particle, with the same range variance as in the previous properties.
- **Acceleration:** This value on x and y axes means where the particles will fly as soon as they appear.
- **Alpha:** This makes the particles transparent and sets the value and range to your liking.

- **Scale:** This is the size of the texture used for the effect. Sometimes you will want smaller particles and sometimes you will want bigger ones, and this property allows you to have only one texture for both of them.
- **Rotation:** This is rarely needed, as each particle lives for so little time that usually rotation is meaningless.
- **Color Blend:** This is one of the most interesting properties. It allows you to set color change in particle life. Particles can start at one color and slowly arrive to another color.

Set particle effect properties to those shown in the preceding screenshot. This will create a nice jet of flame that looks useful to our game. The next thing that we need to do is add it to our player character.

In order to do that, add a new property to `ERGPlayer.h`:

```
@property (strong, nonatomic) SKEmitterNode
*engineEmitter;
```

After that, add the following lines to the `init` method of `ERGPlayer.m`:

```
self.engineEmitter = [NSKeyedUnarchiver
unarchiveObjectWithFile:
[[NSBundle
 mainBundle] pathForResource:@"jet"
 ofType:@"sks"]];
self.engineEmitter.position =
CGPointMake(-12, 18);
self.engineEmitter.name = @"jetEmitter";
[self addChild:self.engineEmitter];
self.engineEmitter.hidden = YES;
```

In the first line, we unarchive the emitter from file. Each Sprite Kit node is archivable and serializable, so here we utilize that fact. Next, we set the position of the emitter in our player sprite, give the emitter a name, and hide it since we don't want it to be on always.

If you remember, we have already handled all physics of acceleration and added the property that tells us whether the player is accelerating with the same name. Now that we need extra behavior from that property, add a new setter to it in `ERGPlayer.m`:

```
- (void) setAccelerating:(BOOL)accelerating
{
    if (accelerating) {
        if (self.engineEmitter.hidden) {
            self.engineEmitter.hidden = NO;
        }
    } else {
        self.engineEmitter.hidden = YES;
    }
}
```

```
    }  
    _accelerating = accelerating;  
}
```

Here we hide or show jet flames based on our state. If you launch the game now, you will see a nice effect that surely added some spice to our game. If Xcode raises a warning, make sure that the `accelerating` property in `ERGPlayer.h` is of non-atomic type.

# Advanced physics

Now that we have added the jetpack effect, we should add core gameplay elements. We will add enemies and power-ups to our game. But we will need to handle collisions somehow. That's where, once again, Sprite Kit and its sophisticated physics engine will save us.

If you remember how we handled collisions before by setting `collisionBitMasks`, the game checks if two things have the same collision bitmasks and handles their collision. This was the way that worked for player-ground collision for us.

But we would like to handle the collisions ourselves. For example, we want the player to lose if he touches enemy, or we want a shield to appear when he picks a shield power-up. Thankfully, there is a working method for that.

The first thing that we need to do is add our scene as conforming to the `SKPhysicsContactDelegate` protocol. To do that, open `ERGMyScene.h` and change the line with `@interface` in it to this:

```
@interface ERGMyScene : SKScene
<SKPhysicsContactDelegate>
```

This means that we implemented some of the methods needed to correctly handle collisions.

Each time our physics bodies collide, our scene will call `(void) didBeginContact:(SKPhysicsContact *)` the contact method. Here we handle collisions and do what we need to do with our bodies and nodes.

To make this method work we need to update our knowledge on bitmasks.

The following are the three types of bitmasks used in Sprite Kit physics:

- `categoryBitMask`: This mask shows us what category this body belongs to. It is used to see what type of collision body this is.
- `collisionBitMask`: This is used to handle collisions automatically. Bodies with overlapping masks will collide and the physics engine will handle everything for you.
- `contactTestBitMask`: This mask is needed for manual collision handling in the `didBeginContact` method. Each time bodies with overlapping contact masks collide, you will get this method invoked. When the collision ends, `didEndContact` will get triggered.

Bitmasks in Sprite Kit are 32 bit, the same as integers, and they work by setting individual bits in an integer. Thus, there are only 32 possible masks,

and you should use them carefully, since mistakes here lead to a lot of confusion.

As bitmasks use each individual bit, you should use these numbers for masks: 1, 2, 4, 8, 16, 32, and so on.

We will need to create new masks in the `Common.h` file and add the following code to it:

```
const static int playerBitmask = 1;
const static int enemyBitmask = 2;
const static int shieldPowerupBitmask = 4;
const static int groundBitmask = 8;
```

We will use these bitmasks to handle different objects on screen. After this, we need to create actual classes to handle power-ups and enemies.

Create a new `ERGENemy` class and make `SKNode` as its parent.

We will need only one property there, that is, `emitter` and add it to `ERGENemy.h` using `@property (strong, nonatomic) SKEmitterNode *emitter.`

Now in the `.m` file, we will need to set up the particle emitter for the job, as follows:

```
- (instancetype) init
{
    self = [super init];
    if (self) {
        [self setup];
    }
    return self;
}

- (void) setup
{
    self.emitter = [NSKeyedUnarchiver
unarchiveObjectWithFile:
    [[NSBundle mainBundle]
pathForResource:@"enemy" ofType:@"sks"]];
    self.emitter.name = @"enemyEmitter";
    self.emitter.zPosition = 50;
    [self addChild:self.emitter];
    self.physicsBody = [SKPhysicsBody
bodyWithCircleOfRadius:20];
    self.physicsBody.contactTestBitMask =
playerBitmask;
    self.physicsBody.categoryBitMask =
enemyBitmask;
    self.physicsBody.collisionBitMask = 0;
```

```

        self.physicsBody.affectedByGravity = NO;
    }

```

We override the `init` method to add the new `setup` method that creates a new emitter and a new physics body. We set `contactBitMask` to `playerBitmask`, since we want to be notified when a player sprite is colliding with the enemy sprite, and we don't want enemies to collide with each other. We set `categoryBitMask` in order to identify the object, `collisionBitMask` to zero since we don't want our engine to handle collision for us, and we will do it manually. We also set `affectedByGravity` to `NO` as we don't want our enemies to fall off the screen.

The next thing we need is power-ups. Create the `ERGPowerup` class (we will use this class to represent objects that grant shields to the player character) and set `ERGENemy` as its `superclass`. We need to redefine the `setup` method in `ERGPowerup.m` as we want them to do different things:

```

- (void) setup
{
    self.emitter = [NSKeyedUnarchiver
unarchiveObjectWithFile:
    [[NSBundle mainBundle]
pathForResource:@"powerup" ofType:@"sks"]];
    self.emitter.name = @"shieldEmitter";
    self.emitter.zPosition = 50;
    [self addChild:self.emitter];
    self.physicsBody = [SKPhysicsBody
bodyWithCircleOfRadius:20];
    self.physicsBody.contactTestBitMask =
playerBitmask;
    self.physicsBody.categoryBitMask =
shieldPowerupBitmask;
    self.physicsBody.collisionBitMask = 0;
    self.physicsBody.affectedByGravity = NO;
}

```

This is same as the previous thing, but we pick a different particle effect and another `categoryBitmask`.

After this, we need to add enemies to our scene. First, import the `ERGENemy.h` and `ERGPowerup.h` files at the top of `ERGMyScene.m`. Second, we will need methods to create enemies:

```

- (ERGENemy *) spawnEnemy
{
    ERGENemy *temp = [[ERGENemy alloc] init];
    temp.name = @"enemy";
    temp.position = CGPointMake(self.size.width +
arc4random() % 800, arc4random() % 240 + 40);
    return temp;
}

```

```
}
```

This method is straightforward, but for the `arc4random()` function that returns a random number, and if we perform a modulo operation on it with certain numbers, we will get a random number from 0 to the previous number of that number. This is useful, since we want our enemies to spawn somewhat randomly. They spawn off screen. We will need the same method to spawn power-ups:

```
- (ERGPowerup *) spawnPowerup
{
    ERGPowerup *temp = [[ERGPowerup alloc] init];
    temp.name = @"shieldPowerup";
    temp.position = CGPointMake(self.size.width +
    arc4random() % 100, arc4random() % 240 + 40);
    return temp;
}
```

In order to move them, add the following code to the update method:

```
[self enumerateChildNodesWithName:@"enemy"
usingBlock:^(SKNode *node, BOOL *stop) {
    ERGEnemy *enemy = (ERGEnemy *)node;
    enemy.position =
    CGPointMake(enemy.position.x -
    backgroundMoveSpeed * timeSinceLast,
    enemy.position.y);

    if (enemy.position.x < -200) {
        enemy.position =
        CGPointMake(self.size.width + arc4random() % 800,
        arc4random() % 240 + 40);
        enemy.hidden = NO;
    }
}];

[self
enumerateChildNodesWithName:@"shieldPowerup"
usingBlock:^(SKNode *node, BOOL *stop) {
    ERGPowerup *shield = (ERGPowerup *)node;
    shield.position =
    CGPointMake(shield.position.x -
    backgroundMoveSpeed * timeSinceLast,
    shield.position.y);

    if (shield.position.x < -200) {
        shield.position =
        CGPointMake(self.size.width + arc4random() % 100,
        arc4random() % 240 + 40);
        shield.hidden = NO;
    }
}];
```

The preceding code will move enemies at the same speed as the background is moved, and will move the enemy to a somewhat random position in front of the screen once it goes off screen. This way we will always use only a certain amount of enemies and won't create any new ones. Re-using things is a good practice, since it saves us memory and processor time. Next, we need to add some constants that handle the number of enemies that spawn and power-ups that can spawn.

We will add these to `Common.h`:

```
const static NSInteger maximumEnemies = 3;
const static NSInteger maximumPowerups = 1;
```

Now back in `ERGMyScene.m`, we will create new enemies and power-ups in the `initWithSize:` method:

```
for (int i = 0; i < maximumEnemies; i++) {
    [self addChild:[self spawnEnemy]];
}
for (int i = 0; i < maximumPowerups; i++) {
    [self addChild:[self spawnPowerup]];
}
```

Now it's time to change the appearance of enemies. Create a new particle effects file for them and name it as `enemy.sks`. You can configure them the way you like, but I used the following properties for enemies:

- Particles: 500
- Lifetime: 0.05
- Position and range: zero
- Angle: 0 with 360 range
- Speed: 400
- Acceleration: Zero on both the axes
- Scale: 0.1 with 0.1 range and 1 speed
- Rotation: At zero and yellow color

And the following properties for power-ups:

- Particles: 400
- Lifetime: 1
- Position range: x at 20 and y at 0
- Angle: 90 with 0 range
- Speed: 100 with 100 range
- Acceleration: x at 0 and y at -50
- Scale: 0.1 with 0.1 range and 1 speed
- Rotation: At zero and blue color

This makes a nice yellow ball that looks dangerous enough for enemies and the blue flame for power-ups. In order to use collisions and collision handling, we need to change the masks in `ERGPlayer.m` and the old values to the following ones:

```
        self.physicsBody.contactTestBitMask =
shieldPowerupBitmask | enemyBitmask;
        self.physicsBody.collisionBitMask =
groundBitmask;
        self.physicsBody.categoryBitMask =
playerBitmask;
```

Again, the preceding code should be straightforward. The first line sets masks that we want to register contact events with. The second line sets automatic collisions with the ground, we want these to work as they did before. And the final line is the category to identify us.

Now we need to modify `ERGMyScene.m` and add methods to handle contacts:

```
- (void) didBeginContact:(SKPhysicsContact
*)contact
{
    ERGPlayer *player = nil;

    if (contact.bodyA.categoryBitMask ==
playerBitmask) {
        player = (ERGPlayer *) contact.bodyA.node;
        if (contact.bodyB.categoryBitMask ==
shieldPowerupBitmask) {
            player.shielded = YES;
            contact.bodyB.node.hidden = YES;
        }
        if (contact.bodyB.categoryBitMask ==
enemyBitmask) {
            [player takeDamage];
            contact.bodyB.node.hidden = YES;
        }
    } else {
        player = (ERGPlayer *) contact.bodyB.node;
        if (contact.bodyA.categoryBitMask ==
shieldPowerupBitmask) {
            player.shielded = YES;
            contact.bodyA.node.hidden = YES;
        }
        if (contact.bodyA.categoryBitMask ==
enemyBitmask) {
            [player takeDamage];
            contact.bodyA.node.hidden = YES;
        }
    }
}
```

```
}
```

The preceding method is provided with the contact that has two properties in it, `bodyA` and `bodyB`. But these bodies are somewhat randomly selected and we need to check their type before we do anything. Here we check if we picked up the power-up on which we set the `shielded` property on our player to `YES`, and if we touch the enemy, we will call the `takeDamage` method on the player.

We will need to modify `ERGPlayer.m` by first removing `self.shielded = YES` and `self.shielded = NO` from where we put them before. They will be handled by collisions from now on. After this, add a new method to handle taking damages:

```
- (void) takeDamage
{
    if (self.shielded) {
        self.shielded = NO;
    } else {
        self.hidden = YES;
        [[NSNotificationCenter defaultCenter]
 postNotificationName:@"playerDied" object:nil];
    }
}
```

Since we want the player node to be as decoupled from our scene as possible, we won't call some method on the scene directly, but we will broadcast notifications to everyone who might be interested in it. In our game, we will handle it by presenting a new Game Over scene to the player.

# Scene transitions

Only one scene may be present on-screen at any time, and in order to handle different scenes that may hold different content we may need smooth transitioning between them. The following is a code sample that will clear things up for you, add it to `ERGMyScene.m`:

```
- (void) gameOver
{
    ERGGameOverScene *newScene =
    [[ERGGameOverScene alloc] initWithSize:self.size];
    SKTransition *transition = [SKTransition
    flipHorizontalWithDuration:0.5];
    [self.view presentScene:newScene
    transition:transition];
}
```



*Our Game Over scene*

First, we create a new scene as usual. Next, we create a `transition` object. There are many kinds of transitions with many different effects. On the third line, we ask `view` to present a new scene. This ends our scene lifecycle. Scenes do not get saved in memory; on each transition the old scene is destroyed, and if you want to use it again, you have to make it all over, as shown in the preceding screenshot.

## Tip

There are many different transition types that you can use, check the `SKTransition` class reference for a list of possible transitions. It is available at

[https://developer.apple.com/library/IOs/documentation/SpriteKit/Reference/SKTransition\\_Re](https://developer.apple.com/library/IOs/documentation/SpriteKit/Reference/SKTransition_Re)  
or in Xcode search.

In order to call this method, we need to subscribe to notifications and add the following code to the `initWithSize:` method of `ERGMyScene.m`:

```
[[NSNotificationCenter defaultCenter]
addObserver:self selector:@selector(gameOver)
name:@"playerDied" object:nil];
```

Now we need to make a new scene, a Game Over scene, to show to the player each time he fails.

Create a new class, name it as `ERGGameOverScene`, and make it a subclass of `SKScene`. Add the following code:

```
- (instancetype) initWithSize:(CGSize)size
{
    self = [super initWithSize:size];
    if (self) {

        SKLabelNode *node = [[SKLabelNode alloc]
initWithFontNamed:@"Chalkduster"];
        node.text = @"Game over";
        node.position =
CGPointMake(self.size.width / 2,
self.size.height / 2);
        node.fontSize = 35;
        node.color = [UIColor whiteColor];
        [self addChild:node];
    }
    return self;
}

- (void) didMoveToView:(SKView *)view
{
    [super didMoveToView:view];
    UITapGestureRecognizer *tapper =
[[UITapGestureRecognizer alloc]
initWithTarget:self action:@selector(newGame)];
    [view addGestureRecognizer:tapper];
}

- (void) newGame
{
    ERGMyScene *newScene = [[ERGMyScene alloc]
initWithSize:self.size];
    SKTransition *transition = [SKTransition
flipHorizontalWithDuration:0.5];
    [self.view presentScene:newScene
transition:transition];
}
```

```
}
```

In the preceding code, we add a new text label to the new scene as usual and set its properties. Next, we add a tap gesture recognizer so that the player can try again by tapping the screen. Finally, in the `newGame` method, we create new `MyScene` and `Transition` to it, effectively starting a new game.

## Summary

In this chapter, we have learned about particle effects and their flexibility in use. We have created enemies and power-ups only with particle effects. We have found out how we can handle contact events and how to use complex physics and collisions in our game. We handled scene transitions and found out how to decouple a node from the scene if needed. Our game is almost fleshed out, we have enemies, power-ups, and lost conditions.

In the next chapter, we will learn how to add game controllers support to our game. We will learn how to discover and use various game controllers that are new in iOS 7.

# Chapter 6. Adding Game Controllers

In this chapter, we are going to discuss a new feature available in iOS 7—game controllers. Apple allowed video streaming from iOS devices to TV a long time ago, but the lack of hardware controller support left many players underwhelmed. Game controllers allow you to focus on the gameplay rather than controls, and if you are using your iPhone or iPad as a controller, you don't receive any physical feedback. This is one of the major drawbacks of touchscreen gaming—you won't be sure whether you tapped the right thing unless you are looking at the screen.

Some companies started to work on this problem, and different controllers such as iCade appeared. But they had one fundamental flaw in them—they had to be supported by the developers in order to work with the game. Another issue was connectivity. Some of them wanted to work over Bluetooth while others were connected directly, and each of these methods had their downsides.

There were very few games that supported hardware controllers. Developers didn't feel like supporting these controllers as the install base was far too small, which would affect the sales, thus increasing the time spent to support them.

Another problem with such a controller is the lack of uniformity. Everybody makes their own interface, and developers need to learn many different frameworks and APIs in order to make these things work.

Another issue is the availability and spread of such controllers—the company might go out of business, it may decide to not make more controllers, or their supply might run out.

That's why supporting third-party controllers didn't work too well.

## Native game controllers

Support for native game controllers first appeared in iOS 7. What are the advantages of native controllers? Here are some of them:

- **A universal API for every controller out there:** You will need to write the code only once to support a myriad of different controllers by different manufacturers.
- **An easy-to-use API written and supported by Apple engineers:** You know that your code will work on the latest versions of iOS and

will generally be supported throughout your application's lifetime.  
Bugs will be removed and everyone will be happy.

- **A standard layout for buttons and joysticks:** You know what buttons each controller will have and where they are expected to be.

Now that we have found out why game controllers are useful, let's incorporate them into our project.

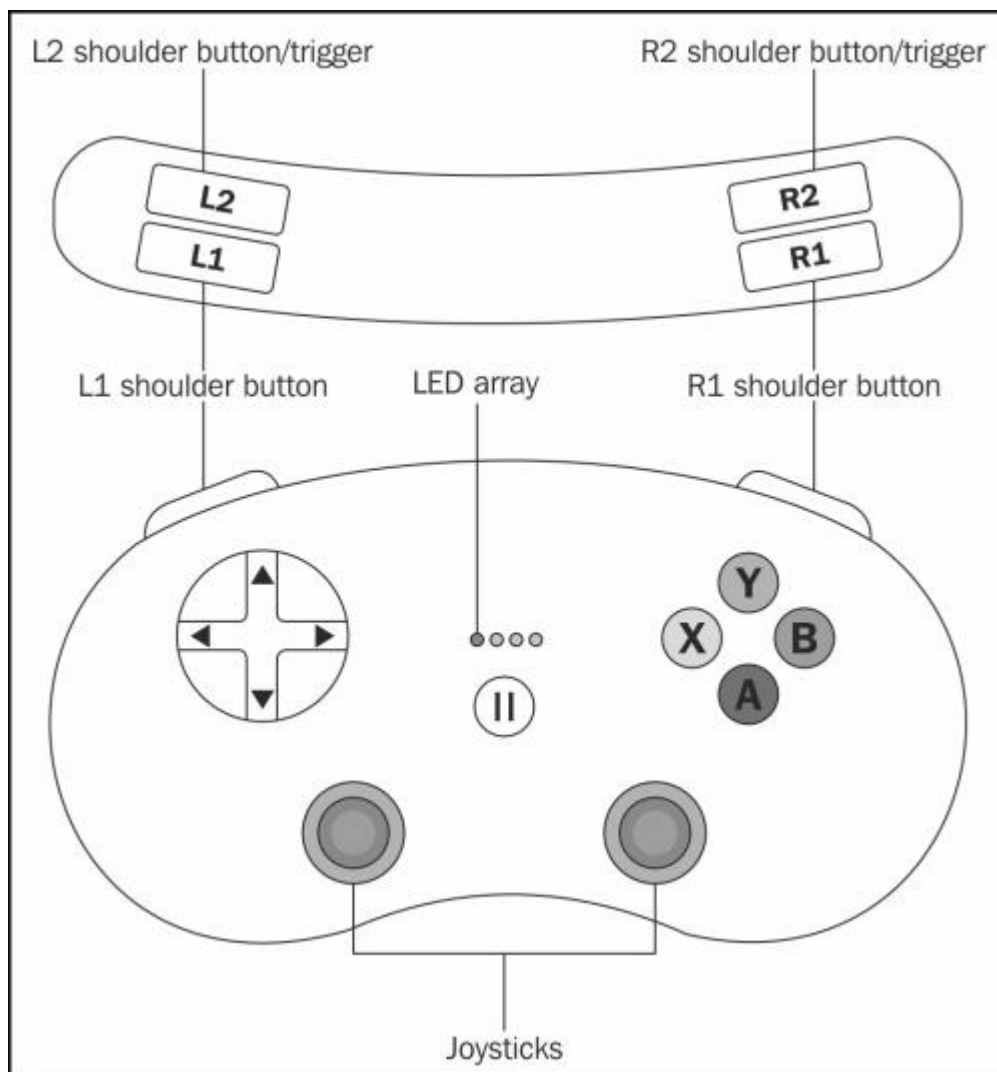
# Game controller basics

The Game Controller API is based on the Game Controller framework. Everything related to controllers happens there.

There are three types of controllers that are supported by the framework:

- **Standard form-fitting controller:** This is a controller in which the iOS device resides. This controller has a direction pad, a pause button, four buttons, and two shoulder buttons.
- **Extended form-fitting controller:** This is the same as the previous one, but it can also have up to two sticks and two more shoulder buttons.
- **Extended wireless controller:** This is the same as the previous one, but it works wirelessly and is an external device without holding your iOS device in it.

The following figure shows the various types of controllers:



## Tip

Apple does not necessarily require a game controller to play a game—there should always be a way to play your game via touchscreen. Game controllers enhance your user's experience, but do not force them to do so.

Wired controllers are automatically discovered, but their wireless counterparts require a one-time setup and can be used freely after that. A controller and a device will automatically connect to each other once they are powered on. There is no difference from the code's perspective for wired or wireless controllers.

Each game controller is represented with the `GCController` object, which you can use in your game. You receive notifications when a new one is being connected or disconnected.

Each game controller has a profile that identifies it as a controller type. Currently, there are two profiles—`gamepad` and `extendedGamepad`.

You can check `GCController` for the properties of `gamepad` and `extendedGamepad`. If they exist, it means that the controller is one of those types. Otherwise, it will return `nil`.

There are two ways to receive data from the controller, which are as follows:

- By reading the properties directly
- By assigning a handler (block) to each button to be executed on button press

The following is a sample method to read inputs if you want to read them at arbitrary times:

```
- (void) readControlInputs
{
    GCGamepad *gamepad =
self.myController.gamepad;
    if (gamepad.buttonA.isPressed)
        [self jumpCharacter];
    if (gamepad.buttonB.isPressed)
        [self shootApple];
    [self moveBy:
profile.leftThumbstick.xAxis.value];
}
```

```
}
```

Here we are checking if the buttons are pressed, and if they are, we execute some methods such as jumping or shooting. On the last line, we read the value of the x axis on the left thumbstick to move our character left or right.

The second way to set handlers is as follows:

```
- (void) setupController
{
    GCExtendedGamepad *gamepad =
self.myController.extendedGamepad;
    gamepad.buttonA.valueChangedHandler =
^ (GCControllerButtonInput *button, float value,
BOOL pressed)
    {
        if (pressed)
            [self shoot];
    };
    gamepad.buttonX.valueChangedHandler =
^ (GCControllerButtonInput *button, float value,
BOOL pressed)
    {
        if (pressed)
            [self openInventory];
    };
}
```

What we do here is set the block to each button (in this case, two buttons) to execute a piece of code each time a button is pressed. One button shoots and the second button opens the inventory. These handlers also get called when the player releases the button, so you should check for the `pressed` variable.

Picking the positions of the thumbstick or the directional pad is a little more complex. Usually, you get x and y values and you can work from there:

```
GCControllerDirectionPadValueChangedHandler
dpadMoveHandler = ^ (GCControllerDirectionPad
*dpad, float xValue, float yValue) {
    if (yValue > 0)
    {
        player.accelerating = YES;
    } else {
        player.accelerating = NO;
    }
};
```

In this block, we check the y value, and if it is higher than 0 (neutral), we start jumping.

Each controller also has a pause button, and you should implement the functionality of pausing your game when the user presses the pause button. The controller holds a block for the pause button that gets triggered each time it is pressed.

## Using a controller in our game

As our game is not too complicated, we use only one button. We will map the same thing to a few buttons so that the players can use any one that they like.

The first thing that we need to do is import the `GameController` framework. In order to do this, add the following line to `ERGMyScene.h` at the beginning of the file:

```
@import GameController;
```

The preceding line of code adds game controller headers and links our project with the game controller library while it is compiling. After this, we will need two new properties in the same file:

```
@property (strong, nonatomic) ERGPlayer *player;  
@property (strong, nonatomic) SKLabelNode  
*pauseLabel;
```

We will need a way to access our `player` object from the code that sets up the game controllers, so we add the first property in the preceding code to hold it.

The second one is the label for the paused state of our game. We will show or hide it based on the current game state.

Right after the line where you add `player` as a child in the `initWithSize:` method in `ERGMyScene.m`, add the following line of code:

```
self.player = player;
```

Here, we set our property to hold a pointer to the `player` object. At the end of `initWithSize:`, we will add the code to handle controller discovery and setup, and create a `pauseLabel`, as shown in the following code:

```
[self configureGameControllers];  
  
self.pauseLabel = [[SKLabelNode alloc]  
initWithFontNamed:@"Chalkduster"];  
self.pauseLabel.fontSize = 55;  
self.pauseLabel.color = [UIColor  
whiteColor];  
self.pauseLabel.position =  
CGPointMake(self.size.width / 2,  
self.size.height / 2);  
self.pauseLabel.zPosition = 110;
```

```
[self addChild:self.pauseLabel];
self.pauseLabel.text = @"Pause";
self.pauseLabel.hidden = YES;
```

Currently, we have not yet written the `configureGameControllers` method, but we will do this shortly. The code to create a label should be fairly familiar to you—we create a label node, set its properties, and set its position to the center of the screen and hide it, as we don't want it to be visible in the unpaused state of the game.

## Tip

Each scene has a `paused` property. When it is set to `YES`, the scene stops evaluating actions for all descendants. But the `update` method is still getting called! So, you may experience a very strange behavior. If we pause our game now, all animations will stop, but the background will still scroll.

The next thing that we need to do is stop updates if the scene is paused. To do that, change the `update` method in `ERGMyScene.m`—add a check for the `paused` property at the start of the `update` method and add the following code:

```
if (self.paused) {
    return;
}
```

This way, the `update` method will do something only if the scene is not paused, which is the intended behavior. If the scene is paused, it will skip all that code and drop out.

After this, we need the code to find any controllers and use them in our game. To do this, add the following code to `ERGMyScene.m`:

```
- (void)configureGameControllers {

    [[NSNotificationCenter defaultCenter]
 addObserver:self
 selector:@selector(gameControllerDidConnect:)
 name:GCControllerDidConnectNotification
 object:nil];
    [[NSNotificationCenter defaultCenter]
 addObserver:self
 selector:@selector(gameControllerDidDisconnect:)
 name:GCControllerDidDisconnectNotification
 object:nil];

    [self configureConnectedGameControllers];
}
```

```

        [GCCController
startWirelessControllerDiscoveryWithCompletionHandler:^(

        // we don't use any code here since when
new controllers are found we will get
notifications
        )];
    }
}

```

The first thing that we do is subscribe for notifications related to the game controller. They are sent when a new controller is connected or disconnected from the device. Next, we call another method to configure the controller.

Also, notice that if you turn off Bluetooth on your device, the game will crash on the `configureGameControllers` method. The same thing will happen if you test on a simulator. Adding error-checking unnecessarily complicates the code in our example, so comment out `[self configureGameControllers];` in the `initWithSize:` method if you don't intend to use it on a Bluetooth-enabled device.

We also start searching for wireless controllers. We don't need any code inside the completion handler as we will be getting notifications when controllers connect or disconnect from the device. Let's get to the next method:

```

- (void)configureConnectedGameControllers {

    for (GCCController *controller in
[GCCController controllers]) {
        [self setupController:controller
forPlayer:self.player];
    }
}

```

This is a simple `for` loop, where we will go through all the available controllers and run a common setup code for each of them. If you are making a more complex or multiplayer game, you might want to differentiate the controllers here. There is a handy `playerIndex` property on the controller that may prove useful for you, but we don't use it here.

Most of the logic happens in the following method, where we set up actual button handlers:

```

- (void)setupController:(GCCController
*)controller forPlayer:(ERGPlayer *)player
{

```

```

        GCControllerDirectionPadValueChangedHandler
dpadMoveHandler = ^(GCControllerDirectionPad
*dpad, float xValue, float yValue) {
    if (yValue > 0)
    {
        player.accelerating = YES;
    } else {
        player.accelerating = NO;
    }
};
if (controller.extendedGamepad) {

    controller.extendedGamepad.leftThumbstick.valueChangedHandler
= dpadMoveHandler;
}
if (controller.gamepad.dpad) {

    controller.gamepad.dpad.valueChangedHandler =
dpadMoveHandler;
}

        GCControllerButtonValueChangedHandler
jumpButtonHandler = ^(GCControllerButtonInput
*button, float value, BOOL pressed) {
    if (pressed) {
        player.accelerating = YES;
    } else {
        player.accelerating = NO;
    }
};

if (controller.gamepad) {

    controller.gamepad.buttonA.valueChangedHandler =
jumpButtonHandler;

    controller.gamepad.buttonB.valueChangedHandler =
jumpButtonHandler;
}
if (controller.extendedGamepad) {

    controller.extendedGamepad.buttonA.valueChangedHandler
= jumpButtonHandler;

    controller.extendedGamepad.buttonB.valueChangedHandler
= jumpButtonHandler;
}

    controller.controllerPausedHandler =
^(GCController *controller) {
    [self togglePause];
};
}

```

At the beginning of this method, we create a handler for the directional pad and thumbsticks—if the player presses the up button on the pad or moves the thumbstick up, we will have the Y value going up from zero. Thus, we start jumping by setting the accelerating property to `YES`.

After that, we check if the controller has an `extendedGamepad` profile, and if it does, we attach this handler to the thumbstick. If it is a regular gamepad, we attach the handler to the directional pad.

The same procedure is repeated when we declare a handler for a button—it checks if the button is pressed and our character jumps; if it is, we set the A and B button handlers to this handler.

The last handler that can be found in this method is the pause handler. It will get called when a player presses the pause button. The following is the code for this method:

```
- (void) togglePause
{
    if (self.paused) {
        self.pauseLabel.hidden = YES;
        self.paused = NO;
    } else {
        self.pauseLabel.hidden = NO;
        self.paused = YES;
    }
}
```

Here we check if the scene is paused, and if it wasn't paused, we pause it and show the **Pause** label. If it was paused, we continue the scene and hide the label.

# Handling controller notifications

Another thing that we would want to handle is new controllers. We need to do something when a controller connects or disconnects.

We can create a complex interface for that, but for our project, simple alerts will do.

The easy part is that when the controller disconnects, we won't get any new input, so we don't need to do anything; we just show an alert and are done with it:

```
-  
(void)gameControllerDidDisconnect:  
(NSNotification *)notification {  
  
    UIAlertView *alert = [[UIAlertView alloc]  
initWithTitle:@"Warning"  
  
    message:@"Game controller has disconnected."  
  
    delegate:nil  
  
    cancelButtonTitle:@"Ok"  
  
    otherButtonTitles:nil, nil];  
    [alert show];  
}
```

We create the `alertView` object and show it. The user cancels it and goes on with their game with touch controls.

But if we connect the controller, we need to ask the user if they actually want to use it in the game. In order to do that, we need to set our scene as a delegate to `UIAlertView` and change the `@interface` line in `ERGMyScene.h` to the following:

```
@interface ERGMyScene : SKScene  
<SKPhysicsContactDelegate, UIAlertViewDelegate>
```

This means that we conform to `UIAlertViewDelegate` and that we will implement certain methods to be executed after the user taps a button in the alert view.

When a new controller connects, we will get a notification and the following method will get called, which should be added to the `ERGMyScene.m` file:

```

- (void)gameControllerDidConnect:(NSNotification
*)notification {

    UIAlertView *alert = [[UIAlertView alloc]
initWithTitle:@"Warning"

    message:@"Game controller connected. Do you want
to use it?"

    delegate:self

    cancelButtonTitle:@"No"

    otherButtonTitles:@"Yes", nil];
    [alert show];
}

```

We create the alert view as before, but here we set its delegate to our scene. The following is the `delegate` method:

```

-(void>alertView:
(UIAlertView *)alertView
clickedButtonAtIndex:(NSInteger)buttonIndex
{
    if (buttonIndex == 1) {
        [self configureConnectedGameControllers];
    }
}

```

When you press a button in the alert view, and if it has the delegate set, it will call this method on its delegate. The button's index starts at zero, and zero is the cancel button that is always present. If there are any other buttons, they will have the subsequent indexes.

Usually, if we have multiple alert views, we have to understand which alert view called what, and what the button at said index should do.

But in our situation, we have only one alert view that has more than one button, and we know what it is, so we don't actually need to check anything here. If the user tapped the **Yes** button on the alert view, we will configure all the connected controllers. If they click on **No**, which is at index 0, nothing will happen.

# Adding sound and music

Our game feels a bit bland without sound effects, so we are going to add some. We will add background music and sound effects.

Sprite Kit is mostly a graphics library, and sound support is limited, but there are ways to add music to our game.

The first thing that we need to do is add music files to the project. To do this, drag-and-drop `Kick_Shock.mp3`, `shieldCharged.wav`, and `shieldSmashed.wav` to our project in Xcode. Select **Copy items into destination group's folder** and make sure that **Add to targets** has your project selected.

In order to play background music, we will need to add the `AVFoundation` library to our project.

The `AVFoundation` library is a collection of classes that allows for audio and video playback on iOS devices. To add `AVFoundation` to our project, add the `@import AVFoundation;` line to `ERGMyScene.h`. We will also need a new property to play music. To do this, add the following property to `ERGMyScene.h`:

```
@property (strong, nonatomic) AVAudioPlayer*
musicPlayer;
```

After this, we need to create a new method to set up the playback sound. Add this new method to `ERGMyScene.m`:

```
- (void) setupMusic
{
    NSString *musicPath = [[NSBundle mainBundle]
pathForResource:@"Kick_Shock" ofType:@"mp3"];
    self.musicPlayer = [[AVAudioPlayer alloc]
initWithContentsOfURL:
[NSURL URLWithString:musicPath] error:NULL];
    self.musicPlayer.numberOfLoops = -1;
    self.musicPlayer.volume = 1.0;
    [self.musicPlayer play];
}
```

In the first line, we created a path to our background music file. Next, we initialized `AVAudioPlayer`—a class that plays music from `AVFoundation`. Then, we set `numberOfLoops`, the property that says how many times the music should be repeated.

Next up is volume; it is set to 1.0 and sends play messages to the player.

After you have done this, call `[self setupMusic]` at the end of the `initWithSize:` method of `ERGMyScene.m` but before the `return` statement.

In a few lines, we managed to add background music to our game. But we will want to stop music playback if the **Game Over** screen is called. To do this, locate the `-(void) gameOver` method in `ERGMyScene.m`, and add the `[self.musicPlayer stop];` line at the beginning of it. This will stop music playback.

We will also add short playback sounds to our game. We will have them for shield charging and depleting. We won't use `AVFoundation` for this, as there is a simple sound playback action in Sprite Kit. It is used in this way (don't add this code to the project):

```
SKAction *soundAction = [SKAction
playSoundFileNamed:@"name.mp3"
waitForCompletion:YES];
[self runAction:soundAction];
```

It works as any other action. You create an action and run it against the node. Wait for completion means that the action is considered done as soon as it runs if `waitForCompletion` is `NO`, and the action is considered running if you pass `YES` to it.

Why haven't we used this action to play background music? It is used for playing short sounds, as somehow removing this action does nothing. It is unclear if it is a bug or a feature, but it is present in the previous version of Sprite Kit. The music just keeps on playing.

As we handle shield effects in `Player.m`, it is a good place to add playback sound. We will change the `setShielded:` method by adding music actions to it. Replace your method with the following:

```
- (void) setShielded:(BOOL) shielded
{
    if (shielded) {
        if (![self.shield
actionForKey:@"shieldOn"]) {
            [self.shield runAction:[SKAction
repeatActionForever:
[SKAction animateWithTextures:self.shieldOnFrames
timePerFrame:0.1 resize:YES restore:NO]]
withKey:@"shieldOn"];
            SKAction *musicAction = [SKAction
playSoundFileNamed:@"shieldCharged.wav"
waitForCompletion:NO];
            [self runAction:musicAction];
```

```

    }
    } else if (!_shielded) {
        [self blinkRed];
        [self.shield
removeActionForKey:@"shieldOn"];
        [self.shield runAction:[SKAction
animateWithTextures:self.shieldOffFrames
timePerFrame:0.15 resize:YES restore:NO]
withKey:@"shieldOff"];
        SKAction *musicAction = [SKAction
playSoundFileNamed:@"shieldSmashed.wav"
waitForCompletion:NO];
        [self runAction:musicAction];
    }
    _shielded = shielded;
}

```

As before, it handled the shield logic, but there is a little bit of playback sound added to it.

Build and run the project; it should run and you should have background music and sound effects fully working.

There is one more issue with our code. If you receive a call or exit from the application, it will crash. This happens due to the fact that we need to close `AVAudioSession` before our application resigns active. This class is used underneath `SKAction` to play sounds.

To fix this, add the `@import AVFoundation;` statement and add the following code to your `AppDelegate.m` file:

```

-
(void)applicationWillResignActive:
(UIApplication *)application
{
    // prevent audio crash
    [[AVAudioSession sharedInstance] setActive:NO
error:nil];
}

-
(void)applicationWillEnterForeground:
(UIApplication *)application
{
    // resume audio
    [[AVAudioSession sharedInstance]
setActive:YES error:nil];
}

```

This code handles `AVAudioSession` and prevents your application from terminating.

## Summary

In this chapter, we have learned what advantages hardware game controllers have and how to work with them. Adding game controller support to your game can certainly provide a better user experience for your players. Some people prefer playing on actual controllers than on touch-screen devices, since the screen is always flat and doesn't provide feedback to touches. A player can use the controller without looking at it, which completely changes the experience of playing if he/she is playing on a TV. One might argue that with game controllers, Apple is making a move into the console gaming territory, and a huge library of games on the iTunes App Store playable on your TV or iPad with a separate hardware controller is opening another new way to experience iOS games. Adding game controller support to your game is not difficult, and if you are not already supporting them, you should really consider doing that. We have also learned how to add background music and sound effects to your game. In the next chapter, we will discuss how you can add your game to the App Store, what provisioning profiles are, and how to prepare your application bundle.

In the next chapter, we will discuss how to prepare your application for the App Store. We will also learn about provisioning profiles and procedures to have your application approved by Apple.

# Chapter 7. Publishing to the iTunes App Store

In this chapter, we will discuss how to prepare your application for the App Store, what steps need to be taken, and the logistics of the process. You will learn how to register yourself as an Apple developer and post your application on the App Store.

## Registering as a developer

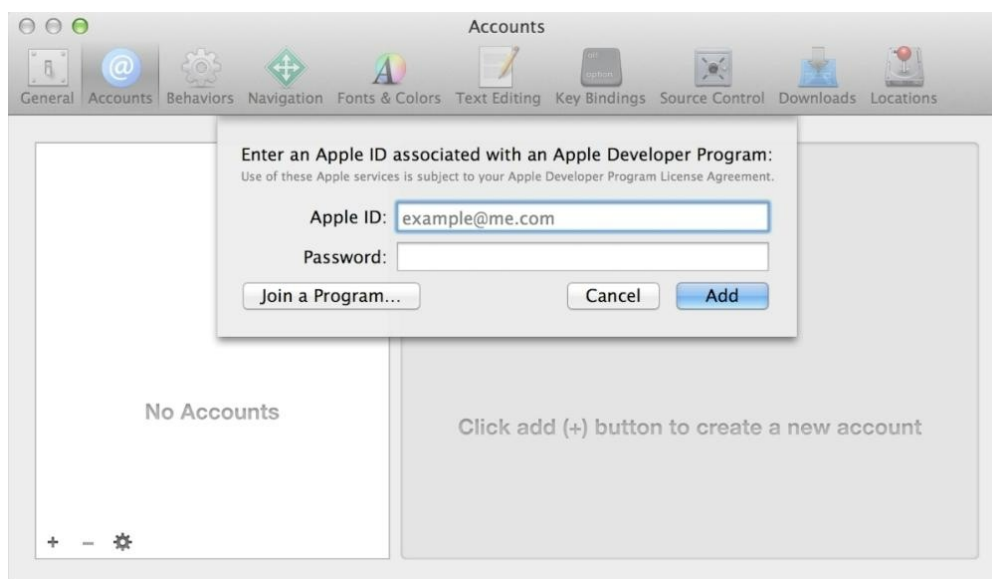
In order to be able to get your applications published on the App Store, you need to be registered as a developer with Apple. There is another added benefit of being a registered developer—you will be able to test your software on the device. This is the only way to do this. And as the simulator's performance is unreliable, you will eventually have to do that, and I suggest that you get it as soon as possible, especially if you are working with games – rendering on iOS simulator is very slow and laggy. You can get very severe frame rate drops in most trivial situations, where even old devices will show a solid 60 frames per second.

In order to register as a developer, we need to perform the following steps:

1. Create an Apple ID or log in to the iOS developer portal available at <https://developer.apple.com/register/index.action>.
2. As the next step, you will have to accept the legal agreement with Apple about your rights and duties as a developer.
3. After accepting the agreement, Apple will enquire you about your role as a developer or designer and what platforms you are planning to develop for.
4. You will gain access to a limited number of resources as a developer, such as videos and documentation, different guides, and other things.
5. The next thing that you should do is join the iOS Developer program at <https://developer.apple.com/programs/ios/>. There are huge benefits to this. You will get access to various developer resources and WWDC videos (materials from yearly conferences for Apple developers, which used to cost few hundreds by itself). On the downside, you will have to pay 99 dollars per year.
6. You will need to pick a program: an individual or company program. The company program offers adding more developers to the same account and enhanced distribution profiles (if you want to distribute your applications in-house). Its license requires much more paperwork, and it takes much longer to register as a company. If you are a sole developer, there are no benefits in registering as a company, as you will get equal possibilities for App Store distribution.

After performing all these steps, you will become a registered iOS developer. If you have chosen to register as a company, it will take you a few weeks to send the paperwork, but if you are a sole developer, the process will be instant. Now you will need to set up your programming environment as follows:

1. The first thing that you need to do is add your developer account to Xcode (remember that you will need Xcode 5 or a higher version for this) by navigating to **Xcode | Preferences**.
2. Click on **Accounts** from the top menu.
3. Click on the **+** button in the bottom-left corner and add **Apple ID**, as shown in the following screenshot:



*Adding your Apple ID to Xcode*

After this, you will be able to use all the features available to registered developers.

# Bundle ID

Bundle ID is a unique identifier that allows Apple to uniquely identify your application among all others. It is used for the following purposes:

- To enter in the Xcode project
- To enter in iTunes Connect (a platform to configure distribution of your applications)
- App ID
- iCloud container ID
- Other services like Game Center or in-app purchases

In order to change a bundle ID of an already existing application, select it in the project navigator, click on **General** present in the top menu, and if needed, open the disclosure triangle near **Identity**. There you will have the bundle ID as shown in the following screenshot. The usual format for the bundle ID is `com.<company>.<project>`, and this helps to keep your ID unique.



*Changing the bundle ID*

Occasionally, you will need to change your bundle identifiers as they are used in many places.

# Provisioning profiles

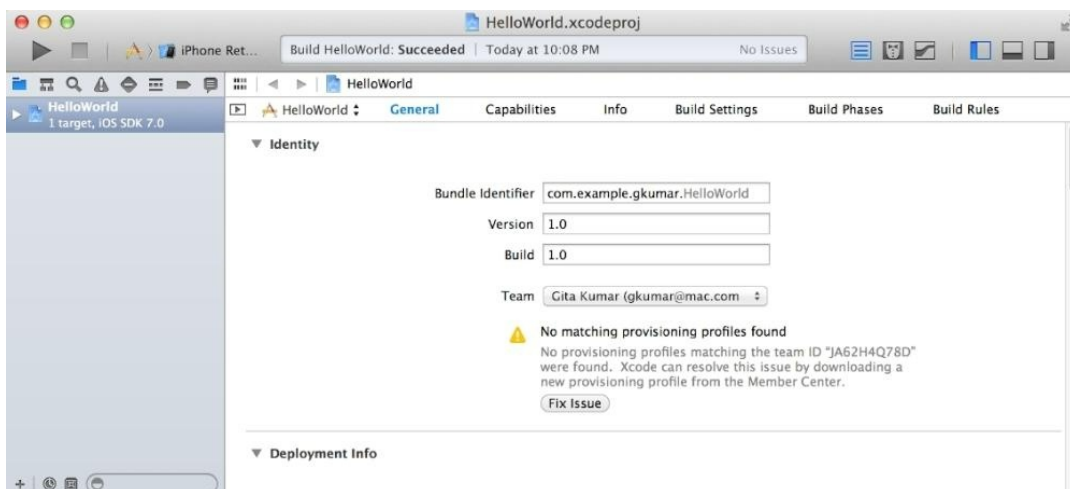
The following are the two kinds of provisioning profiles that are used in development:

- **Developer provisioning profile:** This is a collection of digital certificates and other entities that uniquely identify you as a developer, devices that you chose to use for testing, and enables devices to be used for testing. Each provisioning profile will contain your developer certificates, unique identifiers of devices that are used for testing, and your App ID.
- **A store provisioning profile:** This is a profile that ensures that the application was made by you, and authorizes the use of certain features and technologies. It contains your developer certificate that you need to submit your application to the App Store.

In the past, you had to handle provisioning yourself, but starting with Xcode 5.0, most of this is handled for you. This is a great feature, as provisioning was a huge pain for developers, and you could spend endless hours trying to fix incomprehensible issues.

Now, all you need to do is click on **Use for development** in **Organizer** when your device is connected and build the application. Organizer is an instrument within Xcode that helps you with various tasks such as adding more devices to your profile and handling crash logs. To access it, navigate to **Window | Organizer** in the top menu.

If there are some issues, Xcode will propose to fix them as displayed in the following screenshot. Usually, clicking on the **Fix Issue** button will magically fix everything without your intervention. In some cases, you will need to wait for a few minutes, but mostly that's it.



*The Fix Issue button will help you when something goes wrong*

# Preparing our application for the App Store

In order to prepare our application for App Store, we need to perform the following steps:

1. Set the deployment info by selecting a project in the project navigator and opening **Deployment info** if needed. Here, we set different settings of our application, such as supported versions (Sprite Kit only supports iOS 7 and higher versions), devices (our application is iPhone only), orientations (make sure only **Landscape** ones are selected), and others such as the status bar's style and the first launching storyboard.
2. Add app icons by performing the following steps:
  1. Open [Images.xcassets](#) in the project navigator.
  2. Select **AppIcon**.
    1. Add icons from resources of this chapter [58.png](#), [80.png](#), and [120.png](#) respectively and drop them to the icon places, as shown in the following screenshot. Since our application is iPhone-only, we only need three icons. If we made an iPad version, we would have to add many more icons.



*Adding icons to the project*

# Managing applications in iTunes Connect

iTunes Connect is an Apple and web-based tool that iOS developers use to set up contracts, banking, and tax information, and submit new applications or newer versions of old applications. We will use it to add a new record about our application. This record contains the following:

- Basic application information
- Pricing and territories where the application is available
- Adding languages and keywords
- Uploading large icons and screenshots
- Additional questions about your application (does it use cryptography or have other legally restricted uses)
- Preparing for a binary upload

## Note

Before registering the application in the App Store, you might have to fill some legal paperwork, such as accepting agreements with Apple about various distribution issues and providing Apple with information of your bank account and tax. This is available in iTunes Connect in the **Contracts, Tax and Banking** tab. You won't be able to distribute without accepting those agreements.

In order to register an application in iTunes Connect, perform the following steps:

1. Log in to iTunes Connect at <http://itunesconnect.apple.com>.
2. On the iTunes Connect home page, click on **Manage your apps**.
3. On the **Manage your apps** screen, click on **Add New App**.
4. On the next screens, follow the directions and fill everything that is required, as shown in the following screenshot:

The screenshot shows the 'App Information' form in iTunes Connect. At the top, it says 'iTunes Connect' and 'dimavolevodz@gmail.com'. The title is 'App Information'. Below it, a prompt says 'Enter the following information about your app.' The form contains four fields: 'Default Language' with a dropdown set to 'English', 'App Name' with the text 'Runnr', 'SKU Number' with 'com.dv.Endless-Runner', and 'Bundle ID' with a 'Select' dropdown. Each field has a help icon. Below the Bundle ID field is a link: 'You can register a new Bundle ID here.' At the bottom, there is a 'Cancel' button and a 'Continue' button. A link 'Does your app have specific device requirements? Learn more' is also present.

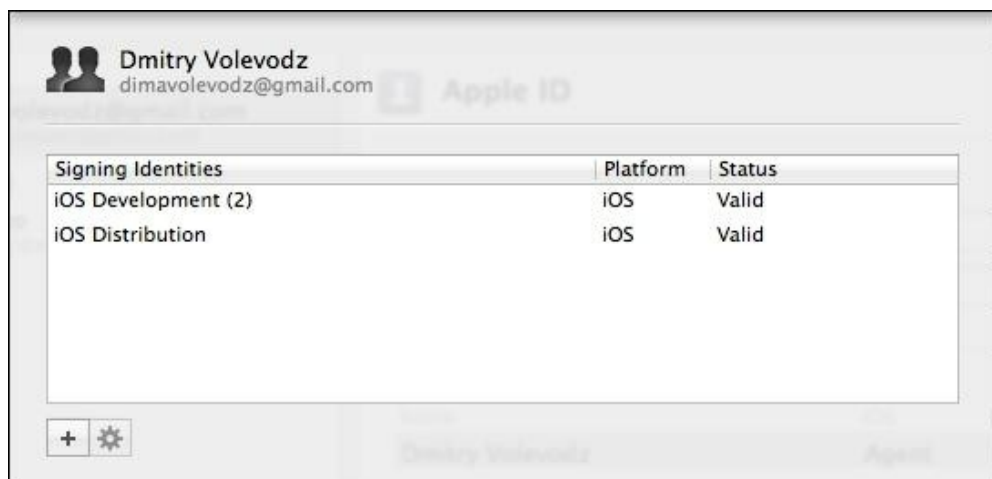
*Entering basic info about an application*

5. You will need to create a new bundle ID for your application. Click on the link to create it and pick it after you create it.
6. After this, you will need to set the version of the application (1.0 will do), copyrights, the category of the application, and ratings.
7. You will need to upload large icons (1024 x 1024) and screenshots from both 3.5 and 4-inch retina devices.

When you are done registering your application in iTunes Connect, you will see the **Waiting for Upload** status in your Application Details page. If it is of some other state, fix the issues that prevent the ready state, such as missing screenshots or text.

Now you will need to build the application for distribution. In order to do that, perform the following steps:

1. Make sure you have distribution certificates. To check this, navigate to **Xcode** in the top menu and then to **Preferences | Accounts** in the top pane. Select your account and click on **View Detail**. Under **Signing Identities**, you should see **iOS Development (2)** and **iOS Distribution**, as shown in the following screenshot. If something is missing, click on the **+** button on the same screen and it will add a distribution certificate to your account. Go to the iTunes developer center and click on **Certificates** and download it. After it is downloaded, run it to add to **Keychain**. After that is done, it should appear on the list.



*Both iOS Development and Distribution profiles are installed*

- After your profiles are set up, you will need to make a build for distribution. To do that, change the iOS device from the iOS simulator near the **Build** button on the left-hand side of the **Xcode** window. After that, click on **Product** in the top menu and archive the profiles there by clicking on the **Archive** option as shown in the following screenshot:



*Archiving your application*

- The Organizer should open with details about your application, as shown in the following screenshot. You can click on **Validate...** there to check if everything is right, and on **Distribute...** as soon as you want to upload it for review. Xcode will prompt you for your Apple ID and password, and if you have your application set up in iTunes Connect, it will be uploaded for review. You can also use the

**Distribute...** button to create a [.IPA](#) file of your game, and you may make it available for your testers as shown in the following screenshot. For further details on this, read about ad hoc distribution on the Apple Developer website.



*Application archived for distribution*

# Life after uploading

Your application can have several states, which are as follows:

- **Waiting for upload:** This is the state after you have registered your application in iTunes Connect but have not uploaded the binary yet.
- **Waiting for review:** This is the state in iTunes Connect after uploading your application.
- **In review:** This is the state when your application is currently in review. It takes any amount of time from minutes to days, depending on a few factors such as the complexity of the application or the reviewers' workload, but mostly they are random.
- **Ready for sale:** This means that if you have not changed the default date of the availability of your application, it will be available for download from the App Store on the day it gets approved.

Sales statistics and other useful data can be found in iTunes Connect. Please make sure you check all the available tools that it offers such as sales and trends, iAd, and catalog reports.

# Summary

In this chapter, we have learned what steps need to be taken in order to post your application to the App Store. You have learned about provisioning profiles, code signing, development and distribution certificates, and a process that needs to be undertaken each time you want to upload your application to the iTunes App Store.

# Index

## A

- accelerometer
  - about / [Accelerometer](#)
  - advantages / [Accelerometer](#)
- actions
  - types / [Actions](#)
- actions, Sprite Kit project
  - about / [Actions](#)
- affectedByGravity property / [Physics simulation basics](#)
- animation
  - about / [What is animation?](#)
  - adding, to Sprite Kit project / [Adding animations to our project](#)
- API, iOS7
  - about / [New APIs](#)
- application
  - preparing, for App Store / [Preparing our application for the App Store](#)
  - managing, in iTunes Connect / [Managing applications in iTunes Connect](#)
  - registering, in iTunes Connect / [Managing applications in iTunes Connect](#)
  - building, for distribution / [Managing applications in iTunes Connect](#)
  - states, after uploading / [Life after uploading](#)
- App Store
  - application, preparing for / [Preparing our application for the App Store](#)

## B

- background image

- adding, to Sprite Kit project / [Adding a background image to our game](#)
- background music
  - adding / [Adding sound and music](#)
- bitmasks
  - about / [Advanced physics](#)
- BodyWithCircleOfRadius method / [Physics simulation basics](#)
- BodyWithEdgeFromPoint
  - toPoint method / [Physics simulation basics](#)
- BodyWithPolygonFromPath method / [Physics simulation basics](#)
- BodyWithRectangleOfSize method / [Physics simulation basics](#)
- Bundle ID
  - about / [Bundle ID](#)

## C

- categoryBitMask
  - about / [Advanced physics](#)
- categoryBitMask property / [Physics simulation basics](#)
- character
  - moving, with actions / [Moving the character with actions](#)
- character state
  - handling / [Character states](#)
  - running state / [Character states](#)
  - jumping state / [Character states](#)
  - in air state / [Character states](#)
- Cocos2d
  - about / [Framework for game development](#)
- collisionBitMask
  - about / [Advanced physics](#)
- collisionBitMask property / [Physics simulation basics](#)

- contactBitMask property / [Physics simulation basics](#)
- contactTestBitMask
  - about / [Advanced physics](#)

## D

- density property / [Physics simulation basics](#)
- developer provisioning profile
  - about / [Provisioning profiles](#)
- draw calls
  - about / [What is animation?](#)
- dynamic property / [Physics simulation basics](#)

## E

- Emitter Node properties
  - setting / [First particle effect](#)

## F

- FPS (frames per second)
  - about / [Sprite Kit basics](#)

## G

- game center
  - features / [Game center renovations](#)
- game controllers
  - native game controllers / [Native game controllers](#)
  - basic concepts / [Game controller basics](#)
  - standard form-fitting controller / [Game controller basics](#)
  - extended form-fitting controller / [Game controller basics](#)
  - extended wireless controller / [Game controller basics](#)
  - using, in Sprite Kit project / [Using a controller in our game](#)
  - notifications, handling / [Handling controller notifications](#)

- game controller support
    - about / [New APIs, Game controller support](#)
  - game development
    - framework / [Framework for game development](#)
    - OpenGL / [Framework for game development](#)
    - UIKit / [Framework for game development](#)
    - third-party libraries / [Framework for game development](#)
    - Cocos2d / [Framework for game development](#)
  - game loop
    - about / [Game loop](#)
  - games
    - developing, for iOS7 / [Developing games for iOS 7](#)
  - gesture recognizers
    - about / [Using gesture recognizers](#)
    - using / [Using gesture recognizers](#)
- 
- infinite scrolling
    - adding / [Adding infinite scrolling](#)
  - iOS7
    - features / [Redesigning the iOS](#)
    - API / [New APIs](#)
    - games, developing for / [Developing games for iOS 7](#)
    - Sprite Kit / [Knowing about Sprite Kit](#)
  - iOS developer portal
    - URL / [Registering as a developer](#)
  - iOS Developer program
    - URL / [Registering as a developer](#)
  - iTunes Connect
    - application, managing / [Managing applications in iTunes Connect](#)

- about / [Managing applications in iTunes Connect](#)
- URL / [Managing applications in iTunes Connect](#)
- application, registering / [Managing applications in iTunes Connect](#)

## M

- mass property / [Physics simulation basics](#)
- Multitasking
  - about / [New APIs](#)

## N

- native game controllers
  - about / [Native game controllers](#)
  - advantages / [Native game controllers](#)
- node
  - about / [Nodes](#)
  - types / [Node types](#)
  - properties / [Node types](#)
  - methods / [Node types](#)

## O

- OpenGL
  - about / [Framework for game development](#)

## P

- parallax
  - about / [Adding a parallax background](#)
- parallax background
  - adding / [Adding a parallax background](#)
- particle effects

- creating / [First particle effect](#)
- particle emitter
  - about / [Particle emitters](#)
- physics body
  - properties / [Physics simulation basics](#)
- physics engine
  - about / [Physics engine](#)
  - physics simulation / [Physics simulation basics](#)
  - implementing / [Implementing the physics engine](#)
- physics simulation
  - about / [Physics simulation basics](#)
- provisioning profiles
  - about / [Provisioning profiles](#)
  - developer provisioning profile / [Provisioning profiles](#)
  - store provisioning profile / [Provisioning profiles](#)

## R

- register as a developer
  - with Apple / [Registering as a developer](#)
- restitution property / [Physics simulation basics](#)

## S

- scene
  - about / [Scenes](#)
- scene transitions
  - handling / [Scene transitions](#)
- score label
  - adding / [Adding a score label](#)
- shield animations

- adding / [Adding shield animations](#)
- SKEffectNode / [Node types](#)
- SKEmitterNode / [Node types](#)
- SKLabelNode / [Node types](#)
- SKNode / [Node types](#)
- SKShapeNode / [Node types](#)
- SKSpriteNode / [Node types](#)
- sound effects
  - adding / [Adding sound and music](#)
- Sprite Kit
  - about / [New APIs, Knowing about Sprite Kit](#)
  - advantages / [Benefits of Sprite Kit](#)
- Sprite Kit physics
  - about / [Advanced physics](#)
  - bitmasks / [Advanced physics](#)
- Sprite Kit project
  - about / [Sprite Kit basics](#)
  - anatomy / [Anatomy of a Sprite Kit project](#)
  - scene / [Scenes](#)
  - node / [Nodes](#)
  - node, types / [Node types](#)
  - actions / [Actions](#)
  - game loop / [Game loop](#)
  - background image, adding / [Adding a background image to our game](#)
  - character, moving with actions / [Moving the character with actions](#)
  - infinite scrolling, adding / [Adding infinite scrolling](#)
  - score label, adding / [Adding a score label](#)
  - animation, adding / [Adding animations to our project](#)
  - character state, handling / [Character states](#)
  - shield animations, adding / [Adding shield animations](#)
  - parallax background, adding / [Adding a parallax background](#)
  - scene transitions / [Scene transitions](#)
  - game controllers, using / [Using a controller in our game](#)
  - sound effects, adding / [Adding sound and music](#)
  - background music, adding / [Adding sound and music](#)
- store provisioning profile

- about / [Provisioning profiles](#)

## T

- Text Kit
  - about / [New APIs](#)
- texture atlas
  - about / [What is a texture atlas?](#)
  - benefits / [What is a texture atlas?](#)
- third-party libraries
  - about / [Framework for game development](#)
- touches
  - handling / [Handling touches](#)
- touchesBegan method / [Handling touches](#)
- touchesCancelled method / [Handling touches](#)
- touchesEnded method / [Handling touches](#)
- touchesMoved method / [Handling touches](#)

## U

- UIKit
  - about / [Framework for game development](#)
- usesPreciseCollisionDetection property / [Physics simulation basics](#)

## X

- Xcode
  - about / [What is a texture atlas?](#)