Community Experience Distilled

Learning Objective-C by Developing iPhone Games

Leverage Xcode and Objective-C to develop iPhone games

Amy M. Booker
Joseph D. Walters

[PACKT] PUBLISHING

www.it-ebooks.info

Learning Objective-C by Developing iPhone Games

Leverage Xcode and Objective-C to develop iPhone games

Amy M. Booker

Joseph D. Walters

BIRMINGHAM - MUMBAI

www.it-ebooks.info

Learning Objective-C by Developing iPhone Games

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 authors, 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: April 2014

Production Reference: 1180414

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

ISBN 978-1-84969-610-4

www.packtpub.com

Cover Image by Joseph D. Walters (joseph@diggablegames.com)

www.it-ebooks.info

Credits

Authors
Amy M. Booker
Joseph D. Walters

Reviewers
Michaël Adaixo
Paris Buttfield-Addison
John P. Doran
Conrad Irwin
Makzan
Clifford Matthew Roche

Commissioning Editor
Martin Bell

Acquisition Editors
Antony Lowe
Gregory Wild

Content Development Editor
Chalini Snega Victor

Technical Editors
Tanvi Bhatt
Gaurav Thingalaya

Copy Editors
Janbal Dharmaraj
Alfida Paiva
Laxmi Subramanian

Project Coordinator
Wendell Palmer

Proofreaders
Simran Bhogal
Paul Hindle

Indexer
Tejal Soni

Graphics
Abhinash Sahu

Production Coordinator
Aditi Gajjar Patel

Cover Work
Aditi Gajjar Patel

www.it-ebooks.info

About the Authors

Amy M. Booker is new to writing. Along the way, she has become a happy gamer and has been raising her family to enjoy and appreciate the art of games, whether card, board, or video. As an advocate of special needs, specifically Autism, she hopes to write a book or design a game to support those with special needs.

Amy lives in San Ramon, California, with her husband and their two boys.

Thanks to my whole-hearted, supportive husband, Randy, who encouraged me to embark on the adventure of writing.

I appreciate and thank my two boys for their meagre allowance of computer time so that I may write.

Thank you mom and dad for always believing in me and expecting me to take the risks needed for me to grow.

Thank you Paige and Joseph for being so astonishingly trustful. You led me on this path, and I thank you for it.

---

www.it-ebooks.info

Joseph D. Walters has been immersed in the creativity, technology, and business of software and game development for over 20 years. He currently is a partner at Third Track Inc., a game development and game technology company. Before Third Track Inc., he was the CEO at MindFuse Games, where he helped raise capital from angel investors and led the development of a massive multiplayer-gaming world. Joseph was also a full partner and technical director at Skunk Studios, one of the leading full-service developers dedicated to bringing casual games to the mass market. Prior to founding Skunk Studios, he was a senior engineer at Shockwave.com, where he developed some of the most popular titles on Shockwave's website. Before starting at Shockwave.com, Joseph owned a top consulting firm that advised some of the largest technology firms in the world, including IBM, Compaq, AT&T, Bell Labs, and countless others.

I would like to thank my family for all of their support throughout the development of this book. Without them, I would not have been able to accomplish this.

I would like to specially thank Mark Lemmons for his help on the game art, and Skunk Studios for allowing me to use their audio.

---

www.it-ebooks.info

About the Reviewers

Michaël Adaixo is a software developer currently working at IS2you developing tools and software for intelligent systems and interactive advertising. He is the creator of an award-winning project called *Wi-GO*.

He currently lives in Portugal and is pursuing his MSc in Computer Science and Engineering. He is writing his thesis on the field of artificial intelligence. Because he is passionate about developing and designing games, he spends his spare time working with Unity 3D to create games. He recently released a casual game for mobile platforms, Memtiles Kids (`http://www.cinderinteractive.net`).

One day, he hopes to make a living out of developing games, either working at a game company or creating his own.

Paris Buttfield-Addison is the co-founder of Secret Lab, a mobile game development studio based in Hobart, Tasmania, Australia. He designs games, runs game development projects, and occasionally writes code.

He is also the co-author of *Learning Cocoa with Objective-C Third Edition*, *O'Reilly Media*, 2012 and *iPhone and iPad Game Development For Dummies*, *Wiley*, 2010. He is currently co-writing *iOS Game Development Cookbook*, *O'Reilly Media*, 2014 and *Mobile Game Development with Unity*, *O'Reilly Media*, 2014.

Secret Lab has built hundreds of mobile apps for people around the world, including Meebo for iOS, Android, and BlackBerry (Meebo, Inc., Mountain View, CA; it was acquired by Google in 2012); Play School Art Maker for iOS (2011); Foodi for iPad (2011); Good Game for iPhone (2012); Play School Play Time for iPad (2013, Australian Broadcasting Corporation); National Science Week for iOS (2012-2013); and many others.

Paris submitted his PhD at the University of Tasmania (UTAS) in late 2013, where he's part of the Information and Interaction Group. His PhD explores Personal Information Management (PIM) and tablet technology (iPads).

www.it-ebooks.info

John P. Doran is a technical game designer who has been creating games for over 10 years. He has worked on an assortment of games in teams that constituted of just himself in the beginning to over 70 in student, mod, and professional projects.

He previously worked at LucasArts on *Star Wars: 1313* as a game design intern. He later graduated from DigiPen Institute of Technology in Redmond, WA, with a Bachelor of Science degree in Game Design.

John is currently working at DigiPen's Singapore campus as the lead instructor of the DigiPen-Ubisoft Campus Game Programming Program, instructing graduate-level students in an intensive, advanced-level game-programming curriculum. In addition to this, he also tutors and assists students on various subjects while giving lectures on C++, Unreal, Flash, Unity, and more.

He is the author of the following books:

- *Getting Started with UDK*, *Packt Publishing*
- *Mastering UDK Game Development*, *Packt Publishing*

He has also co-authored *UDK iOS Game Development Beginner's Guide*, *Packt Publishing*.

Conrad Irwin is a computer programmer who specializes in developer and productivity tools. He helped build Rapportive in a way through which anyone could become more effective at e-mailing people, and now works on making Bugsnag, the best tool for tracking exceptions in production. His other projects include Pry, the featureful development console for Ruby, and `showterm.io`, which lets developers share what they see on the terminal easily. When not programming, he enjoys Irish and American dancing and blogs at `http://cirw.in/`.

www.it-ebooks.info

Makzan is a developer with a specialty in web development and game design. He has over 10 years' experience in building digital products, including real-time multiplayer interaction games and iOS applications.

He has written two books and one screencast series for building a Flash virtual world and creating games with HTML5 using the latest web standards. He is now organizing different kinds of courses in Hong Kong and Macao.

I would like to thank my wife, Candy Wong, for supporting all my writings.

Clifford Matthew Roche is a self-taught game programmer who has been working on AAA projects since 2008, focusing on engine development, performance, and animation. He is credited on several games, including NHL 09, FIFA 10, and Motion Sports: Adrenaline.

Recently, he started his own game development company, Firestarter Games, where he works to bring high-quality games to mobiles and consoles with Unity3D. Firestarter Games' first project, Globulous, was nominated for Best Audio (Casual/Indie) at the Audio Network Guild Music Awards.

www.it-ebooks.info

www.PacktPub.com

Support files, eBooks, discount offers, and more

You might want to visit `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` and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at `service@packtpub.com` for more details.

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

`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`, you can use this to access PacktLib today and view nine entirely free books. Simply use your login credentials for immediate access.

www.it-ebooks.info

www.it-ebooks.info

Table of Contents

Preface **1**

Chapter 1: The Environment – Installing and Working with Xcode **7**

Becoming an Apple developer **7**
Registering as an Apple developer 8
Becoming an iOS developer **10**
Registering as an iOS developer 10
Downloading and installing Xcode **13**
The Xcode IDE **14**
Opening Xcode for the first time 14
The Xcode workspace **18**
The toolbar 18
The Stop button 19
The Scheme pop-up menu 19
The Activity viewer 19
The Editor buttons 19
The View buttons 19
The Navigation area 20
The Editor area 22
The Debugger area 23
The Utility area 24
The iOS Simulator **26**
Summary **28**

Chapter 2: Programming and Objective-C **29**

Understanding Objective-C **29**
Programming basics **30**
Basic data types 30
Text data types 30
Numeric data types 31

www.it-ebooks.info

Other data types 31
Variables 32
Constants 33
Comments 33
Operators 35
Arithmetic operators 35
Comparison operators 35
Logical operators 36
Increment and decrement operators 37
Compound operators 37
Control statements 38
The if statement 38
The if…else statement 38
The if…else if statement 39
The switch statement 39
Loops 41
The for loop 41
The while loop 42
The do while loop 42
The break statement 43
The continue statement 43
Functions 44
Variable scope 45
Object-oriented programming 46
Objects 47
Classes 47
Methods 47
Pointers 48
Objective-C data types 49
NSString 49
NSMutableString 50
NSNumber 51
NSSet 52
NSMutableSet 52
NSArray 53
NSMutableArray 54
Hello World! 54
Programming with Objective-C 58
@interface 58
@implementation 58
Model-view-controller 62
Summary 63

www.it-ebooks.info

Chapter 3: iPhone Game Development Basics – The Matching Game **65**
Scenes and views **65**
Understanding storyboards 70
The memory game code 80
Frameworks 85
Building our game scene 86
Summary **104**
Chapter 4: Drawing Pictures – Space Invaders **105**
Understanding screens and graphics **105**
Adding graphics programmatically **106**
Moving our ship sprite **112**
Creating our enemies **117**
Using a sprite sheet **124**
Creating the enemy bombs **130**
Summary **136**
Chapter 5: The Game Engine – Space Invaders Part 2 **137**
Understanding the game logic **137**
Understanding state machines **138**
Learning the game state **140**
Collision detection **151**
Timers and frame rates **152**
Debugging your code **153**
Summary **159**
Chapter 6: The Audio Engine – The Simon Memory Game **161**
Introduction to audio **161**
File formats **162**
Bit rates 162
Sample rates 162
Data formats **162**
The Simon Says game **163**
Summary **182**
Chapter 7: User Interactivity – Mini Golf **183**
Using user input and touch events **183**
Using gestures in iOS apps **184**
Using touch start, move, and end **193**
Initializing our code 195
Touch events 198
Updating the screen 199

www.it-ebooks.info

Using other mechanisms **201**
Shake 202
Orientation 202
Tilt 204
Motion 206
Summary **207**
Chapter 8: The Galaxy Game **209**
Planning the game **209**
Creating the game concept document 210
Creating the game design document 210
Creating the technical design document 211
Considering the design and flow **212**
Focusing on resource management **213**
Creating game screens **215**
Building the user interface **218**
Setting up the gameplay environment **231**
Creating and controlling the player **232**
Creating and controlling the enemies **236**
Collision detection **237**
Adding sounds **238**
Summary **239**
Chapter 9: Releasing Your Game **241**
Preparing your app for distribution **241**
Selecting your identity **242**
The bundle identifier 242
The version number and build string 242
Team 243
The deployment information **244**
The deployment target 245
Devices 245
The device orientation 245
The status bar style 245
Selecting app icons and launch images **245**
Linked frameworks and libraries 247
Beta testing and ad hoc distribution **247**
Register all test devices 247
App ID 249
Ad hoc provisioning profile 250
Archive and create your app package 251

www.it-ebooks.info

Installing on test devices 251
iTunes Connect app 251
Avoiding rejection from the App Store 252
Summary 252
Chapter 10: Conclusion 253
Developing an app for iPad 254
Using Apple's game center 254
In-app purchasing 254
Implementing iAd 254
The social framework 254
Core Location and Map Kit 255
Saving space with iCloud 255
Exploring game engines 255
Summary 256
Index 257

www.it-ebooks.info

www.it-ebooks.info

Preface

The introduction of the Apple App Store has empowered thousands, even millions of people to embrace software development. Using Objective-C and the Xcode IDE, you can produce awesome games and launch them on the App Store, allowing you to make and sell games quickly and easily.

Welcome to *Learning Objective-C by Developing iPhone Games*. This book will offer you a step-by-step process to develop iPhone games using the Objective-C programming language. You will be intrigued by the use of old games, such as Galaga and Space Invaders, as teaching tools to receive a hands-on understanding of game development. We will teach you everything you need to know in order to use Apple's Xcode integrated development environment. When you are finished, you will be able to program your own games from the ground up using graphics, animations, sounds, and interactivity.

What this book covers

Chapter 1, *The Environment – Installing and Working with Xcode*, covers how to download, set up, and install Xcode. You will also learn what Xcode is and how we will be using it. This will introduce you to the development environment and the space they will be working on.

Chapter 2, *Programming and Objective-C*, is a soft introduction to programming languages in general, or a programming language primer if you prefer. When we are done with this chapter, you will have a general understanding of how most programming languages are structured and how this works in games. We will take you through the basics of object-oriented programming and a description of how Objective-C uses classes. We will wrap up the chapter with an explanation of how the model-view-controller design pattern applies to iOS programming.

www.it-ebooks.info

Chapter 3, *iPhone Game Development Basics – The Matching Game*, explains the CoreGraphics, Foundation, and UIKit frameworks. We will build a simple user interface using Xcode's interface builder in a single view application. Then, we will use this to create a simple memory game using some basic game logic, UIButtons, and other standard library components.

Chapter 4, *Drawing Pictures – Space Invaders*, teaches you to program a Space Invaders game using images, views, and the view controller. You will learn how to display a background image for your game and program your alien to drop bombs. You will use these to try and destroy your player using CoreAnimation. You will also create timers in your game using NSTimer methods.

Chapter 5, *The Game Engine – Space Invaders Part 2*, will expand the Space Invaders game by examining NSObjects and basic object-oriented programming structures to create enemies. You will learn the basics of artificial intelligence, collision detection, and storing information about your invader in objects, properties, and arrays. You will also learn the basics of how to debug your code.

Chapter 6, *The Audio Engine – The Simon Memory Game*, will create a simple audio-based memory game. In this chapter, you will learn about all of the different audio formats and data formats that can be used for iOS audio. We will explain the AVAudioFoundation frameworks in order for you to add sounds and music to your app. You will learn how to use different types of audio, which can be played back inside your game for both background music and as audio effects.

Chapter 7, *User Interactivity – Mini Golf*, teaches different ways of programming touch controllers for your app using touch events. We will show you how to program your graphics to move using buttons, gestures, and multitouch interactivity.

Chapter 8, *The Galaxy Game*, talks about resource management and some basics on planning for what is being used, what is planned to be used, and how these resources are stored. We will use this and everything you have learned previously in order to create your final game similar to Galaga.

Chapter 9, *Releasing Your Game*, speaks about what needs to be done once you have completed your game; it's not yet finished. You will still need to get your app into the App Store. Preparing your game for the App Store can be a very confusing process; however, we will try and make it a bit easier for you. In this chapter, we are going to introduce you to iTunes Connect, provisioning profiles, and development profiles. We will show you how to set up your code and prepare it for release in the Apple Store and how to avoid some issues that all developers run into at one point or another.

Chapter 10, *Conclusion*, gives an overview of what we covered in the book and covers in brief other technologies that Apple has to offer.

www.it-ebooks.info

What you need for this book

In order to get the most out of this book, there are a few essentials you will need:

- A Mac computer running OS X 10.8.4 or higher (no, you cannot program an iPhone in Windows; sorry)
- Basic knowledge of programming is helpful
- Xcode 5.1 (we will explain how and where to get this in the first chapter)
- Safari or another web browser
- Internet access
- If you would like to distribute your apps in the App Store, you are required to have a paid developer account. We will explain how to sign up for a developer account in the first chapter.

Who this book is for

If you are a beginner or an enthusiast who dreams about creating games, or if you have already worked in this domain but need some additional inspiration and knowledge, this book is for you. No programming experience is expected.

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, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "Your debugger will go to the first line in the `loadingScreen` method."

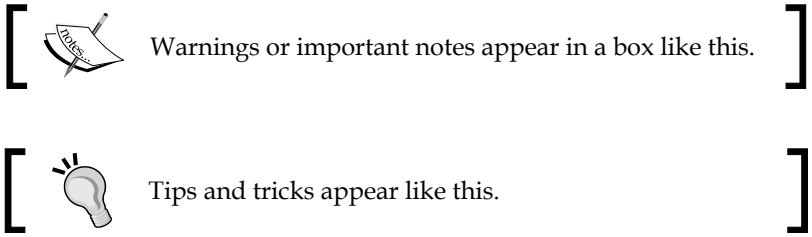
A block of code is set as follows:

```
-(void)intersectCheck{
  BOOL isConnecting =
    CGRectIntersectsRect(self.enemies.enemiesBullet.
        bombRect, self.playerOne.playerRect);
  if(isConnecting == true){
    [NSTimer scheduledTimerWithTimeInterval:2
        target:self
        selector:@selector(endScreen)
        userInfo:nil
```


www.it-ebooks.info

```
            repeats:NO];
        [self changeState:RELOADING];
      }
    }
```

New terms and **important words** are shown in bold. Words that you see on the screen, in menus or dialog boxes for example, appear in the text like this: "Navigate to **View | Debug Area | Show Debug Area.**"



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`, and mention the book title via 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`.

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.

www.it-ebooks.info

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/submit-errata`, selecting your book, clicking on the **errata submission form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded on our website, or added to any list of existing errata, under the Errata section of that title. Any existing errata can be viewed by selecting your title from `http://www.packtpub.com/support`.

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` 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` if you are having a problem with any aspect of the book, and we will do our best to address it.

www.it-ebooks.info

www.it-ebooks.info

1

The Environment – Installing and Working with Xcode

In this chapter, you will learn how to download, set up, and install the Xcode development tools and **Software Development Kit** (**SDK**). You will also learn what the Xcode IDE is and how you will be using it. This will introduce you to the development environment and the space you will be working on most of the time while you are developing.

In this chapter, we will cover the following topics:

- Becoming an Apple developer
- Becoming an iOS developer
- Downloading and installing Xcode
- The Xcode IDE
- Opening Xcode for the first time
- The Xcode workspace

Becoming an Apple developer

Before you can begin building apps and games for the iPhone, you will need to become a certified Apple developer, which gives you free access to Apple's developer website, programming tools, tutorials, documentation, and much more. As a certified Apple developer, you are able to create apps and games for Mac, Safari, and iOS.

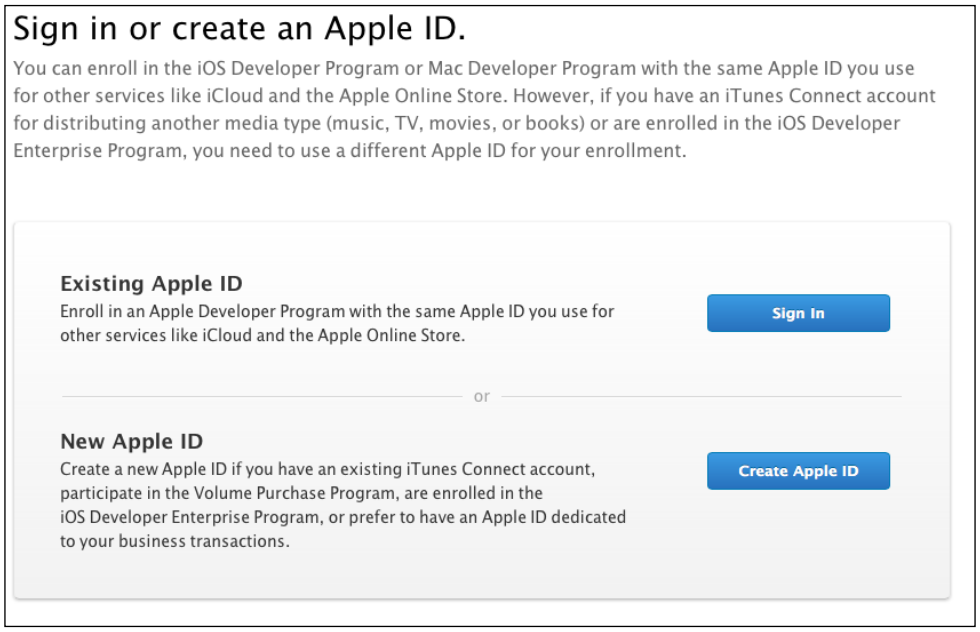
www.it-ebooks.info

Apple requires you to register as an Apple developer first. The registration is free and provides you with access to all of the tools and the iOS SDK required to develop apps for iPhone and iPad. Registering also gives you access to exclusive Apple content such as development videos, guidelines, tutorials, Q&As, and sample code. You will also have access to Apple's developer forums should you run into any questions that can be answered by the developer community. You can register at `https://developer.apple.com/register/`.

The free tier of Apple registration gives you access to the basic levels of development for Apple products. This limited access allows you to test your game on your computer (simulator) to make your game work, but without the ability to sell your game at the Apple Store.

Registering as an Apple developer

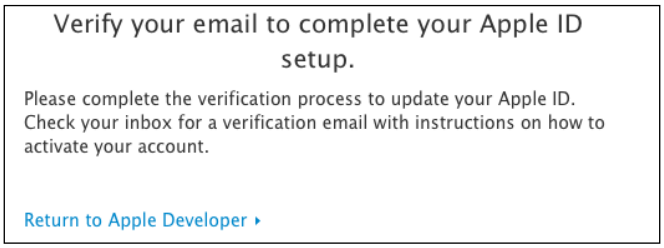
If you are not already an Apple developer, you just have one option available and that is to create an ID. The registration page looks as follows:



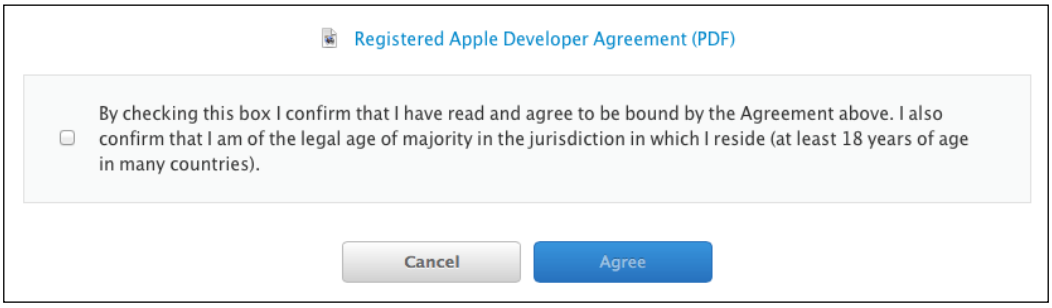
www.it-ebooks.info

The following are the steps available for you depending on whether you are registered as an Apple developer:

1. **Sign in or create an Apple ID**: Select whether you will need an existing Apple ID or create a new Apple ID. If you don't have an Apple ID, you will need to create one by clicking on the **Create Apple ID** button. If you have one already, click on the **Sign In** button and move ahead to step 4.
2. Supply all of the personal information they require. Read the terms of service and privacy policy and then click on the **Create Apple ID** button.



3. You will need to verify your Apple ID by responding to their e-mail. You will also need to repeat this process with the rescue e-mail address you provided. Then, click on **Return to Apple Developer** and you will be redirected to the iOS Dev Center (`http://developer.apple.com/`) to log in to your account. This will take you back to step 1, but this time you will sign in using your new Apple ID.
4. Read through the Apple Developer Agreement. Check the box once you are ready to continue and click on the **Agree** button.



5. Fill out the **Tell us about yourself** screen and select the **Register** button. You are now an official Apple developer.

www.it-ebooks.info

Becoming an iOS developer

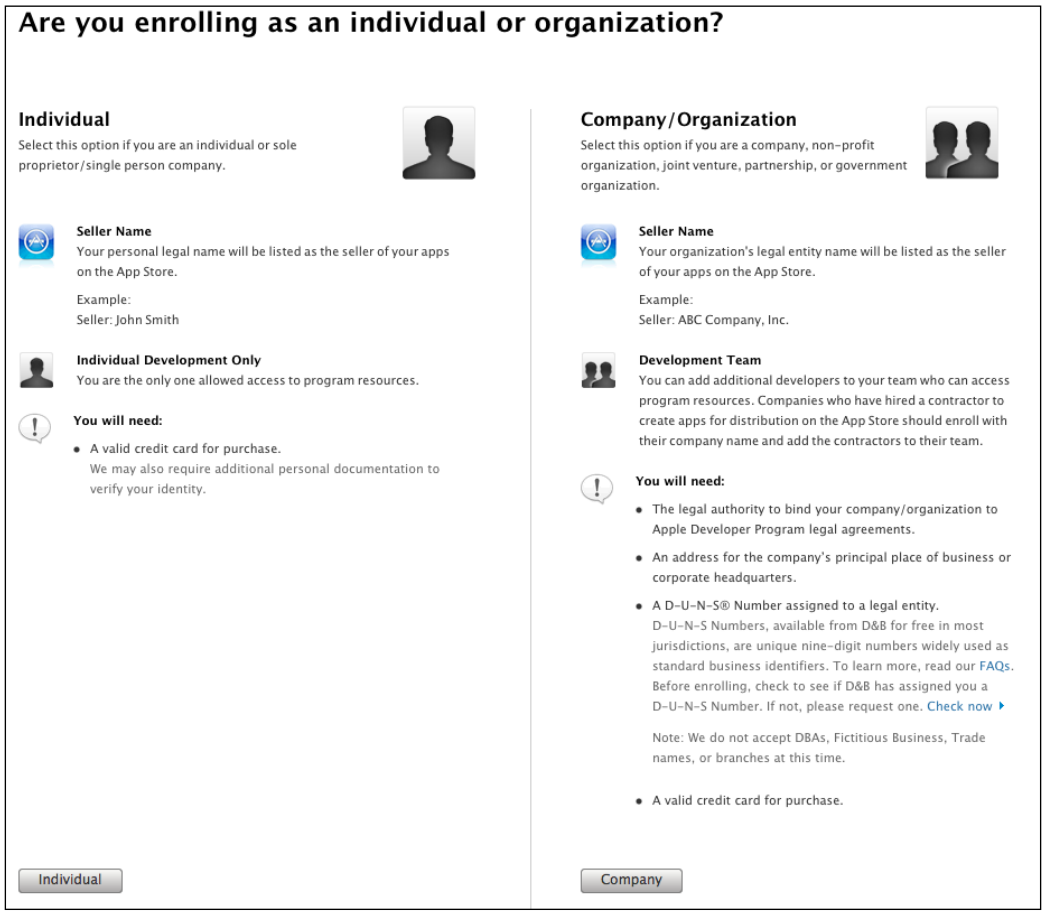
Going a step further and becoming a certified iOS developer is a requirement to release your app or game for iOS devices. In other words, to perform testing on iPhones and iPads, gain access to beta software, or to release products in the Macintosh Apple Store or the iOS Apple Store, you will need to join one of the paid developer programs. You can join either the iOS Developer Program or the Macintosh Developer Program depending on which store you prefer to release from, or join both. It costs 99 USD per year per program, and it offers several resources that are helpful for programming and releasing your apps to the public. It is recommended to sign up for the iOS Developer Program as it will make testing much more realistic, especially when testing anything that has to do with touch.

Registering as an iOS developer

Register for the iOS Developer Program at the link that is shown in the following screenshot using the following steps:

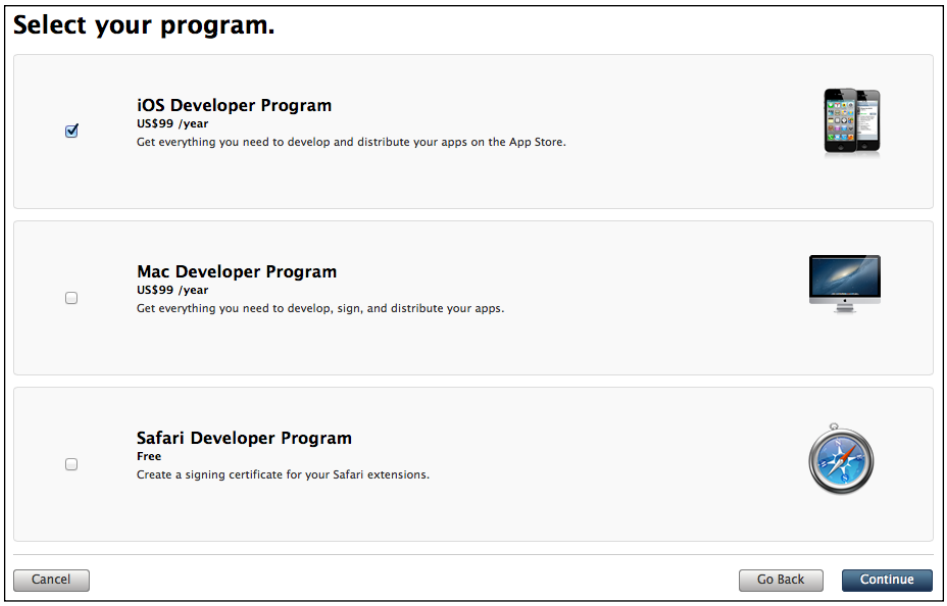
www.it-ebooks.info

1. Once you are at the website, select the **Enroll Now** button. The next screen describes everything you need to develop and distribute apps for iOS and OS X. Review the steps and click on the **Continue** button.
2. You will need to log in with your newly created Apple ID and hit the **Continue** button.
3. You will be redirected to the **Sign in or create an Apple ID** page. You should already be logged in with your Apple ID, so you can just hit the **Continue** button.
4. Choose whether you are registering as a company or as an individual. If you register as a company, you will be able to add additional team members who can access the program's resources and a company name will appear in the seller information in the App Store. If you register as an individual, your name will appear as the seller in the App Store and you will be the sole developer in your team.



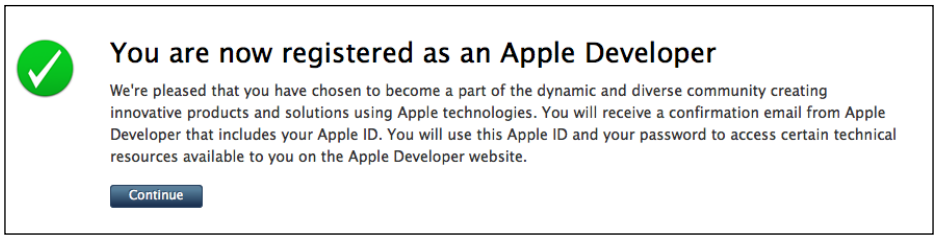
www.it-ebooks.info

5. On the **Enter your billing information for identity verification** page, you will need to enter your credit card information and then hit **Continue**.
6. On the **Select your program** page, check the box for the iOS Developer Program and any other developer programs you would like to join and then hit the **Continue** button.



7. On the **Review your enrollment information and submit** page, take a look through all of the information you have provided and make changes where necessary or click on the **Place Order Now** button to go to the next screen.
8. In the **Program License Agreement** page, read through the license agreements and click on the **I Agree** button.
9. Add the developer program to your cart. Click on **Continue** once you have entered all of the necessary information to pay for your account.
10. You will receive an e-mail from Apple with a verification code. Enter the code into the field provided in the dialog box. If you did not receive an e-mail, select the **verification email** button in the bottom-right corner of the dialog box, wait for the e-mail, and then enter it into the field. Click on the **Continue** button. Bask in the glory; you are now an official iOS developer.

www.it-ebooks.info



11. Click on the **Continue** button when you are ready.

And that's all there is to it; you are now an official iOS developer. You can download the beta software, test on the actual hardware, and distribute your apps to your heart's content.

Downloading and installing Xcode

Once you are registered, you will need to download and install the Xcode developer tools by performing the following steps:

1. You can download and install Xcode from the Macintosh App Store at `http://itunes.apple.com/us/app/xcode/id497799835?ls=1&mt=12`.
2. Select the **Free** or **Install** button on the App Store page. Once it has completed the install process, just launch Xcode from your `Applications` folder.
3. You can find additional development tools from the Apple developer website at `https://developer.apple.com/`.

Removing Xcode

For Xcode 4.3 and later versions, if you need to remove Xcode, just drop it in the trash like most Mac applications. If you run into issues installing a new version of Xcode or just want to remove an older version and you need to completely remove Xcode from your system, you will need to run a special command-line instruction:

```
sudo <Xcode>/Library/uninstall-devtools --mode=all
```

This will ensure all of the associated files are removed from your library directory. `<Xcode>` is the directory where tools are installed. For typical installations, the full path is `/Xcode4/Library/uninstall-devtools`.

Before doing this, it is best to make sure this is really what you want to do, because once it is done, it is removed completely and thoroughly. You will always have the option of reinstalling Xcode, but you will need to go through the entire install process from the beginning and any additional resources you installed will also need to be reinstalled.

www.it-ebooks.info

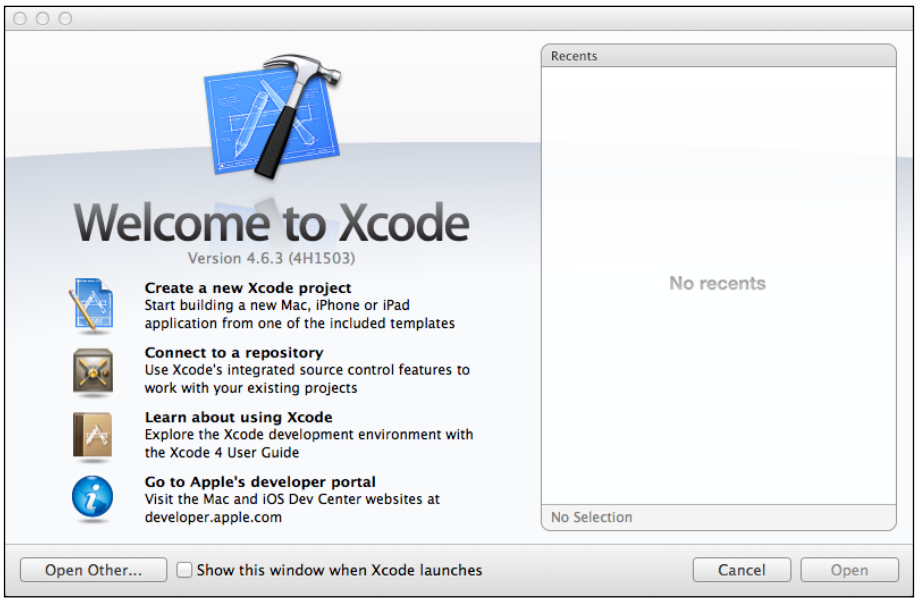
The Xcode IDE

Xcode is Apple's **Integrated Development Environment** (**IDE**) for developing for both iOS and Mac OS X. In other words, most of what you will need in order to develop software for the Mac or iOS devices is integrated into one nice, clean application. When you install Xcode on your Mac, you get everything needed to develop for both the Mac and iOS devices. The IDE takes everything from your program and puts it all together into a single executable application; this is referred to as compiling. If you are compiling on your Mac or the iOS simulator, you can run your application once everything is compiled. However, it gets a bit more complicated when compiling for iOS devices, but we will run through that later.

Opening Xcode for the first time

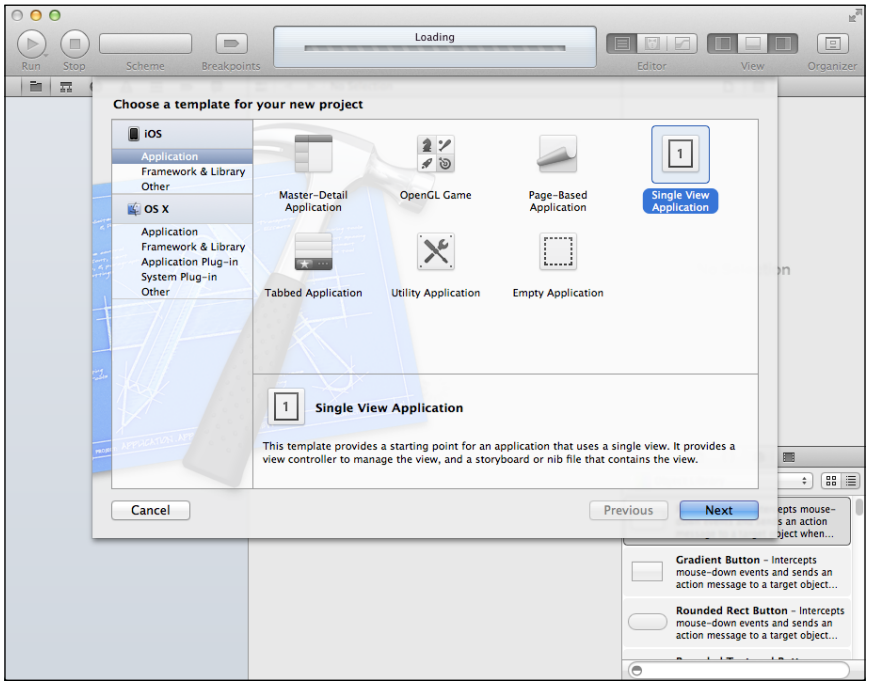
Let's begin by opening the Xcode IDE and explore how to create an app for the first time:

1. Open `Xcode.app` either through your finder in the applications directory or through Apple's **Launchpad**. If you have never created or opened an Xcode project before, you should be presented with a screen that looks similar to the following:



www.it-ebooks.info

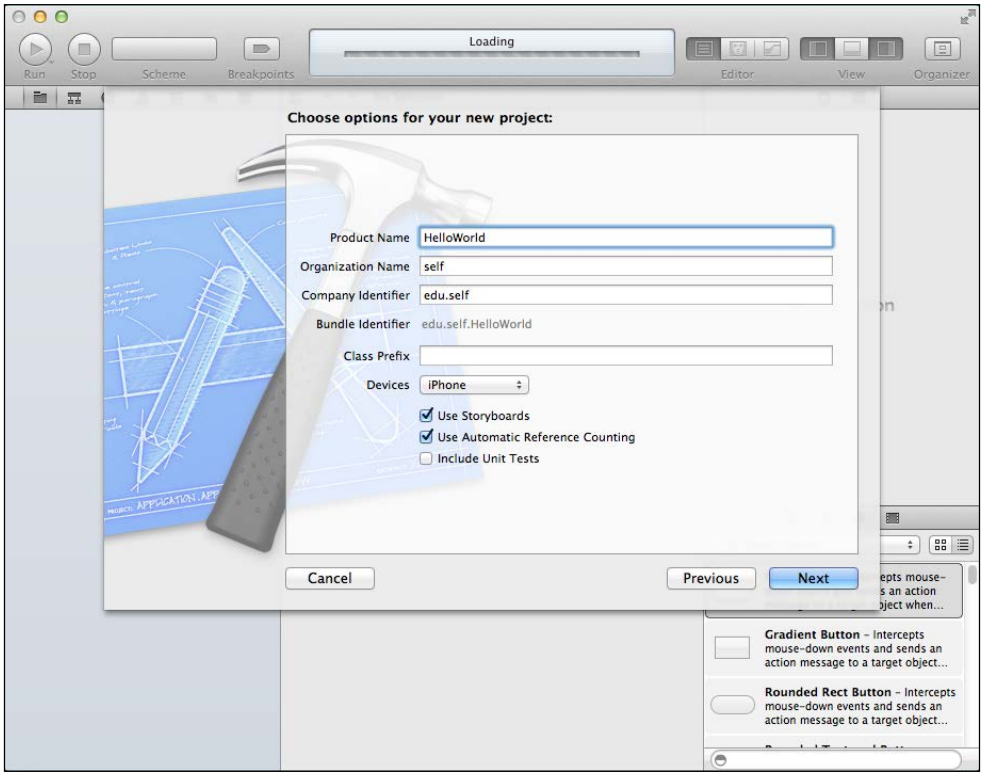
2. In the **Welcome to Xcode** dialog, select the **Create a new Xcode project** button. If you have created a project with Xcode previously, this dialog panel may not show up. If this is the case, you can navigate to **File | New | Project...**. This will open a new project window with the **Choose a template for your new project** dialog, as shown in the following screenshot:



3. In the left-hand side column, you will see a choice of operating systems and a list of types of templates. Apple provides several precreated templates for both OS X and iOS apps. We will discuss some of the different template types later in the book. For now, select **Application** under the iOS operating system. Select **Single View Application** from the precreated templates area and then click on the **Next** button.

www.it-ebooks.info

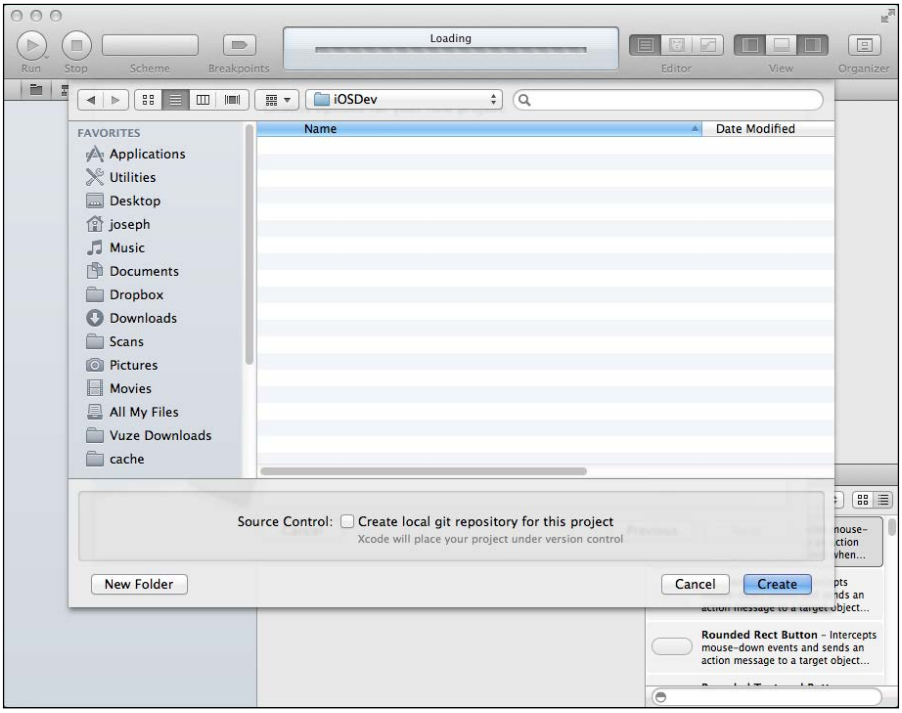
4. The next dialog is your project options panel. This is where you will give your project a name and set some additional options. For our first project, we will set **Product Name** to `HelloWorld` in accordance with standards set aside for all first projects in all programming languages. The organization name can be set to anything but is usually set to the name of your company or organization; for now, we will just use `self`:



5. The company identifier is a unique name that is used with the product name to create the bundle identifier for the product in order to distinguish your app from other apps. Apple recommends using a reverse domain name convention for your company identifier. This is like reversing the domain name or URL for your company's website; for example, if you were an educational company named *Teaching Games*, your website would be `teachinggames.edu` and the reverse domain would be `edu.teachinggames`. This would make your bundle identifier show up as `edu.teachinggames.helloworld` in the next line. You do not need to set this right now if you are just building and running your app locally. As it is used to create your app's bundle identifier, it would be best to assign it before you start testing your app externally or submit it to the Apple Store.

www.it-ebooks.info

6. The class prefix is a string of letters that will show up in front of the names of any custom classes you create for your project. This should be a short prefix. Some people will use their company's name, an acronym for their project, or their initials. Although this is not necessary, it does prevent collisions with other classes that may end up having the same names.
7. Once you have all of your information entered, click on the **Next** button.
8. Choose a place to save your project and click on the **Create** button.



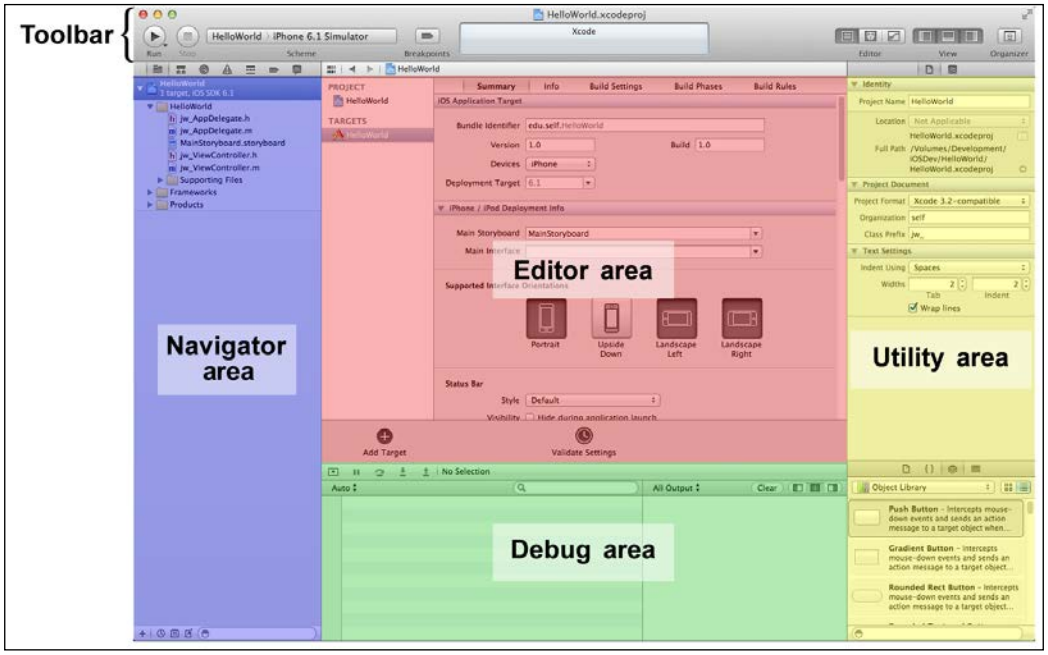
9. You have now created your first iOS application. Once it is created, you are brought to the workspace window of Xcode. If you click on the Run button in the upper-left corner of the window, it will open the iOS simulator and run your app. As we have not added any code to tell the app what to do, it will only bring up a blank white screen.

> You may have noticed the **Create local git repository for this project** checkbox below the file list. We will not be covering Git or any version control system in this book, but if you know what Git is and you are comfortable creating a repository, feel free to check this box. Otherwise, just leave it unchecked since it creates a lot of hidden files on your hard drive. If you are interested in learning about Git, feel free to check out their site: `http://git-scm.com`.

www.it-ebooks.info

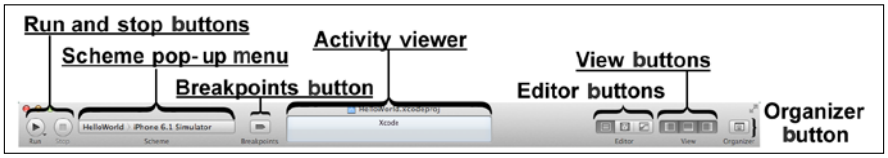
The Xcode workspace

Once you have created your project, you are presented with the Xcode workspace. Most of what you will need to program for the iOS environment is contained within a single development environment: the **workspace**. The workspace is broken up into several sections, as shown in the following screenshot:



The toolbar

The toolbar rests at the top of your workspace area and stores your frequently used workspace management tools for quick access and reference. Not to state the obvious, but the toolbar is a bar where all of your tools like to live. The following screenshot portrays the toolbar:



Going from left to right, the toolbar contains the following tools.

www.it-ebooks.info

The Run button

The Run button can actually change based on the task you are running, but the primary function is to build and run your code. If your code builds correctly, it will run your app and start a debug session.

The Stop button

If you have a running session, the Stop button will activate and allow you to stop the active app and the current session.

The Scheme pop-up menu

A scheme is a collection of targets that your app is set up to build. This pop-up menu allows you to choose which collection you wish to build.

The Activity viewer

The Activity viewer gives you information on everything going on in your app. If something is wrong with your code, this will be the first place you see a notification. If you do get a warning or alert, you can get detailed information from the issues navigator (see the **Navigation area** in the screenshot of the workspace).

The Editor buttons

The **Editor** buttons allow you to choose from three different types of editors. The first button is the standard editor, which is what will be used in most cases and changes based on what you are editing. The second button is the assistant editor, which brings up an additional text editor window. The third button is the version editor, which is used when your app is tied to a source control system.

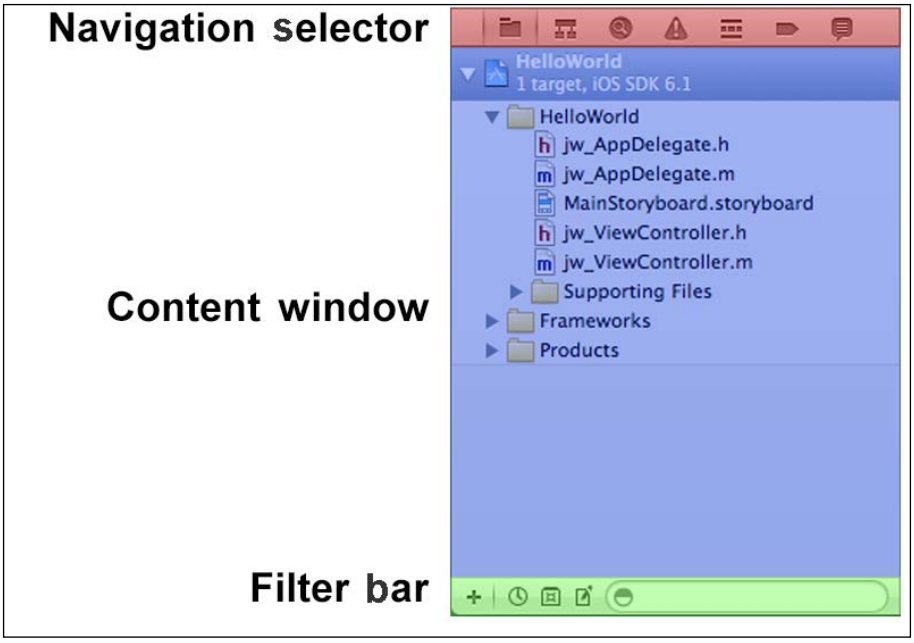
The View buttons

The **View** buttons turn on/off the different panes in the workspace window. The **Editor area** will always be available, but you can turn on/off (in this order) the **Navigator**, **Debugger**, and the **Utility areas** by clicking on the buttons corresponding to their position.

www.it-ebooks.info

The Navigation area

The **Navigation area** is broken up into three sections (from top to bottom): the **Navigation selector**, **Content window**, and **Filter bar**. The entire navigation area can be turned on/off using the leftmost **View** button on the toolbar. The **Navigation area** can be seen in the following screenshot:



The **Navigation selector** bar has a series of buttons that you can use to change the data in the **Content window**:

- The **project navigator** : This is used for navigating, managing, and organizing your project files. By selecting your file from the **Content window**, you will be shown the selected file in the **Editor area** where the file can be modified. The filter bar changes to let you add and import files, show recent files, show source-controlled files, show unsaved files, or only show files with a specific name.
- The **symbol navigator** : This is used for viewing a list of all of the symbols in your project. By selecting an item from the content list, you will be taken to the file and position of the symbol's location. The **Filter bar** lets you show only class symbols and project-defined symbols, hide members, and filter by a specific name.

www.it-ebooks.info

- The **search navigator** : This is used for searching and optionally replacing the content of your data, returning any matches to the **Content window**. You can search your project by typing what you would like to find in the search field at the top of the content area. If you select anything in the **Content window**, the **Editor area** will take you to the line of content matching your search. The filter bar changes so you can filter your results to show only specific filenames.
- The **issue navigator** : This shows a list of active errors and warnings in the **Content window**. If you select an item from the list of errors, the editor will identify the item with the issue and if appropriate will take you to the line of code with the problem. Your **Filter bar** changes to show only items from the current build, to show only errors, or to display specific content.
- The **debug navigator** : When in debug mode, this will show a list of the running threads and the related stack information. When an item is selected in the **Content window**, information about the item is displayed in the **Editor** and **Debugger** areas. The **Filter bar** changes to show a **Thread Filter** button, which only shows relevant threads and a **Call Stack Slider**, which shows a list of stack frames. You can adjust the slider to show the most relevant to all of the stacks.
- The **breakpoint navigator** : This displays a list of active breakpoints in the **Content window**. The **Filter bar** lets you add or delete specific breakpoints, show only active breakpoints, or filter by specific matching content.
- The **log navigator** : This shows an active history of your tasks and session events that have happened during your development process. When an item is selected in the **Content window**, the matching log displays in the **Editor area**. You can filter content by most recent or by specific names using the **Filter bar**.

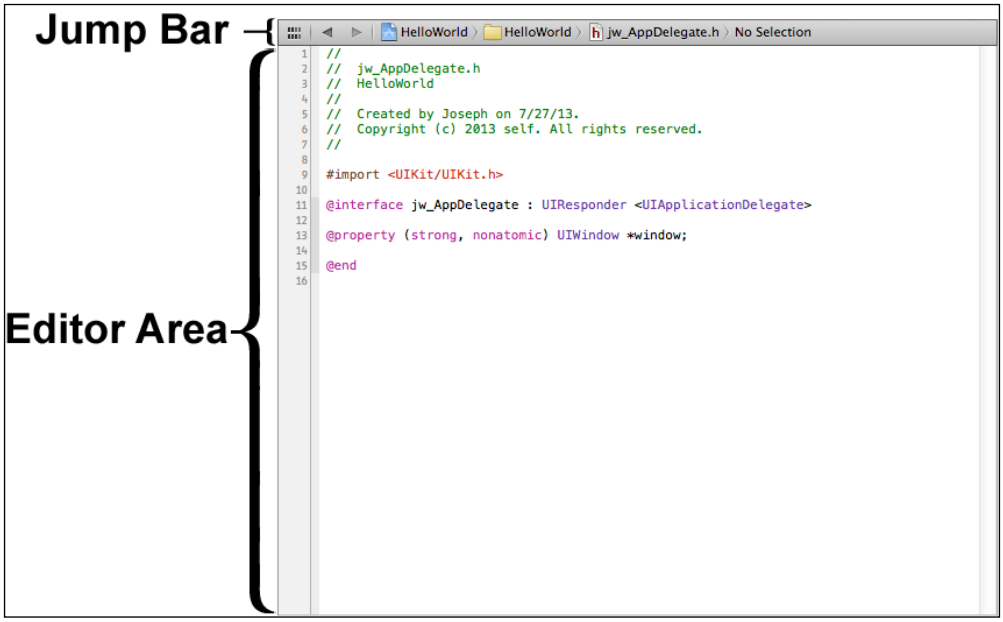
The **Content window** is the area where your data is displayed based on which navigation button you have selected. Typically, if you select an item in this area, the file's data will be displayed for modification in the **Editor area** of the workspace window. For specifics on what is displayed in the **Content window**, see the corresponding navigation button listed previously.

www.it-ebooks.info

The **Filter bar** reduces the amount of information returned to the **Content window**. The options available to the **Filter bar** change based on the navigation button that is selected. For details on which options become available, see the corresponding navigation button information listed previously.

The Editor area

The **Editor area** is where most of your development will take place. The type of editor window can be changed in the toolbar using the **Editor** buttons (see the screenshot of the toolbar). For now, we are going to go through the standard editor window:



No matter which type of editor you are using, you will have access to a jump bar. There are three items available in the **Jump Bar**; they are as follows:

- The Related items button lists additional selections relevant to your currently opened file.
- The Next and Previous history buttons are used to jump back and forth through the files you have opened. If you press and hold either of the history buttons, a drop-down list will appear with a list of your entire history.

www.it-ebooks.info

- The hierarchy list is used to navigate to the currently opened file:

The type of editor that will be displayed will depend on what you have selected in the **Content window**:

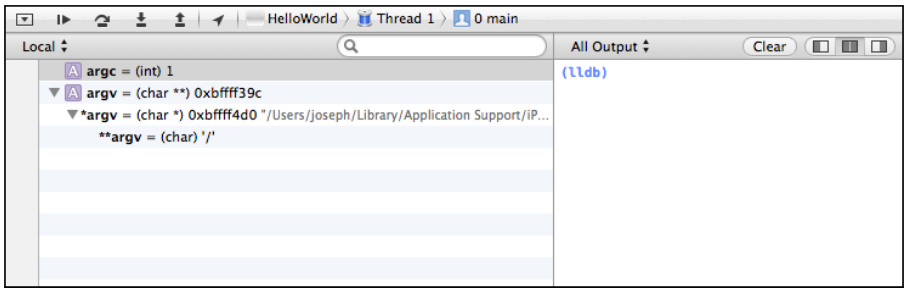
- **The source editor**: This enables the user to write and edit source code, set and facilitate or stop breakpoints, as well as enable the user to implement the control program.
- **The project editor**: This enables the user to view and edit project and target settings such as build options.
- **The property list editor**: This enables the user to view and edit a multitude of types of property lists or plists, which store a series of objects.
- **The rich text editor**: This enables the user to open, create, and edit rich text (`.rtf`) files similar to TextEdit.
- **The Core Data model editor**: This enables the user to execute and/or adjust a Core Data copy.
- **The mapping model editor**: This enables the user to graphically create and edit a mapping between an old Core Data model and a new Core Data model.
- **The interface builder**: This enables the user to graphically generate and/or edit user interface files.
- **The script editor**: This enables the user to create and/or edit AppleScript script files.
- **The scripting dictionary editor**: This enables the user to create and edit the scripting definition (`.sdef`) file for your application.

The Debugger area

The **Debugger area** of the workspace window is turned off by default, but it can be turned on or off using the middle **View** button on the toolbar. The **Debugger area** is used for examining detailed information about your application while it is running. The buttons at the top-left corner of this area are for stepping line-by-line through your code and for viewing what is happening in your code as it runs. The three buttons on the right-hand side of the top bar control what is displayed in the **Content window** of the **Debugger area**. The left button is to show your variables, the right is for the console only, and the middle button shows both. We will discuss how this works in greater detail in the debugging section of the book.

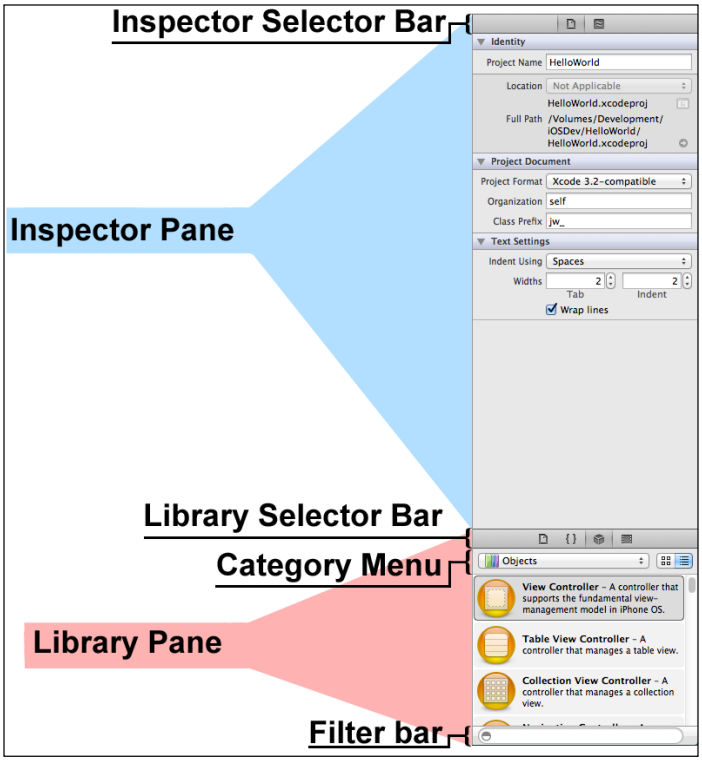
www.it-ebooks.info

The **Debugger area** can be seen in the following screenshot:



The Utility area

The **Utility area** of the workspace window, sometimes referred to as the **Inspector area**, can be turned on or off using the rightmost **View** button on the toolbar. The **Utility area** displays specific details for what is shown in the **Editor area**. Depending on what is in the **Editor area**, the Utility area could be limited to information about the file itself, such as size and date created, or it could end up offering multiple types of utilities for the data. The Utility area can be seen in the following screenshot:



www.it-ebooks.info

The **Inspector selector bar** lets you change the content in the **Inspector pane**. There are several options that will appear in the bar depending on what kind of content is in the **Editor area**. The file inspector and quick help will always be available, but the rest will appear or hide depending on what is needed.

The following are items you will find on the **Inspector selector bar**:

- **The file inspector** : This shows all of the metadata that is available for the selected file. This may be the name and file information or it could include subcategorized information associated with the file. We will describe these items as necessary throughout the book, but for now, just know that the information can change.
- **Quick help** : This will display the declaration, description, platform, and operating system (OS) availability, as well as where it is declared, reference information, guides, or sample locations.

The following is a list of inspectors that may be available in some editors:

- The **identity inspector** : This enables the user to view and manage metadata for an object, such as its class name.
- **The attributes inspector** : This enables the user to configure attributes of the selected object. The attributes available are directly related to the selected object.
- **The size inspector** : This enables the user to assign characteristics, such as autosizing, for an object.
- The **connections inspector** : This enables the user to view the outlets and actions for an object, make new connections, and sever existing connections.
- The **bindings inspector** : This enables the user to generate, view, and/or configure bindings for view objects.
- The **effects inspector** : This enables the user to identify settings for animation and other visual characteristics of a chosen object.

www.it-ebooks.info

Use the library pane to access libraries of resources that are ready to use in your project:

- **File templates** : These are templates for general types of files and code constructs. To use a template, drag it from the library to the project navigator.
- **Code snippets** : These are frequently used, short pieces of source code for use in your software. To use one, drag the snippet into your source file.
- **Objects** : These are interface objects. To use them, drag them directly into your nib file in the interface builder editor window.
- **Media files** : These comprise of graphics, icons, and sound files. To use them, drag them directly to your nib file in the Interface Builder editor window.

The iOS Simulator

The iOS Simulator is a standalone app that imitates everything about iOS devices. It saves you a tremendous amount of time by allowing you to run and test your code in a software-based environment before releasing it on the hardware. The iOS Simulator can simulate iPhone and iPad using different versions of the operating system.

You can access the iOS Simulator by going to **Xcode | Open Developer Tool | iOS Simulator**. A second way to access the simulator is by running an iOS product and selecting one of the simulator schemes; in that way, the iOS Simulator will automatically launch.

The **Hardware** menu lets you access the simulated hardware and modify it to better suit your needs:

- **Rotate left / Rotate right**: These rotate the simulator window to the left or right.
- **Shake Gesture**: This simulates a shake on your device.
- **Home**: This simulates the Home button on your iOS device.
- **Lock**: This will lock the simulator and take you to the slide to unlock the screen that you would normally see on your iOS device.
- **Simulate Memory Warning**: This will send the active app a low memory warning.

www.it-ebooks.info

- **Toggle In-Call Status Bar**: This brings up the flashing green **Active Call** bar and replaces the standard status bar. This is very useful to see how your app's user interface will react while a phone call is running.
- **Simulate Hardware Keyboard**: This turns on/off the iOS keyboard.
- **TV Out**: This simulates a TV out signal from the simulator.

To simulate a touch or gesture, you can use the following mouse clicks and motions:

- **Tap**: This is achieved by a mouse click
- **Touch and hold**: Hold the mouse button
- **Double tap**: This is acheived by double-clicking a mouse button
- **Swipe**: Click, drag, and release your mouse
- **Flick**: Click, drag, and release your mouse
- **Drag**: Click, hold your mouse button down, and release at your final position
- **Pinch**: Continuously hold the *option* key down on your keyboard, move the finger position in circles by moving your mouse, hold the *shift* key on your keyboard to move the center point of the pinch, and click-and-drag to simulate the pinch

www.it-ebooks.info

Summary

In this chapter, we covered how to navigate Apple's website to become an Apple developer, download and install Xcode, and the basics of working your way around the Xcode IDE. Everything you learned in this chapter will be referenced throughout the book and you will most likely find yourself back here trying to find out where something in Xcode is located.

In our next chapter, we will introduce you to programming in Xcode and learning the Objective-C language. We will be covering the basics on how to program as well as details on object-oriented programming.

www.it-ebooks.info

2
Programming and Objective-C

In this chapter, we will cover the basics of programming and the Objective-C language. If you already know how to program or have programmed in the past, this will be a great refresher for general programming concepts. We will also cover the basics of object-oriented programming and Objective-C.

In this chapter, we will cover the following topics:

- Understanding Objective-C
- Programming basics
- Objective-oriented programming
- Hello World!
- Programming with Objective-C
- Model-view-controller

Understanding Objective-C

Every computer processor uses a machine code, and iPhone is no different. So that people like you and me can write a program to run in these environments, extremely brilliant people have created easy-to-use software, which incorporates universal languages. Programmers can then learn this universal language and utilize it on multiple machines. Of course, there is a long and glorious history of these brilliant people doing this, but we will not cover this historical phenomenon in this book. Sorry! Suffice to say, Objective-C is a combination of two of these really brilliant programming languages: the **C language**, as it is universally recognized and powerful, and **SmallTalk** for its ease of use. In fact, since Objective-C is a superset of C, anything you program in C, you can use in Objective-C.

www.it-ebooks.info

Objective-C was the primary language behind NeXTSTEP's OPENSTEP operating system. Apple purchased NeXT, inc. in 1996 and used NeXT's OPENSTEP operating system as the core for Apple's operating system (OS X). This made Objective-C the core programming language behind OS X and in turn iOS. Fundamentally, this is how you talk to the iOS Simulator or iOS devices, as it is a derivative of multiple programming languages, most notably the C language and SmallTalk.

To write an Objective-C program for iPhone, there is a specific process to follow. All of the code you write will need to abide by the rules of this language, first entering it into an ACSII text file and then feeding that text file into Xcode. Once your text file is fed into Xcode, it will be compiled into an application (or app) that your iPhone can then read and play back. Pretty simple, huh?

Programming basics

Every programming language uses specific ways of storing and distributing information through the code. These examples translate to nearly every programming language and are not necessarily restricted to the Objective-C language. All of this information is entered into a standard text file and is read in with the compiler. Xcode has its own standard text editor which is built into the IDE that actually helps you type in the correct code. Let's begin by talking about the different types of data used in code.

Basic data types

Everything in your software is made up of various types of data such as text and numbers; these are conveniently called **data types**. These data types are stored in containers called **variables**. You can assign data to a variable and retrieve it in other areas of your program. Objective-C has expanded most of these data types to make them more useful. When defining the data types in the next three sections, I included how you would typically see these data types written in code within parentheses.

Text data types

Text data types are used for single characters or words, for example, player names or sentences and paragraphs for the help sections and character descriptions:

- **Character** (`char`): This is a single character and is used very rarely, but it is still important for programming. A character can represent any letter, number, common punctuation mark, nonprintable characters (such as a carriage return), or white space, for example, `1`, `a`, `A`, `$`, or `" "`.
- **String** (`NSString`): A string is one or more characters strung together, for example, `This is a string of characters.` or `I`.

www.it-ebooks.info

Numeric data types

In general, numeric data types are just numbers. The details on what kind of number determines the type. Integers are usually used to increment and count, including player's scores. Floats and doubles will be used for more precise things such as a player's position, measurements, and currency. The numeric data types are as follows:

- **Integer** (`int`): Integers are whole numbers or any number without a fractional part, for example, `4`, `35`, `103`, or `1234`.
- **Float** (`float`): These are floating-point numbers or numbers with a fractional part, for example, `1.23`, `5.12345`, `123.456`, or `99.999`.
- **Double** (`double`): Basically, this is the same type of number as a float; however, a double allows you to use double-precision floating-point numbers. A double can be interpreted as an integer as long as it doesn't have a fractional point, so 3.0 would be used as though it were 3, for example, `1.7320508`, `2.0`, or `3.14159`.

Other data types

The following is a list of other data types:

- **Boolean** (`bool`): A Boolean data type is a representation of either true or false. It always has to be one or the other and can never be anything else. Boolean's are great for quick comparisons, for example, `bool playerIsAlive = true;` or `if(playerIsAlive == false) NSLog(@"Your Player has unexpectedly expired and gone to meet his maker!");`.
- **Date** (`date`): The `date` data type is used for calculating information based on time.
- **Array** (`array`): There are several different types of arrays, but at its core, an array is just a list of other variables. Arrays let you store, organize, and recall information very quickly and efficiently. If you had an array with elements `a`, `b`, `c`, `d`, and `e`, you could send this to another area of your code and then get specific items from the array. Although it can be confusing at times, it is important to remember that when talking about items in an array, the list counts from `0` (zero). Using the previous example, if you called item `0` from your array, it would return the letter `a`, or item `3` would return `d`. You can also use an array to store lists of objects. In the next chapter, we will use an array to store image objects that will be used as our game pieces. There are several other types of arrays, but we will discuss these later in the book.

www.it-ebooks.info

Later in the chapter, we will discuss a few data types that are specific to Objective-C as well as other data types that we will discuss later in the book. For now, that should cover the basics and help you begin programming right away.

Variables

A variable is simply a name that you can assign to an object. The name you want to assign is up to you, but you must follow some basic steps when using variables in programming. There are three vital steps when using data in variables: **declaration**, **assignment**, and **use**. When you decide that you want to use a variable, you must first figure out the type of variable you want to use and declare it as a variable with that type. This is very important because once you set the type of a variable, you cannot change it. **Type casting** allows you to disguise a variable as a different type, but we will discuss this later. The following is an example of declaration: if I decided to set a variable to describe how many enemies I have in my game, I would define it as an integer and give it a variable name, `enemyCount`. This is what the code would look like:

```
int enemyCount;
```

Now that you have a declaration saying that you have a variable named `enemyCount`, you can now assign an integer to that variable. Use the `=` sign or **assignment operator** to assign a value to the variable:

```
enemyCount = 24;
```

As you can get errors when you try to get a value from a variable before it has been assigned, it is important to assign a value to the variable when you define it, as shown in the following code:

```
int enemyCount = 24;
```

We are going to show an example of **use** in the control statements that are discussed later. **Use** is where we can examine, change, and compare variable data.

You may have noticed that we used `;` at the end of our lines in the example code. The semicolon is used to tell the compiler that we are at the end of our statement. You could insert several semicolons and Objective-C would just think they were empty statements.

www.it-ebooks.info

Constants

A **constant** is very similar to a variable with one exception; once declared a constant, it cannot be changed. Get it? It's constant... oh no, bad pun. Anyway, you initialize a constant by adding the `const` keyword before declaring the variable type and name. The following is an example of the `const` keyword:

```
const int constantForceX = 10;
```

Now, if you tried something funny like changing a constant somewhere else in your code, you would get an error such as `Read-only variable is not assignable`:

```
const int constantForceX = 10;
constantForceX = 30;
```

Comments

If you wanted to put something into your code that you didn't want the code to compile, you can write **comments** into your code. In order to define something as a comment, you can do it in one of two ways.

In the first type, if you want to comment one line at a time, you would use `//` at the beginning of your statement to tell the compiler to *ignore* this. This is a great way to leave a note for yourself or anyone else who may be reading your code later. The following are all the conventional examples:

```
// This is a single line comment
// Ignore this line as well
int enemyCount = 24;   // You can put a comment after a semicolon
// Although it is not standard practice.
int groupCount = 25;
//////////////////////////////////
//        Comment Area          //
//////////////////////////////////
bool fullGroup = false;

//--------------------
// Yes this works too
//--------------------
```

You might use the `//` single-line comment to block off large sections of code; however, this is not recommended because it gets confusing and could end up being messy for anyone who has to read through your code.

www.it-ebooks.info

The second type of comment in Objective-C is the **block** comment. You can block off large sections of code by using the `/* */` block comment. The way it works is you put `/*` at the beginning of your comment area and `*/` at the end of the comment area. An example of this is as follows:

```
/* All of the lines before
   The enemyCount variable will be ignored
   Because we have a block comment
   Keeping them from being read
*/
int enemyCount = 24; /* Yes this works as well */
int groupCount = 25;
/* But be very careful with this
   Since you could end up commenting out
   A section of your code by accident

bool fullGroup = false;

   Do you see what I mean */
```

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.

Although all of these examples will work in your code, everyone comments their code differently. You can comment your code however you like, but you may want to check with some of the people who will be reading your code and ask them what some of the standards they use are. After all, they are probably the ones who will bail you out when something goes wrong.

Here is one more thing to consider about commenting your code. You should always do it! If not for other people, do it for yourself. At some point in the future, usually long after you originally wrote your code, you will need to go back and look at what you did. Unless you commented your code, you will end up trolling through thousands (not an exaggeration) of lines of code to try and figure out what you did. And if one day you have to read through someone else's code on which they have done a poor job of commenting it, you will curse them the entire time you are trudging through it. There are few things worse than digging through code and complaining about it only to find out you were the one who didn't comment it.

www.it-ebooks.info

Operators

Operators let you perform different procedures on your variables. In this section, we will go over some of the basic operators you will be working with throughout the next few chapters. You will find a full list of the Objective-C operators and their functions at the back of the book. A few examples are supplied throughout the book; most of which will be either very obvious or covered when needed. Keep in mind that standard mathematics order of operation rules apply whenever you are using multiple operators.

Arithmetic operators

Arithmetic operators are pretty much what you learned in grade school. They take two operands and apply the operator to give a result. The following table consists of arithmetic operators:

Operator	Description
`+`	Addition
`-`	Subtraction
`*`	Multiplication
`/`	Division
`%`	Modulo

The following is an example for arithmetic operators:

```
int enemyCount = 24;
int x = 0;

x = enemyCount + 10;
//x is now equal to 34
```

You could also use multiple operators in your expression as follows:

```
x = enemyCount + (10 * 3);
//x is now equal to 54
```

Comparison operators

Comparison operators are used when comparing two operators, returning `true` or `false` based on the results. The following table consists of comparison operators:

Operator	Description
`==`	Equal to
`!=`	Not equal to

www.it-ebooks.info

Operator	Description
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to

An example for comparison operators is as follows:

```
bool fullGroup;
int enemyCount = 24;

fullGroup = (enemyCount == 24);
//fullGroup is now set to True because enemyCount is equal to 24
```

Logical operators

Logical operators are also set up to return a Boolean answer; you will see these throughout the next few chapters. The following table consists of logical operators:

Operator	Description
!	NOT
&&	Logical AND
||	Logical OR

An example for logical operators is as follows:

```
bool fullGroup = true;
bool emptyGroup;
bool logicalAndGroup;
bool logicalOrGroup;


emptyGroup = !fullGroup;
// emptyGroup is now set to false which is what fullGroup is not.

logicalAndGroup = (emptyGroup == true && fullGroup == false);
// logicalAndGroup is now set to true since
// emptyGroup is equal to true AND fullGroup is equal to false

logicalOrGroup = (emptyGroup == true || fullGroup == true);
// logicalOrGroup is now set to true since either
// emptyGroup is equal to true OR fullGroup is equal to false
```


www.it-ebooks.info

Increment and decrement operators

Increment and **decrement** operators will either add or subtract one from your variable. The following table consists of increment and decrement operators:

Operator	Description
++	Increment by 1
--	Decrement by 1

An example for increment and decrement operators is as follows:

```
int x = 10;

x++;
//x is now equal to 11

x--;
//x is now equal to 10
```

Compound operators

Compound operators are designed to replace statements where an operator takes a variable as one of its arguments and then deposits the results back into the same variable. The following table consists of compound operators:

Operator	Description
==	Equal to
!=	Not equal to
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to

The following is an example for compound operators:

```
int x = 10;
int y = 3;

y *= x;
// take y and multiply it by x (or 10) then replace y with the
returned value
//y is now equal to 30
```


www.it-ebooks.info

Control statements

Control statements are a way to check different conditions in your code. Various control statements are described in this chapter.

The if statement

Continuing the enemy group example from the previous section, let's say we wanted to find out if we had enough enemies to make a full group. Let's create a variable named `enemyCount`, which is the number of enemies that we currently possess. A second variable is the number of enemies in the group called `GroupCount`. We can then set a control statement that checks if we have enough to form a full group. An `if` statement is exactly what it seems to be; it tests a chunk of code to see if something is true. If the code is true, it acts on it; likewise, if it's not, it ignores everything in the curly braces (see the example). Let's take a look at the following `if` statement:

```
int enemyCount = 24;
int GroupCount = 25;
bool fullGroup = false;

if(enemyCount >= GroupCount)
{
   // Do something here
   fullGroup = true;
}
```

Essentially, the code expresses that we have 24 enemies, but we know that a group has 25 enemies. We are going to start by assuming we don't have a full group of enemies. If the number of enemies is greater than or equal to a group of enemies, we set our `fullGroup` variable to `true`; otherwise, we move on.

The if…else statement

Our next control statement is an `if...else` statement. This is used if a condition is true, but you also want to check to see if another condition is false. An example for an `if...else` statement is as follows:

```
int enemyCount = 24;
int GroupCount = 25;
bool fullGroup;

if(enemyCount >= GroupCount)
{
   // Do something here
   fullGroup = true;
```


www.it-ebooks.info

```
}
else
{
   // Do something else here
   fullGroup = false;
}
```

The code basically does the same thing as the previous code, yet with the `if…else` statement, if my number of enemies is greater than or equal to a group of enemies, we will set our `fullGroup` variable to `true`; otherwise, we set `fullGroup` to `false`.

The if…else if statement

Our last `if` control statement is an `if...else if` statement. Basically, it lets you check a specific condition and then check against another condition. You can wrap it up with an `else` statement to catch any remaining conditions. You can have as many `else if` statements as you like, but it is going to get messy after a while. An example for an `if...else if` statement is as follows:

```
int enemyCount = 3;
int enemyPointValue = 0;

if(enemyCount == 1)
{
   // Do something here
   enemyPointValue = 1;
}
else if (enemyCount == 2)
{
   // Do something else here
   enemyPointValue = 10;
}
else
{
   // Do something else here
   enemyPointValue = 100;
}
```

The switch statement

This brings us to the `switch` statement. The `switch` statement is similar to an `if...else if` statement but much more organized and manageable. The `switch` statements are structured a bit differently; only when you need one will you really appreciate its strengths.

www.it-ebooks.info

The structure of a `switch` statement starts with an expression that supplies a value. Following the expression is a series of `case` statements that will be compared against the expression. It is important to note that the data type of the expression's value must match the data type of the `case` statement that we are comparing it to and the code we want to run for that value. We close the `case` statement with a `break` statement; in order to break out of the `case` statement. In the following example, `value1` does not have a `break` statement, so it will continue to execute statements of `value2` and then break out of the `switch` statement. Next, we follow up with the `default` statement, which is equivalent to the `if...else` statements; it catches anything that was not picked up by the prior `case` statements. An example for the `switch...case` statement is as follows:

```
switch (expression)
 {
     case value1:
          statements
     case value2:
          statements
          break;
     case value3:
          statements
          break;
     default:
          statements
          break;
 }
```

In the following example, we have set our `enemyCount` integer to `1`, and then we check it against three different `case` statements. If `enemyCount` had been set to `0`, `enemyPointValue` would have been set to `2`, and then it would break out of the `switch` statement all together. But as `enemyCount` is set to `1`, we set `enemyPointValue` to `35` and increment it in the next statement as there is no break to force us out of the `case` statement. In general, it is not recommended to eliminate the `break` statement in your code as it makes it very difficult to debug later and can be very confusing.

```
int enemyCount = 1;
int enemyPointValue = 0;

switch (enemyCount)
{
   case 0:
      enemyPointValue = 2;
      break;
   case 1:
```


www.it-ebooks.info

```
        enemyPointValue = 35;
    case 2:
        enemyPointValue++;
        break;
    default:
        enemyPointValue = 10;
        break;
}
// enemyPointValue is now = 36 since it was
// set to 35 in case 1 and then incremented in case 2
// then it breaks out before it can be set to 10
```

Loops

In programming, **loops** are second in importance only to control statements. Loops allow you to loop through your code and execute its contents multiple times. You are going to require a lot of repetitive tasks when you are programming, and this is the way in which to do it. You can loop in three basic ways: `for`, `while`, and `do while`. All three loop through your code in completely different ways.

The for loop

Let's start with the `for` loop, as this is probably the most used example of a loop. The `for` loop needs three different things in order to work, and all are placed in parentheses and separated by semicolons. You place your code inside curly braces. First, you will need an **initializer** to tell the loop where to start. Second, you need the **conditional statement**, which tells the loop in what condition you wish to stop the loop. The third piece is the **increment/decrement expression**; each time the loop repeats, it will execute your increment or decrement expression. An example for the `for` loop is as follows:

```
for ( initializer; conditional; increment/decrement )
{
      // loop in here
}
```

In the following example, we have two integer variables, `tileCount` and `tileID`, and `tileID` is set to a value of `9`. Then, in our `for` loop, we start by setting `tileID` to `0`. We are going to loop until `tileID` is greater than or equal to `tileCount` (or `9`), and every time we loop through, we are going to increment `tileID` by `1`, as shown in the following code snippet:

```
int tileCount = 9;
int tileID;
```


www.it-ebooks.info

```
for (tileID = 0; tileID >= tileCount; tileID++){
   // tileID will start off at zero
   // but will increment by 1
   // 9 times then it will exit the loop
}

// tileID is now equal to 9
```

The while loop

The `for` loop will not always be the most efficient way of handling your code. It is possible that you will run into situations where you will need to loop in an area until a certain condition is met and when you don't know how many times you will need to repeat. This is where the `while` loop comes in handy. In a `while` loop, the code will keep looping inside your curly braces until your condition is met, as shown in the following code snippet:

```
while (condition)
{
      // loop in here
}
```

In this example, we are going to set a condition to check if our `tileCount` variable is greater than `10`. Inside our code, we are going to increment `tileCount` by `1` each time we loop. Once `tileCount` reaches `10`, we will exit the loop, as shown in the following code snippet:

```
int tileCount = 0;

while (tileCount < 10)
{
      // loop in here
   tileCount++;
}
// tileCount is now equal to 10
```

The do while loop

Even though while loops are very helpful, they will always check the condition before running the code in the curly braces. Suppose you want to check the condition after the code has run; this is where the `do while` loop comes in, as shown in the following code snippet:

```
do
{
       // loop in here
} while ( condition )
```


www.it-ebooks.info

In the following example, we are going to do exactly the same thing we did in the `tileCount` loop in the previous example, but we are going to use a `do while` loop instead to show the difference. First, we set a condition to check if our `i` variable is greater than or equal to `10`. Inside the code, we are going to increment `i` by `1` each time we loop. Once `i` reaches `10`, we will exit the loop, as shown in the following code snippet:

```
int tileCount = 0;

do
{
      // loop in here
   tileCount++;
} while (tileCount < 10)

// tileCount is now equal to 10
```

The break statement

You may run into situations where you want your loop to stop running under a different circumstance than your loop condition. In those situations, you would use a `break` statement. In this example, we compare `x` and `y` after incrementing `x` to see if they are equal. If they are, we exit the `while` loop. The code for the `break` statement is shown as follows:

```
int x = 0;
int y = 10;

while (x < 30)
{
   x++;
   if (x == y){
      break;
   }
}
//x is now equal to y
```

The continue statement

While the `break` statement will take you completely out of the loop, the `continue` statement will allow you to stay in your loop but not execute code following the statement, as shown in the following code snippet:

```
int i = 9;
int x;
```


www.it-ebooks.info

```
for (x = 0; x >= 10; x++){
   //x is getting incremented each loop
   if(x > 3){
      continue;
   }
   i++;
}

// x is now equal to 10 but i is only equal to 3
// because we told the code to continue the loop
// from that point if x is greater than 3
```

Of course, we can't cover everything in this section of the book, but this should give you a good leg up.

Functions

Let's say we were writing several very similar lines of code over and over again in our code. Rather than rewriting the same chunk of code every time, we can write a function to do it for us, and then we can just call one line of code each time.

Functions conform to the following format:

```
<returned type> <name of function> (<arg_1 type> <arg_1 name>,
   <arg_2 type> < arg_2 name>, < arg_n type> < arg_n name>, ...)
```

Let's say we want to set up a chunk of code to be called each time a projectile hits our enemy characters. Sometimes we use a laser, and it takes 25 points off, and sometimes it is a bullet, which takes 10 points off. A function is then written that would accept a different variable each time but would still remove that amount from the enemies' total hit points. Let's take a look:

```
int hitPoints = 100;
int timesHit = 0;

bool adjustHitPoints(int amount)
{
   bool playerIsDead = false;
   hitPoints -= amount;
   timesHit++;
   if(hitPoints <= 0)
   {
      playerIsDead = true;
   }
   return playerIsDead; //
}
```


www.it-ebooks.info

In this example, we would call it from another piece of code as follows:

```
bool isDead = adjustHitPoints(20);
```

As you can see, the returned data type is `bool`; that means your function will always have to return a `true` or `false` to whatever called it. If it doesn't, you will get errors in your code and you probably will not be able to compile. The next field is the function name, and it is separated by a space; the name is `adjustHitPoints`. The function name can be anything you like, but it must be one word; usually, your code standards dictate how the name should be formatted. For our purposes, we will always start the name off with a lower case letter and capitalize each word's first letter. Next, your arguments are encapsulated in parentheses. **Arguments** are any information you would like to pass on to your function preceded by your argument's type. If you don't have any arguments, you should have empty parentheses `()`. In our example, we only have one argument, but there is no limit to the number of arguments you can send in, although you should keep it below three if at all possible. From there, all of your code is contained in curly braces.

Variable scope

In our `adjustHitPoints` example, you may have noticed that we declared a new Boolean variable inside our function named `playerIsDead`. Where a variable is declared determines that variables' **scope**, and it can't be read outside of its scope. There are basically two different scope rules you should know.

The first is a global variable, which is when your variable is defined anywhere in your file outside a function like our early examples of the `hitPoints` and `timesHit` variables. These are stored and can be accessed in any function anywhere in your code and are always in memory while your program is running.

The second type is a local variable. A local variable gets defined inside a set of curly braces and can be accessed in the block of code below where they were declared, or any of the nested areas of the code below where it was declared. Once your block of code has run, your local variables are removed from memory and are no longer available anywhere in your code. The following code demonstrates variable scope:

```
// These are global variables and can be accessed
// in main or adjustHitPoints
int hitPoints = 100;

int main()
{
   // laserDamage is only available in main but
   // We pass its value to adjustHitPoints as an argument
```


www.it-ebooks.info

```
    int laserDamage = 20;
    adjustHitPoints(laserDamage);

    laserDamage = 10;
    adjustHitPoints(laserDamage);
    adjustHitPoints(laserDamage);

    laserDamage = 50;
    adjustHitPoints(laserDamage);
}

void adjustHitPoints(int amount)
{
    // testNumber is only available in adjustHitPoints
    int testNumber = 0;
    testNumber += 5;

    //we are modifying the hitPoints variable
    hitPoints -= amount;

    printf ("values are %d and %d, \n",  hitPoints, testNumber);
}
```

The previous code would print out as follows:

```
values are 80 and 5
values are 70 and 5
values are 10 and 5
```

Notice how `testNumber` gets refreshed each time `adjustHitPoints` is called, but `hitPoints` stays in memory and continues to get updated.

Object-oriented programming

So far, we have covered code elements that are in nearly every programming language. Now we are going to take a little time to explain **object-oriented programming** (**OOP**) languages. In order to understand an OOP, it is important to know what an object is in terms of code. Now, there are entire volumes of books dedicated to explaining what OOP is and its function, but as you bought this book for other reasons, we won't address it here.

www.it-ebooks.info

Objects

I have heard an **object** explained as poorly as simply is anything, or explained over pages and pages and described in such minute detail that anyone would eventually just give up on the whole idea. Basically, an object is a group of code that is self-contained and can be reused over and over again. There are complex notions with objects, such as, you can put whatever code you want in an object and you can even make sections of your code available to other objects, or write sections of the code that can only be accessible inside the object. However, in its fundamental form, it is just a group of reusable code.

Classes

A **class** is an object's container, or at the very least, an outline for what your object will look like once it is instantiated. Most of the time, we create a class in its own text file (although it is not mandatory). In that file, we create functions (or methods) and variables and we define how they can be accessed outside the class. Within functions is where all of our code exists. A class keeps all of this organized for us, so when we have code in another section of our program that needs information from our class, we know exactly what to ask for and how to ask for it. Very exciting stuff, huh?

Whenever you create a class and store it, you are creating what is called an **instance** of a class. This becomes very important when differentiating how code is called in your class.

Objective-C has a standard base class named `NSObject`, which we use to derive most of our classes. This is a very powerful parent class, and we will be discussing its uses throughout the book.

Methods

A class can contain groups of code that can perform specific tasks in relation to your class. These are referred to as methods. **Methods** are very similar to functions with the exception that a method is defined inside a class and has a few distinct differences.

There are two different types of methods in Objective-C: class methods and instance methods. An **instance** method will run code on one specific instance of a class you have created and can pick up data particular to one instance. A **class** method however is a set of code that is shared with all instances of a class but still cannot access instance-specific methods or data.

www.it-ebooks.info

Methods can vary drastically depending on the language you are using, so I am just going to explain how methods are set up using Objective-C. When we are declaring methods in Objective-C, we differentiate our methods by preceding them with a plus (+) sign for class methods or a minus (-) sign for instance methods, as shown in the following code snippet:

```
+(int) myClassMethod;
-(int) myInstanceMethod;
```

Pointers

Pointers are one of the more difficult things you are going to have to learn about, simply because they can be so incredibly confusing.

A pointer *points* to a place in memory where an object is stored. When we want to retrieve the object and place it into a variable, we preface our variable with an asterisk (`*`). So, if we create a custom class named `MyCustomClass` and we wanted to store an instance of that class in a variable, we would type it like the following:

```
MyCustomClass *myClassInstance;
```

The primitive variables we have used so far store the variable's data directly in memory, but the `myClassInstance` variable just points to a place in memory where the object is stored.

Now, in order to use the `myClassInstance` variable, we have to initialize it and allocate its own memory. This is done by first sending an `alloc` message to the object class and then sending an `init` message to the returned allocation. This is what it looks like in code:

```
MyCustomClass *myClassInstance;
myClassInstance = [[MyCustomClass alloc] init];
```

Notice that once the variable has been declared, we don't need to use the asterisk. That is because our code knows that we were only using that as a reference. Now that we have our own instance, we can access the class' data through the new variable.

If we wanted to call a method in our new instance, we could use the following code:

```
[MyCustomClass myMethodName];
```

If we wanted to have an integer as a parameter to our method, it would look like the following code snippet:

```
myInteger = 3;
[MyCustomClass myMethodName:myInteger];
```


www.it-ebooks.info

Objective-C data types

As you look through the code in Objective-C, you will see several data types that are preceded by the characters `NS`. These are a part of the Foundation Frameworks classes and were added into Objective-C to provide some basic functionality that was not provided by the basic Objective-C language. I am not going to cover them all here, because at the time of this writing, there were almost 200 classes in Foundation Frameworks, but I will cover a few of the basics that we will use throughout the book.

NSString

`NSString` is an immutable object, meaning that like a `const` variable, once you declare it, you cannot change it. It has several methods that make it much better than just using a standard string variable.

To declare an `NSString`, you start with the object type followed by a pointer and variable name, then you can set the string using the literal `@"string contents"` syntax:

```
NSString *myString = @"string contents";
```

`NSString` provides support for Unicode and can contain any UTF-8 characters, which means you can have your code look something more like this:

```
NSString *myString = @"Au contraire, mon frère. The answer is 42!";
```

You can define an `NSString` with a preformatted string:

```
  NSString *myFrenchString = @"mon frère";
  int answer = 42;
  NSString *myString = [NSString stringWithFormat: @"Au contraire, %@.
The answer is @d!", myFrenchString, answer];
```

You can compare one `NSString` with another using the following `isEqualToString` method:

```
  NSString *myString = @"Practicalities";
  if ([myString isEqualToString:@"Practicalities"]) {
    NSLog(@"Very Practical");
  }
```

You can append one string to another, replace substrings, and pull substrings from `NSString`. Keep in mind that since `NSString` is immutable, you can only do this by creating a new variable, as shown in the following code:

```
NSString *myFrenchString = @"Au contraire, mon frère. ";
NSString *myAnswerString = @"The answer is 42!";
```


www.it-ebooks.info

```
NSString *myString = [myFrenchString stringByAppendingString:myAnswer
String];

//This will print "Au contraire, mon frère. The answer is 42!"
NSLog(@"%@", myString);

//This will print "Au contraire, My brother. The answer is 42!"
NSString *newString = [myString stringByReplacingCharactersInRange:
NSMakeRange(14, 9) withString:@"My brother"];
NSLog(@"%@", newString);

//This will print "Au contraire"
NSLog(@"%@", [myString substringToIndex:12]);
```

We always seem to find a reason to change the case of a string. The following is the code containing three examples of ways to change the case of your `NSString`:

```
NSString *myString = @"Au ContRaire, moN frère";
NSLog(@"%@", [myString lowercaseString]);//au contraire, mon frère
NSLog(@"%@", [myString uppercaseString]);//AU CONTRAIRE, MON FRÈRE
NSLog(@"%@", [myString capitalizedString]);//Au Contraire, Mon Frère
```

`NSString` has two methods for checking your strings. The `length` method returns the number of characters in `NSString`. The `characterAtIndex` returns an unsigned short value for the Unicode decimal of the character at the index provided, as shown in the following code snippet:

```
NSString *myString = @"Practicalities";
//These two lines will both print 14 out to your debug panel
NSLog(@"%lu", (unsigned long)myString.length);
NSLog(@"%lu", (unsigned long)[myString length]);

//This will print the number 80 then the letter "P"
NSLog(@"%hu", [myString characterAtIndex: 0]);
NSLog(@"%@", [NSString stringWithFormat:@"%C", [myString
characterAtIndex: 0]]);
//This will print the letter "c" from the word "Practicalities"
NSLog(@"%@", [NSString stringWithFormat:@"%C", [myString
characterAtIndex: 3]]);
```

NSMutableString

`NSMutableString` is very similar to `NSString` with the exception that `NSMutableString` can be modified. In fact, `NSMutableString` is derived from `NSString`, so all of the methods from `NSString` can be used with `NSMutableString`. This also means that whenever you append or grab a substring, you will not need to make a new variable.

www.it-ebooks.info

You can create a new `NSMutableString` by copying an `NSString` into your new `NSMutableString`, or you can define it with a capacity. If you define a string with a specific capacity, it will not restrict the number of characters in your string; it is just a hint at the number of characters for efficient data storage purposes. The following example demonstrates this:

```
NSMutableString *myString =
[NSMutableString stringWithString:@"Au Contraire"];
NSLog(@"%@", myString);

[myString setString:@"Au Contraire, Mon Frère"];
NSLog(@"%@", myString);

NSMutableString *myNewString =
[NSMutableString stringWithCapacity:25];
[myNewString setString:@"Au Contraire, Mon Frère"];
NSLog(@"%@",myNewString);
```

You can delete and change characters in `NSMutableString` similar to how we did it previously, but we will not need to create a new variable, as shown in the following code snippet:

```
NSMutableString *myString =
[NSMutableString stringWithString:@"Au Contraire, Mon Frère"];
[myString replaceCharactersInRange:
NSMakeRange(14, 9) withString:@"My brother"];
//This will print "Au contraire, My brother.
NSLog(@"%@", myString);

[myString deleteCharactersInRange:NSMakeRange(12, 12)];
//This will print "Au contraire
NSLog(@"%@", myString);
```

NSNumber

`NSNumber` is a generic number variable that can store most numbers: `signed char`, `unsigned char`, `short int`, `int`, `long int`, `long long int`, `float`, `double`, or `bool`. The `NSNumber` data type is immutable and cannot be changed once created. Declaring your variable is similar to the way we did it with `NSString`:

```
NSNumber *myInt = [NSNumber numberWithInt:1234567890];
// or
NSNumber *myInt = @1234567890;
```


www.it-ebooks.info

NSSet

The NSSet data type is an immutable unordered collection of Objective-C objects. You would use NSSet as an alternative to an array if you were only using Objective-C objects. They are not concerned with the order of the objects and whether performance is important when testing to see if an object is contained within the set. To declare NSSet, you can use either a predefined list of objects, or you can create a set from an existing array. Even though NSSet is immutable, the objects within it can be modified. The following is an example to demonstrate this:

```
NSSet *playerSet = [NSSet setWithObjects:
@"Player 1", @"Player 2", @"Player 3", nil];
// or
NSArray *opponentArray = @[@"Enemy 1", @"Enemy 2", @"Enemy 3"];
NSSet *enemySet = [NSSet setWithArray: opponentArray];
```

If you need to check NSSet for a specific object, you can use either containsObject or member. The NSSet data type is extremely fast at these operations and should be used if you are just checking objects against a predefined list, as shown in the following code snippets:

```
if ([playerSet containsObject:@"Player 4"]) {
  NSLog(@"Player 4 has entered the game");
}
// or
NSString *player = [playerSet member:@"Player 4"];
if (player != nil) {
  NSLog(@"%@ has entered the game!", player);
}
```

You can get the number of items in a set:

```
NSUInteger count = [playerSet count];
```

You can combine sets together using setByAddingObjectsFromSet to merge two NSSet variables into a new NSSet variable:

```
NSSet *allSets = [playerSet setByAddingObjectsFromSet:enemySet];
```

NSMutableSet

Similar to strings, NSSet has NSMutableSet that you can modify. To create NSMutableSet, we use the following code:

```
NSMutableSet *playerSet = [NSMutableSet setWithObjects:
@"Player 1", @"Player 2", @"Player 3", nil];
// or
NSMutableSet *enemySet = [NSMutableSet setWithCapacity:3];
```


www.it-ebooks.info

To add or remove objects from an NSMutableSet, we use the following code:

```
[playerSet addObject:@"Player 4"];
[playerSet removeObject:@"Player 3"];
```

NSArray

NSArray is not as fast at checking for objects as NSSet, but it is sortable and much more reliable in its ordering. NSArray is immutable but it does have an NSMutableArray counterpart, which we will discuss in the next section. There are two ways to define your arrays; the first is using the @[] syntax and is recommended. I included the arrayWithObjects definition as it was more prevalent in the older versions of Xcode:

```
NSArray *playerSet = @[@"Player 1", @"Player 2", @"Player 3"];
NSArray *enemySet = [NSArray arrayWithObjects:@"Enemy 1", @"Enemy 2",
@"Enemy 3", nil];
```

You can get the number of items in NSArray, which is similar to NSSet:

```
NSUInteger count = [enemySet count];
```

When checking NSArray for objects, you can use containsObject for confirmation or you can get the location of the object in NSArray using indexOfObject to get its position:

```
if ([playerSet containsObject:@"Player 4"]) {
  NSLog(@"Player 4 has entered the game");
}
NSUInteger playerLoc = [playerSet indexOfObject:@"Player 3"];
```

There are several ways to sort NSArray, but using a selector to sort makes it a bit easier. There is a huge list of options to sort NSArray, each one specific to your sorting needs. The following example is for a simple alpha case-insensitive string sort:

```
NSArray *playerNames = @[@"Jimmy", @"Katie", @"Claire", @"Gracie"];
NSArray *sortedPlayers = [playerNames sortedArrayUsingSelector:
@selector(localizedCaseInsensitiveCompare:)];
//returns [@"Claire", @"Gracie", @"Jimmy", @"Katie"]
```

You can create arrays using a range of items from another array using subarrayWithRange, which will return the last two players in the list:

```
NSArray *playerSet = @[@"Player 1", @"Player 2",
@"Player 3", @"Player 4"];
NSArray *lastTwoPlayers = [playerSet
subarrayWithRange:NSMakeRange(2, 2)];
```


www.it-ebooks.info

You can merge two `NSArray` variables into a new `NSArray` variable using `arrayByAddingObjectsFromArray`:

```
NSArray *allPlayers = [playerSet arrayByAddingObjectsFromArray:player
Names];
```

NSMutableArray

As with earlier examples, `NSMutableArray` has a few additions to the methods available for `NSArray`. We can't declare `NSMutableArray` the same, so let's start with our declaration:

```
NSMutableArray *playerSet = [NSMutableArray arrayWithObjects:@"Player
1", @"Player 2", @"Player 3", nil];
```

We can add objects similar to `NSMutableSet`, but we can also insert an object into a specific location in `NSMutableArray`:

```
[playerSet addObject:@"Player 4"];
[playerSet insertObject:@"Player 5" atIndex:0];
```

Similarly, we can remove an object, remove a specific object, or remove an object from a specific index location:

```
[playerSet removeLastObject];
[playerSet removeObject:@"Player 1"];
[playerSet removeObjectAtIndex:1];
```

Finally, we can replace an object at a specific location with a new object:

```
[playerSet replaceObjectAtIndex:2 withObject:@"New Player"];
```

There are several theories for what the `NS` stands for, but it is generally accepted that the `NS` stands for either NeXTSTEP or NeXT/Sun for the collaborated work that NeXT and Sun did on the OpenStep API.

Hello World!

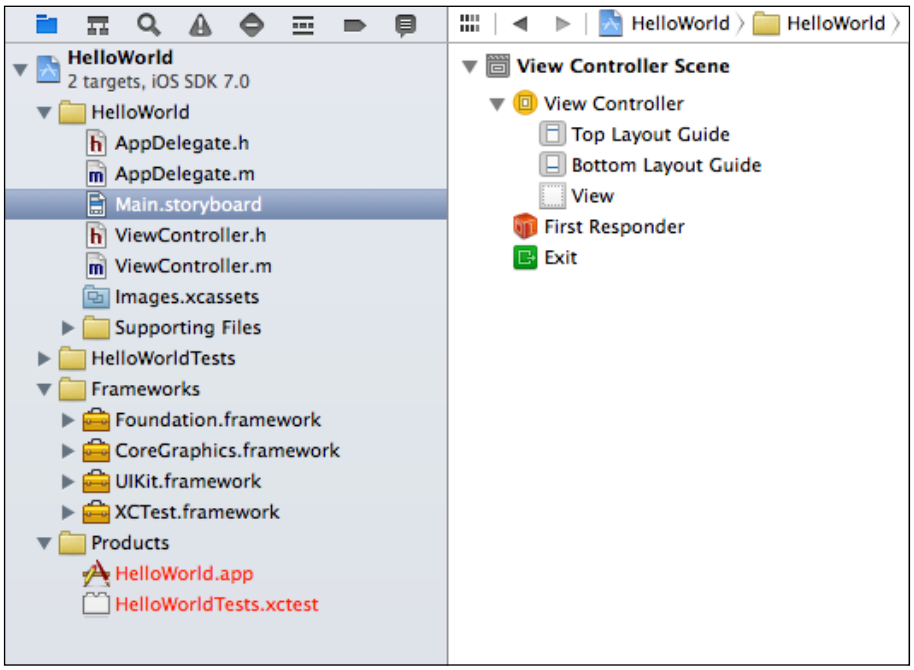
Just so we don't break too far from tradition, we are going to create the standard Hello World! program for our first iPhone app. This can be accomplished using Xcode with no programming whatsoever. Sounds odd, I know, but it really works in the Xcode IDE.

www.it-ebooks.info

Let's go back to the steps we took in the section *Opening Xcode for the first time* in the previous chapter. We stopped once the project was created, so let's continue it now. Hit the Run button and it will load up your simulator and run your app, but notice it is only a blank white screen. Although it really doesn't look like much of an app, it is your first iPhone app. Hooray for you! To make it a bit more interesting, let's add in a text string that says `Hello World!`.

First, allow me to introduce you to a new editor window. If your navigator selector bar is set to **Project navigator**, then you should see a list of your project files. Inside your `HelloWorld` folder, you should see a file named `Main.storyboard`. If you go ahead and click on it, you will notice your editor window will switch over to the storyboard editor window. The storyboard is the user interface layer of your iOS application. Your project can have multiple scenes that you can connect together; for now, let's just focus on a single view. Your **Utilities** window will also change to show several more options than what is shown normally.

On the left-hand side of the storyboard editor window, notice your document outline for the current storyboard. In your document outline, you should see a **View Controller** with a little triangle next to it for expanding it out (believe it or not, that triangle has a name; it is called a Disclosure Triangle), a First Responder, and an Exit. If you expanded the **View Controller**, you will see a **View**, as shown in the following screenshot:



www.it-ebooks.info

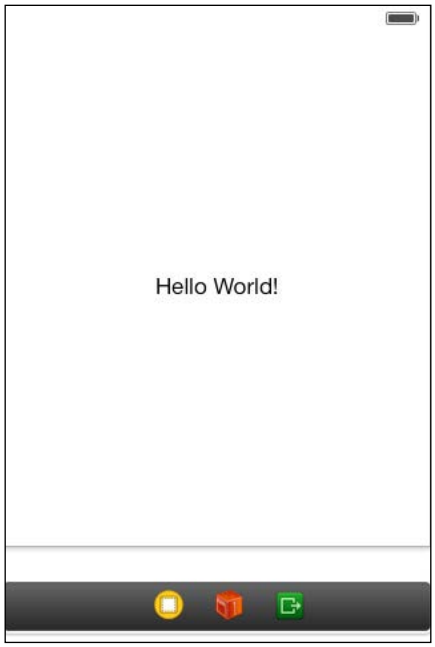
If your controller is set up to simulate the longer iPhone 5 screen, you will have an icon at the bottom of your storyboard editor window like this:

Go ahead and click on it to set it to the shorter Retina 3.5 form factor and the icon should change to this:

This is so that everything will line up in our simulator exactly as expected.

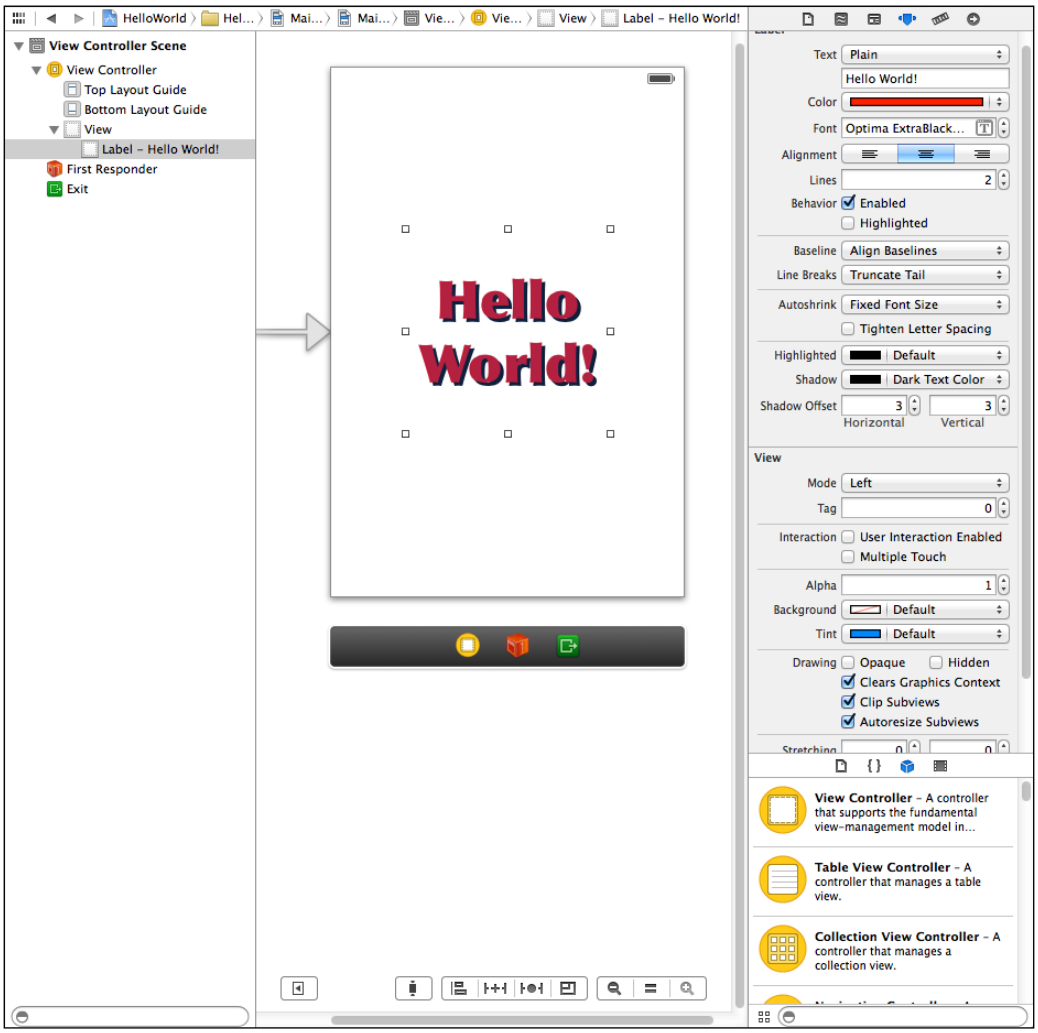
In the bottom panel of your **Utilities** window, select the **Objects** library and scroll down to the **Label** object. If you just drag a **Label** object onto the **View** controller in your scene, you will see an editable object appear with the word **Label**. You can drag this anywhere in the **View** controller area; it doesn't really matter, but if you drag it to the approximate center of the screen, it will snap to the exact center position. Next, double-click on **Label** in order to edit the text, and enter the words `Hello World!` into the field. You can put anything you like in here; however, as this is the `Hello World!` demo, let's just stick with `Hello World!`.

This is approximately what your storyboard should look like at this point:



www.it-ebooks.info

At this point, if you run your app, you should see a white screen with the words `Hello World!` printed across the screen in black text. Nice, huh? Although, you might be thinking, *Well, that's nice but it's not a really interesting demo. I expected a lot more from an iPhone demo*. Fair enough! Let's play a bit before moving on to the meatiness of programming. Go back and select your label and click on the attributes editor in your **Utilities** window. You will see a lot of options available to your little text field. Go ahead and play around with the options, for example, set the font and color, add a shadow, reset the alignment, and do whatever you like. Just make it your own. Now run your app; doesn't that look a little more like something you would make—a truly unique `Hello World!` iPhone app:



www.it-ebooks.info

Everything you just did you can also do by typing in code to control your scene. You may make the modifications by simply telling your code to set up the screen, place a label, change the attributes, and then display them in your app. We are going to start learning how to use Objective-C to do these things, but first, let's talk a little bit about programming with Objective-C.

Programming with Objective-C

You got a brief description of classes earlier in the chapter, but now we need to show you how to declare and implement classes. In order to have a class in Objective-C, it requires two pieces: the **interface** and **implementation**.

@interface

The interface is used to define the class, its instance variables, and the methods that will be used in the class. The interface is usually stored in the header file; that's the file with the `.h` extension. In this code example, we are creating an interface for `ViewController` with a parent class of `UIViewController`, and within that `ViewController class`, we want an instance method called `showAlert` that will return `IBAction`. This is everything you will want to add into the interface section; it all begins with an `@interface` declaration, and once we have everything defined, it ends with the `@end` statement:

```
@interface ViewController : UIViewController

-(IBAction)showAlert;

@end
```

@implementation

The implementation file (the file with the `.m` extension) contains most of the class code. We use the declaration `@implementation` in order to declare the class implementation. In other words, it is the implementation file that matches our header file. The code starts off by importing a link to the header file. Then, the class implementation starts with the `@implementation` declaration and the name of the class. In this manner, you have your instance class method, which is just waiting for your button code:

```
#import "ViewController.h"

@implementation ViewController
```


www.it-ebooks.info

```
-(IBAction)showAlert
{
   // add button code here;
}
@end
```

So now that you understand the structure, let's create some code.

Here we go! We are going to start a new project; create another project exactly like the one we created in the *Opening Xcode for the first time* section in the previous chapter, but call this project `HelloWorld2`. In essence, we are going to combine what we learned in the previous section with some basic programming.

Select your `ViewController.h` header file. Let's add the `showAlert` button action method before the `@end` line, shown as follows:

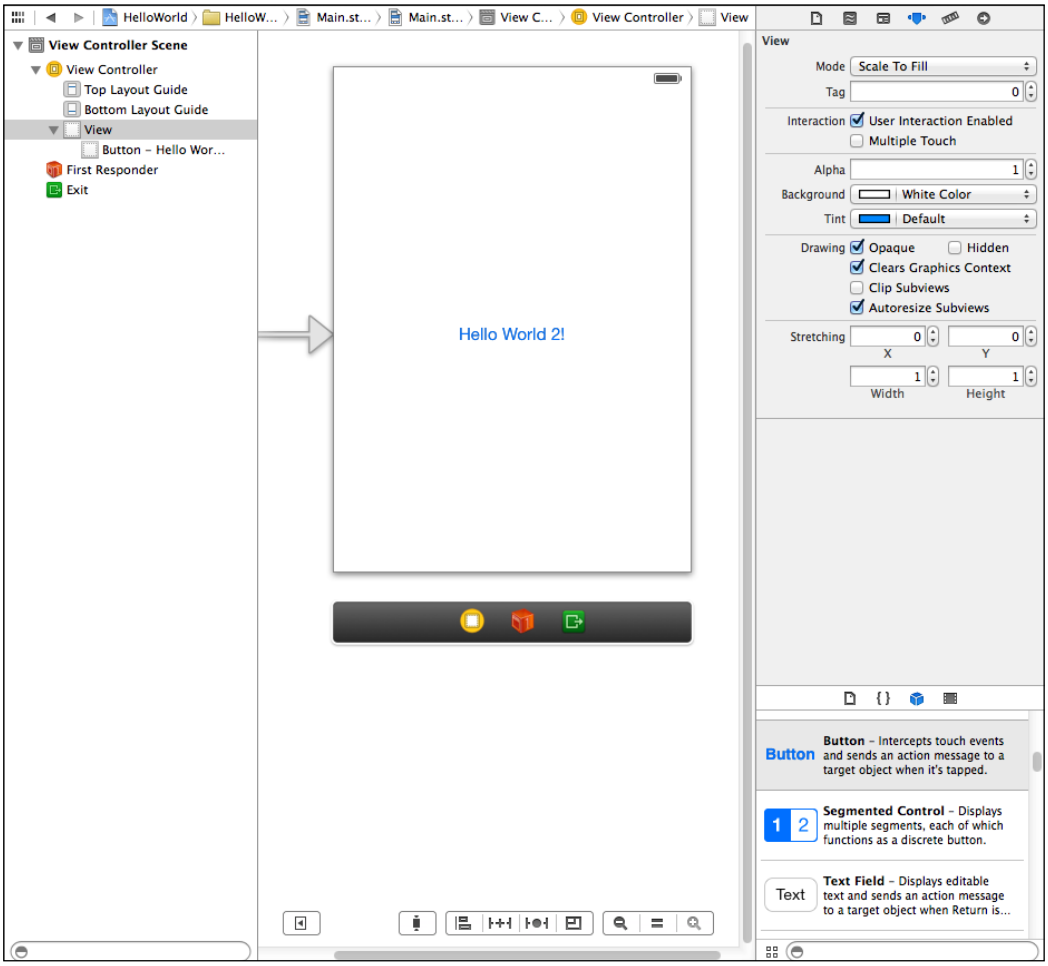
```
@interface ViewController : UIViewController

-(IBAction)showAlert;
@end
```

Then, select your `ViewController.m` implementation file so that we can add our `showAlert` method. Inside `showAlert`, you are going to create an instance of the `UIAlertView` class and name it `helloWorldAlert`. You don't really need to understand all of the this right now, but I do encourage you to take a close look at the arguments. The next line displays `helloWorldAlert`. Let's wrap it all up with the `@end` statement. For now, you can just replace all of the code in your file with the following code:

```
#import "ViewController.h"

@implementation ViewController

-(IBAction)showAlert
{
UIAlertView *helloWorldAlert =[[UIAlertView alloc]
                       initWithTitle:@"MyTitle"
                       message:  @"Hello World 2!!"
                       delegate:nil
                       cancelButtonTitle:@"OK"
                       otherButtonTitles: nil];
  [helloWorldAlert show];
}

@end
```

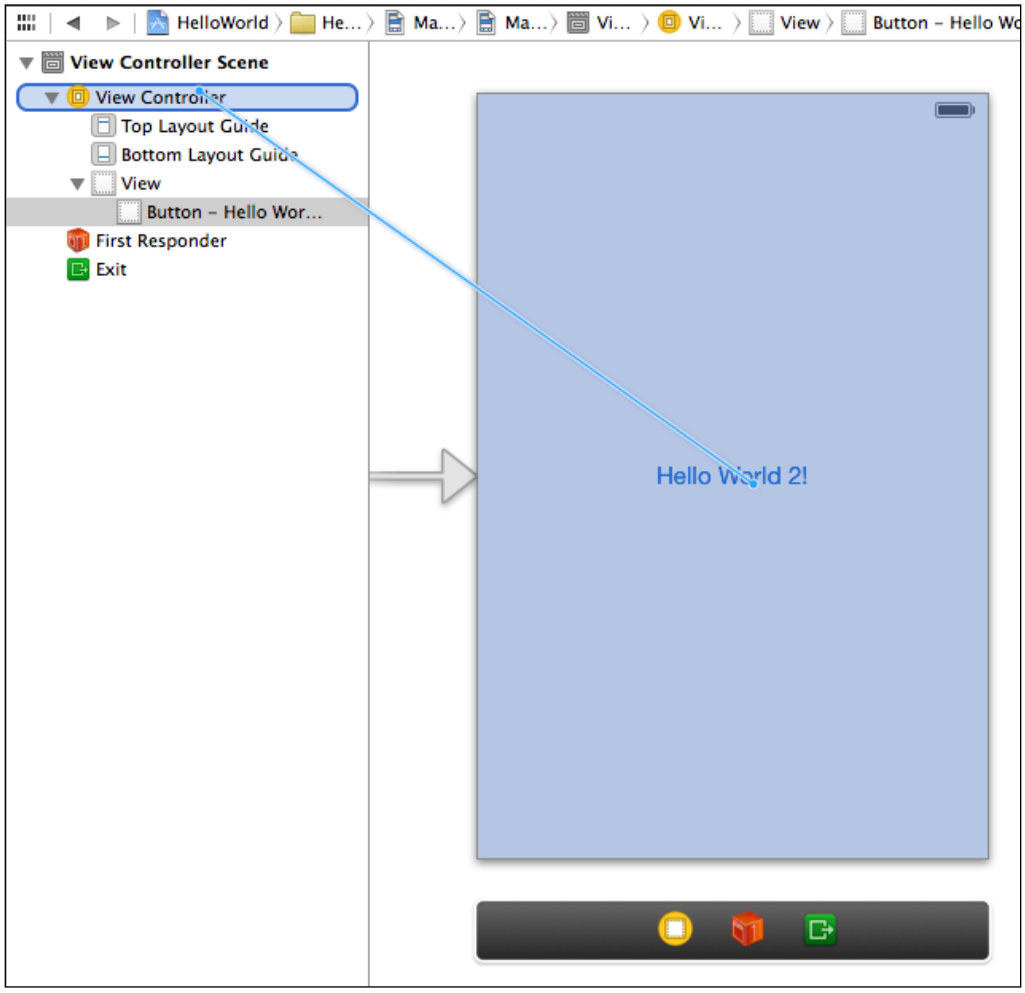

www.it-ebooks.info

Next, go into your storyboard file and add a `Button` object from your object library to your main view. Double-click and change the button's text to `Hello World 2!`. The result of these operations can be seen in the following screenshot:



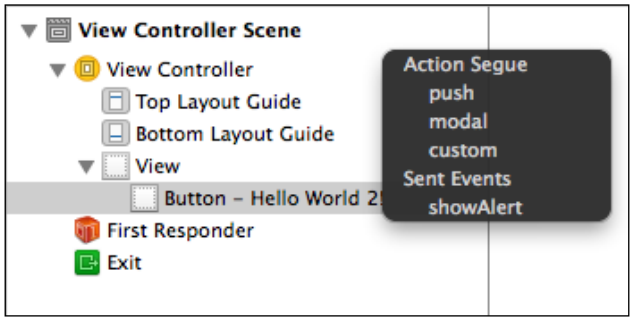
www.it-ebooks.info

If you press *control* and click on your new button and drag your mouse, you will see a blue link line. Press *control* and drag your mouse over to the **View Controller** in your document outline:



www.it-ebooks.info

When you release your *control* + drag operation, you will get a pop-up menu. Select the **showAlert** menu option from the list. This will link your `showAlert` method to your **Hello World 2** button:



If you run your app now, you will see your button where you placed it in comparison to your storyboard. Click on this button. You should see a dialog box appear that says **Hello World 2!** with the title **My Title**. If you look closely at your `UIAlertView` dialog arguments, you will notice both of these strings of text. Feel free to go in and change what it says until you get it to say exactly what you want. Another win for the unique `Hello World!` apps.

Model-view-controller

Oh, and there is one more thing. I am not going to get too far into design patterns, as (once again) there are volumes of information on this subject. As you build apps, you may notice that they are built with a certain methodology in mind. In the iPhone programming world, a design pattern is used called model-view-controller. Everything we build for the iOS environment should fall under one of following three roles:

- **Model objects**: These objects are your stored data information, rules, logic, and processes. This could be a character in your game, one of your enemies, saved player information, or the artificial intelligence that tells an enemy what to do. The model object gets manipulated by the controller object but usually updates the **View** object.

www.it-ebooks.info

- **View objects**: These are objects we create that the user sees. The user interface objects on the storyboard are a good example of **View** objects. **View** objects allow the user to interact and change data in the application. **View** objects talk to controller objects and also display the model objects' information, but they are usually disconnected from the model objects. The communication to the model object usually happens through the controller object.
- **Controller objects**: These objects receive information from your **View** objects and manipulate the model objects.

If you design your code with the model-view-controller pattern, it will make your life a lot easier in the long run by making your code more reuseable and your user interfaces much cleaner.

Summary

In this chapter, we covered the basics of programming along with details of programming using the Objective-C language. You learned about OOP and how Xcode uses the model-view-controller pattern to create apps. You also built a `Hello World!` application in Xcode and ran your first app.

In the next chapter, we will learn how to use scenes and views to create a simple puzzle game.

www.it-ebooks.info

www.it-ebooks.info

3

iPhone Game Development Basics – The Matching Game

In this chapter, we will be talking about setting up multiple screens in iOS and how to transition from screen to screen. We will be writing our first game, a simple matching or memory game, and we will have a brief discussion of the Apple Framework's code libraries.

In this chapter, we will be covering the following topics:

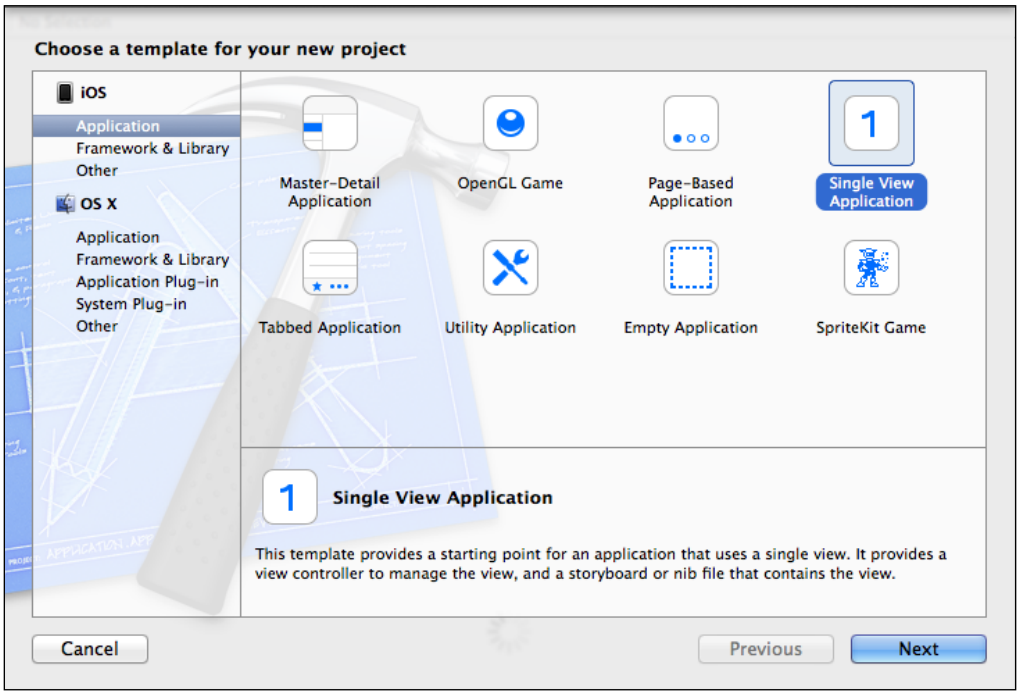
- Understanding storyboards
- The memory game code
- The frameworks
- Building our game scene

Scenes and views

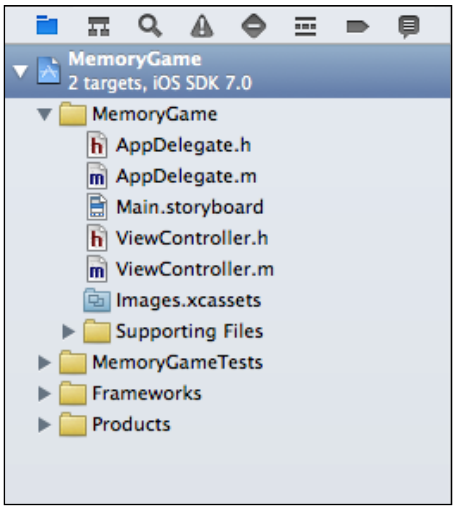
We played around with storyboards in the previous chapter. In this chapter, we are going a little more in depth by showing you how **views** and **scenes** work together. Let's start with a new project using our standard **Single View Application**. We will call this one `MemoryGame`, because that is what we will have created by the end of this chapter.

www.it-ebooks.info

The **Single View Application** template can be seen in the following screenshot:



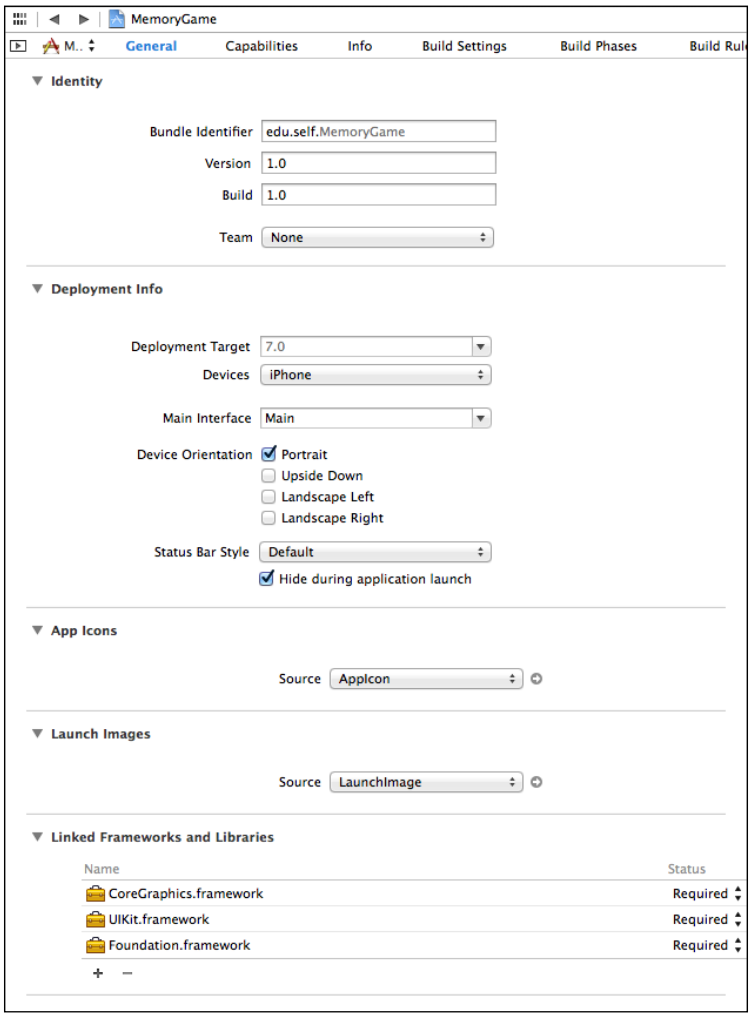
When we first start a new project, we are brought to our project editor screen. If you do not see it, just select your project at the top of your project navigator:



www.it-ebooks.info

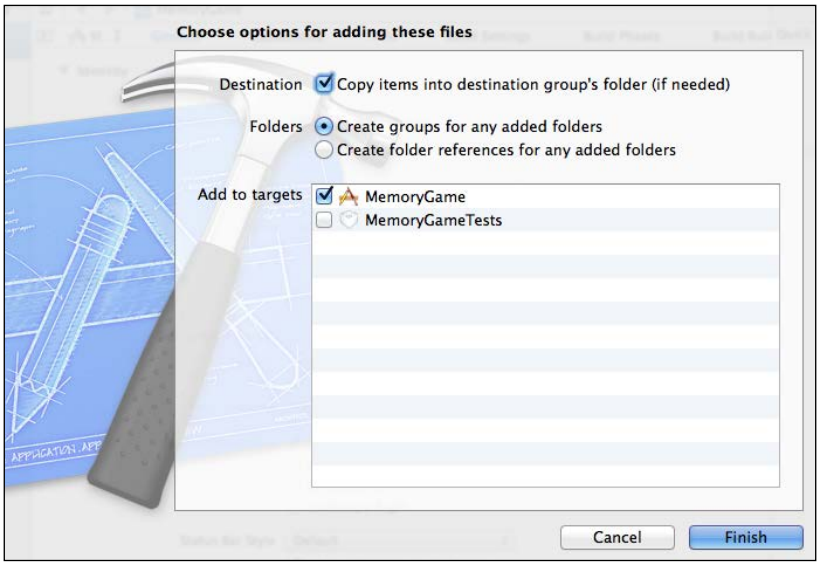
We are going to make some very easy changes to the project, just to tidy things up a bit for the app. First, we are going to change the **Device Orientations** so that only the **Portrait** orientation is turned on, which will prevent the users from rotating their phones and changing the screen layout we put together.

Second, we are going to change the visibility of the status bar by making sure **Hide during application launch** is checked. The status bar is the bar that runs along the top of your iPhone that shows your carrier, connection, battery life, and time. Depending on the type of app you finally create, you may want to keep this on, but for now, we want to have all the screen space possible. The changes made can be seen in the following screenshot:

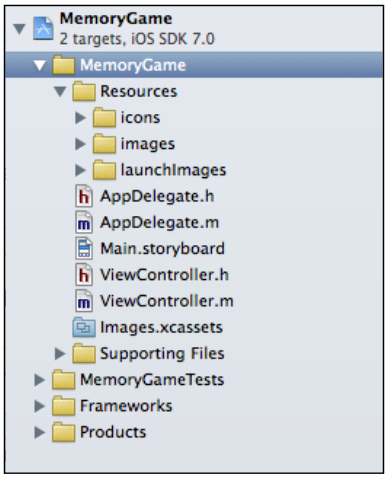


www.it-ebooks.info

Finally, we will add some icons to the application. None of these are necessary, but they are nice to have. Adding an app icon early allows you to distinguish the apps you are working on in the Simulator. Drag the supplied `Resources` directory from your finder to the `MemoryGame` group in your project. When adding files from an outside directory, make sure to check the **Copy items into destination group's folder (if needed)** checkbox and select the **Create groups for any added folders** radio button, as shown in the following screenshot:

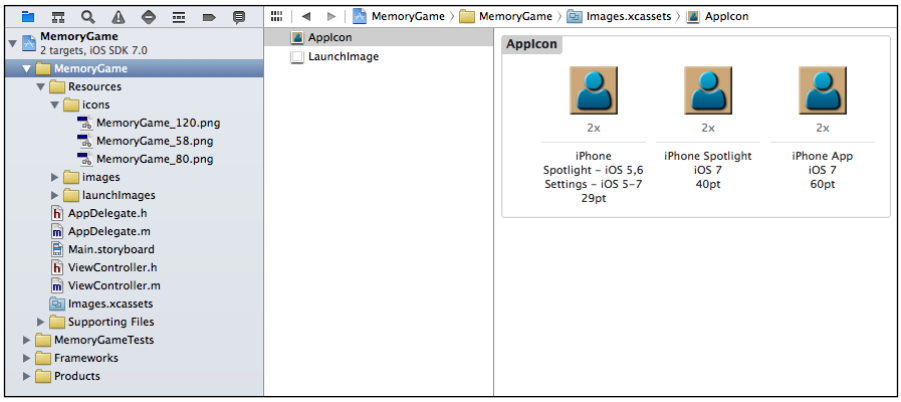


This will create a folder in your project with all the art you will need to complete this game:



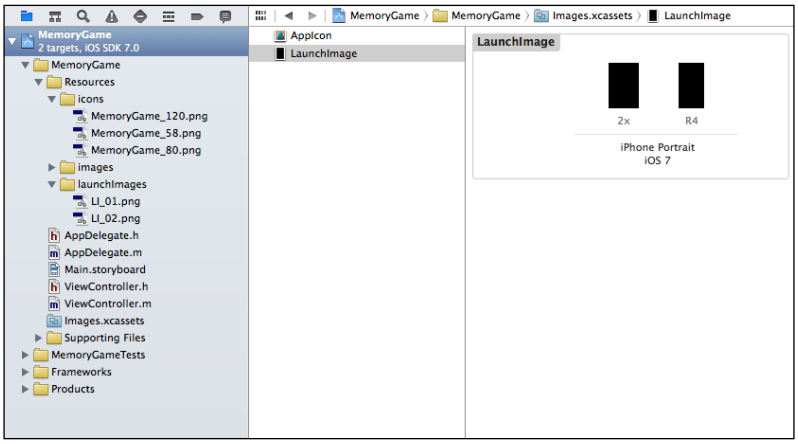
www.it-ebooks.info

Back in the project editor, in the **AppIcon** section, click on the right arrow icon next to the **AppIcon** pop-up menu. This will take you to the **Images.xcassets** section of the project editor. Drag the three icons from the icons directory to the placeholder areas in the **AppIcon** section, as shown in the following screenshot:



You will drag the icons from the smallest icon (**MemoryGame_58.png**) on the left to the medium sized icon (**MemoryGame_80.png**) in the middle and the largest icon (**MemoryGame_120.png**) on the right. When you create your app, these icons will automatically load up on the appropriate device.

We are going to do the same thing with **LaunchImage** now. Launch images are images that open up before anything in the app loads. It is extremely fast, so if you have a large app that loads slowly, you can put an image in here that tells your users that the app is loading so the users know that your app is in the process of starting up. As this is a small app that will load really quickly, we are just going to load a blackout image, as shown in the following screenshot:



www.it-ebooks.info

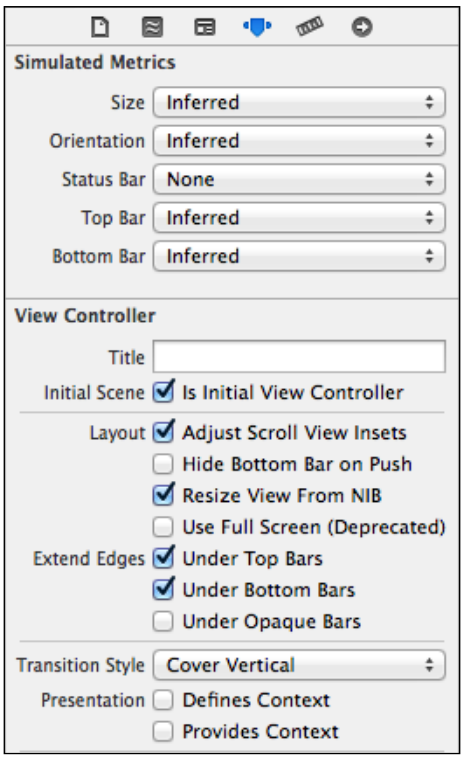
Understanding storyboards

Let's take a look at the storyboard screen. As with our `Hello World!` demos, the storyboard starts out with a single scene. If you were going to run your entire app in a single screen, this would be great; however, I have never seen anyone make an app with just one screen.

Apple has built-in storyboards to help you visually lay out your apps. With `Hello World 2`, we played around with the first layout scene, but now we are going to create a multiscene app in order to introduce you to **view controllers** and **scenes**. For our example, we are going to create four scenes and show you how to control them using code, as well as how to drag-and-drop objects in your storyboard.

Let's begin by performing the following steps:

1. Select **Main.storyboard** in your project navigator.
2. Then, select the **View Controller** in your **View Controller Scene**.
3. Set your **Utilities** panel to **Attributes Inspector**.
4. You may now set your **Status Bar** drop-down menu to **None** so that when you are doing your layouts, it will match your app while it is running:



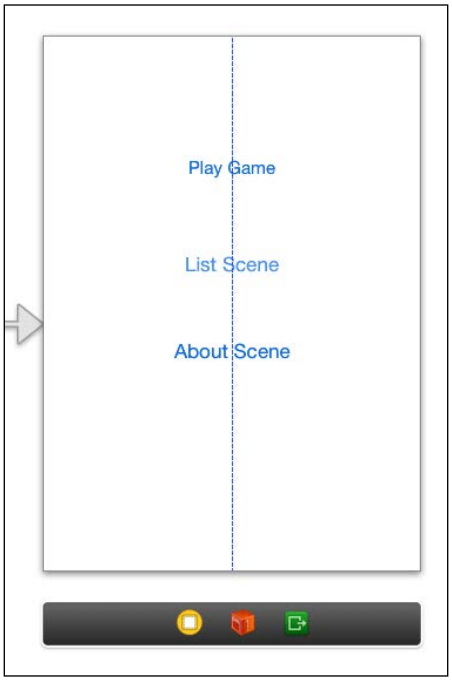
www.it-ebooks.info

5. Double-clicking on the background of your storyboard area will allow you to zoom out.
6. Next, we are going to drag out three more **View Controller** objects from your **Objects Library**. You may notice the status bar is available on your new view controllers, but don't worry, as that will get fixed soon.
7. Double-click on your first **View Controller** and you will zoom back in. The zoom controls are available on the bottom-right corner of the storyboard window as shown in the next screenshot. The = symbol zooms in to actual size.

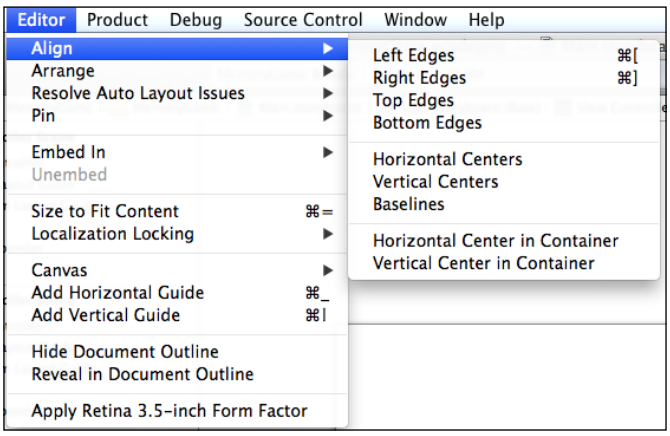
8. Add three **Button** objects from your **Objects** menu.
9. Double-click on the first button and change the text to say `Play Game`; change the second button's text to `List Scene` and the third one's text to `About Scene`. You have probably come to the conclusion that each of these buttons is going to jump to the different views, but first let's make the view a bit more appealing.

www.it-ebooks.info

10. If you drag the buttons around on the screen, dashed blue lines show up when you are aligned with something else on the screen, or you are lined up with another object on the screen:

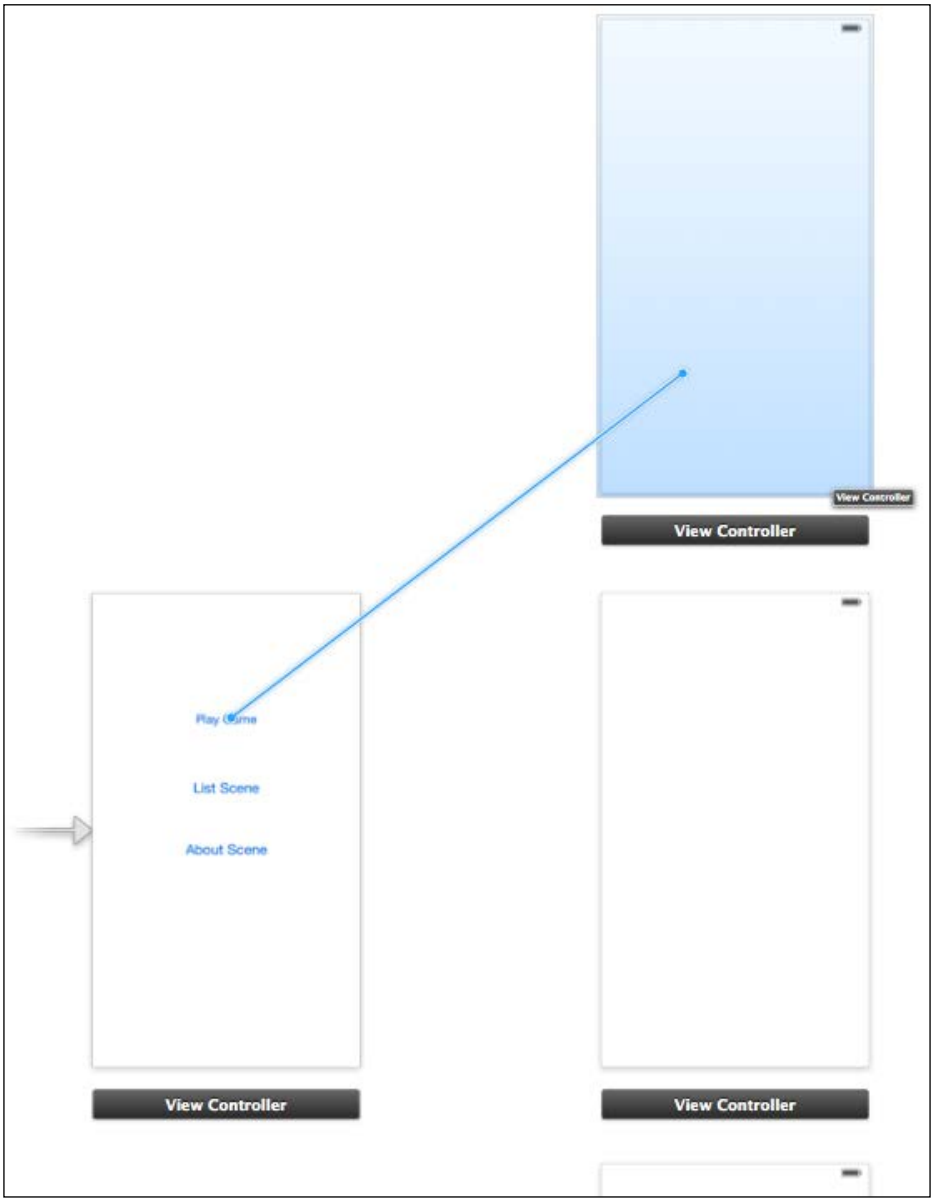


11. Objects can move around on the screen and can be manually aligned to other objects. You can adjust the position using the size inspector in your **Utilities** menu, or you can navigate to **Editor** | **Align** and choose from the alignment options available:



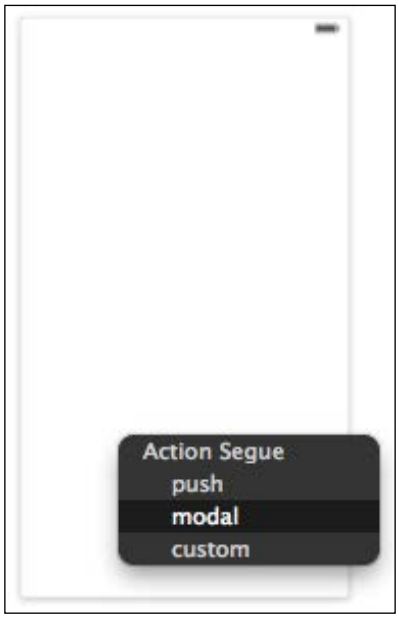
www.it-ebooks.info

12. Now that our main menu is laid out the way we want, let's make it such that when someone clicks on one of our buttons, it goes to another scene by adding a segue. Segues manage transitions between view controllers. This is a simple three-step process. Press *control* and click-and-drag from your **Play Game** button to your first **View Controller**. You will see a blue line connecting your two elements:



www.it-ebooks.info

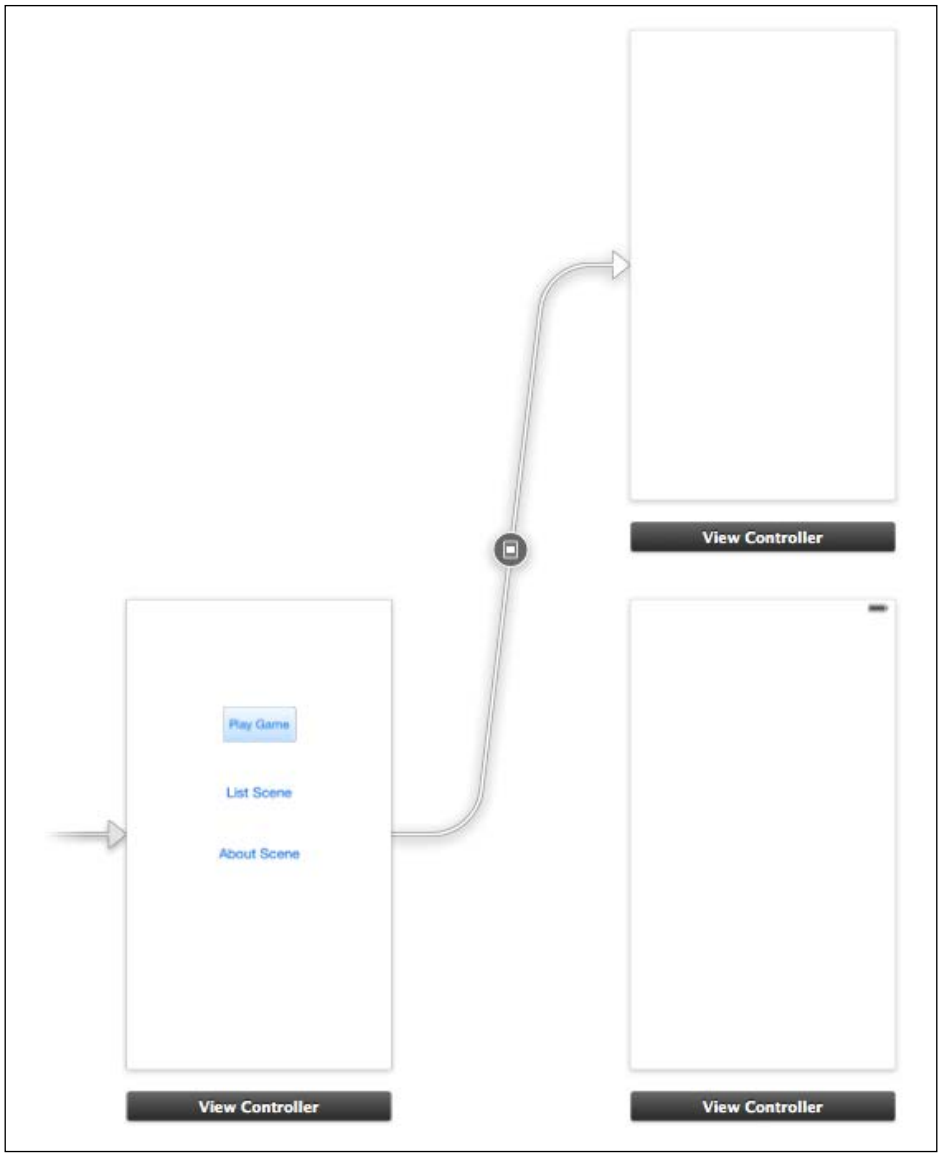
13. Release your mouse once you are over your **View Controller**. You should see an **Action Segue** pop-up menu appear; choose **Modal** from the menu:



> A modal item is a window that appears over another window, thus preventing the user from being able to interact with anything behind it.

14. Once you have chosen a segue from the list, a line between the two elements appears in your storyboard window showing which elements are connected:

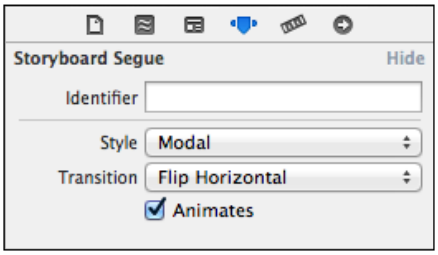
www.it-ebooks.info



You will notice that once a segue has been added, the **View Controller** removes the status bar from the top of the controller. Segued windows will automatically pick up the linked windows metrics.

www.it-ebooks.info

15. You can change and make adjustments to the type and properties of the segue by clicking on the segue line and viewing the attributes in the attributes editor in your **Utilities** area. Don't change the **Style**, but let's go ahead and change the `Transition` type to the **Flip Horizontal** animation, as shown in the next screenshot. You can choose which transition you prefer.



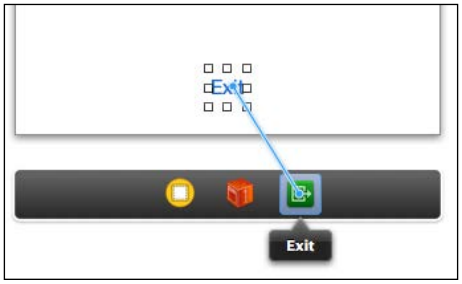
16. At this point, you should run your app. Click on your **Play Game** button and go to your new **View Controller** scene. Note that you cannot go anywhere from there, unfortunately. So, before you run your app, let's set up a button in your game scene to take you back to the main menu.
17. Double-click on your background to zoom out, and then double-click on your linked **View Controller**.
18. Add a new **Button** object and call it `Exit`.
19. We could create a segue back to the main menu, but that would create a new instance of the main menu scene. Instead, we will unwind the scene using code.
20. Select your `ViewController.m` file and add the `unwindToThisViewController` method right before the `@end` statement:

```
#import "ViewController.h"

@interface ViewController ()

@end

@implementation ViewController

- (void)viewDidLoad
{
    [super viewDidLoad];
  // Do any additional setup after loading the view, typically
from a nib.
}
```

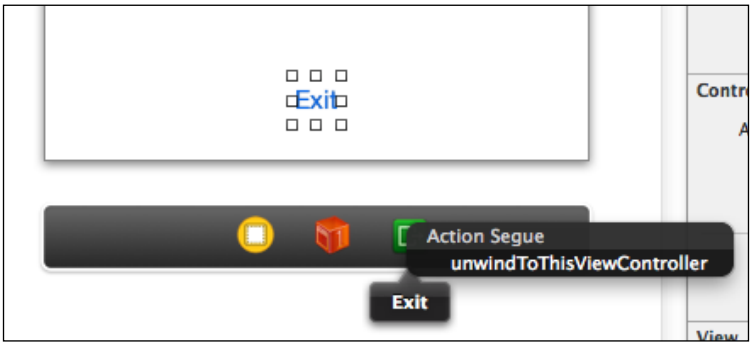

www.it-ebooks.info

```
- (void)didReceiveMemoryWarning
{
    [super didReceiveMemoryWarning];
    // Dispose of any resources that can be recreated.
}
- (IBAction)unwindToThisViewController:(UIStoryboardSegue *)
unwindSegue
{

}

@end
```

21. Since this is your method, you can change the method name to whatever you prefer, but keep in mind that I will be referring to it as `unwindtoMainMenu`. Wherever you see reference to this method, just replace it with whatever you named your method.
22. We are going to link our **Exit** button to this method. Press *control* and click on your **Exit** button and drag it to the `Exit` object and you will be rewarded with yet another pop-up menu:

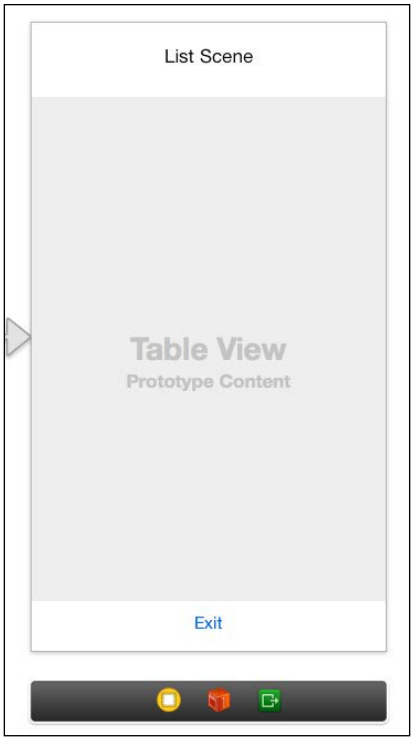


The pop-up menu will list the available, segue button actions in your code. For us, there is only one available so let's select that one:



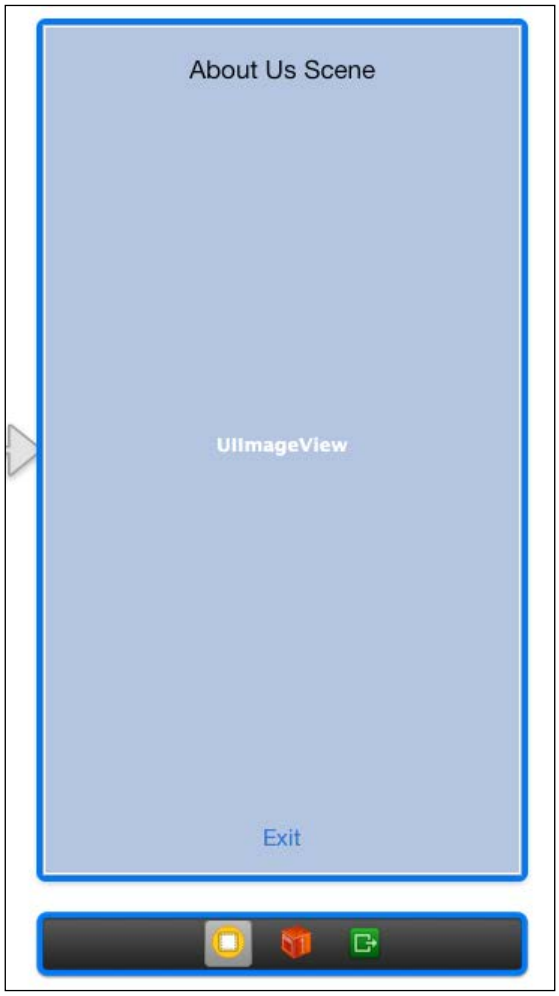
www.it-ebooks.info

23. There you have it! You can run your code. Click on your **Play Game** button and return back to your main menu as many times as you wish. Did you notice your flashy window transitions? If you built it exactly like we talked about, you should see your window flip horizontally when you clicked the **Play Game** button. Then, when you hit **Exit**, it automatically flips back, even though we did not add another transition anywhere. Of course, the other buttons don't do anything at this point; so, let's connect the **List Scene** and **About Scene** buttons to the other two views.
24. Don't forget to add **Exit** buttons and transitions to those scenes. If you follow the same instructions for adding an **Exit** button that was explained for your game scene, you would be able to see the same pop-up menu appear for unwinding back to your main menu. This is a very good and simple example of using our view controller class as a shared controller and our scene as our view that our users interact with.
25. Let's add a few elements to help distinguish our scenes from the game scene. In our list scene, let's add a **Table View** object (not a **Table View** controller) and a **Label object**. Go ahead and put something in the **Label** object so that we can tell where we are. I just inputted **List Scene**, but you can input whatever you prefer:



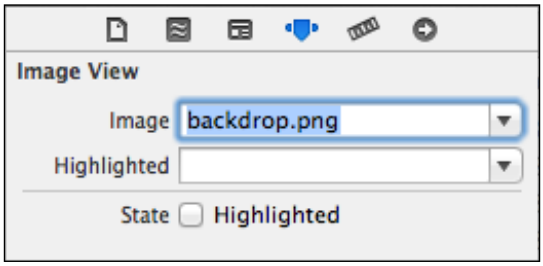
www.it-ebooks.info

26. For our **About Us** scene, go ahead and add a picture for the background. Select your **About Us** Scene in the storyboard and add in a **Label** object and an **Image View** object.
27. Set your size inspector for your **Image** view as follows: `x` to `160`, `y` to `240`, `width` to `320`, and `height` to `480`. In your **View Controller Scene** layout on the left, drag your **Image** view to the top of the view section.
28. Set the text of your label to `About Us Scene` or whatever you like:



www.it-ebooks.info

29. Select your **Image** view and set your utilities to the attributes inspector. If you change the image to **backdrop.png**, you should see your scene update with the backdrop image as the new background for your **About Us** scene:



30. For the final element, we are going to add a **Text View** object to our scene. This would be where we add the credits for the game. In the attributes inspector, make sure you turn off the editable checkbox. Other than that, feel free to play around with the settings. If you make a mistake, you can always undo. I made mine a transparent background so that I could see my image behind the scrolling text.

I think this is about enough on these scenes for now. Let's move on with making our first game.

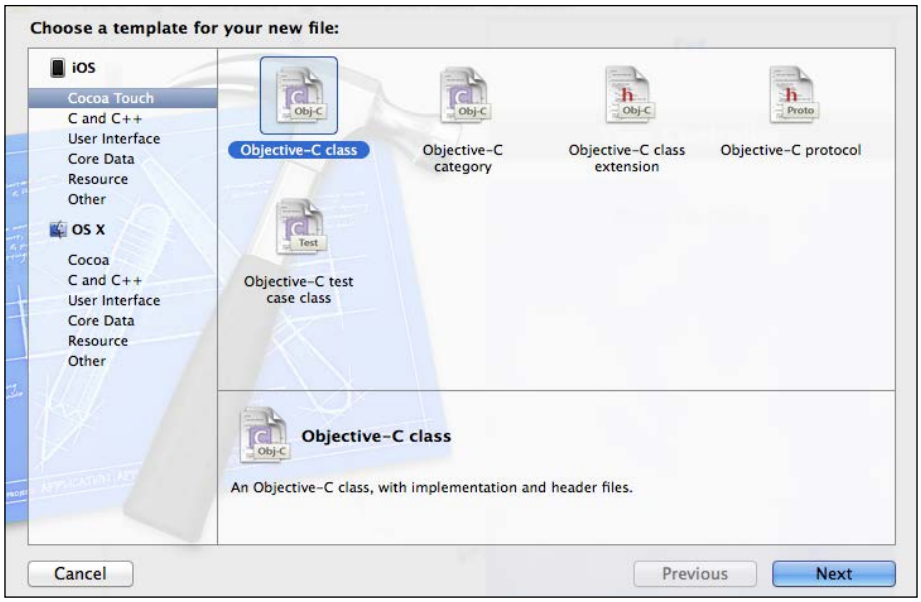
The memory game code

Here is a brief description of the memory game we will now create. We will design a bunch of buttons and treat them like they are selectable tiles that can be flipped over. On one side of the tiles, it will be blank and the color brown, and on the other side, there will be an image that we will randomly assign. We are going to check the tiles when the user has turned over two of them. If they match, we will remove them from the board, and if they don't, the tiles will automatically turn back over. We are going to create a basic timer that lets the user see the tiles for a full second before we turn them over or remove them. Once the user has removed all the tiles, we will tell the player how many times they guessed. If their score is better than previous players, we will display their score on the main menu.

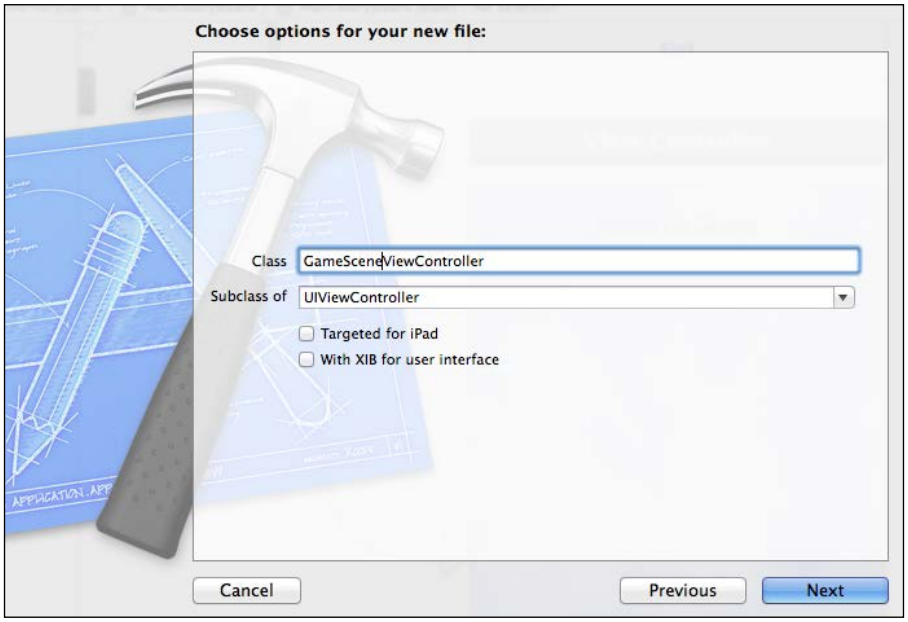
Let's start by creating our first class and associating it with one of our scenes:

1. To create new classes, you need to navigate to **File | New | File...**. This brings up the file template panel; **templates** are just text files that have the basic information you will use and are already set up for you.
2. You should select the **Objective-C class** file type from the template files and click on the **Next** button, as shown in the following screenshot:

www.it-ebooks.info

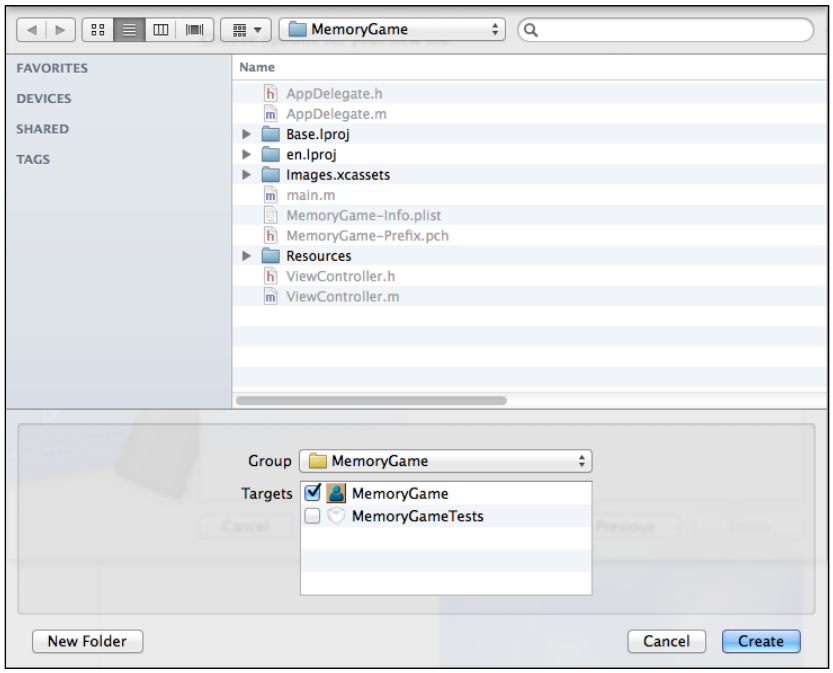


3. We are going to name our class `GameSceneViewController`. Set the subclass to `UIViewController`.
4. Make sure that **Targeted for iPad** and **With XIB for user interface** are not checked, as shown in the following screenshot:

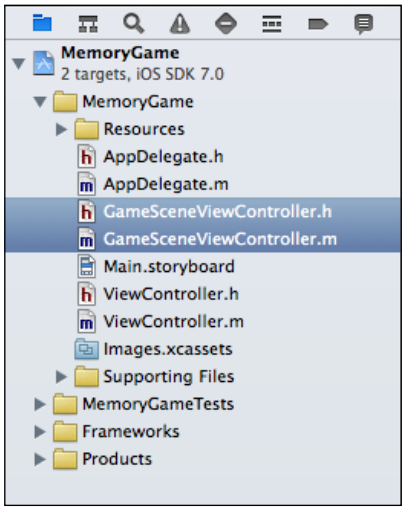


www.it-ebooks.info

5. Click on **Next** and save it to your project's directory:



6. Xcode will automatically import files into your project for you.
7. Two new files will appear in your project: **GameSceneViewController.m** and **GameSceneViewController.h**:



www.it-ebooks.info

The `.h` extension defines a file as a header file. In Objective-C, the **header file** defines the interface of our class; it is where we declare all of our class' methods and instance variables. The `.m` extension defines a file as an implementation file. There are several arguments as to what `m` stands for; I have heard method, iMplementation, and messages. I am not going to feed that hungry troll; therefore, we are going to refer to it as an implementation file.

Let's take a look at our header file when we first start up. Please note that we need to ignore the comments at the top of the file; this is how we start:

```
#import <UIKit/UIKit.h>

@interface GameSceneViewController : UIViewController

@end
```

`#import <UIKit/UIKit.h>` tells our class that we will need access to Apple's UIKit framework. We will talk more about frameworks in the next section, but for now, you should at least know that the UIKit is a set of code that gives your code access to the items in your storyboard.

`@interface GameSceneViewController : UIViewController` is where we declare our class. `@interface` is the declaration, `GameSceneViewController` is the new class name, and `UIViewController` is the subclass that we are building from. Finally, the `@end` statement closes out the class declaration. Lines of code between the `@interface` line and the `@end` statement line are public variable declarations and public method declarations, which means that it will allow classes outside our class to have access to this data, including our storyboards.

Now let's take a look at our implementation file:

```
#import "GameSceneViewController.h"

@interface GameSceneViewController ()

@end

@implementation GameSceneViewController

- (id)initWithNibName:(NSString *)nibNameOrNil bundle:(NSBundle *)
nibBundleOrNil
{
    self = [super initWithNibName:nibNameOrNil bundle:nibBundleOrNil];
    if (self) {
```


www.it-ebooks.info

```
            // Custom initialization
        }
        return self;
    }

    - (void)viewDidLoad
    {
        [super viewDidLoad];
        // Do any additional setup after loading the view.
    }

    - (void)didReceiveMemoryWarning
    {
        [super didReceiveMemoryWarning];
        // Dispose of any resources that can be recreated.
    }

    @end
```

`#import "GameSceneViewController.h"` links our header file to our implementation file. This line is the equivalent of pasting everything from our header file into our implementation file and deleting the header file. You could do that, but it is not recommended; besides, this is much cleaner for your code.

Next is `@interface GameSceneViewController ();` a class extension. You are probably saying, "Didn't we do this in the header?" The answer is, of course, no! The second `@interface` statement is for declaring private properties and private class methods. Private properties and class methods are only accessible from within this class. We will discuss this and show some examples of this later in this chapter. The `@end` statement closes the class extension. `@implementation GameSceneViewController` begins the area of code where we set up all of our functionality. This is where we put our class's methods and local variables.

As we created our `GameSceneViewController` class using Xcode's class template, we have a few methods predefined for us. The `initWithNibName` method is used when initializing XIB/NIB files, but as we are using segues, we will not be using this for now.

www.it-ebooks.info

The `viewDidLoad` method is automatically called as soon as our scene's view information is loaded. Should you receive a memory warning from the system, `didReceiveMemoryWarning` is a predefined method for you to dispose of anything that you may have in memory. You will notice that we are using a new keyword in `didReceiveMemoryWarning` called `super`. By using the `super` keyword, we are making a call up the hierarchy chain to call the method `didReceiveMemoryWarning` in our super class.

Frameworks

Frameworks are code libraries and **APIs** (**Application Programming Interfaces**) that have been prewritten for programmers to make their job easier. For iPhone, Apple has provided the **Cocoa Touch** collection of frameworks, which gives you access to some of the amazing technologies built into Apple's devices. We will be talking about several frameworks throughout the book, but you should at least be aware of a few.

UIKit gives you access to your UI and storyboards, including your views, buttons, and text. Usually if you are calling anything that has a prefix of UI, it is most likely calling UIKit. The following is an account of the frameworks:

- **Foundation Kit Framework** is the core for a lot of your Objective-C programming. Most of the classes you see with `NS` as a prefix come from your Foundation framework.
- **Core Animation** lets you easily program visual effects, compositing, and other graphics effects.
- **Core Audio** gives you access to professional level tools for programming audio recording, playback, and manipulation for your app.
- **Core Data** is for all of your data management. It makes it easy for you to share data across apps or just store information in a database.

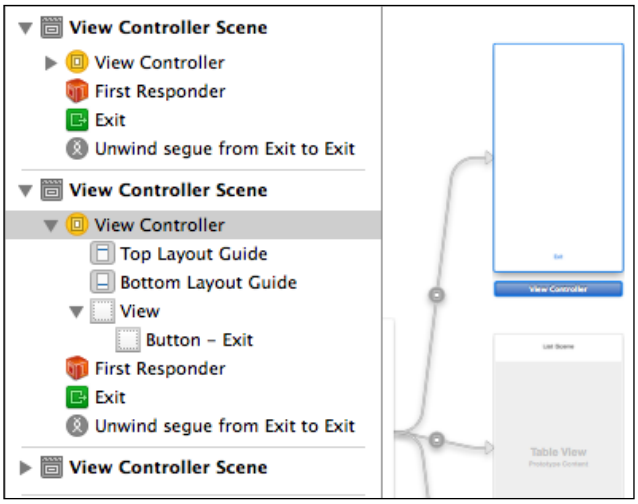
We will discuss more about these libraries as we access them throughout the book.

www.it-ebooks.info

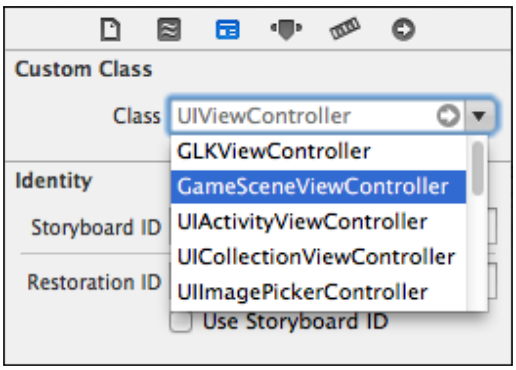
Building our game scene

It's time to start building our game scene:

1. We are going to start by connecting our newly created class to our game scene. Select the **View Controller** object for our game scene, as shown in the following screenshot:

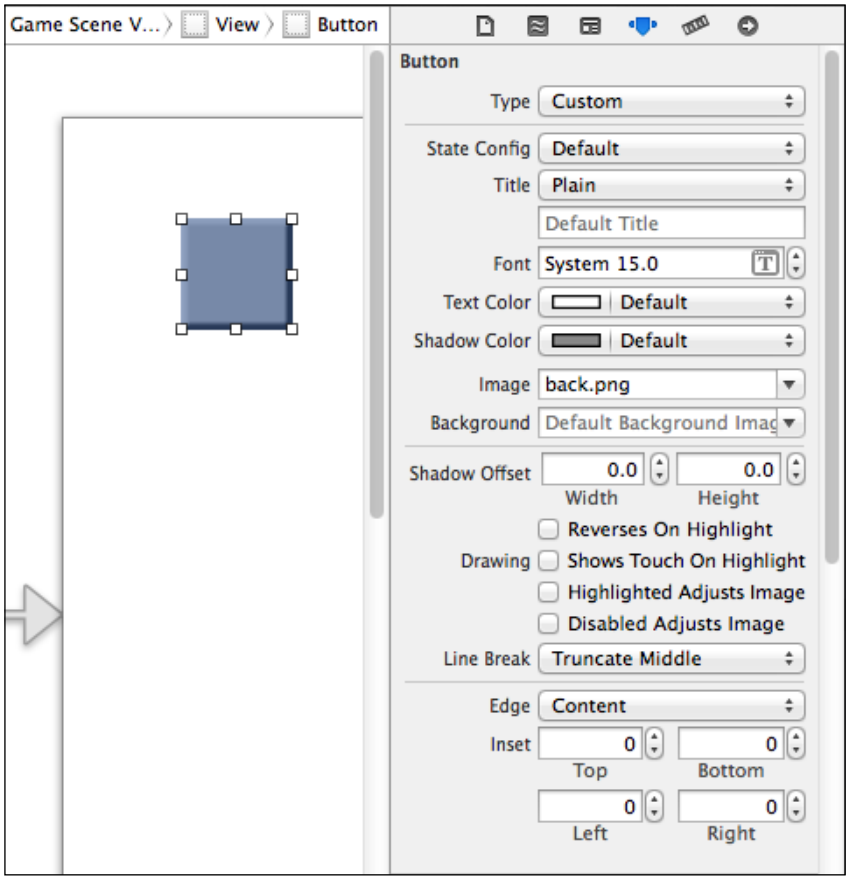


2. In your identity inspector, set your class to `GameSceneViewController`. Your view is now using your new class rather than the standard `UIViewController` class. This will allow us to write code that will only affect your game scene and still have access to the code in the class of `UIViewController`. We can do this because our interface extends the `UIViewController` class in our header file.



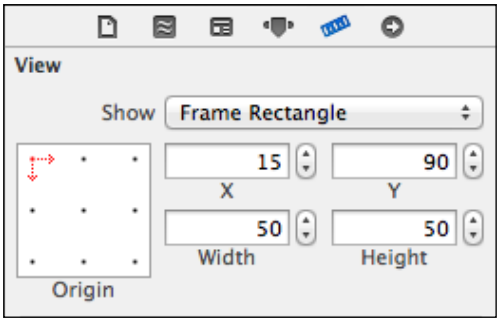
www.it-ebooks.info

3. Next, we are going to add our first tile button.
4. Let's drag in a new **Button** object.
5. With the new button selected, change the button **Type** in your attributes inspector to **Custom**.
6. Delete the name **Button** from the title area.
7. Uncheck **Highlighted Adjusts Image** and **Disable Adjusts Image**.
8. And, finally in the **Image** drop-down box, choose the **back.png** file.



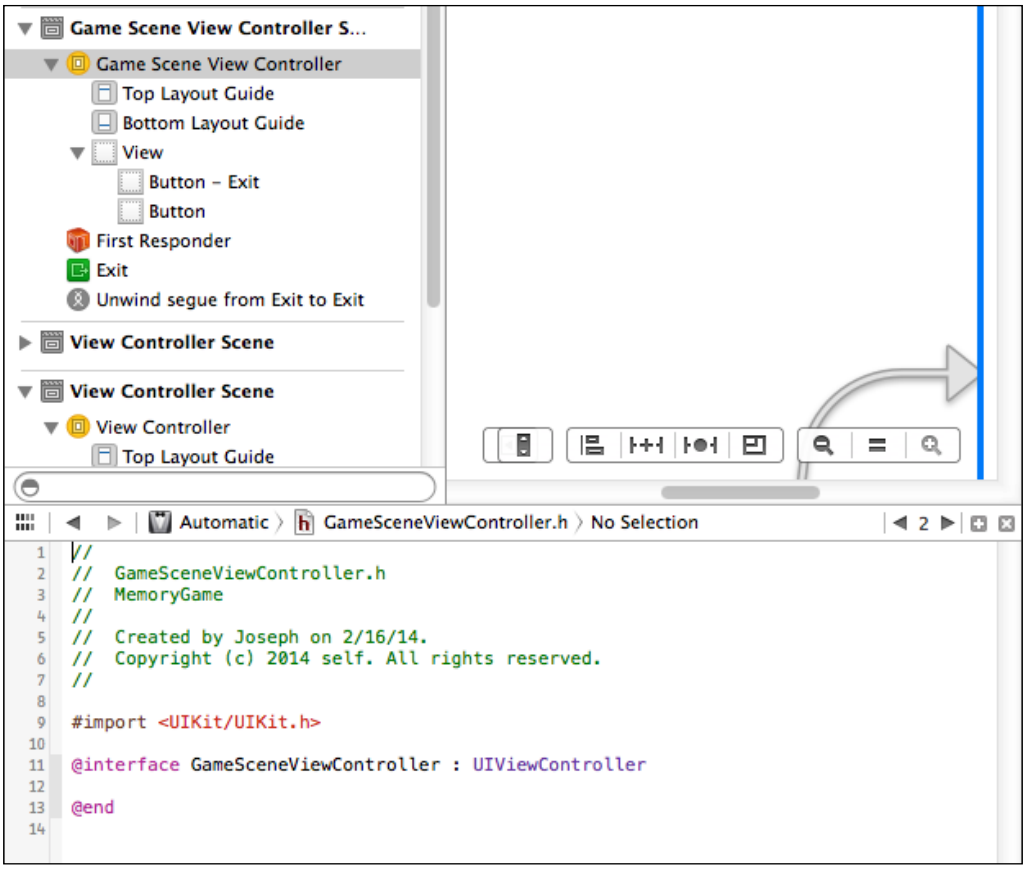
www.it-ebooks.info

9. Keeping the button selected, go to your size inspector. The **View** section of your size inspector lets you set the properties of your objects numerically. In here, we want to set the *X* position to 15, *Y* to 90, width to 50, and height to 50, as shown in the next screenshot. These measurements are in screen pixels; a **pixel** is one dot on your computer's screen and is a standard measurement for graphics that are displayed on computer screens.



10. Now we are going to create an **Outlet** property in our header file for our button. Apple was kind enough to create a fast and easy way to create properties from their interface builder to code using the assistant editor.
11. Select your **Game Scene View Controller Scene** and navigate to **View | Assistant Editor | Show Assistant Editor**. You will notice that a code window will show up below the editor area in your workspace. It should automatically have your `GameSceneViewController.h` file selected. If not, click on the **Automatic** button at the top of the assistant editor and select it from the list of files, as shown in the following screenshot:

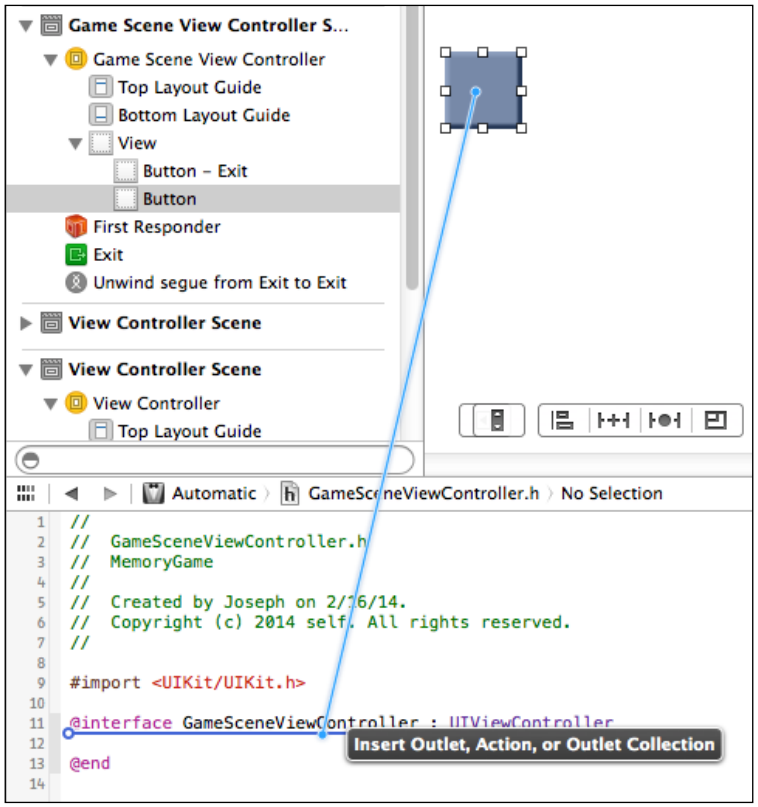
www.it-ebooks.info



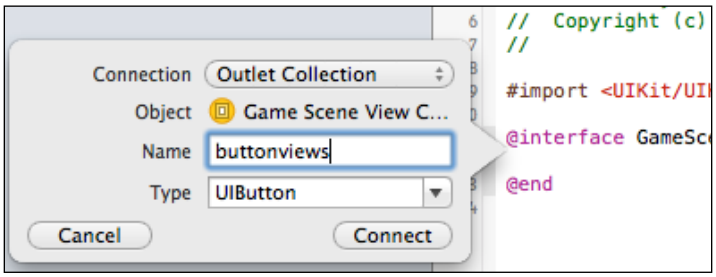
12. If you press *control* and click-and-drag your new button, you will see your blue connector line show up again.

www.it-ebooks.info

13. If you drag your button into your `@interface` area of your header file, it will bring up a property pop-up panel, shown as follows:



14. We are creating a lot of buttons, so we will want to make a collection and put them into an array in order to make them easier to track. In order to do this, we want to set our connection type to **Outlet Collection**, give it a name such as `buttonviews` and a type of `UIButton`, and then click on **Connect**, shown as follows:



www.it-ebooks.info

15. This will add `@property` into your code, as shown in the following screenshot, and it will be preformatted exactly as it needs to be as it creates a link to the button. You can tell if it is linked to a button when there is a filled circle symbol next to the line number of your property. If it is not linked, the circle will not be filled.

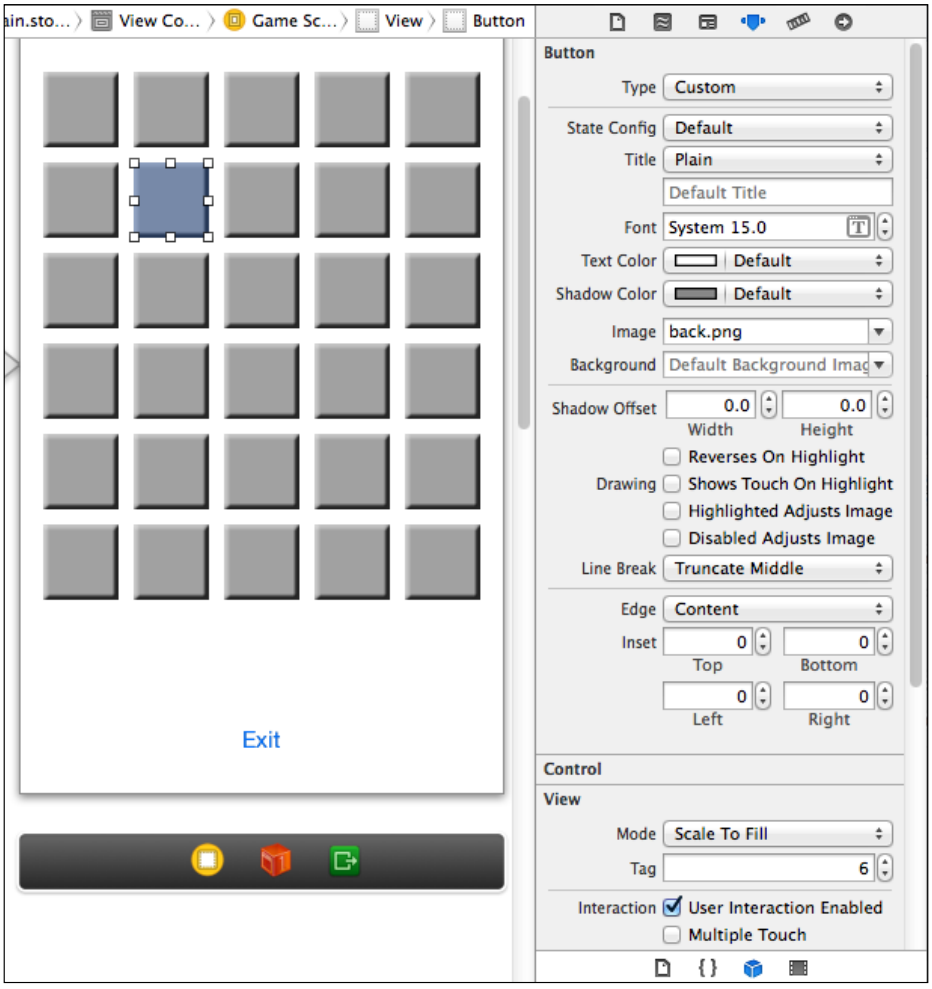
16. Our newly generated code shows our property as `IBOutletCollection` of `UIButtons` in `NSArray`, and it is named `buttonviews`. If we decided not to use a standard outlet for our connection type, it would show up as an `IBOutlet` with a type of `UIButton` but would not have `NSArray`, as shown in the following code snippet:

```
@property (strong, nonatomic) IBOutletCollection(UIButton) NSArray
*buttonviews;
```

17. Now, let's create a few more buttons and link them to the `buttonviews` array. We are going to create a grid of buttons 5 x 6 to look like the game screen displayed in the next screenshot. Copy and paste the first button and set its size inspector to (`x` = `75` and `y` = `90`) without changing the width or height. As we have already created our property, you can just press *control* and click-and-drag the button from your new button to the property.
18. When your mouse is over the property, you should see a label that says **Connect Outlet Collection**. You will need to do this until all 30 buttons are equally spaced 10 pixels apart on the left, right, top, and bottom. The final button should be placed at (`X` = `255` and `Y` = `390`).

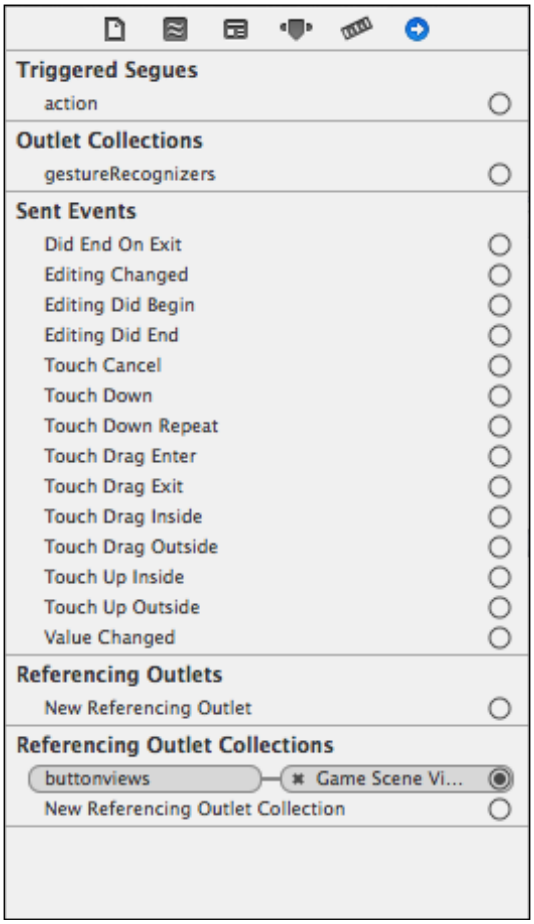
www.it-ebooks.info

19. In your attributes inspector, scroll down to the **View** section, as shown in the next screenshot (it's a little easier if you roll up the button and control sections). You will see a field named **Tag**. We will need to add a unique tag for each of the buttons; this is how our code will know which button the user touched. Moving from left to right, top to bottom, number them from `0` (zero) to `29` (twenty nine).



www.it-ebooks.info

20. If you take a look at the connections inspector for any of the buttons that have been linked, you will see the button within the connection in the **Referencing Outlet Collections** section. If at any point a mistake is made, you can always remove the link by clicking on **X** next to the **Game Scene View Controller** connection, as shown in the following screenshot:



21. Drag out a new **Label** object from the **Object Library** to the top area of the scene. Set the alignment to centered text and change the width of the label to `300` pixels wide.

www.it-ebooks.info

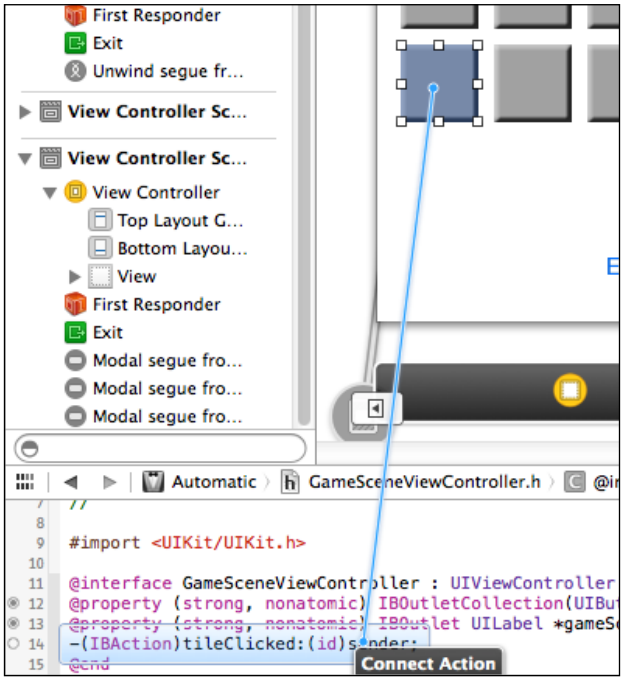
22. Let's create a property for it by pressing *control* and clicking-and-dragging it to the next line in our `GameSceneViewController` header. This time we are going to create a connection type of **Outlet** and name it `gameScoreLabel` with a strong storage type. The next code snippet will show up in your header file where you released your mouse button. You now have another linked property. As with our buttons, the link will show up in our connections inspector, but this will show up under **Referencing Outlets** as we are not using a collection outlet.

    ```
    @property (strong, nonatomic) IBOutlet UILabel *gameScoreLabel;
    ```

23. While we are here in the header file, let's add in a definition for a method when a button is clicked. Add the following line of code in below our `gameScoreLabel` property:

    ```
    -(IBAction)tileClicked:(id)sender;
    ```

24. To send an event when a user touches a button, we will need to link it to our `tileClicked` action. To do this, we use a similar method as we did when linking `buttonviews`. Press *control* and click-and-drag from your button to the `tileClicked` declaration. When you see the **Connect Action** label show up, you can let go. You will need to do this for all of the buttons. In your connection inspector, you will see your new event show up next to the **Touch Up Inside** item in the **Sent Events** area, as shown in the following screenshot:



www.it-ebooks.info

25. That's all we will need to do with our storyboard and header file for now. At this point, be prepared for some typing. As we defined an instance method called `tileClicked` in our header, we will need to add the method into our implementation file. If we ran the code without doing this and we clicked on a button to which it is linked, we would crash our app and get an error in our debugger.
26. Open our `GameSceneViewController.m` file. Right now, you should see a little Yield sign next to your `@implementation` statement with a message that reads **Incomplete implementation**. Why is this showing up? The reason is that we are missing the `tileClicked` method. Let's start by fixing that. It gets on my nerves to see those little yield signs, as it probably does yours as well. To get rid of the sign, type the following somewhere between the `@implementation` and `@end` statement:

    ```
    -(IBAction)tileClicked:(id)sender
    {

    }
    ```

    Cool! The yield sign is gone; now we can all sleep at night.

27. Every time someone clicks on one of our tile buttons, this instance method will get called, and it will run through the code line-by-line; it is not exactly helpful as there are no lines in here. So, let's remedy that. We have an `id` variable that is being sent in, and that `id` has a bunch of information tied to it that is associated with the buttons. One of the items associated with the buttons is the **Tag number** we set up earlier in the scene.
28. We start by defining a local integer variable (`senderID`) and set it equal to the `sender` object's `tag`. Then, we are going to tell our debugger to print a message saying what our ID is. Adjust your `tileClicked` code to match this:

    ```
    -(IBAction)tileClicked:(id)sender
    {
      int senderID = [sender tag];
      NSLog(@"Our ID is: %d\n", senderID);

    }
    ```

29. Now, run your code and click on your tiles. Each time you click on a tile, you should get the ID associated with that tile printed out in your debugger area. The result is as follows:

    ```
    Our ID is: 0
    Our ID is: 1
    Our ID is: 3
    Our ID is: 2
    Our ID is: 29
    ```


www.it-ebooks.info

30. If you are new to programming, read the note below about **NSLog**; if not, you have probably been using some form of NSLog in whichever language you have been working in.

NSLog

To show this example, we are going to use a standard Objective-C `NSLog` function. Whenever you want to have your code output something to your debugger console, you can use `NSLog` followed by an `NSString` containing what you want it to show. This is how most `Hello World!` demos are done, although I think my way is a bit more entertaining. This is very handy when trying to figure out what is going on in your code or debugging your code. The following is how it works:

```
NSLog(@"Sample text with a number: %d\n", 32);
```

This would print the message **Sample text with a number: 32** out as an error message. The `%d` adds the signed decimal integer argument in the function to the string and the `\n` adds a carriage return to the string, so the next string will show up on the next line. The format specifiers are as follows:

- `%d`: Signed decimal integer
- `%f`: Floating point number
- `%c`: Character
- `%s`: String

31. Now that we are sure that our buttons are linked to our code correctly, let's start setting up the code that we want to load up every time we start our game. For this section, I am going to go line-by-line to explain the code for our `GameSceneViewController.m` to explain how it works.
32. First, we use `#import` to link our header file to our implementation file:

```
#import "GameSceneViewController.h"
```

33. In the `@interface` section of our `.m` file, we are going to declare our instance properties:

```
@interface GameSceneViewController ()
```

34. We start by declaring a couple of image properties for our blank tile and the face down tile:

```
@property UIImage *blankTileImage;
@property UIImage *backTileImage;
```


www.it-ebooks.info

35. Next, we are going to declare an array that will hold image objects for all of the icon images we are going to use when the tiles are showing, as shown in the next code snippet. The `Mutable` in the `NSMutableArray` just means that we can change the array if we need to.

```
@property NSMutableArray *tiles;
```

36. The next mutable array is an array of numbers; these numbers are going to line up images array. Its declaration is shown as follows:

```
@property NSMutableArray *shuffledTiles;
```

37. `matchCounter` and `guessCounter` are integers that get incremented by `1` every time someone makes a correct guess and every time they make a guess respectively:

```
@property NSInteger matchCounter;
@property NSInteger guessCounter;
```

38. The `tileFlipped` property is the ID of the first flipped tile. Whenever it is not set to a tile ID, it is set to `-1`, so we know the first tile of a set has not been clicked. Its declaration is shown as follows:

```
@property NSInteger tileFlipped;
```

39. When a player clicks on a tile, we will set `tile1` for the first tile and `tile2` for the second time they click:

```
@property UIButton *tile1;
@property UIButton *tile2;
```

40. These are our private methods that we are using in our class. We are just going to declare methods here along with any arguments they have (these don't have any) and what they return (these three return nothing or void) with the help of the following code snippet:

```
- (void)shuffleTiles;
- (void)resetTiles;
- (void) winner;
@end
```

41. This is where our methods get implemented. As we declared everything in either the header file or in the private interface section, this is where we actually make everything work. Let's take a look:

```
@implementation GameSceneViewController
```


www.it-ebooks.info

42. These are just a couple of local Boolean variables that we are going to use in our code. The `isDisabled` variable is set to `true` when you click a second tile to stop the player from clicking on tiles before we confirm we have a match. The `isMatch` variable is set to `true` when the second tile is clicked and it matches the first tile clicked:

```
static bool isDisabled = false;
static bool isMatch = false;
```

43. The `viewDidLoad` method, as we mentioned earlier, is called as soon as our game scene is done loading:

```
- (void)viewDidLoad
{
  [super viewDidLoad];
```

44. This is where we assign an `UIImage` class to our two image properties and map our image file by name to that class:

```
  self.backTileImage = [UIImage imageNamed:@"back.png"];
  self.blankTileImage = [UIImage imageNamed:@"blank.png"];
```

45. We initialize `self.tileFlipped` to `-1` so our code knows we have not clicked on a first tile yet:

```
  self.tileFlipped = -1;
```

46. Set our counters to `0`:

```
  self.matchCounter = 0;
  self.guessCounter = 0;
```

47. We are going to set our game score label's text to show the number of matches and the number of guesses. You will notice `%d` in the text is exactly like the notation from `printf` that we used earlier. In this example, the `matchCounter` number will appear where the first `%d` specifier is in the text string, and the guess counter will appear where the second `%d` specifier occurs:

```
  self.gameScoreLabel.text = [NSString stringWithFormat:@"Matches:
%d Guesses: %d", self.matchCounter, self.guessCounter];
```

48. This is where we instantiate our first mutable array. First, we allocate `NSMutableArray`. Then, we initialize it with all of our `UIImage` objects, as shown in the next code snippet. When we are done with our entire list, we send a `nil` to terminate the array. It is important to remember that when retrieving setting information in an array, start counting from zero, not one.

```
  self.tiles = [[NSMutableArray alloc]initWithObjects:
            [UIImage imageNamed:@"icons01.png"],
            [UIImage imageNamed:@"icons01.png"],
```


www.it-ebooks.info

```
                [UIImage imageNamed:@"icons02.png"],
                [UIImage imageNamed:@"icons02.png"],
                [UIImage imageNamed:@"icons03.png"],
                [UIImage imageNamed:@"icons03.png"],
                [UIImage imageNamed:@"icons04.png"],
                [UIImage imageNamed:@"icons04.png"],
                [UIImage imageNamed:@"icons05.png"],
                [UIImage imageNamed:@"icons05.png"],
                [UIImage imageNamed:@"icons06.png"],
                [UIImage imageNamed:@"icons06.png"],
                [UIImage imageNamed:@"icons07.png"],
                [UIImage imageNamed:@"icons07.png"],
                [UIImage imageNamed:@"icons08.png"],
                [UIImage imageNamed:@"icons08.png"],
                [UIImage imageNamed:@"icons09.png"],
                [UIImage imageNamed:@"icons09.png"],
                [UIImage imageNamed:@"icons10.png"],
                [UIImage imageNamed:@"icons10.png"],
                [UIImage imageNamed:@"icons11.png"],
                [UIImage imageNamed:@"icons11.png"],
                [UIImage imageNamed:@"icons12.png"],
                [UIImage imageNamed:@"icons12.png"],
                [UIImage imageNamed:@"icons13.png"],
                [UIImage imageNamed:@"icons13.png"],
                [UIImage imageNamed:@"icons14.png"],
                [UIImage imageNamed:@"icons14.png"],
                [UIImage imageNamed:@"icons15.png"],
                [UIImage imageNamed:@"icons15.png"],
               nil];
```

49. The last thing we do is shuffle the arrays. To keep this a little cleaner, I created a method to do this. As the method is in this class, we use the `self` class reference to call the `shuffleTiles` method, as shown in the next code snippet. Once this is done, our code will just sit around until someone touches a tile.

```
  [self shuffleTiles];
}
```

50. And here is our `shuffleTiles` method we just called from `viewDidLoad`:

```
- (void)shuffleTiles
{
```


www.it-ebooks.info

51. We need to know how many tiles we are going to shuffle. I could have written 30, but if I ever changed my code to only 20 tiles, I would need to make sure I changed that number here as well. Instead, let's ask the tile's mutable array to give a count of how many items are in its array. Then, we assign that number to the `tileCount` variable, as shown in the next code snippet. As this variable is defined inside the `shuffleTiles` method, when we are done with this method, that variable will disappear from memory. The same goes for the `tileID` and `i` variables as well.

```
int tileCount = [self.tiles count];
```

52. We are going to create a `for` loop to populate our `shuffledTiles` array with numbers. We do this by initializing a local variable `tileID` and setting it to `0`. We tell the loop that we want to keep looping while `tileID` is less than our variable `tileCount` divided by two. We also tell the loop to increment `tileID` by one each time we loop back around. Every time we go through our loop, we are going to add a number object to our `shuffledTiles` mutable array. When we are adding a number to the `shuffledTiles` array, we need to populate it with objects (as it is `NSMutableArray`), so we can't just use an `int` variable; we will have to convert it to `NSNumber`. When we are done with the loop, our `shuffledTiles` array will be filled with number objects counting from zero to fourteen two times. It will look like `[0,0,1,1,2,2 …13,13,14,14]`. The code for this `for` loop is shown as follows:

```
for (int tileID = 0; tileID < (tileCount/2); tileID++)
{
  [self.shuffledTiles addObject:[NSNumber
numberWithInt:tileID]];]];
  [self.shuffledTiles addObject:[NSNumber
numberWithInt:tileID]];]];
}
```

53. Next, we are going to do the same type of the `for` loop, but this time we are going to count to 30. In this loop, we are going to count down an integer `nElements` from 30 with a variable integer `n` that will pick a random number between zero and `nElements`, and we are going to add the current loop number to it. In the next two lines, we take `NSMutableArrays` and swap the objects from position `i` and replace them with object `n`:

```
for (NSUInteger i = 0; i < tileCount; ++i) {
  NSInteger nElements = tileCount - i;
  NSInteger n = (arc4random() % nElements) + i;
  [self.shuffledTiles exchangeObjectAtIndex:i
withObjectAtIndex:n];
```


www.it-ebooks.info

```
    [self.tiles exchangeObjectAtIndex:i withObjectAtIndex:n];
  }
}
```

54. Let's take a look at what we want to do when a player touches one of our tiles. We already know that when a player touches or clicks on our tiles, it calls the `tileClicked` function and sends us its sender object with it:

```
-(IBAction)tileClicked:(id)sender
{
```

55. If the clicks are disabled, exit out of our method as shown in the next code snippet. You will see later in this code where we disable our clicks and how we turn our clicks back on.

```
  if(isDisabled == true)
    return;
```

56. Let's get the tag ID of the button the user clicked and assign it to our `senderID` local variable, as shown in the next code snippet. We are going to need this throughout this method.

```
  int senderID = [sender tag];
```

57. We need to check to see if `self.tileFlipped` has been set to a button's ID yet and `senderID` is not equal (`!=`) to `self.tileFlipped`; we can then continue with this block of code. If these are both `true` statements, we click on our second unique tile, flip it, and compare it to the previous tile. If they match, do something; if not, flip them back over. The code to perform these operations is as follows:

```
  if(self.tileFlipped >= 0 && senderID != self.tileFlipped)
  {
```

58. As we already know someone clicked on the first tile before we got here, we are going to load the current sender's information into the `tile2` button:

```
    self.tile2 = sender;
```

59. Next, we are going to load the image information from the `self.tiles` array at the `self.tileFlipped` position and the image information from the `self.tiles` array at the `senderID` position:

```
    UIImage *lastImage = [self.tiles objectAtIndex: self.
tileFlipped];
    UIImage *tileImage = [self.tiles objectAtIndex:senderID];
```

60. Calling the sender class, we are going to set the image for the clicked tile to the `tileImage` we just collected:

```
    [sender setImage: tileImage forState:UIControlStateNormal];
```


www.it-ebooks.info

```
Increase the guessCounter by 1.
    self.guessCounter++;
```

61. Check whether our two images match:

```
    if(tileImage == lastImage)
    {
```

62. Disable our two tiles so that no one can use them again:

```
      [self.tile1 setEnabled:false];
      [self.tile2 setEnabled:false];
```

63. As the tiles match, increase `matchCounter` by 1:

```
      self.matchCounter++;
```

64. Save a variable for outside the method to let the code know we have a match:

```
      isMatch = true;
```

65. Set the `isDisabled` variable to `true` to prevent anyone from clicking on a tile while we are waiting for our timer to finish, as shown:

```
    isDisabled = true;
```

66. We want to set up a timer that will flip our tiles over, but we want to wait a second so that the player can see the tiles before they flip back over or get removed from the game play area.
67. This is our timer class. The `NSTimer` class is a very powerful and useful class but still very easy to understand. We are going to set our `NSTimer` class to wait for `1.0` seconds. After that, we are going to call the class `self` and the method `resetTiles` without `userInfo` and we are not going to repeat, as shown in the following code:

```
    [NSTimer scheduledTimerWithTimeInterval:1.0
                                     target:self
                                   selector:@selector(resetTiles)
                                   userInfo:nil
                                    repeats:NO];
```

68. Set our `self.tileFlipped` property back to `-1` so that our code knows that we are back to looking for the first tile in a clicked set:

```
    self.tileFlipped = -1;
  }
  else
  {
```


www.it-ebooks.info

69. So, referring back to our `if` statement, if our `self.tileFlipped` property is less than zero, we need to flip over our first of two tiles. Set `self.tileFlipped` to `senderID` that we picked up earlier, and set `self.tile1` to the sender coming in from the method:

```
self.tileFlipped = senderID;
self.tile1 = sender;
```

70. Get `tileImage` from the `self.tiles` array at position `senderID`:

```
UIImage *tileImage = [self.tiles objectAtIndex: senderID];
```

71. Set the image of the sender button to `tileImage`:

```
    [sender setImage: tileImage forState:UIControlStateNormal];
  }
```

72. Whether you have a match or not, make sure you update your game information for the user to see by performing the next code snippet. This is the same string you saw earlier in the code.

```
  self.gameScoreLabel.text = [NSString stringWithFormat:@"Matches:
%d Guesses: %d", self.matchCounter, self.guessCounter];
}
```

73. This is the method that gets called when our timer, from the `tileClicked` method, runs out:

```
- (void)resetTiles
{
```

74. If we set `isMatch` to true before we called the timer, we set the images to blank; otherwise, we set the images to the flipped over tile image:

```
  if(isMatch)
  {
    [self.tile1 setImage: self.blankTileImage
forState:UIControlStateNormal];
    [self.tile2 setImage: self.blankTileImage
forState:UIControlStateNormal];
  }
  else
  {
    [self.tile1 setImage: self.backTileImage
forState:UIControlStateNormal];
    [self.tile2 setImage: self.backTileImage
forState:UIControlStateNormal];
  }
```


www.it-ebooks.info

75. Reset our variables so that we are ready to start the next round:

```
  isDisabled = false;
  isMatch = false;
```

76. If we got 15 correct guesses, we have reached the end and we will call our winner code:

```
  if(self.matchCounter == (self.tiles.count/2))
    [self gameWon];
}
```

77. This is our winner code. The `viewWillDisappear` method is what gets called in response to a view getting removed:

```
- (void)  gameWon
{
    self.gameScoreLabel.text = [NSString stringWithFormat:@"You
won with %d Guesses", self.guessCounter];

}

-(void) viewWillDisappear:(BOOL)animated
{
  [super viewWillDisappear:animated];
}

@end
```

Wow, that's a huge leap from a simple `Hello World!` demo. Do you get the feeling that you've learned an enormous amount about iPhone programming at this point?

Summary

In this chapter, we've taken a look at views and scenes and produced a basic game. These will provide the basis for our iPhone game going ahead throughout the rest of the book.

In our next chapter, we will discuss graphics programming. We will show you how to display graphics programmatically as well as how to animate and move your graphics around the screen. We will use what we have learned to create a single scene Space-Invaders-style game.

www.it-ebooks.info

4

Drawing Pictures – Space Invaders

In this chapter, we will create a Space Invaders-style game. In order to build it, we will need to learn a bit about graphics, animation, screens, displays, and coordinate systems. By the end of this chapter, you will be able to run your Space Invaders-like game in your simulator or iOS device.

In this chapter, we will cover the following topics:

- Understanding screens and graphics
- Adding graphics programmatically
- Moving our ship sprite
- Creating our enemies
- Using a sprite sheet
- Creating the enemy bombs

Understanding screens and graphics

Graphics are fairly easy to understand once you have grasped a few concepts. Let's start small, very small. A **pixel** has become the standard unit of measurement when discussing computer graphics and monitors. If you were to zoom in to a graphic image using any graphics program, eventually, you would zoom in to the point where the image is constructed of multiple squares, in other words, pixels. Monitors and screens work in the same way; if you were to use a magnifying glass to look at your monitor, you would see that it is constructed of many pixels as well. On iOS devices, Apple decoupled the screen's coordinate systems from the device by using points instead of pixels. By doing this, you now have a logical coordinate system which is not tied to the display.

www.it-ebooks.info

The result is that if you have two devices with different resolutions, the content you place at a specific coordinate on the screen will appear in relatively similar positions on both devices.

Different iPhone models have different screen sizes or resolutions. The iPhone 4's screen size is 640 by 960 pixels or 640 x 960. The screen size of the iPhone 5 is 640 x 1136. If we go back to the iPhone, 3, you will see that the screen size is only 320 x 480. Since iPhones have different screen sizes, you will need to make the sizes of multiple images match with the different iPhone screen sizes. While creating a new project in Xcode, you will notice that there are several launch image graphics that are automatically created for you to match these sizes. The images have a standard image and a duplicate image with a @2*x* notation added on to the name, which is twice the size. When you create an image with the @2*x* notation, it will automatically be loaded by the system in place of your standard image.

It's helpful to know that as you place an image on the screen, you are usually going to measure from the top-left corners (although on rare occasions, it is from the bottom-left). Let's take, for example, a solid square image that is 64 x 64 pixels. We want it to be shown 20 points from the left and 10 points from the top. Let's start drawing the square with a dimension of `[20, 10]`. The rectangle would be programmed as `CGRectMake(20,10,64,64)`—you will see me use this throughout our following examples.

In our games, we are not going to create the graphics; after all, we are not teaching you graphics, we are teaching you how to program.

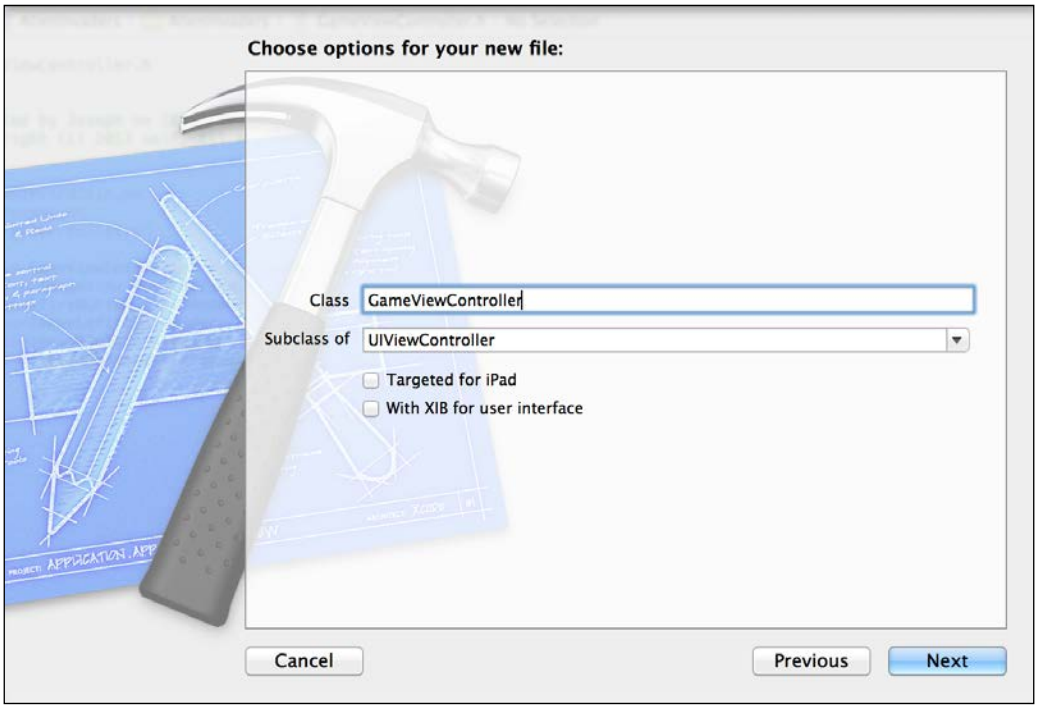
Adding graphics programmatically

In the previous chapter, you learned how to add a graphic to your scene using the storyboard with drag-and-drop methods. Unfortunately, this will not work if you want to create a graphic from a list of images, duplicate it, and place it in a very specific place on the screen. For that, we need to use code. Let's start building our next app.

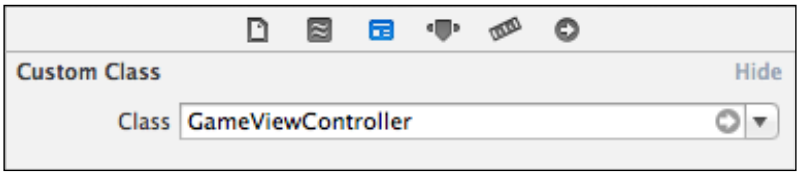
Start a new **Single View Application** project and call it `AlienInvaders` using the same method we used in the previous chapters. Set your build scheme to **iPhone Retina (3.5-inch)**. A build scheme is a predefined set of targets that we are setting our app to build for. In this case, we are setting our app to target the 3.5 inch Retina iPhone. We can always change these settings in our project if we ever need to.

www.it-ebooks.info

Create a new class file called `GameViewController` with a `UIViewController` subclass, as shown in the following screenshot:

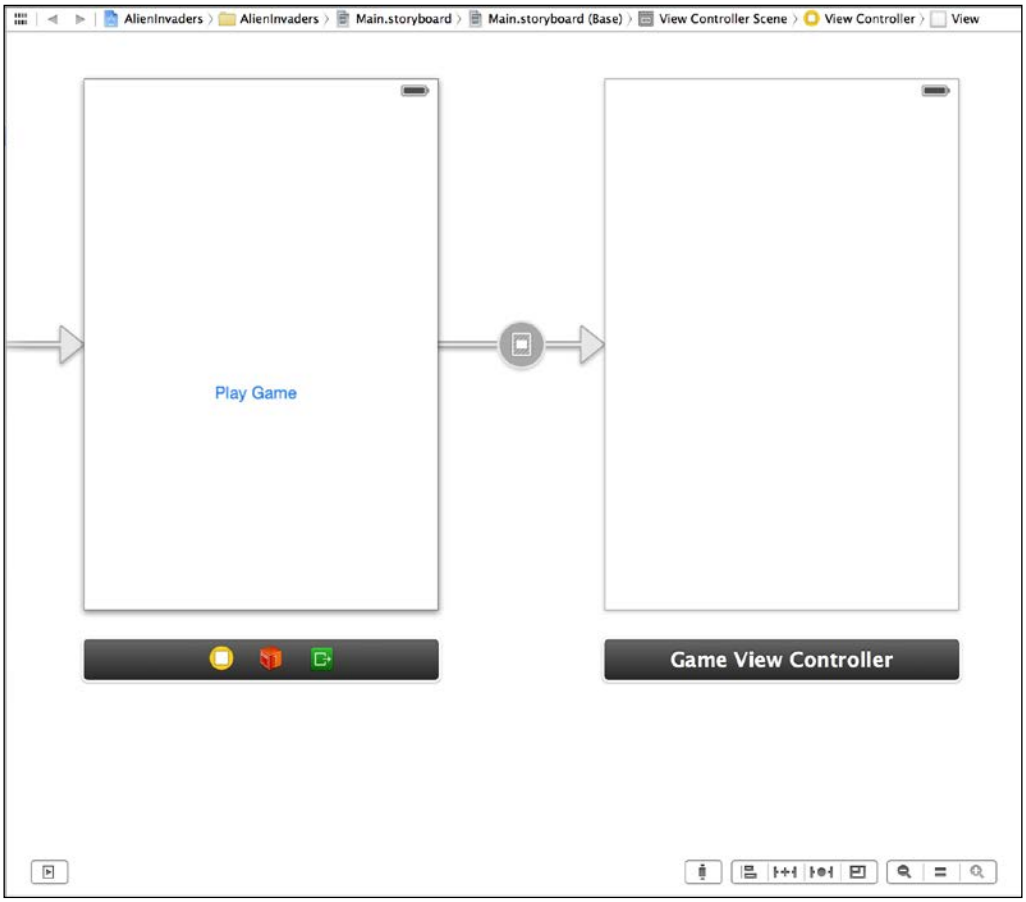


In our storyboard, add a new view controller object and set its class to **GameViewController,** as shown in the following screenshot:



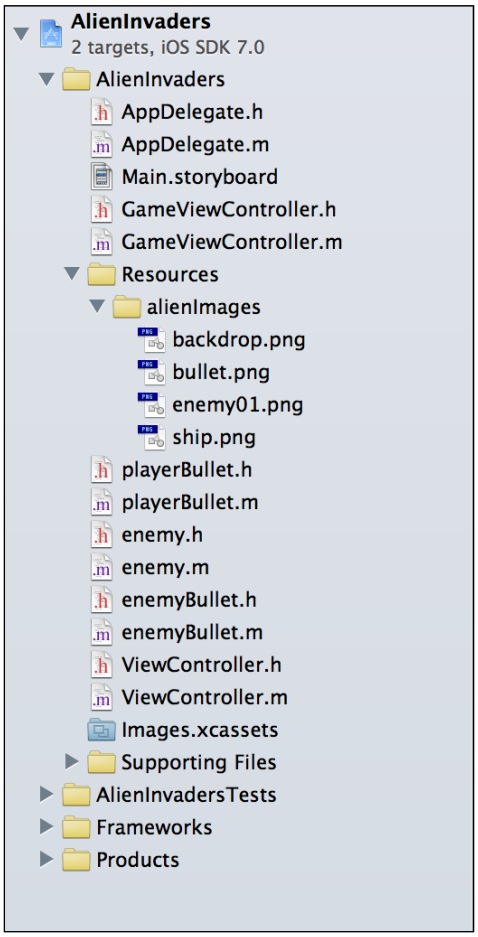
www.it-ebooks.info

Add a button called **Play Game** and create a modal segue between the main view controller and the new view, as shown in the following screenshot:

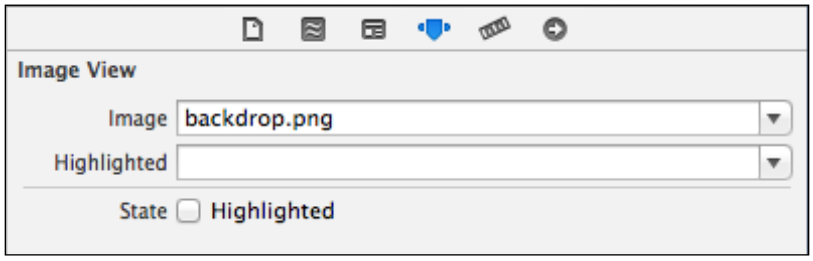


www.it-ebooks.info

Import our graphics into the `alienImages` folder as shown in the following screenshot:

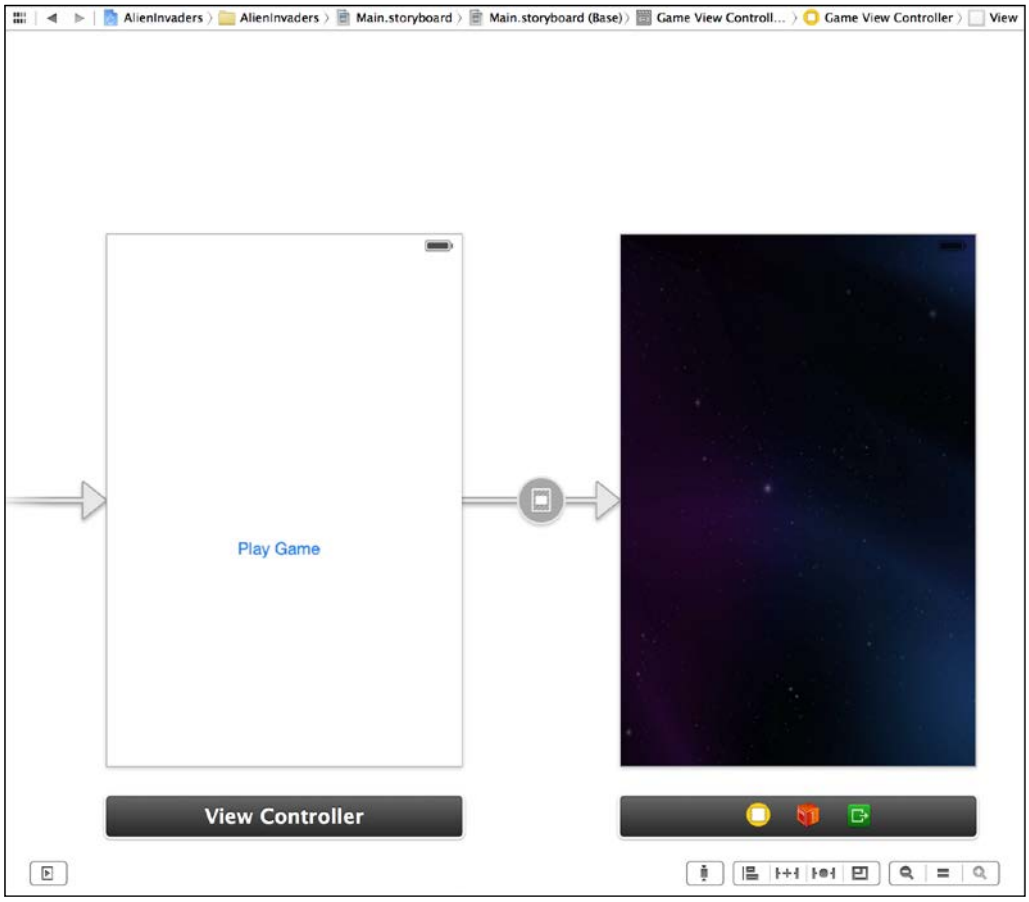


We are going to go ahead and add a background to our game view. Add an image view from the object library and enlarge it to the full size of the game view. Set your image to **backdrop.png** as shown in the following screenshot:



www.it-ebooks.info

Stretch and position your image so that it fills the screen of **Game View Controller**, as shown in the following screenshot:



At this point, we are ready to start programming. Let's start by adding a property for an image and an image view for a player's ship into our private interface, as follows:

```
@interface GameViewController ()
@property (strong) UIImage *playerImage;
@property (strong) UIImageView *playerView;
@end
```


www.it-ebooks.info

In our `viewDidLoad` method, we are going to assign the `ship.png` image to the `playerImage` property. We need to do this after `[super viewDidLoad]`, which calls the implementation of the super class `viewDidLoad` as follows:

```
self.playerImage = [UIImage imageNamed:@"ship.png"];
```

Now that we know what image is associated with the player, let's go ahead and assign the view for the player by initializing a new `UIImageView` container with that image. `UIImageView` is a container for displaying images. Once we have the image view, we can set its frame position and size using `CGRectMake` and then add it to Game View Controller as follows:

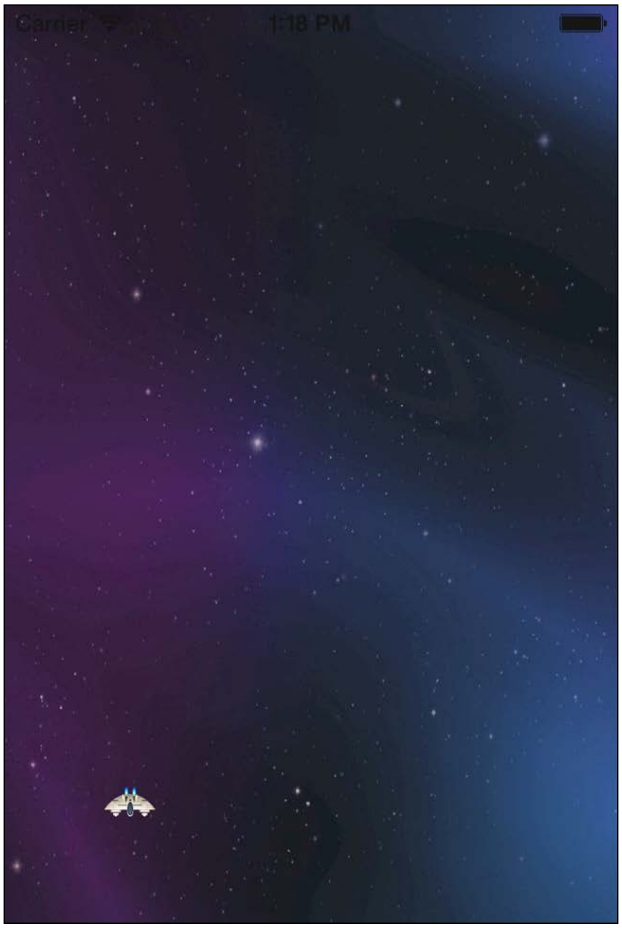
```
  self.playerView = [[UIImageView alloc] initWithImage:
    self.playerImage];
  self.playerView.frame = CGRectMake(50,400, 32,32);
  [self.view addSubview: self.playerView];
```

Let's review how we added an image to the screen just by using code. Assuming that everything went well, your code should look something similar to the following:

```
- (void)viewDidLoad
{
    [super viewDidLoad];
  //Assign an image to a variable
  self.playerImage = [UIImage imageNamed:@"ship.png"];

  //Create an image view and start it up with our image,
  self.playerView = [[UIImageView alloc] initWithImage: self.
playerImage];

  //Set the image views bounding rectangle to scale the image.
  self.playerView.frame = CGRectMake(50,400, 32,32);

  //Show the image on the screen.
  [self.view addSubview: self.playerView];
}
```


www.it-ebooks.info

Run the code and see how it looks. You should be able to see the player ship on the screen with the image of the sky as the background, as shown in the following screenshot:



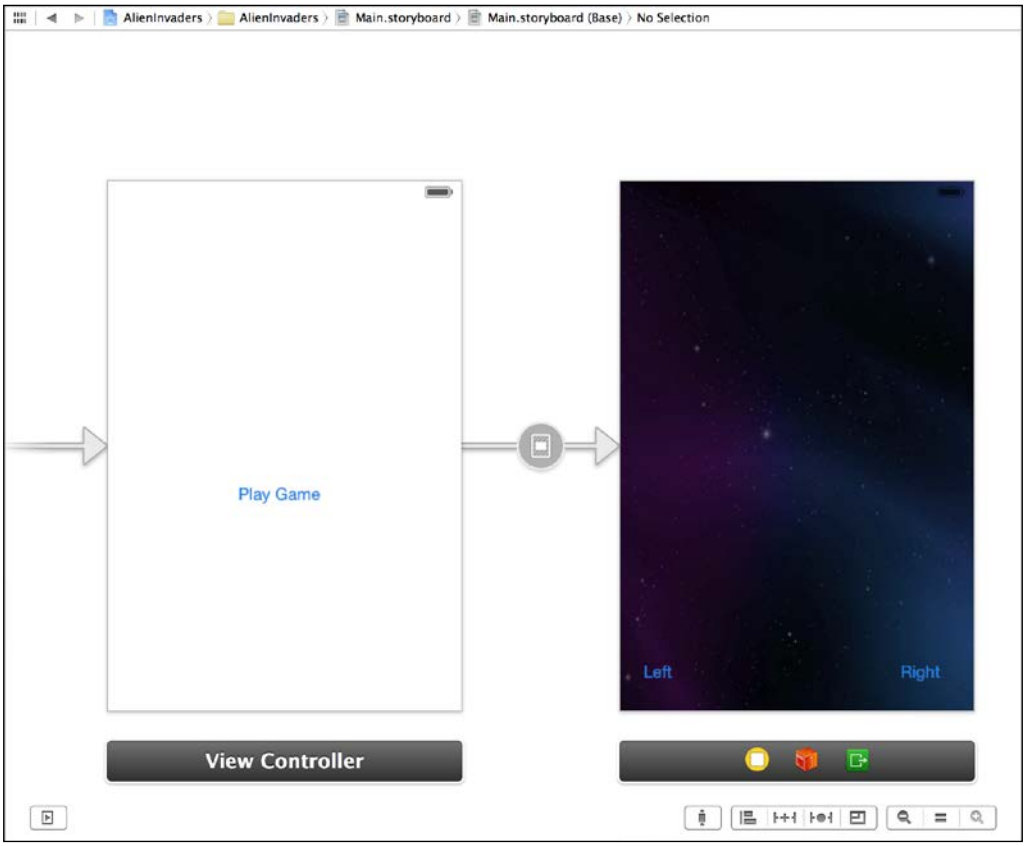
Moving our ship sprite

A **sprite** is a graphic/icon/image that is displayed inside a larger area of a screen. Think of it this way; if a fairy (or sprite) were flying around the screen with a background image of a princess castle, the sprite would still move independently of the background as an entity in itself.

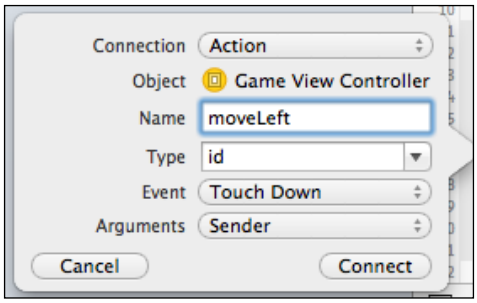
For our current game, we have a ship sprite that we want to move from left to right. For our game example, we are going to add in a move-left and move-right button. Our sprite will move along its *x* axis when the buttons are pushed.

www.it-ebooks.info

Let's start with creating the buttons. Create two button views in our storyboard, as shown in the following screenshot:

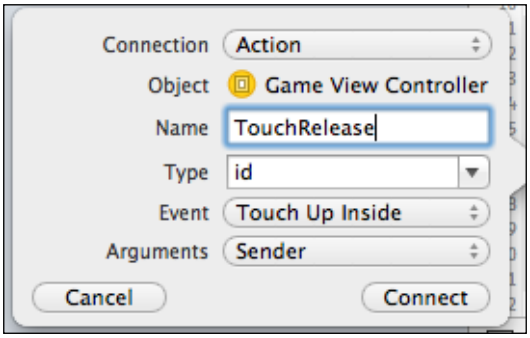


Open your assistant editor. Press the *Ctrl* key and click and drag your buttons to `GameViewController.h` in order to create button actions for each of your buttons. Call your left button's action `moveLeft` and your right button's action `moveRight`. Set the action's **Event** to **Touch Down** as shown in the following screenshot:



www.it-ebooks.info

Create another action named **TouchRelease** with the **Event** option set to **Touch Up Inside** and link it to both of your buttons. Once you have created the **TouchRelease** action, you can link the second button to it by holding the *Ctrl* key and clicking and dragging the button to the `TouchRelease(IBAction)` line of code in your `GameViewController.h` file. This is how we are going to tell our ship to stop moving when we release the button:



In our **GameViewController** interface, we are going to add a few properties for managing our bullets, as shown in the following code:

```
@interface GameViewController ()
@property CGRect playerRect;
@property (strong) UIImage *playerImage;
@property (strong) UIImageView *playerView;
@property (strong) NSTimer *moveTimer;
@end
```

In our `GameViewController` implementation code, we are going to add some code to the newly created methods. But first, we are going to initialize our player's image variables in our `viewDidLoad` method. We start by creating `UIImage` with the ship's image file, and then we create a view for the image to be displayed. Finally, we create a bounding rectangle for the player's image using `CGMakeRect` with the position and sizes for where the image will be displayed on the screen. We set the `playerView` object to be displayed in the `playerRect` area and then add the `playerView` object to the game's view using `addSubview`.

```
- (void)viewDidLoad
{
  [super viewDidLoad];

  //Assign an image to a variable
  self.playerImage = [UIImage imageNamed:@"ship.png"];
```


www.it-ebooks.info

```
    //Create an image view and start it up with our image,
    self.playerView = [[UIImageView alloc]
                  initWithImage:self.playerImage];

    //Set the image views bounding rectangle to scale the image.
    self.playerRect = CGRectMake(50,400, 32,32);
    self.playerView.frame = self.playerRect;

    [self.view addSubview:self.playerView];

    [[[Enemy alloc]init] initEnemies:self.view];
  }
```

Let's add three more methods to our code, which will be called from our button methods. Using timers, the first two methods are going to be called whenever the user presses on either of the buttons. The `releaseTouch` method will be called whenever we want to remove our timers. Each time the `movePlayerRight` or `movePlayerLeft` methods are called with the timer, we move the ship's position either three points to the left or right and then set the ship's frame to update the view on the screen. Use an `if` statement to prevent the ship from moving off the screen. The `releaseTouch` method tells our timer to invalidate the timer and set it to `nil` to clear out the property, as follows:

```
-(void) movePlayerRight{
  if(self.playerRect.origin.x <= 320){
    self.playerRect = CGRectOffset(self.playerRect, 3, 0);
    self.playerView.frame = self.playerRect;
  }
}
-(void) movePlayerLeft{
  if(self.playerRect.origin.x >= 10){
    self.playerRect = CGRectOffset(self.playerRect, -3, 0);
    self.playerView.frame = self.playerRect;
  }
}
-(void)releaseTouch{
  if(self.moveTimer != nil){
    [self.moveTimer invalidate];
    self.moveTimer = nil;
  }
}
```


www.it-ebooks.info

Now, we can set up the timers to be called from our buttons. For all three of our button actions, the first thing we must do is release any previous timers for the button presses. We do this by calling the `[self releaseTouch];` method. Then, for the `moveLeft` and `moveRight` methods, we start a timer that calls the `movePlayerLeft` or `movePlayerRight` methods 30 times per second and repeats until we invalidate the timer as follows:

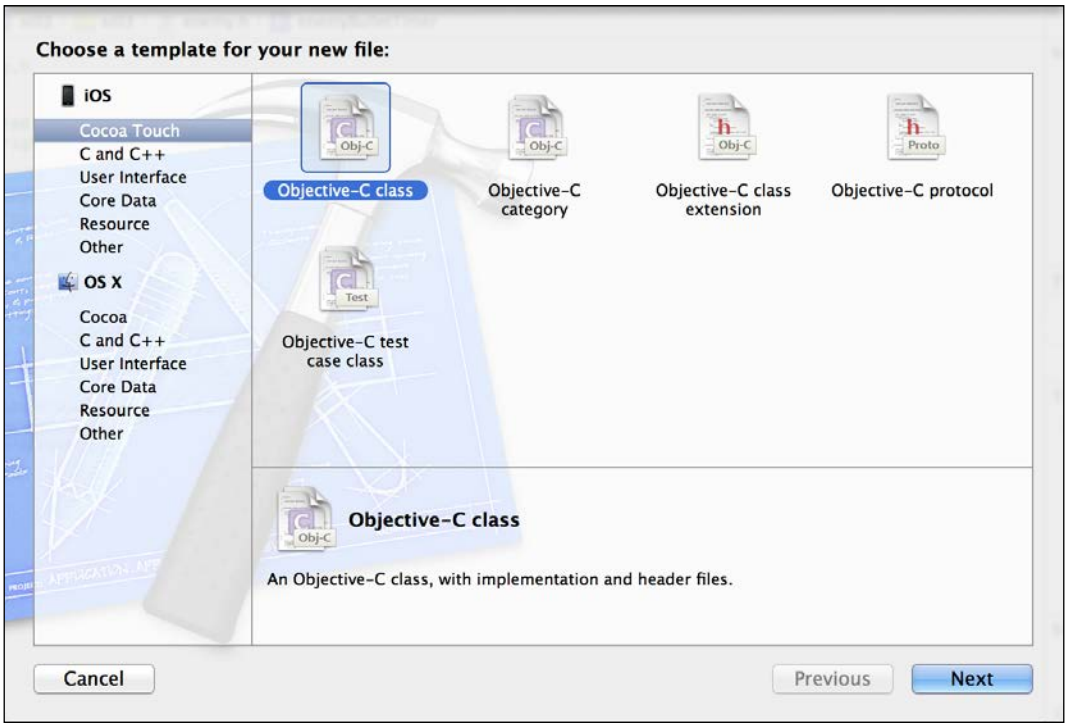
```
- (IBAction)moveLeft:(id)sender {
  [self releaseTouch];
  self.moveTimer = [NSTimer scheduledTimerWithTimeInterval:.03
                      target:self
                      selector:@selector(movePlayerLeft)
                      userInfo:nil
                      repeats:YES];
}

- (IBAction)moveRight:(id)sender {
  [self releaseTouch];
  self.moveTimer = [NSTimer scheduledTimerWithTimeInterval:.03
                      target:self
                      selector:@selector(movePlayerRight)
                      userInfo:nil
                      repeats:YES];
}

- (IBAction)touchRelease:(id)sender {
  [self releaseTouch];
}
```

You can run your code at this point, and by using the left and right buttons, you can see that you can animate your ship left and right. This is the most basic movement for animation as it is used in many different games. We are only animating along the *x* axis for now, but in later chapters, there will be enemies who will fall from the sky along a curved path. Even though this is a more complex path, the concept remains the same.

www.it-ebooks.info

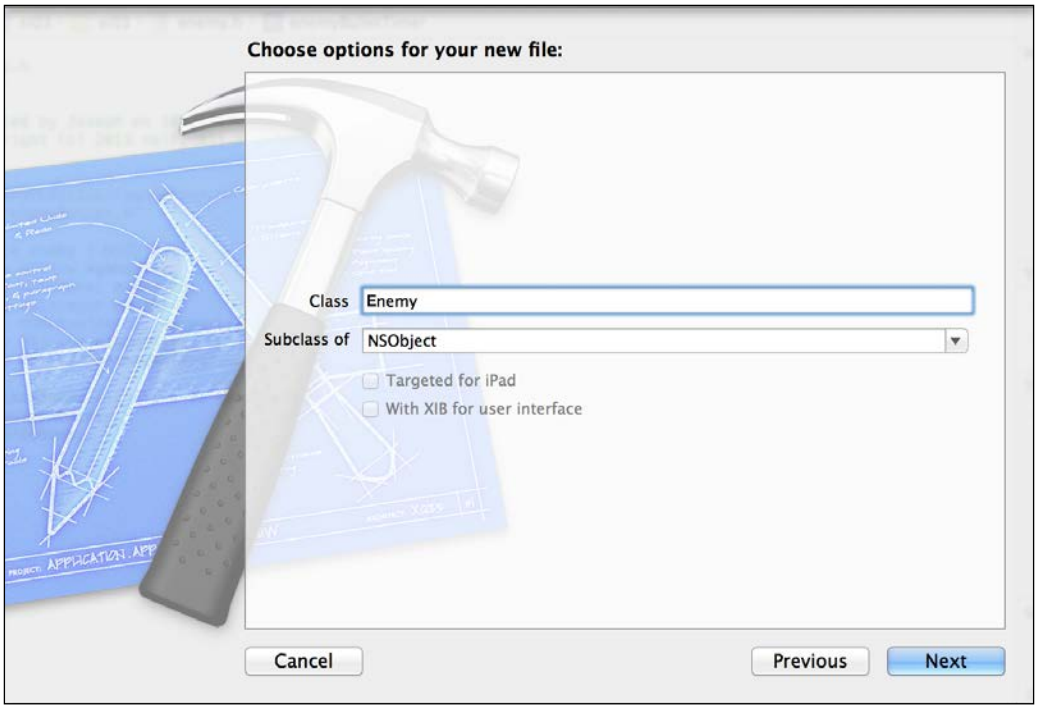
Creating our enemies

Cool! That was a great start. Now we need a bunch of enemies. We are going to start by creating a new class for our enemies. Under your **File** menu, add a new file. Choose the file type as **Objective-C class**, as follows:



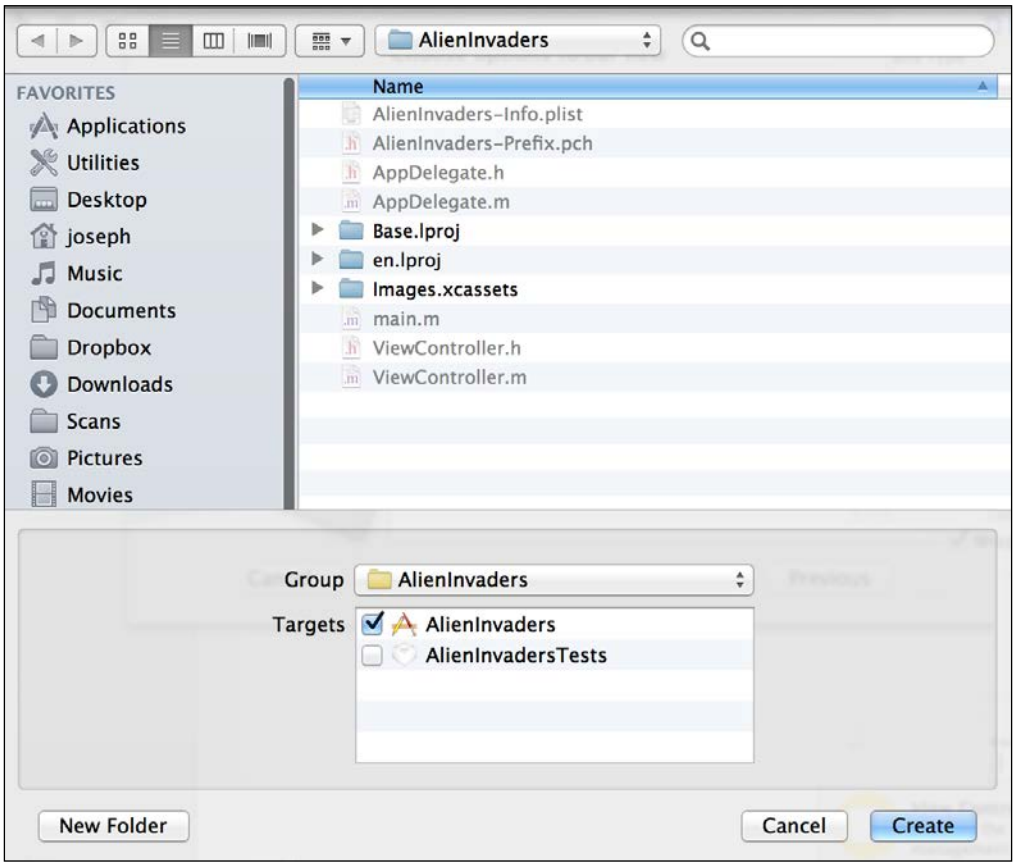
www.it-ebooks.info

Name your class `Enemy` and set the subclass to `NSObject` as shown in the following screenshot:



www.it-ebooks.info

Save your file to the default project directory as shown in the following screenshot:



Once this is done, we will link the `Enemy` class to our `GameViewController` class by using the `#import` command in our `GameViewController.m` implementation file. Add the `#import` command with the name of your enemy header's filename after where we link the `Enemy` class to the `GameViewController.h` file as follows:

```
#import "GameViewController.h"
#import "enemy.h"
```

This will create a direct link to our new enemy file from within our `GameViewController`, which will allow us to control our enemies as a new object. Let me show you.

We will start by setting up our `Enemy` class with a method that will initialize our enemies, save the game's view, and create a list of enemies. Then, we will show you how to call the new object and initialize it.

www.it-ebooks.info

In our header file, we will add the following code in our interface area:

```
@interface Enemy : NSObject
@property UIView *gameView;
@property NSInteger minXPos;
@property NSInteger maxXPos;
@property NSMutableArray *enemyList;

-(void) initEnemies:(UIView *) gameView;
@end
```

This will set up some properties for our minimum and maximum *x* positions that we want our enemies to stay within, a link to our game view, and our array of enemies. We are also going to add an `initEnemies` method that will contain an argument and the game view.

In our implementation file, we will add our `initEnemies` method so that we have something to call. We will not be returning anything, but our game view that we will be receiving will be `UIVew`, so we will define our method as follows:

```
-(void) initEnemies:(UIView *)gameView{

}
```

We will have to call this method just after our player is created, so let's go back to our `viewDidLoad` method in our `GameViewController` implementation file. At the end of our `viewDidLoad` method, add the following line of code:

```
[[[enemy alloc]init] initEnemies:self.view];
```

In this one line of code, we are allocating a new enemy object, initializing it, and calling our `initEnemies` method and passing it `UIView` which is associated with our `GameViewController` class.

Congratulations! You have just created your first complete custom class in Objective C and called it from another class. You are going to be doing this a lot in your programs, so it is very useful and important to learn and get used to this.

Now, let's get back to creating our enemies. In our enemy implementation file, we are going to add some variables. We are going to add these variables just below our `@implementation` definition as follows:

```
int eSize = 32;
bool goingLeft = false;
int enemyRows = 5;
int enemyColumns = 5;
```


www.it-ebooks.info

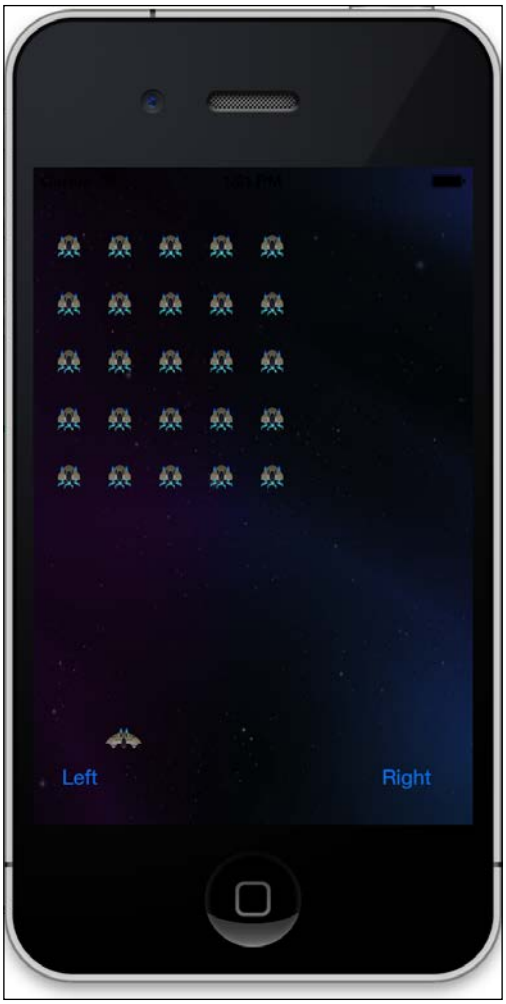
These variables tell us the size of our enemies in pixels, whether they are going left or not, how many rows of enemies there are, and how many columns of enemies there are. Feel free to play around with these variables once we have our enemies working.

Using the same concepts that we used for creating our player, we are going to make our method dynamically generate a multitude of enemies. Let's add the following code to generate all of our enemies:

```
-(void) initEnemies:(UIView *)gameView{
  self.minXPos = 10;
  self.maxXPos = 278;
  self.gameView = gameView;

  //keep track of which row has been created
  int rowCount=0;
  //start creating the enemies this position from the left
  int startX = 10;
  //start creating the enemies this position from the top
  int startY = 0;

  //define the enemy image
  UIImage *enemyImage = [UIImage imageNamed:@"enemy01.png"];
  self.enemyList = [[NSMutableArray alloc]init];

  //loop for each enemy
  for(int i = 0; i < (enemyRows * enemyColumns); i++){
    //use modulo to calculate the column position for each enemy
    int columnMod = i % enemyColumns;
    if(columnMod == 0)
      rowCount++;
    //caluclate the frame position for the enemy
    int xPos = startX + ((eSize*columnMod)+(columnMod*5));
    int yPos = startY + ((eSize*rowCount)+(rowCount*10));

    //create an image view
    UIImageView *enemyView = [[UIImageView alloc]initWithImage:
      enemyImage];
    //set the position and size for the newly created enemy
    enemyView.frame = CGRectMake(xPos, yPos, eSize, eSize);

    [self.enemyList addObject: enemyView];

    //add the new enemy to the game view
    [self.gameView addSubview:enemyView];
  }
}
```


www.it-ebooks.info

In this method, we are basically doing the same thing we did when creating our player. The only difference is that we are looping through 25 different enemies and moving their starting positions on the screen. This gives us a pretty good sense of where our game is going to end up graphically, as shown in the following image:



www.it-ebooks.info

One way in which our enemies are drastically different is the manner in which they are moved. We obviously can't have our player controlling our enemies, which leaves it up to the code to control them. Let's create a new method to move our enemies. Start by adding the `moveEnemies` declaration just below our `initEnemies` declaration in our `Enemy.h` header file as follows:

```
-(void) moveEnemies;
```

Next, we are going to create the method in our implementation file. This is where the `goingLeft` variable is going to come in. We are going to start by checking the position of our left-most and right-most enemies in our `enemyList` array to make sure they are not moving off the screen. If they are getting close, we will make them turn around in their next move. The next thing we do is create a `for` loop that cycles through all of our enemies and moves them either +3 point to the right or -3 point to the left. We update the frame and the game view just like we did with our player. That's it for movement, and this is shown in the following code:

```
-(void) moveEnemies{
  UIImageView *enemyView = self.enemyList[0];
  if(enemyView.frame.origin.x <= self.minXPos){
    goingLeft = false;
  }

  enemyView = self.enemyList[enemyColumns-1];
  if(enemyView.frame.origin.x >= self.maxXPos){
    goingLeft = true;
  }

 //cycle through all of the enemies and
  //move them left or right by 3

  for(int i=0;i< self.enemyList.count; i++){
    enemyView = self.enemyList[i];
    int xPos;

    if(goingLeft){
      xPos = enemyView.frame.origin.x-3;
    }else{
      xPos = enemyView.frame.origin.x+3;
    }

    enemyView.frame = CGRectMake(xPos, enemyView.frame.origin.y,
eSize, eSize);

    //add the new enemy to the game view
    [self.gameView addSubview:enemyView];
  }
}
```


www.it-ebooks.info

We are going to add a movement timer to our class after we initialize our enemies. Furthermore, we are going to add a property to our enemy header for the timer. You can just add this line of code below our `enemyList` definition as follows:

```
@property NSTimer *enemyTimer;
```

This will continuously call our `moveEnemies` method 30 times a second until we end our game. Just add the timer to the last line of our `initEnemies` method as follows:

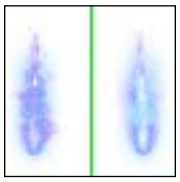
```
self.enemyTimer = [NSTimer scheduledTimerWithTimeInterval:.03
                       target:self
                       selector:@selector(moveEnemies)
                       userInfo:nil
                       repeats:YES];
```

Using a sprite sheet

When sprites were first being used for animation, a technique was created to reduce the time necessary to load and unload images from memory. Programmers (or artists) would create one large image with all of the images needed for that sprite. This became known as a **sprite sheet**.

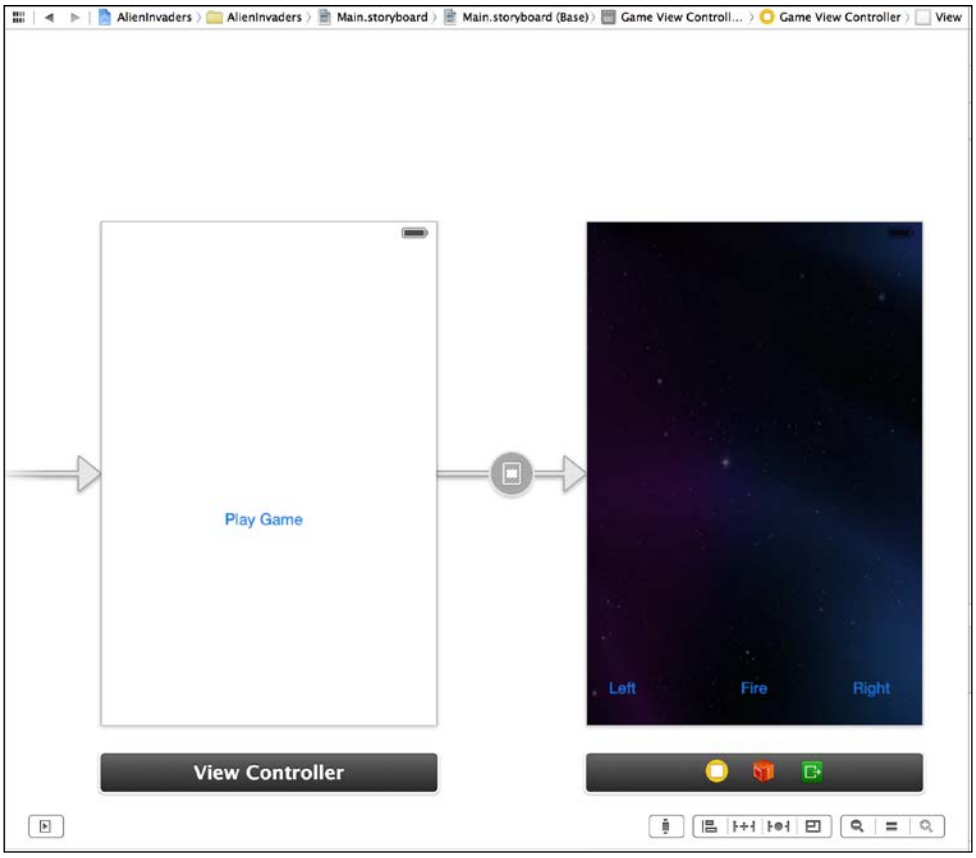
In this next section, we are going to take a simple image and save it as two different images, flipping back and forth between the two to give the illusion of an animated bullet. This should give you a glimpse of how larger sprite sheets might work with something along the lines of character animation.

This is our bullet image file. To give the effect of animation, we will create one image of the left bullet and another image for the right bullet. Then, we will create an image view that will flip between the two images:



Let's begin our next task by adding a **Fire** button. In your storyboard editor, add a new button and align it in the center of your screen. Name the button **Fire** as shown in the following screenshot:

www.it-ebooks.info



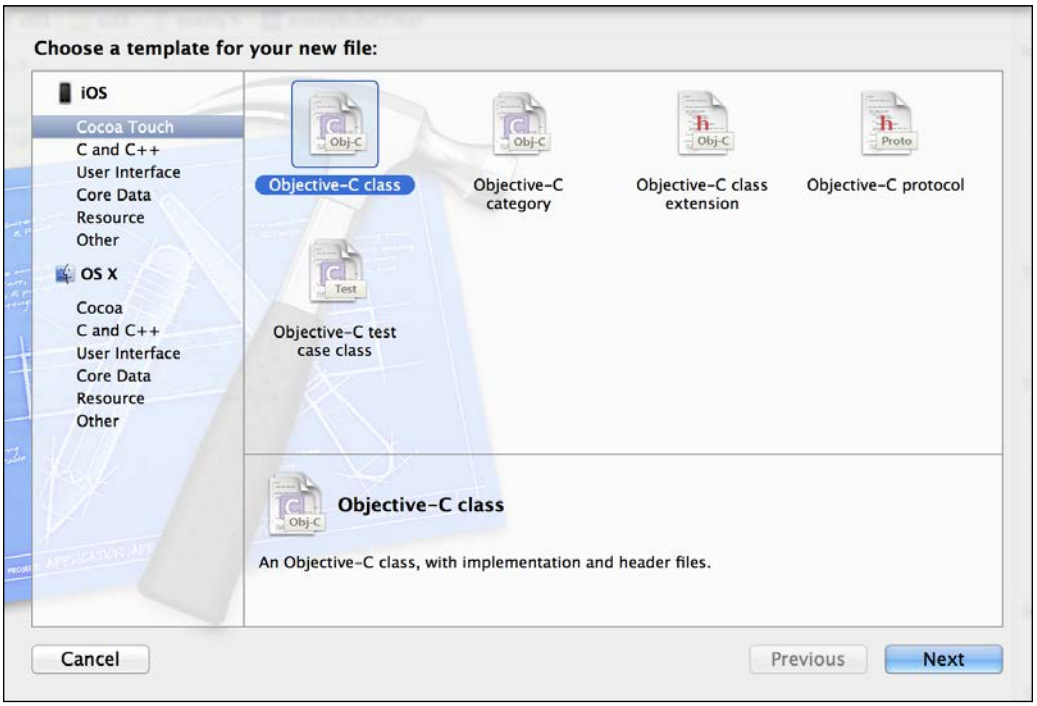
Make sure your assistant editor is still up and running and press the *Ctrl* key and click on your fire button to create a new button action. Call your button's action `fireButton`. Set the action's **Event option** to **Touch Down,** as shown in the following screenshot:

www.it-ebooks.info

Now, your header file should look something like the following code:

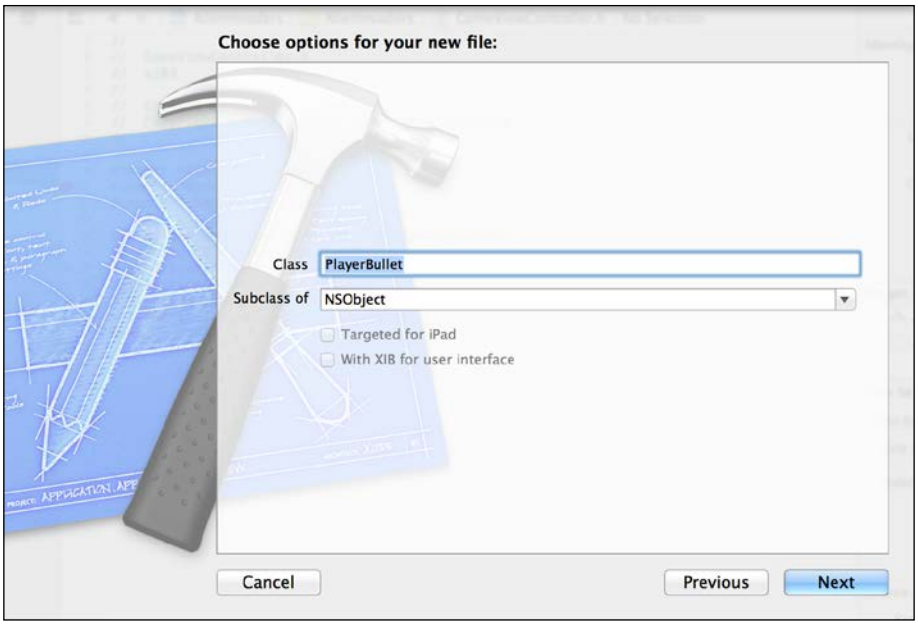
```
#import <UIKit/UIKit.h>
#import "enemy.h"

@interface GameViewController : UIViewController
@property UIImageView *bulletView;
- (IBAction)moveLeft:(id)sender;
- (IBAction)moveRight:(id)sender;
- (IBAction)touchRelease:(id)sender;
- (IBAction)fireButton:(id)sender;
@end
```

We are going to add another class to our game. The purpose of this class is to manage our player's bullets, and it will allow us to create multiple instances of the same bullet. Under your file menu, add a new file. Choose the file type as **Objective-C class**, as shown in the following screenshot:

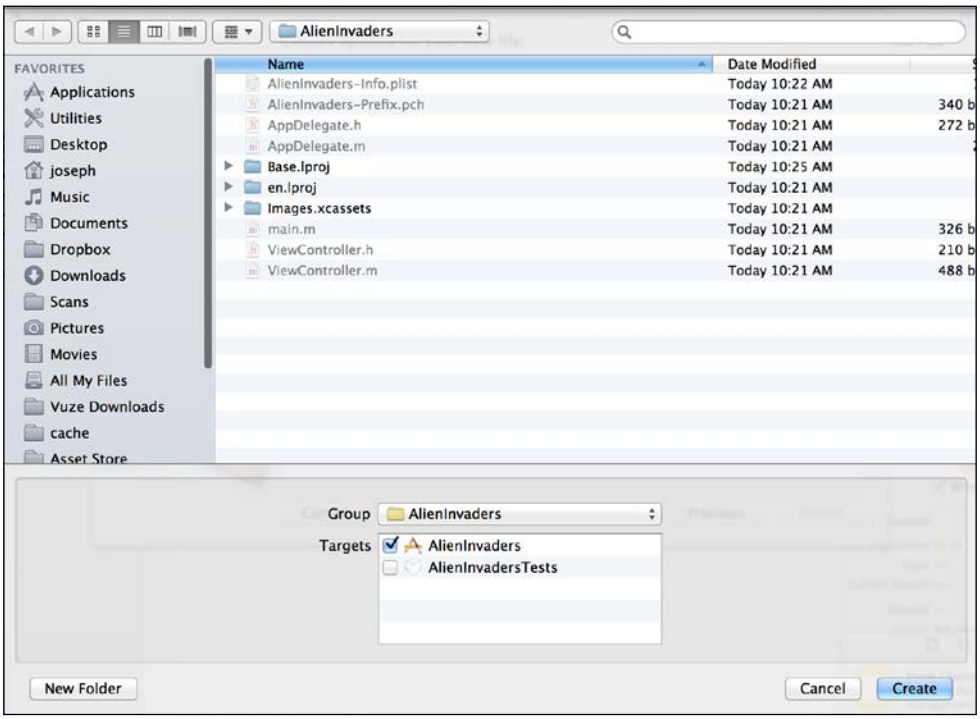


Name your class `PlayerBullet` and set the subclass to **NSObject** as shown in the following screenshot:

www.it-ebooks.info



Save your file to the default project directory as follows:



www.it-ebooks.info

Once this is done, we will link the `playerBullet` class to our `GameViewController` class by adding `#import` to our `GameViewController` file:

```
#import "GameViewController.h"
#import "enemy.h"
#import "playerBullet.h"
```

We will have to set up `playerBullet` before we can continue in `GameViewController`. Start by setting up the header file. The properties that we are adding are for the rectangle around the bullet, the image view for the bullet, and a timer to manage the bullet's motion. We are also passing the game's view from `GameViewController`:

```
#import <Foundation/Foundation.h>

@interface playerBullet : NSObject
@property CGRect bulletRect;
@property UIImageView *bulletView;
@property (strong) NSTimer *bulletTimer;

@property UIView *gameView;
-(void) fireBullet: (UIView *) gameView :(UIImageView *)
  playerView;
@end
```

In our `PlayerBullet` implementation file, we can go ahead and add our initialization method that we declared in our interface as follows:

```
-(void)fireBullet:(UIView *)gameView :(UIImageView *) playerView {
}
```

In our `GameViewController` implementation, we are going to tell our `fireButton` method to call our new `fireBullet` method in the `playerBullet` class. This will create a new bullet every time we push the fire button as follows:

```
- (IBAction)fireButton:(id)sender {
  [[[playerBullet alloc]init] fireBullet: self.view: self.playerView
];
}
```

We are again going to go back to our `playerBullet` implementation file and add the rest of our `fireBullet` code. Our `fireBullet` code will create a new image based on the `bullet.png` file we showed earlier:

```
-(void) fireBullet: (UIView *) gameView :(UIImageView *) playerView{
  self.gameView = gameView;
```


www.it-ebooks.info

```
//get our bomb image
UIImage *bombImage1 = [UIImage imageNamed:@"bullet.png"];
```

Let's find the start location for our bullet. In this case, we want to find out where the center of the player's ship is at the moment they hit the fire button and launch the bullet from there. This is done with the help of the following code:

```
//get the center x position of the player's ship
int bulletWidth = 8;
int bulletStartX = playerView.frame.origin.x +(
  playerView.frame.size.width/2)-(bulletWidth/2);
```

Next, get the bullet image, cut it into two images, and add them to an array. Create an image view in which the bullet to play inside. Set the image view's `animationImages` to the array as follows:

```
//make two images from the bomb image using these rects
  CGImageRef imageRef = CGImageCreateWithImageInRect([bombImage1
CGImage], CGRectMake(0, 0, 32, 64));
  CGImageRef imageRef2 = CGImageCreateWithImageInRect([bombImage1
CGImage], CGRectMake(33, 0, 32, 64));

  //create an array of both images for the animation
  NSMutableArray *bombArray = [[NSMutableArray alloc]init];
  [bombArray addObject:[UIImage imageWithCGImage:imageRef]];
  [bombArray addObject:[UIImage imageWithCGImage:imageRef2]];

  //create an image view for the bomb to display in
  self.bulletRect = CGRectMake(bulletStartX, playerView.frame.
origin.y, bulletWidth, bulletWidth*2);
  self.bulletView = [[UIImageView alloc] initWithFrame: self.
bulletRect];

  //set the start image
  [self.bulletView setImage: [UIImage imageWithCGImage:imageRef2]];

  //animate the image view with the bomb array of images
  self.bulletView.animationImages = bombArray;
  //how often do we want to change the image? every 1/3rd of a second
  self.bulletView.animationDuration = .3;

  //add the animation to the Game's View
  [self.gameView addSubview: self.bulletView];
```


www.it-ebooks.info

The `[self.bulletView startAnimating];` is what actually tells the image view to start the animation. If we stopped here, we would have created a bullet right where the player is, and it would just animate in place as follows:

```
    //start the bomb's image animation
    [self.bulletView startAnimating];

    //move the bomb up the screen
    self.bulletTimer = [NSTimer scheduledTimerWithTimeInterval:.03
                            target:self
                            selector:@selector(moveBullet)
                            userInfo:nil
                            repeats:YES];
}
```

Let's add a timer to move the bullet up towards the top of the screen by calling a new method, `moveBullet`. Our `moveBullet` method is similar to our `moveLeft` and `moveRight` methods, with the exception that it invalidates the timer and removes the bullet from the screen when it gets too close to the top of the screen (50 pixels from the top, to be exact) as follows:

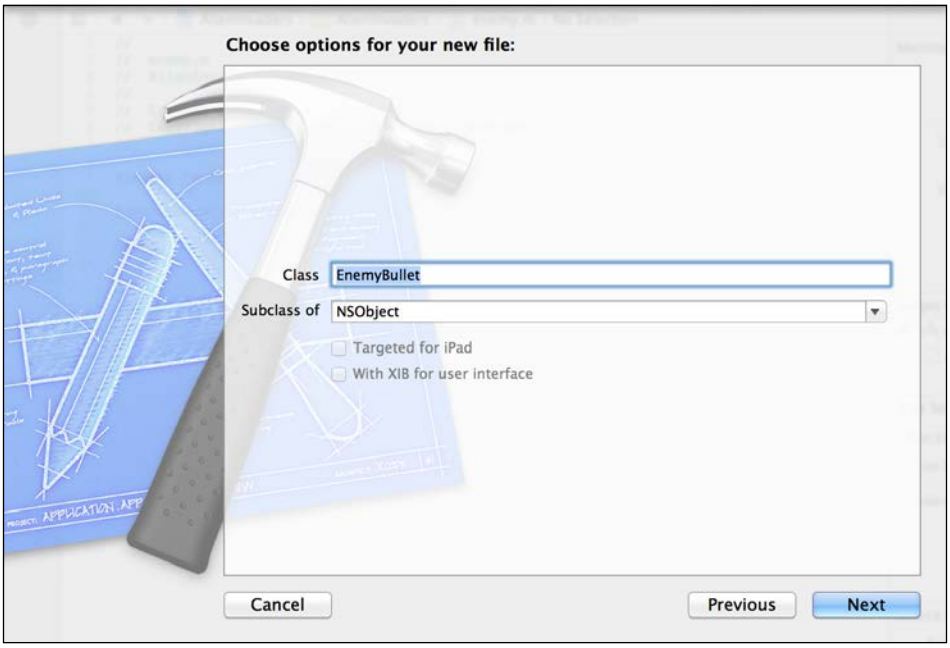
```
-(void) moveBullet
{
  self.bulletRect = CGRectOffset(self.bulletRect, 0, -5);
  self.bulletView.frame = self.bulletRect;
  if(self.bulletRect.origin.y < 50){
    [self.bulletTimer invalidate];
    self.bulletTimer = nil;
    [self.bulletView removeFromSuperview];
  }
}
```

Creating the enemy bombs

Okay. This works well for the player, but the enemies have nothing to do except get hit. Let's give the enemies a little fire power as well. We can pretty much duplicate the same features that we added to the player's bullets, but we will be dropping bombs instead. So wash, rinse, and repeat. Create a new class for the enemy bullet. Under your **File** menu, add a new file. Choose the file type of **Objective-C class** as shown in the following screenshot:

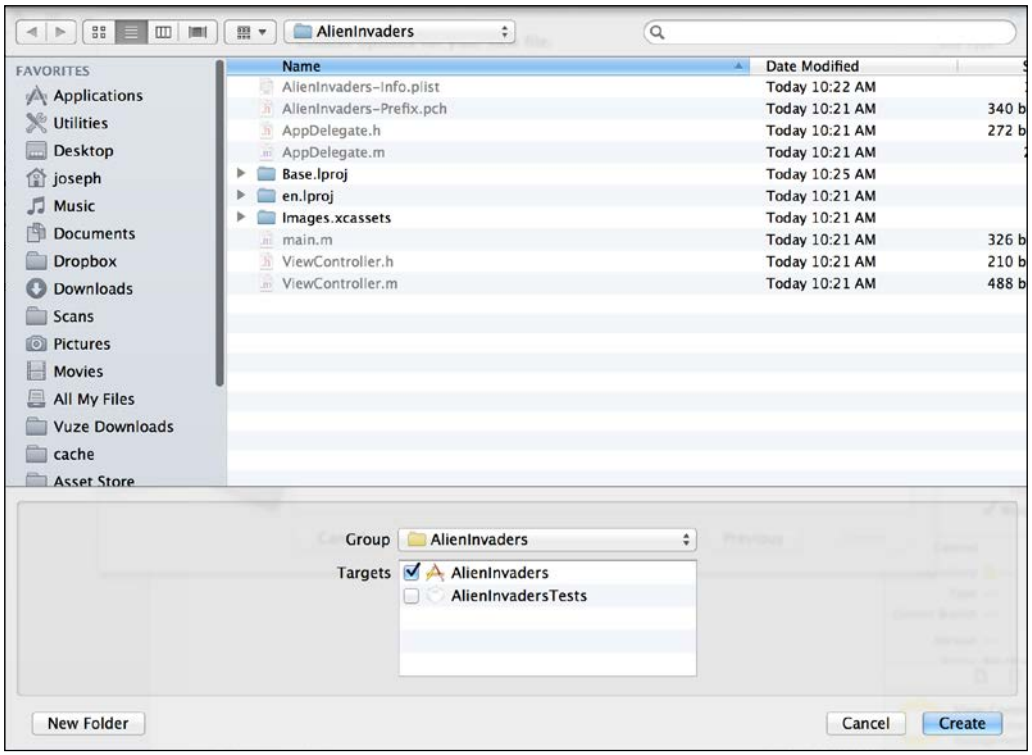
www.it-ebooks.info

Name your class `EnemyBullet` and set the subclass to **NSObject** as shown in the following screenshot:



www.it-ebooks.info

Save your file to the default project directory as shown in the following screenshot:



Once this is done, we will link the `EnemyBullet` class to our `enemy` class by adding `#import` to our enemy implementation file just below the `#enemy.h` import as follows:

```
#import "Enemy.h"
#import "EnemyBullet.h"

@implementation Enemy
```

We are adding it to our `enemy` class because that is where we will be calling it. We will need to add `NSTimer *enemyBulletTimer` to our properties along with a `dropBomb` method to our header file as follows:

```
#import <Foundation/Foundation.h>

@interface enemy : NSObject
@property UIView *gameView;
@property NSInteger minXPos;
@property NSInteger maxXPos;
@property NSMutableArray *enemyList;
```


www.it-ebooks.info

```
@property NSTimer *enemyTimer;
@property NSTimer *enemyBulletTimer;

-(void) initEnemies:(UIView *) gameView;
-(void) moveEnemies;
-(void) dropBomb;
@end
```

Our implementation file is very similar to the `playerBullet` implementation, with a few minor exceptions. In our `fireBullet` method, we take an array instead of `UIIMageView` as an argument. This is because we are working with an array of enemies and not just a single player. For our enemy bullets, we pick a random number between 1 and the number of enemies we created. Then, we launch an enemy bullet from that enemy's location. Other than these differences, it is very similar to the `playerBullet` implementation, and looks as follows:

```
-(void) fireBullet: (UIView *) gameView :(NSArray *) enemyList{
  int randEnemy = random() % enemyList.count;
  UIImageView *enemyView = enemyList[randEnemy];
  self.gameView = gameView;

  //get our bomb image
  UIImage *bombImage1 = [UIImage imageNamed:@"bullet.png"];

  //get the center x position of the player's ship
  int bombWidth = 8;
  int bombStartX =enemyView.frame.origin.x;

  //make two images from the bomb image using these rects
  CGImageRef imageRef = CGImageCreateWithImageInRect([bombImage1
    CGImage], CGRectMake(0, 0, 32, 64));
  CGImageRef imageRef2 = CGImageCreateWithImageInRect([bombImage1
    CGImage], CGRectMake(33, 0, 32, 64));

  //create an array of both images for the animation
  NSMutableArray *bombArray = [[NSMutableArray alloc]init];
  [bombArray addObject:[UIImage imageWithCGImage:imageRef]];
  [bombArray addObject:[UIImage imageWithCGImage:imageRef2]];

  //create an image view for the bomb to display in
  self.bombRect = CGRectMake(bombStartX, enemyView.frame.origin.y,
    bombWidth, bombWidth*2);
  self.bombView = [[UIImageView alloc] initWithFrame:
    self.bombRect];
```


www.it-ebooks.info

```
    //set the start image
    [self.bombView setImage: [UIImage imageWithCGImage:imageRef2]];

    //animate the image view with the bomb array of images
    self.bombView.animationImages = bombArray;
    //how often do we want to change the image? every 1/3rd of a
    //second
    self.bombView.animationDuration = .3;

    //add the animation to the Game's View
    [self.gameView addSubview: self.bombView];

    //start the bomb's image animation
    [self.bombView startAnimating];

    //move the bomb up the screen
    self.bombTimer = [NSTimer scheduledTimerWithTimeInterval:.03
                           target:self
                           selector:@selector(moveBomb)
                           userInfo:nil
                           repeats:YES];
}
```

Our `moveBomb` method is very close to the `moveBullet` method in our `playerBullet` class. Since we want our bullets to fall instead of rise, we need to make a few adjustments. Instead of changing our rectangle's origin by decreasing 5 pixels, we are going to increase it by 5 pixels. When we check our `bombRect` method's *y* axis location, we need to make sure it doesn't pass the 440 pixel mark on our screens:

```
-(void) moveBomb
{
  self.bombRect = CGRectOffset(self.bombRect, 0, 5);
  self.bombView.frame = self.bombRect;
  if(self.bombRect.origin.y > 440){
    [self.bombTimer invalidate];
    self.bombTimer = nil;
    [self.bombView removeFromSuperview];
  }
}
```

Now, we can set up a trigger that tells our bombs to drop. Back in our enemy implementation file, we can add a timer that will call a `dropBomb` method as follows:

```
    self.enemyBulletTimer = [NSTimer
                             scheduledTimerWithTimeInterval:1
                             target:self
```

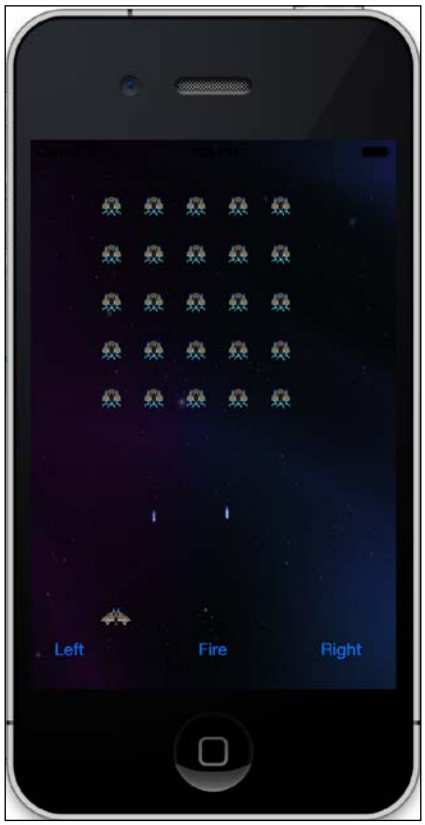

www.it-ebooks.info

```
                                selector:@selector(dropBomb)
                                userInfo:nil
                                repeats:YES];
```

Our `dropBomb` method will just create a new bomb and tell it to fire as follows:

```
-(void) dropBomb{
  //pick a random enemy and drop a bomb from their position
  EnemyBullet *newBullet = [[EnemyBullet alloc]init];
  [newBullet fireBullet: self.gameView: self.enemyList];
}
```

Let's check out the results. If everything went well, you should be able to move your player left and right, as well as fire by pushing one of the three buttons at the bottom of your screen. Your enemies should be dropping bombs from different enemy positions, but bombs and bullets just slide right through everyone, almost like they don't really exist, as shown in the following image:



We are going to stop our Space Invaders example here.

www.it-ebooks.info

Summary

In our next chapter, we will get into issues such as bullet/bomb/player/enemy collisions. For now, however, we want to make sure that you understand how images work and how to control them. In this chapter, we learned about graphics and visuals, as well as how to display images programmatically, move them around on the screen, and how to do a basic sprite animation. From a programming standpoint, we learned how to create a new object class and the basics of linking them and calling them from other classes. We also learned how to call a specific method from a new class that we created.

In the next chapter, we will learn about `NSObjects`, basic object, oriented programming, game logic, state machines, creating your enemy class, and more.

www.it-ebooks.info

5

The Game Engine – Space Invaders Part 2

In the previous chapter, we began exploring how to create a Space Invaders style game. We got a great introduction to graphics; starting small. We learned about bullets and sprites and how to use the shoot bullet code. Now, we are ready to go deeper in our understanding with the following points:

- Understanding the game logic
- Understanding state machines
- Learning the game state
- Creating your enemy class
- Creating multiple enemy objects
- Timers and frame rates
- Debugging your code
- Collision detection using intersecting rectangles and debugging

Understanding the game logic

Let's talk logic. If you have ever played a board game—and I am assuming you have since you are reading this book—then game logic is the equivalent of the rulebook for your game. There are rules for every part of your game, down to the minutest detail. Game logic is best planned out as much as possible before starting to program. Keep in mind, however, that a lot of your code and design will change as you program your game, along with changes your game logic will establish as well.

www.it-ebooks.info

When programming the game, we will need to figure out how the game is going to work. Let's take our *Space Invaders* game, for example. We know that when we start our game, we are going to have a very simple main menu, a game screen, a player, lots of enemies that will be shooting at the player, and a scoring system. In order to manage all of these elements, we are going to need something to manage them and keep track of all of this stuff, such as where elements are going and what is happening on the screen. In order to manage all of these elements, we need to incorporate some management and logic into our code. This is the **Controller** arm of the **Model-View-Controller** architecture pattern.

For our games, we are going to try and have each piece (or object) in our game have its own controller. With that said, we should have an enemy controller, player controller, game controller, score controller, sound controller, and so on. In the upcoming chapters, we will create other controllers for attributes such as sound and scores. When the game is playing, we will have the game's view controller act as the game controller. This way, when the player leaves the game screen, the game's logic is not being used when it is not necessary.

In this chapter, we are going to rewrite some of the `GameViewController` code so that it only keeps track of game information. We will need to move the player information out to its own class and add a few more classes for managing other objects. This will free up the game controller to be exclusively for controlling the game environment.

Understanding state machines

When we rewrite the `GameViewController` code, we are going to add in an enemy controller, which will be a special type of controller called a finite state machine or simply a state machine. A state machine is a controller that keeps track of the active state of an object. It can only be in one state at any given time but can change states whenever necessary.

State machines are one of the coolest concepts I have ever learned in programming. State machines are simple, elegant, and extremely useful. Let's take a look at how a simple state machine works using ball 8 in a pool game as an example. When ball 8 is sitting still on a table and nothing is interacting with it, we can call the state of the ball *idle* since it is sitting idle and waiting for something to happen. The states of other objects in the game don't necessarily affect the state of ball 8.

www.it-ebooks.info

For example, if someone shoots the cue ball and hits ball 3 into a pocket, ball 8 is still in its idle state since it didn't move. Now if the player shoots the cue ball into ball 4, then ball 4 hits ball 8 into a pocket. The state of ball 8 would then change accordingly. Our ball starts out in an *idle* state, changes to *connecting*, then *moving* until it gets to the pocket, then it becomes *sinking* until it connects with the bottom of the pocket, at which point it becomes *inactive*. If we set this up in code, it might look something similar to the following code snippet:

```
NSString *activeState = NULL;

typedef NS_ENUM(NSInteger, GAME_STATE){
  IDLE=1,
  CONNECTING=2,
  MOVING=3,
  SINKING=4,
  INACTIVE=5
};

-(void)changeState: (int)newState{
  switch (newState)
  {
    case IDLE:
      //do nothing until something happens
      activeState = @"idle";
      break;
    case CONNECTING:
      activeState = @"moving";
      [self playSoundEffect: @"BallHit"];
      break;
    case MOVING:
      [self doRoll];
      break;
    case SINKING:
      [self sinkBall];
      break;
    case INACTIVE:
      [self checkForWin];
      [self destroyMe];
    default:
      break;
  }
}
```


www.it-ebooks.info

If you start thinking about all of the objects in your game sitting in a specific state at any given moment, a state machine becomes very powerful very quickly. Not all of your objects will need a state machine; for example, the pool table in this scenario wouldn't need a state machine since it is only in one state. It is helpful if you have a game state machine, which we are going to cover next.

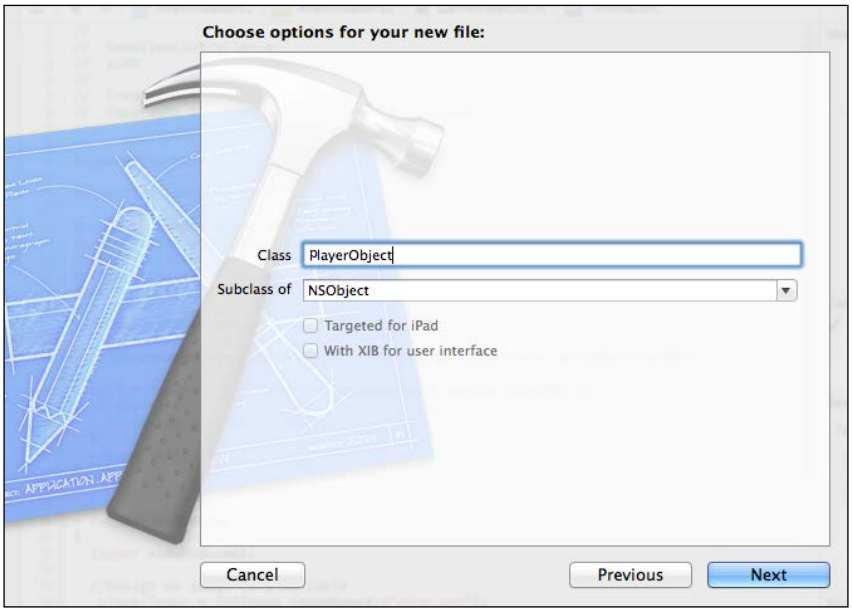
Learning the game state

Let's pick up the *AlienInvaders* game we started putting together in the previous chapter. In this chapter, we are going to set up our game to have its own state machine. If we go back and look over the `GameViewController` code from the previous chapter, you will notice that everything that we put in here doesn't really have anything to do with the game itself; it is all about the player. That doesn't make any sense if we want our player to have its own state machine and the game to have its own state machine. Let's start by performing the following steps:

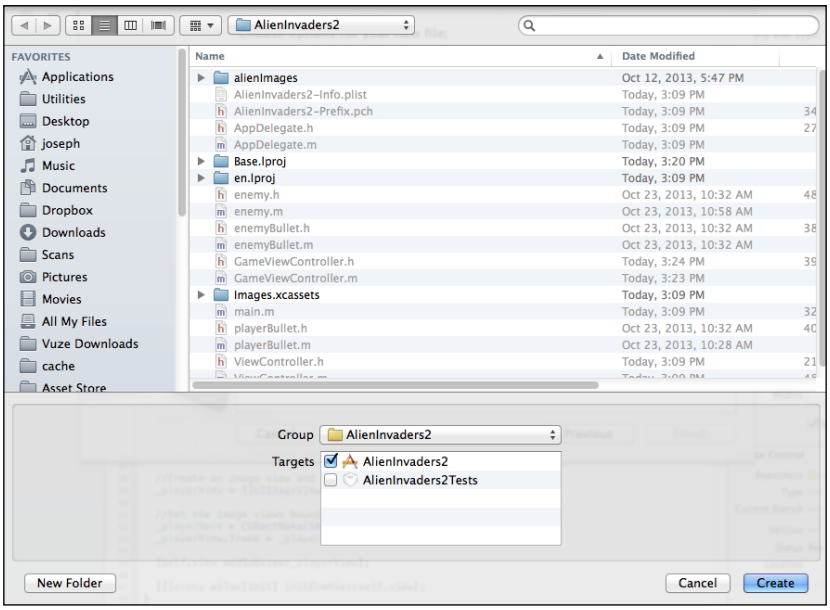
1. Create a new player class and move everything for the player out of `GameViewController`.
2. Create a new class for `EnemyBullet` under your **File** menu and add a new file.
3. Choose the file type of **Objective-C class** as shown in the following screenshot:

www.it-ebooks.info

4. Name your new class `PlayerObject` with a **NSObject** subclass as shown in the following screenshot:



5. And finally, save your file to the default project directory as shown in the following screenshot:



www.it-ebooks.info

If we look at our `GameViewController` header file, the only things in the interface are button functions that hook into the game's interface. We are going to keep these linked to the game controller since the interface is not part of the player. We, however, want to remove the link to the `PlayerBullet.h` file since that is going to be called by the new player object, not the game. Copy and paste `#import PlayerBullet.h` into your `playerObject.h` file, just below the foundation class's import. We are going to create a new property for our game to keep track of our player and the player's view. Here is our new `GameViewController` header with all of our changes:

```
// GameViewController.h
#import <UIKit/UIKit.h>
#import "PlayerObject.h"
#import "Enemy.h"

@interface GameViewController : UIViewController
@property (strong) NSTimer *moveTimer;
@property PlayerObject *playerOne;
@property UIView *playerView;
@property Enemy *enemies;

- (IBAction)fireButton:(id)sender;
- (IBAction)moveLeft:(id)sender;
- (IBAction)moveRight:(id)sender;
- (IBAction)touchRelease:(id)sender;
@end
```

Next, we are going to store our player object in our new `self.playerOne` property and move all of the player information from the `GameViewController` implementation file and put it into the `playerObject` implementation file. Since we are going to keep all of our button actions in our `GameViewController` file, we will have to set all of the `moveTimer` targets to the `self.playerOne` property. We will have to call our `self.playerOne fireBullet` method now from the `fireButton` method. Here is what our new `GameViewController` should look like when we are all done:

```
#import "GameViewController.h"
@implementation GameViewController

- (id)initWithNibName:(NSString *)nibNameOrNil bundle:(NSBundle *)
nibBundleOrNil
{
  self=[super initWithNibName:nibNameOrNil bundle:nibBundleOrNil];
  if (self) {
    // Custom initialization
```


www.it-ebooks.info

```
    }
    return self;
  }
```

Once our game's view is loaded, initialize the `self.playerOne` object and its view as follows:

```
- (void)viewDidLoad
{
  [super viewDidLoad];

  self.playerOne = [[[PlayerObject alloc]init] initPlayer:self.view];
   self.playerView = [self.playerOne getPlayerView];
  [[[Enemy alloc]init] initEnemies:self.view];
}
```

Set up our left and right button actions using timers as follows:

```
- (IBAction)moveLeft:(id)sender {
  [self releaseTouch];
  self.moveTimer = [NSTimer scheduledTimerWithTimeInterval:.03
                        target: self.playerOne
                        selector:@selector(movePlayerLeft)
                        userInfo:nil
                        repeats:YES];
}

- (IBAction)moveRight:(id)sender {
  [self releaseTouch];
  self.moveTimer = [NSTimer scheduledTimerWithTimeInterval:.03
                        target: self.playerOne
                        selector:@selector(movePlayerRight)
                        userInfo:nil
                        repeats:YES];
}

- (IBAction)touchRelease:(id)sender {
  [self releaseTouch];
}

-(void)releaseTouch{
  if(self.moveTimer != nil){
    [self.moveTimer invalidate];
    self.moveTimer = nil;
  }
}
```


www.it-ebooks.info

Finally, call our player's fire button as follows:

```
- (IBAction)fireButton:(id)sender {
  [self.playerOne fireBullet];
}

- (void)didReceiveMemoryWarning
{
  [super didReceiveMemoryWarning];
  // Dispose of any resources that can be recreated.
}
@end
```

Our new `playerObject` header file should look like this:

```
#import <Foundation/Foundation.h>
#import "PlayerBullet.h"

@interface PlayerObject : NSObject
@property UIView *gameView;
@property CGRect playerRect;
@property (strong) UIImage *playerImage;
@property (strong) UIImageView *playerView;

-(PlayerObject *) initPlayer:(UIView *) gameView;
-(void) movePlayerRight;
-(void) movePlayerLeft;
-(void) fireBullet;
-(UIView *) getPlayerView;
@end
```

As you can see, everything we moved out of the `GameViewController` file has been moved here. We are returning `PlayerObject` from the `initPlayer` method so that `GameViewController` will be able to keep track of the newly created player. Using objects this way in future chapters will allow us to create a second player that could share the same code base. You will notice that the bullet is being launched from this player, so we are keeping track of it with this player rather than from `GameViewController`. Since the player shot the bullet, we want to make sure the correct `PlayerObject` will get the information when an enemy gets hit.

www.it-ebooks.info

Now, let's take a look at the new `PlayerObject` implementation file. We are passing the `gameView` object by using the same technique we used with the enemy object. When we create our `self.playerView` object, we are using `[UIImageView alloc];` the `alloc` keyword is used for allocation. When we create a new `UIImageView`, we are allocating the necessary memory for a new image view. The `init` method that follows is to initialize its attributes. You can also see that at the end of the `initPlayer` method, the `self` pointer is returned back to the object that called our `initPlayer` method. In Objective-C, `self` is a pointer to the object executing the method. In this case, `self` would be the `PlayerObject` as follows:

```
#import "playerObject.h"
#import "PlayerBullet.h"

@implementation PlayerObject

-(PlayerObject *) initPlayer:(UIView *) gameView{
  self.gameView = gameView;

  //Assign an image to a variable
  self.playerImage = [UIImage imageNamed:@"ship.png"];

  //Create an image view and start it up with our image,
  self.playerView = [[UIImageView alloc] initWithImage:self.
playerImage];

  //Set the image views bounding rectangle to scale the image.
  self.playerRect = CGRectMake(50,400, 32,32);
  self.playerView.frame = self.playerRect;

  [self.gameView addSubview:self.playerView];

  return self;
}

-(UIView *) getPlayerView{
  return self.playerView;
}

-(void) movePlayerRight{
  CGRect screenRect = [[UIScreen mainScreen] bounds];
  if(self.playerRect.origin.x <= screenRect.size.width
     - self.playerRect.size.width - 10){
```


www.it-ebooks.info

```
      self.playerRect = CGRectOffset(self.playerRect, +3, 0);
      self.playerView.frame = self.playerRect;
    }
  }

  -(void) movePlayerLeft{
    if(self.playerRect.origin.x >= 10){
      self.playerRect = CGRectOffset(self.playerRect, -3, 0);
      self.playerView.frame = self.playerRect;
    }
  }

  -(void) fireBullet{
    [[[PlayerBullet alloc]init] initBullet: self.gameView :
      self.playerView ];
  }
  @end
```

Now that the code is all dismantled again, let's get back to the game's state machine. In our `GameViewController` header file, let's add in a new `typedef` declaration for an **enumerator** (**enum**). An enumerator will allow us to create a list of names that are going to be associated with an integer. This will allow us to create a switch statement in our code that will be a lot more readable for our state machine than a standard switch statement. We are also going to add in a new property to track the current state of the game controller. While we are at it, let's add a new image view for loading a splash screen as follows:

```
#import <UIKit/UIKit.h>
#import "PlayerObject.h"
#import "Enemy.h"

@interface GameViewController : UIViewController
@property (strong, atomic) NSTimer *moveTimer;
@property (strong, atomic) NSTimer *collisionTimer;
@property PlayerObject *playerOne;
@property UIView *playerView;
@property Enemy *enemies;

typedef NS_ENUM(NSInteger, GAME_STATE){
  INITIALIZING=1,
  PLAYING=2,
  RELOADING=3,
  ENDING=4
};
```


www.it-ebooks.info

```
@property GAME_STATE currentState;

@property (strong) UIImageView *loadingView;

- (IBAction)fireButton:(id)sender;
- (IBAction)moveLeft:(id)sender;
- (IBAction)moveRight:(id)sender;
- (IBAction)touchRelease:(id)sender;
@end
```

We have made a lot of changes to our `GameViewController` file, so let's go through them. This is where the changes begin. We changed our `viewDidLoad` method to change the state of `GameViewController` to `INITIALIZING`, which according to our `GAME_STATE` list is the first step in our state list:

```
- (void)viewDidLoad
{
  [super viewDidLoad];
  [self changeState: INITIALIZING];
}
```

When we look at the new method, `changeState`, you see that we take an integer as the argument since `INITIALIZING` is equal to the integer 1; we will be able to check it in our switch statement. Our switch statement tells us to call the `loadScreen` method, which contains the code that was used to call in our `viewDidLoad` method. The main reason for moving the code is that if we ever want to add in a state that was called before our `loadingScreen` method, we would simply need to change the argument to our `changeState` call in the `viewDidLoad` method.

We added the rest of our states from our enum to the `case` statement. You will notice that we are not doing anything with our playing case. Since we have prepped everything with our initialization state, we do not need to do anything here. Our reloading state will get called whenever our player gets hit with a bomb from the enemies, thus ending a round, which will cause the screen to go into a pause state and clear out the timers as follows:

```
-(void)changeState: (int)newState{
  switch (newState)
  {
    case INITIALIZING:
      //throw up a loading screen
      [self loadingScreen];

      break;
    case PLAYING:
```


www.it-ebooks.info

```
        break;
      case RELOADING:
        [self.view addSubview: self.loadingView];
        [self.enemies stopTimers];
        [self.collisionTimer invalidate];
        self.collisionTimer = NULL;
        break;
      case ENDING:
        break;
      default:
        NSLog (@"Integer out of range");
        break;
    }
    self.currentState = newState;
  }
```

Our new `LoadingScreen` method takes care of all of our initialization. This is because we can stop the entire game and start the scene over again if we need to. We added in a `loadingView` property, which stores `UIImageView` of a screen that pops up over our game to tell the player to *Prepare For Battle*. This could of course be anything you would like the image to be. I am just using it as a way to show the state changes during gameplay. I added a timer at the end that will keep the screen up for two seconds before calling the `closeScreen` method as follows:

```
-(void)loadingScreen{
  self.playerOne = [[[PlayerObject alloc]init] initPlayer: self.view];
  self.playerView = [self.playerOne getPlayerView];

  self.enemies = [[[Enemy alloc]init] initEnemies:self.view];

  self.loadingView = [[UIImageView alloc] initWithImage:[UIImage
imageNamed:@"loading.png"]];
  self.loadingView.frame = self.view.frame;

  [self.view addSubview: self.loadingView];
  [NSTimer scheduledTimerWithTimeInterval:2
                        target:self
                        selector:@selector(closeScreen)
                        userInfo:nil
                        repeats:NO];
}
```


www.it-ebooks.info

The `closeScreen` method closes the loading screen image and sets the state to playing. It then tells the enemy class to start all of their timer methods. I am adding a repeating timer that calls a new method (`intersectCheck`), which we will go over later in the chapter:

```
-(void)closeScreen{
  [self.loadingView removeFromSuperview];
  [self changeState: PLAYING];
  [self.enemies startTimers];
  self.collisionTimer =[ NSTimer scheduledTimerWithTimeInterval:.1
                         target:self
                         selector:@selector(intersectCheck)
                         userInfo:nil
                         repeats:YES];
}
```

Our end screen is called after a bomb has hit the player; this is the last state before exiting the `GameViewController` implementation. This is also where we would do any kind of clean up before exiting the gameplay area. In order to exit the gameplay area, let's add an exit button to the storyboard screen along with an unwind segue. I gave the segue an identifier name of `unwind` and we will call it when we are ready to leave the screen:

```
-(void)endScreen{
  [self.loadingView removeFromSuperview];
  [self changeState: ENDING];
  [self performSegueWithIdentifier:@"unwind" sender:self];

}
```

This is where we will add our collision code for checking to see if the player has been hit:

```
-(void)intersectCheck{

}
```

The button controllers didn't change from our previous version because we want to keep our `GameViewController` implementation as the main control point for all of our game's movement control:

```
- (IBAction)moveLeft:(id)sender {
  [self releaseTouch];
  if(self.currentState != PLAYING)
```


www.it-ebooks.info

```
      return;
    self.moveTimer = [NSTimer scheduledTimerWithTimeInterval:.03
                           target: self.playerOne
                           selector:@selector(movePlayerLeft)
                           userInfo:nil
                           repeats:YES];
  }

  - (IBAction)moveRight:(id)sender {
    [self releaseTouch];
    if(self.currentState != PLAYING)
      return;
    self.moveTimer = [NSTimer scheduledTimerWithTimeInterval:.03
                           target: self.playerOne
                           selector:@selector(movePlayerRight)
                           userInfo:nil
                           repeats:YES];
  }

  - (IBAction)touchRelease:(id)sender {
    [self releaseTouch];
  }

  -(void)releaseTouch{
    if(self.moveTimer != nil){
      [self.moveTimer invalidate];
      self.moveTimer = nil;
    }
  }

  - (IBAction)fireButton:(id)sender {
    if(self.currentState != PLAYING)
      return;
    [self.playerOne fireBullet];
  }

  - (void)didReceiveMemoryWarning
  {
    [super didReceiveMemoryWarning];
    // Dispose of any resources that can be recreated.
  }
  @end
```


www.it-ebooks.info

Collision detection

Ah, collision detection, another really cool part of programming games. Collision detection is simply using code to detect when objects touch or overlap each other. There are a few ways to handle collision detection. If you are developing a 3D game, it gets a lot more complicated, and a lot of game engines have physics engines integrated in them, which you can request to provide collision information when objects collide. Over the last few years, 2D game engines have started integrating physics libraries as well. With some of the more complex libraries, if you request the collision data, you will receive a list of what happened with that collision. For instance, you can find out what objects collided, how fast they were going, what the exact points of collision for each of the colliding objects were, and so on. In the games that we will be building, we are not really interested in all of this kind of data; we will just want to know if we collided.

This brings us to our code. What we are looking for is simple collision. Our code here is just tracking two CGRect's to check if they are intersecting. If they are set to the reloading state, they call the end screen after a two second delay. For this demo, we are just ending the game here. It would probably be a lot more realistic if you gave your player three tries before ending the game. Apple was kind enough to supply us with the `CGRectIntersectsRect` method, which returns a true value if they are crossing and a false value if they are not:

```
-(void)intersectCheck{
  BOOL isConnecting =
    CGRectIntersectsRect(self.enemies.enemiesBullet.
                         bombRect, self.playerOne.playerRect);
  if(isConnecting == true){
    [NSTimer scheduledTimerWithTimeInterval:2
                         target:self
                         selector:@selector(endScreen)
                         userInfo:nil
                         repeats:NO];
    [self changeState:RELOADING];
  }
}
```

For our purposes, we are only checking the player's location and the enemy's bullet. Since the enemy can only have one bullet at a time, this is a pretty simple method. What we would do in a more complex game is create an array that keeps track of all of the active bullets on the screen along with all of the active players and enemies. We would then do a check to see if the player's bullets intersected with any of the enemies. Then, we would check to see if any of the enemy's bullets in the array are intersecting with the active player.

www.it-ebooks.info

As you can imagine, this can get very complex, and if you are making too many, it could end up slowing the game down. This brings us to our next topic; timers and frame rates.

Timers and frame rates

By this point, you have seen us use a lot of timers in the game. You may have even picked up on the fact that our collision timer was set a little lower than most of our other timers. We have set it to 0.1 seconds rather than 0.03 (30 frames per second) because we were checking the collision on a couple of pretty big rectangles. The bullet couldn't get through the entire player's rectangle in a tenth of a second; so, in order to keep the number of times I check the collision down, I just reduced the number of times I called the method. This may seem a little extreme at first, but the reality is that the more you add to your screen, the more it is going to slow down. Similarly, the number of times you are checking your methods in a timer, the more you are going to slow your game down.

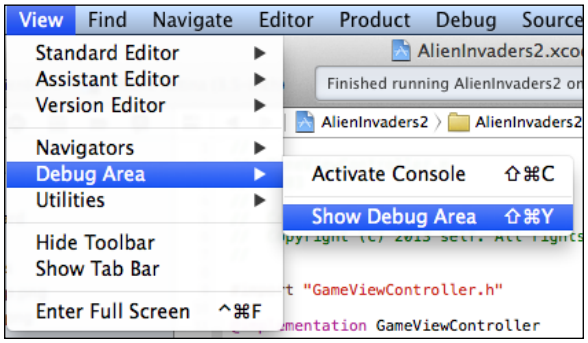
The motion picture industry standard for film is 24 frames per second. In the games industry, we usually lock our frame rates to 30 or 60 fps. There are some 3D games that don't lock their frame rates that report more than 100 fps. If you go through some of the game reviews on the Internet or in magazines, you will find a focus on frame rates. It is a benchmark that is used to determine how well a game plays. It can also be used to see how efficient you are running your code. As your projects get larger, you are going to notice your frame rates getting slower and slower. When this happens, you will start to see your animations skip frames, or jump more pixels ahead than they should be.

In order to reduce this, we are going to start using an `update` method. We can call this from a timer on initialization. Inside the `update` method, we will check everything in our class in which we have been using multiple timers. You can see an example of the `update` method in the `MyScene` implementation file of our *Galaxy game* in *Chapter 8*, *The Galaxy Game*. This will at least allow us to manage our classes for each frame and in turn allow us to control our frame rates a bit.

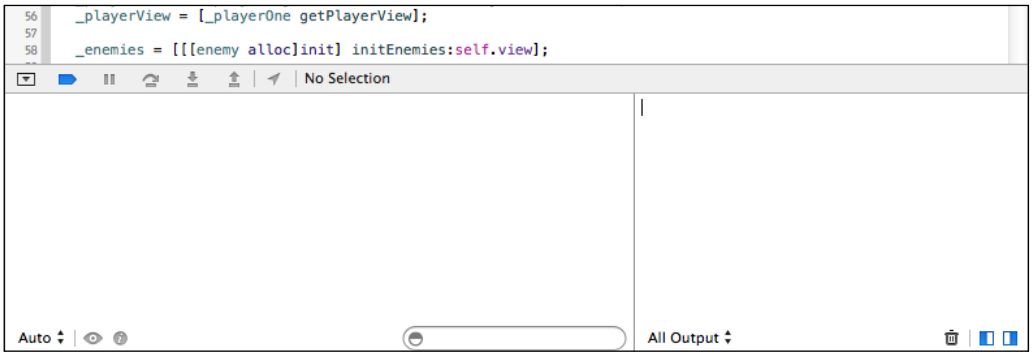
www.it-ebooks.info

Debugging your code

So how do we check our code when something is going wrong? There are two things that you can do: optimize your code and debug your code. Optimizing your code is when you take existing code that is working and make it run more efficiently. Debugging is when you are trying to find problem areas and fixing the problems. Xcode has some excellent tools for debugging our code. Let's start with the debugging area. Navigate to **View | Debug Area | Show Debug Area** as shown in the following screenshot:



This will open a panel below your workspace with the debugger as shown in the following screenshot. We discussed the basics of the debugger in *Chapter 1, The Environment – Installing and Working with Xcode*, but let's talk about how it can help you with your code.



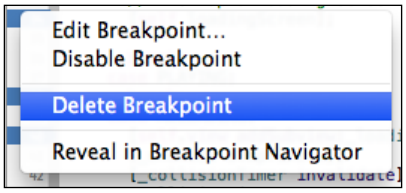
www.it-ebooks.info

Let's start with breakpoints. When you are programming and you run into something that doesn't seem to work quite right, and you can't figure out where the problem is originating, it helps to step through your code one line at a time while it is running. You can insert a breakpoint anywhere in your code. When you run your project and the code runs into the breakpoint, the game will stop and bring you back to your code where you can examine all of your properties and variables at that point in time. You can step through your code line-by-line and see how your code is reacting. Let's give it a shot.

Take a look at the new `GameViewController.m` code. Let's add some breakpoints in our code. Add one each time we change states. Scroll to the area in your code you want to add the breakpoints and click on the line numbers to the left of the line in which you want it to stop. You should see a blue marker show up next to the line as shown in the following screenshot:

```
29  -(void)changeState: (int)newState{
30    switch (newState)
31    {
32      case INITIALIZING:
33        //throw up a loading screen
34        [self loadingScreen];
35
36        break;
37      case PLAYING:
38        break;
39      case RELOADING:
40        [self.view addSubview:_loadingView];
41        [_enemies stopTimers];
42        [_collisionTimer invalidate];
43        _collisionTimer = NULL;
44        break;
45      case ENDING:
46        break;
47      default:
48        NSLog (@"Integer out of range");
49        break;
50    }
51    _currentState = newState;
52  }
```

If you click it again, the breakpoint will fade out, which disables the breakpoint but leaves the image behind in case you want to add it back again. If you want to completely remove it, Press *Ctrl* and click on the breakpoint and select **Delete Breakpoint**:



www.it-ebooks.info

You can add these wherever you like, but for now, let's run our code.

Your game should load just like it normally would. If you click on **Play Game**, you will be brought into your game screen. But before anything can happen, our initializing state is called and we are brought back into our code as follows:

```
28
29  -(void)changeState: (int)newState{
30    switch (newState)
31    {
32      case INITIALIZING:
33        //throw up a loading screen
34        [self loadingScreen];
35
36        break;
37      case PLAYING:
38        break;
39      case RELOADING:
40        [self.view addSubview:_loadingView];
41        [_enemies stopTimers];
42        [_collisionTimer invalidate];
43        _collisionTimer = NULL;
44        break;
45      case ENDING:
46        break;
47      default:
48        NSLog (@"Integer out of range");
49        break;
50    }
51    _currentState = newState;
52  }
```

The green highlighted line you are seeing is where the code has stopped. If you set your breakpoints in the same places I did, this should be at the `[self loadingScreen];` line in the `changeState` method. In your debug area, you will see the `self = (GameViewController *)` object. If you open that up, you will see all of the variables in your object as they currently stand. You will also see a local variable at the bottom for `newState`, which is currently set to 1 since we are passing in `INITIALIZING` as the argument.

The buttons over your debug area are collapse debugger, globally turn on/off breakpoints, continue running code, step over code, step into code, step out of code, and simulate location in code.

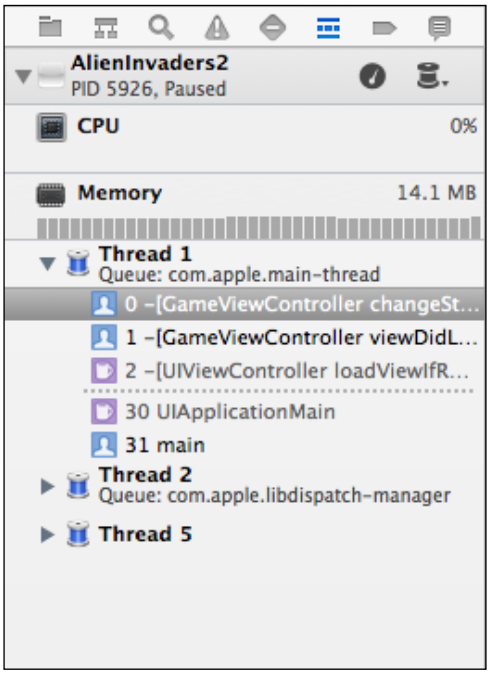
If you hit the step into button, your debugger will go to the first line in the `loadingScreen` method; hence, step into. If you press the step over button, it will run the code in the player object but your debugger will just step to the next line for `self.playerView`. If you press the step out button, you will jump back up the case statement at the breakpoint.

www.it-ebooks.info

If you press the continue button, your game will continue where we left off. You should be brought back into the code at the breakpoint for the `PLAYING` state. You will notice the `newState` variable is set to 2 now, which is the `PLAYING` position of our `GAME_STATE` enum.

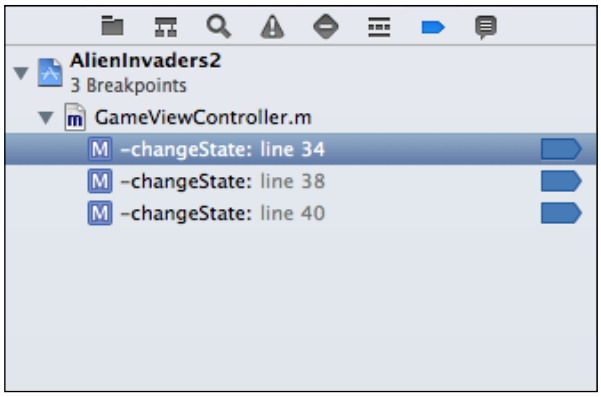
Press the continue button again and if you let your player get hit, the code will break out at the reloading statement. You will now be exited out of the game and our debugging is complete.

As you can imagine, this can be very useful on several levels. Most importantly, being able to step through code lets you see where any errors are occurring. It also lets you view what is running in your class so that you will be able to see if a timer is loaded, a variable is set incorrectly (or correctly for that matter), and you can see how your code has been called.



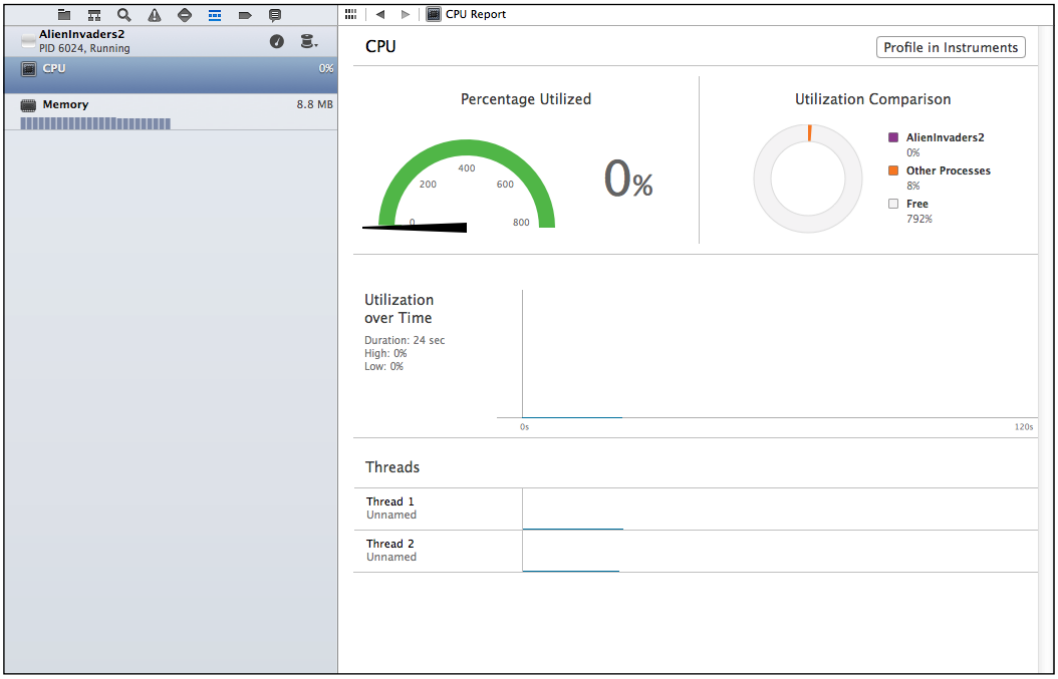
If you look to the left of the debug navigator, you can see that the current thread was called (from the bottom up) `UIApplicationMain` that called `UIViewController loadViewIfRequired`. This thread then called `GameViewController viewDidLoad`, that finally called our `GameViewController changeState` method. This can prove helpful if you have a method that gets called and you can't figure out what is calling it (yes, it actually happens sometimes).

www.it-ebooks.info



You can also see a global view of where all of your breakpoints are in the breakpoint navigator. You can delete individual breakpoints by selecting them and hitting *Delete* on your keyboard. If you click on the breakpoint marker to the right of the line number, you can disable that breakpoint. It may not seem like much when you only have three breakpoints, but when you are dealing with a large codebase, it comes in quite handy.

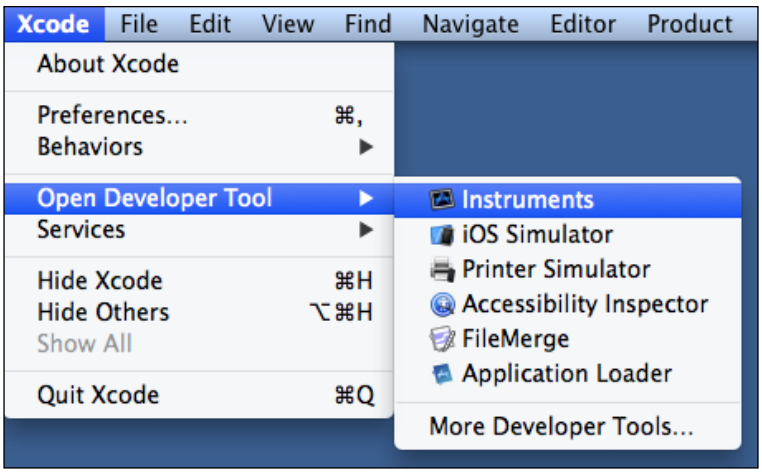
Debug gauges can be used to track your CPU, energy, and memory usage over time.



www.it-ebooks.info

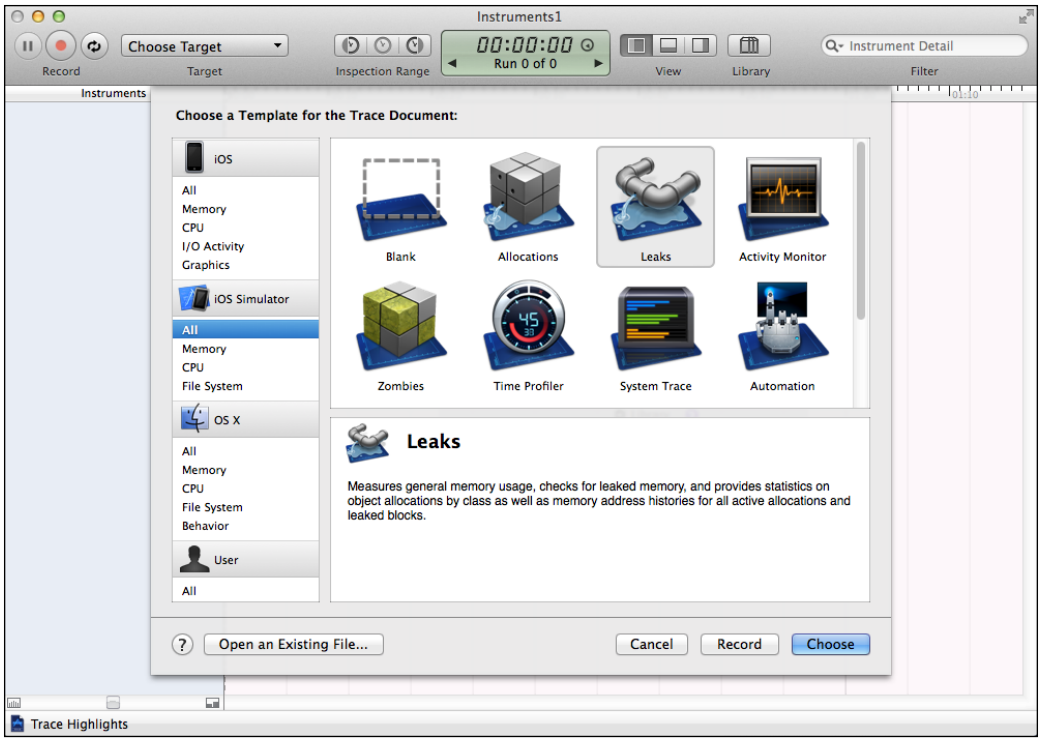
If you select the **Debug Navigator** option while your app is running (not stepping through code), this screen will load. It shows the graphs that you have selected for either CPU or memory. You can use this to see where you may have memory spikes or too much pressure on your processor.

All in all, these are just some basic tools put in place to help make developing easier. Apple provides some additional developer tools outside Xcode that can help you get even deeper in debugging. We are not going to cover them in this book, but if you navigate to **Xcode | Open Developer Tool,** you will see a list of the tools available for you as shown in the following screenshot:



Although these are not all made for iOS development, most of them can be used for testing iOS products. The **Instruments** tool can be used in conjunction with your **iOS Simulator** tool in order to track memory or performance issues. With the **Instruments** tool, you may use multiple tools to record multiple processes, then replay them in order to examine your app and see how it interacts within the operating system. This is very helpful in finding difficult-to-track issues in your code that you wouldn't normally be able to see, which could be causing performance issues or crashing. When you are in the testing phase of your project, you could also use **Instruments** to create automated processes in order to reduce your testing time. I find them most helpful for finding areas where I can increase performance, where I wouldn't normally look for issues.

www.it-ebooks.info



Summary

In this chapter, we expanded the Space Invaders game by examining `NSObjects` and basic object-oriented programming. We learned about game logic, understanding state machines, game state, creating your enemy class, timers and frame rates, collision detection, and debugging.

In the next chapter, we will learn about the audio engine by creating a Simon Says-style game. Working with audio files is difficult; however, the following chapter will thoroughly address all of your questions and concerns.

www.it-ebooks.info

www.it-ebooks.info

6

The Audio Engine – The Simon Memory Game

In the previous chapter, we completed the Space Invaders game while looking at NSObjects and basic object-oriented programming. We learned about game logic, state machines, game state, creating your enemy class, timers and frame rates, collision detection, and debugging.

In this chapter, we will cover the following topics:

- Introduction to audio
- File formats
- Data formats
- The Simon Says game

Introduction to audio

Let me start off by saying, *audio is hard*. Getting the right sound that matches your game is tricky. On top of that, all of the different file formats and encoding techniques gets very confusing and extremely difficult. On a positive note, we are going to do everything we can to make your first experience with audio a pleasant one. Let's jump right in!

Audio files are made of two parts: the **file format** and the **data format**. The file format is how the file is stored and the data format is how the file is encoded. When you save an audio file, you will need to know a few things about how the files are saved, which we will cover next.

www.it-ebooks.info

File formats

File formats are the containers for your audio files. There are several to choose from, but personally, I prefer **WAV** or **Audio Interchange File Format** (**AIFF**) for uncompressed (or lossless compressed) audio and MP3 (lossy) formats since they have been universal for a while. Apple prefers that you use the **Advanced Audio Coding** (**AAC**) or **Core Audio Format** (**CAF**) file formats, even though they support MPEG-1 (`.mp3`), MPEG-2 ADTS (`.aac`), AIFF, CAF, WAV, and probably countless others before this book is even released.

Bit rates

The **bit rate** is the number of bytes per second that a file takes up. Bit rates usually range from 32 Kbps (kilobits per second) to 320 Kbps. The most important thing to remember is the lower the bit rate, the lower the quality. 32 Kbps is, usually, only used for recording speech; it is too low to be used in most games but is fine if you are just doing dictation. On the higher end, 320 Kbps is the highest quality supported by MP3s and is very similar to CD quality, which is a 16-bit 44.1 k stereo audio file. There are higher and lower settings, but if you are compressing audio, it is a pretty good sign that you are trying to save disk space. So, any lower value than the required quality is a bad quality file and any higher is taking up too much space.

Sample rates

The **sample rate** is how often an analog signal is sampled when converting it to a digital format. The standard sampling rate used for converting CD audio is 44.1 kHz, and this is recommended for most sampling rates. If you have to reduce the sampling rate, I suggest 22.05 kHz since it is the next lowest standard.

Data formats

Before a file is stored, it is considered to be in a raw format, which consists of very large uncompressed files. However, most audio files compress the data and store it in a specific format.

The following table shows various playback file formats:

Playback file formats	Acronym
AAC	Advanced Audio Coding
HE-AAC	High Efficiency Advanced Audio Coding
ALAC	Apple Lossless Audio Codec

www.it-ebooks.info

Playback file formats	Acronym
iLBC	Internet Low Bitrate Codec
IMA4	Interactive Multimedia Association 4:1 compression
Linear PCM	Linear Pulse-Code modulation
MP3	Moving Picture Experts Group-1 Audio Layer 3
u-law/a-law	Not acronyms but rather logarithmically encoded 8-bit files

The following table shows various recording file formats:

Recording file formats	Acronym
AAC	Advanced Audio Coding
ALAC	Apple Lossless Audio Codec
iLBC	Internet Low Bitrate Codec
IMA4	Interactive Multimedia Association 4:1 compression
Linear PCM	Linear Pulse-Code modulation
u-law/a-law	Not acronyms but rather logarithmically encoded 8-bit files

Phew, enough of the heavy stuff, let's get to the fun stuff!

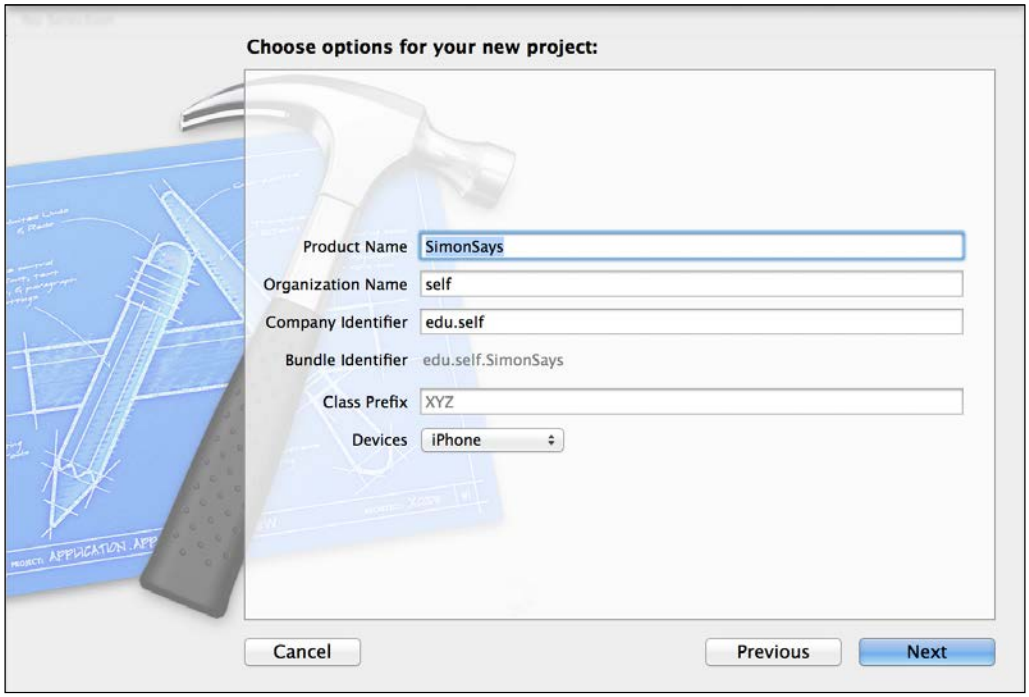
The Simon Says game

Here is a little information about the Simon Says game. Simon Says is a child's game where one of the players is *Simon* and the other players must follow his/her instructions. Simon begins each instruction with "Simon says, jump two times", for example, and the players follow suit. If, however, a player performs the instruction without the prompt of Simon Says, that player automatically takes the role of Simon.

Let's start by figuring out what the game is going to do. First and foremost, this is an introduction on how to play audio on the iPhone, so we are going to keep this pretty simple. We want to make this a two-page game with a main menu and a game scene. The game play area will need four large colored buttons and an indicator to let the player know their level. The game controller will call out a color and the player will need to match the color. Then we will keep adding a color to the list after each round, which the player will need to match. When the player misses a turn, the round will be reset back to zero. Oh, and we are going to need really cool thumping background music that matches the game.

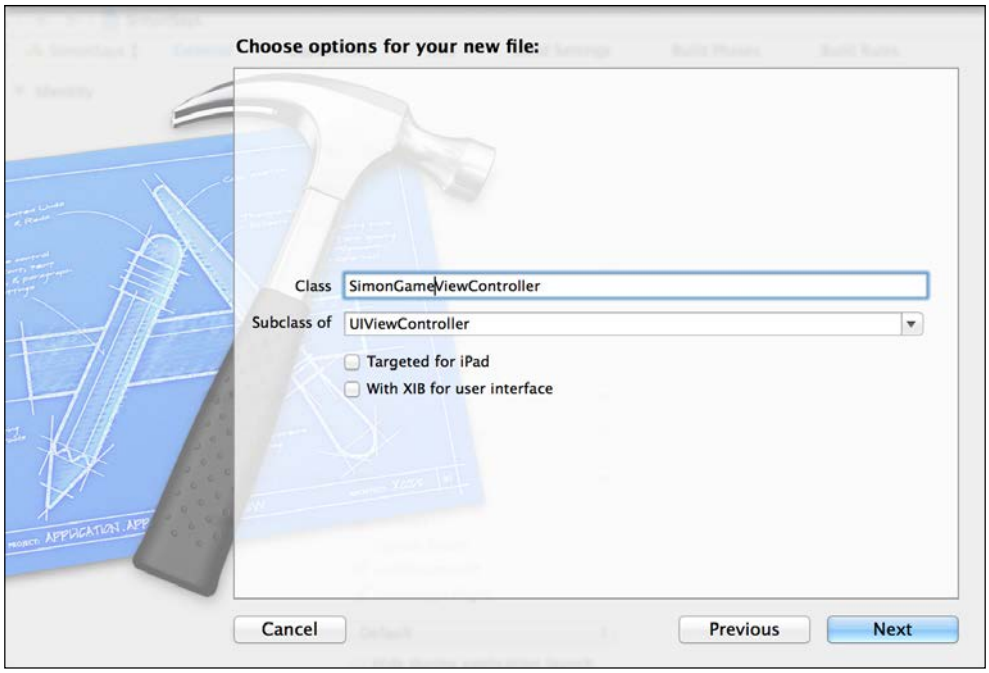
www.it-ebooks.info

Now that we have a basic idea of what the game will do, let's make our standard *Single View Application Project*. We will call this one `SimonSays` and save it to a new directory:

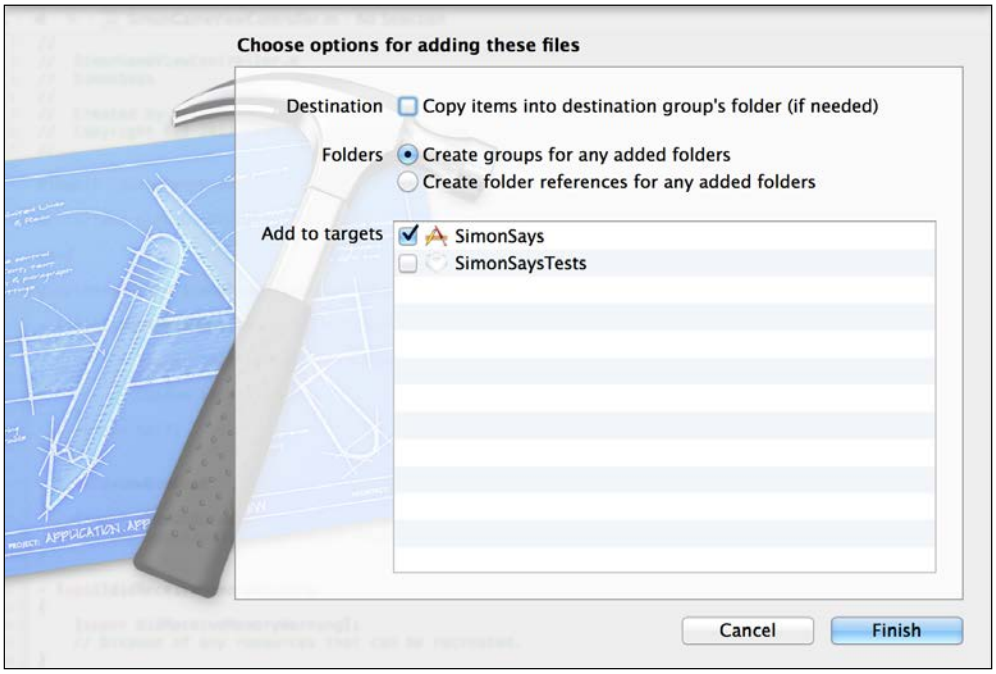


Create a new Objective-C class file for our game view controller with a subclass of `UIViewController`. We will call it `SimonGameViewController`, as shown in the following screenshot, so that we don't confuse it with other game view controllers:

www.it-ebooks.info

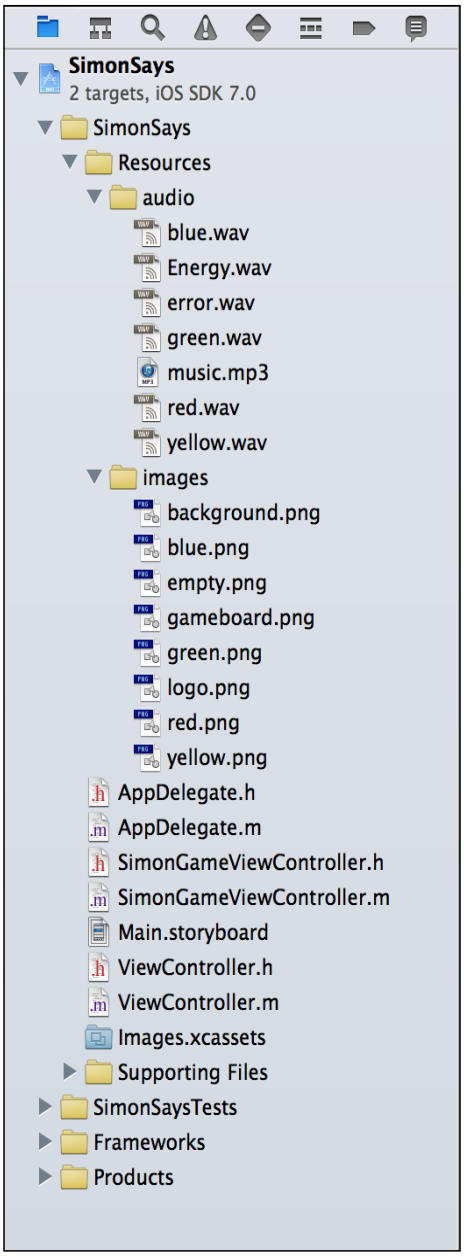


Next, you will need to drag in the `resources` folder for the Simon Says game:



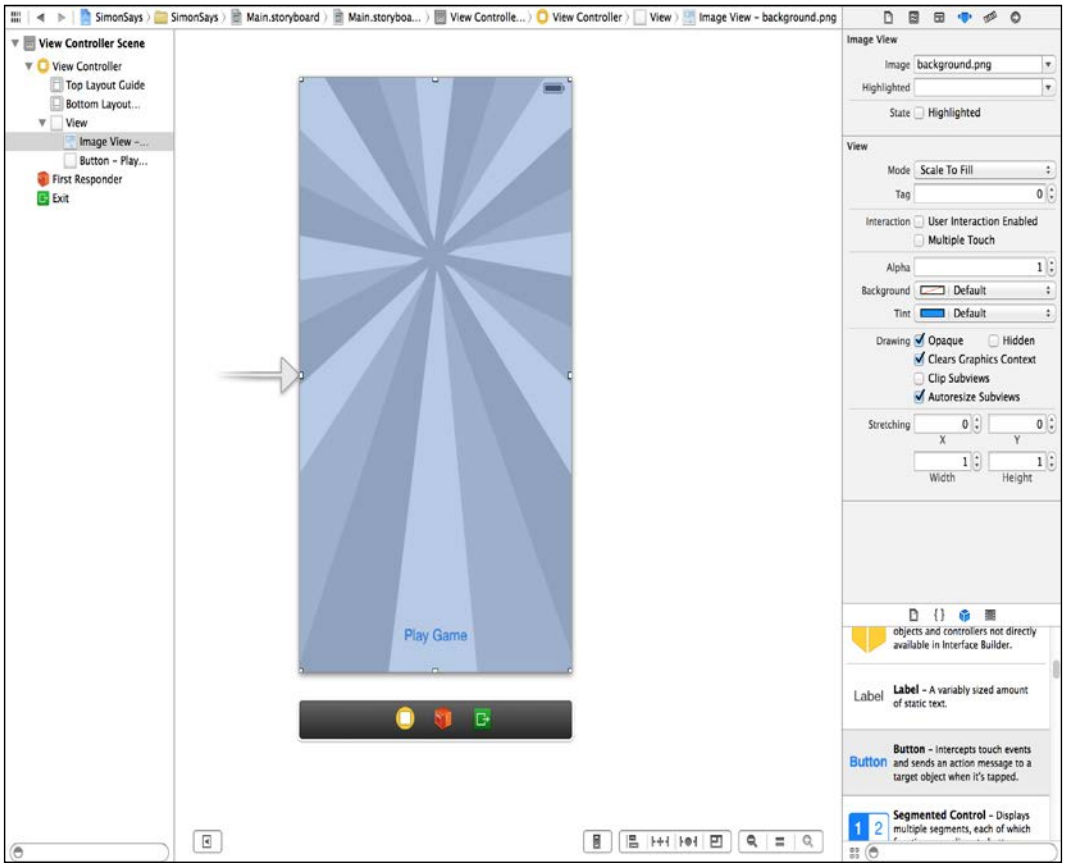
www.it-ebooks.info

The `resources` directory contains all of the graphics and audio we will be using for this chapter. As you can see, we are going to be using WAV files for our sound effects and an MP3 file for our background music. Make sure **Copy items into destination group's folder (if needed)** is checked when you drag your files in.



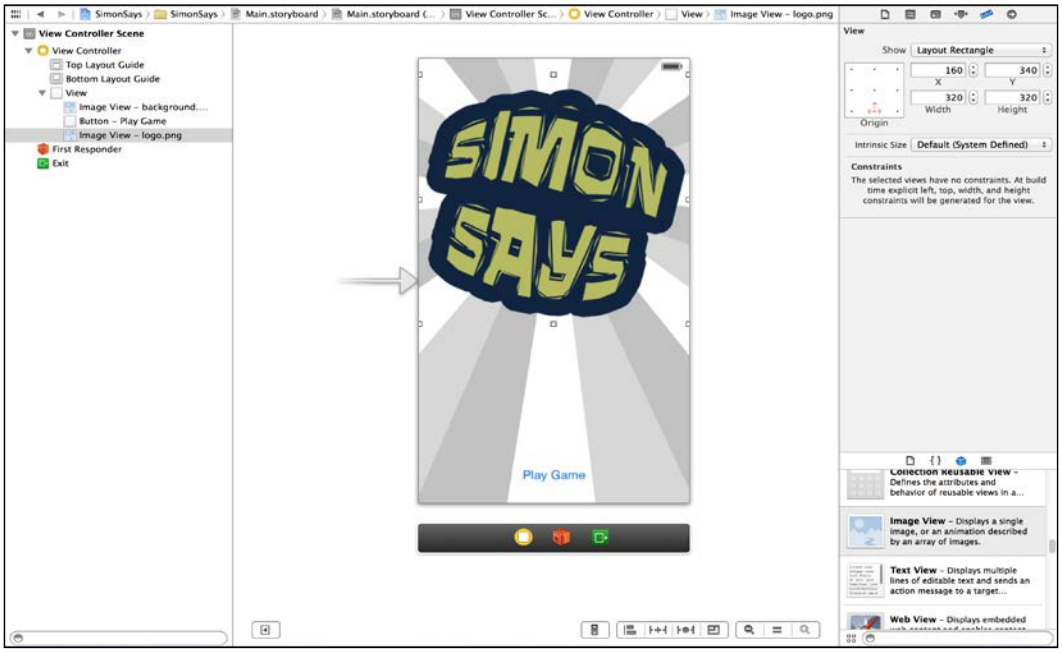
www.it-ebooks.info

Let's set up our main menu area first. For this project, we are not going to set our phone's screen size to the smaller size, and I will explain more about this in a few minutes. Drag an image view from the **Object Library** and then a **Button** from our **Object Library**. Name the button `Play Game` and set the **Image** option of the **Image View** panel to the `background.png` image file from our `resources` directory. Scale the image so that it takes up the full screen of the playable area, as shown in the following screenshot:



www.it-ebooks.info

Next, drag another image view and set its **Image** to `logo.png`. In your size inspector, set the size information as follows: **X** to `160`, **Y** to `320`, **Width** to `320`, and **Height** to `320`, and it will place it perfectly for the logo as shown in the following screenshot:



Now let's create our gameplay area. In our `ViewController.h` file, we need to add an unwind method to let our player exit the game cleanly. So, add the following line of code to your header:

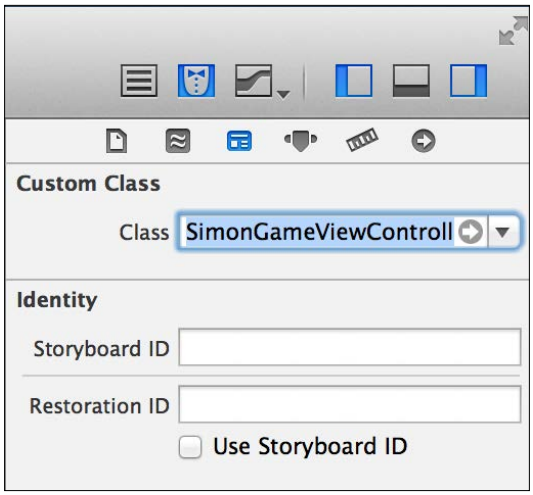
```
- (IBAction)unwindToThisView:(UIStoryboardSegue *)unwindSegue;
```

In our `ViewController.m` implementation file, we need to add the following method:

```
(IBAction)unwindToThisView:(UIStoryboardSegue *)unwindSegue
{
}
```


www.it-ebooks.info

Back in our storyboard, we need to drag out a new view controller from our Object Library. Select your view controller's identity inspector and set the class to **SimonGameViewController**, as shown in the following screenshot, so that our class is tied to our view:

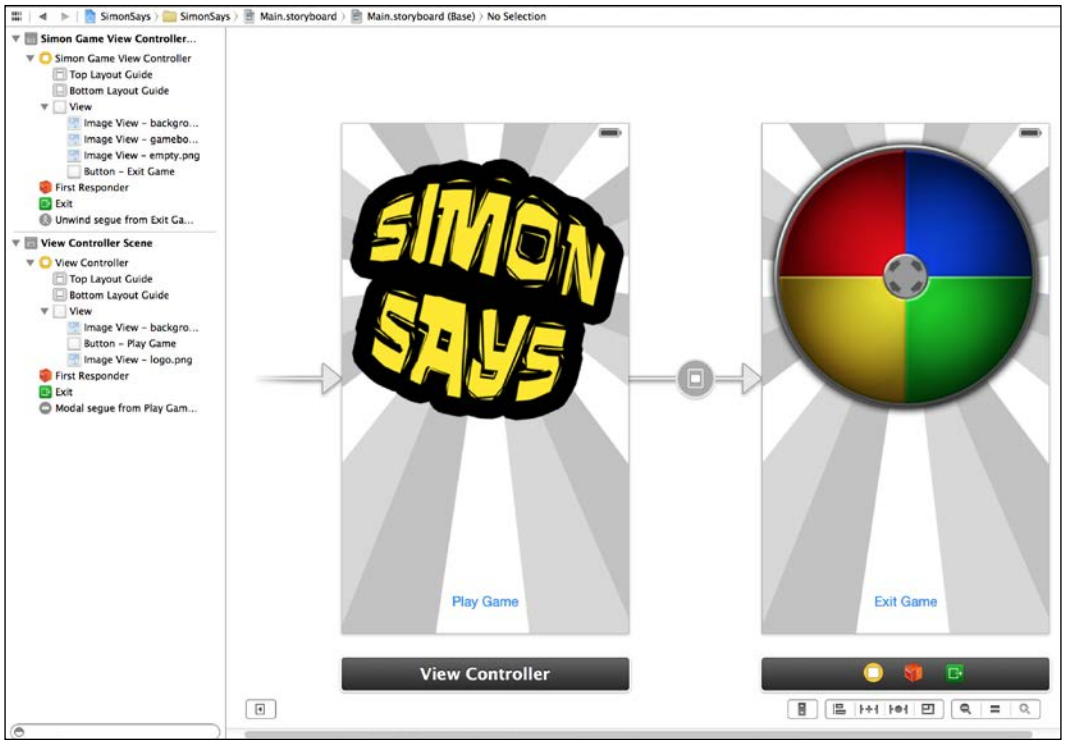


Hold your *Ctrl* key and click and drag your **Play game** button to your new view controller and set **Action Segue** to **Modal**. Feel free to set the transition however you like, but I really like the way that **Cover Vertical** looks.

Drag out an image view to **SimonGameViewController** and set it to fullscreen with our background image, exactly like we did earlier. Then, drag out a **Button** object and name it `Exit Game`. Hold the *Ctrl* key and click-and-drag the **Exit** button to the exit scene button below your view. Select the `unwindToThisView` method that will show up in the action segue list:

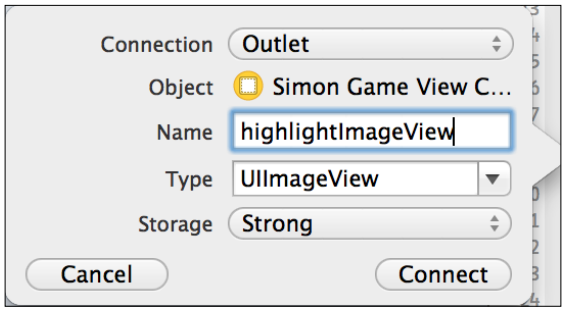
www.it-ebooks.info

Drag in two more image views and set the first image to `gameboard.png` and the second one to `empty.png`. In your size inspector, set the size information for both images as follows: **X** to `160`, **Y** to `330`, **Width** to `320`, and **Height** to `320`. This will place it perfectly for the logo as shown in the following screenshot:

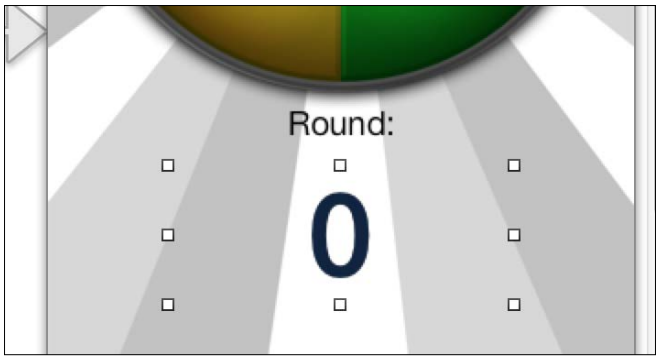


You will notice that in the list on the left of our background, image is at the very top, followed by the game board, then the empty image. If you think of these as layers of paper, the empty image is on top. We are going to use this area to swap out the empty image with the highlight color button images whenever we want the button to look like it is lighting up. We will switch it back to the empty image when we are done with it so it looks like it is going dark again. In order to do this, we will have to create an outlet for our empty image view. Open your assistant editor from your edit menu and set the **Asset Editor** view to **SimonGameViewController.h**. Drag out a new **Label** from the **Object Library** to be used as a round number display area. Hold the *Ctrl* key and click-and-drag your new round number display to the header to create an outlet for your empty image. An **outlet** is a connection between your code and an object in your storyboard. The outlet's connection is established as soon as your file is loaded. Name it `highlightImageView` with the storage set to **Strong**. You should see a new line of code in your header file that says `@property (strong, nonatomic) IBOutlet UIImageView *highlightImageView;`.

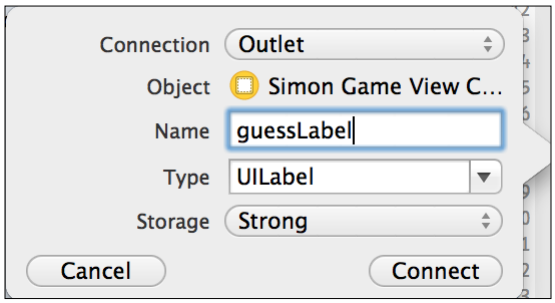
www.it-ebooks.info



We are going to add two labels to our layout. The first is going to be static and does not change. Just set the text to `Round:` and set the font to around 18-point and align it to the middle of the screen, placing it just below the game board. The second is going to be our round number. Set the text to `0`, as shown in the following screenshot, and the font to 64-point aligned in the center of the screen:



Open your assistant editor from your edit menu and set the bottom view to `SimonGameViewController.h`. Hold the *Ctrl* key and click-and-drag your new round number to the header to create an outlet for your round number. Name it `guessLabel`; you should see a new line of code in your header file that says `@property (strong, nonatomic) IBOutlet UILabel *guessLabel;`.

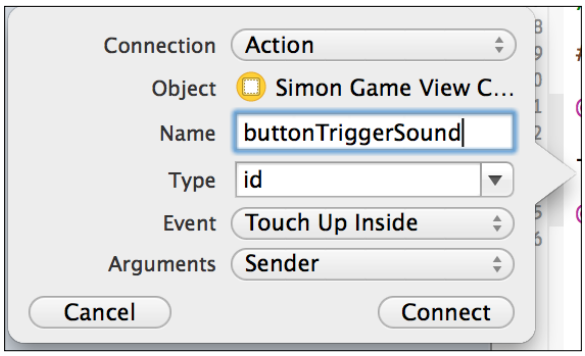


www.it-ebooks.info

We are going to create four invisible buttons for the user to tap to select a color. No name will be given for the text but they are going to have tag numbers so that we can distinguish them in the code. Tags are a very useful way to pass information into your code from the view level of your project. Drag a button out from your object library. In your attributes inspector under **View**, set the **Tag** number to **1**, then in your size inspector, set the size information as follows: **X** to `20`, **Y** to `30`, **Width** to `140`, and **Height** to `140`. You need to do this three more times. Here are all of the settings for your buttons:

- For Tag 2, set the values as follows: **X** to `160`, **Y** to `30`, **Width** to `140`, and **Height** to `140`
- For Tag 3, set the values as follows: **X** to `20`, **Y** to `170`, **Width** to `140`, and **Height** to `140`
- For Tag 4, set the values as follows: **X** to `160`, **Y** to `170`, **Width** to `140`, and **Height** to `140`

Open your assistant editor from your edit menu and set its view to `SimonGameViewController.h`. Hold the *Ctrl* key and click-and-drag your new button to your header file to create a new button action called `triggerButtonSound`:

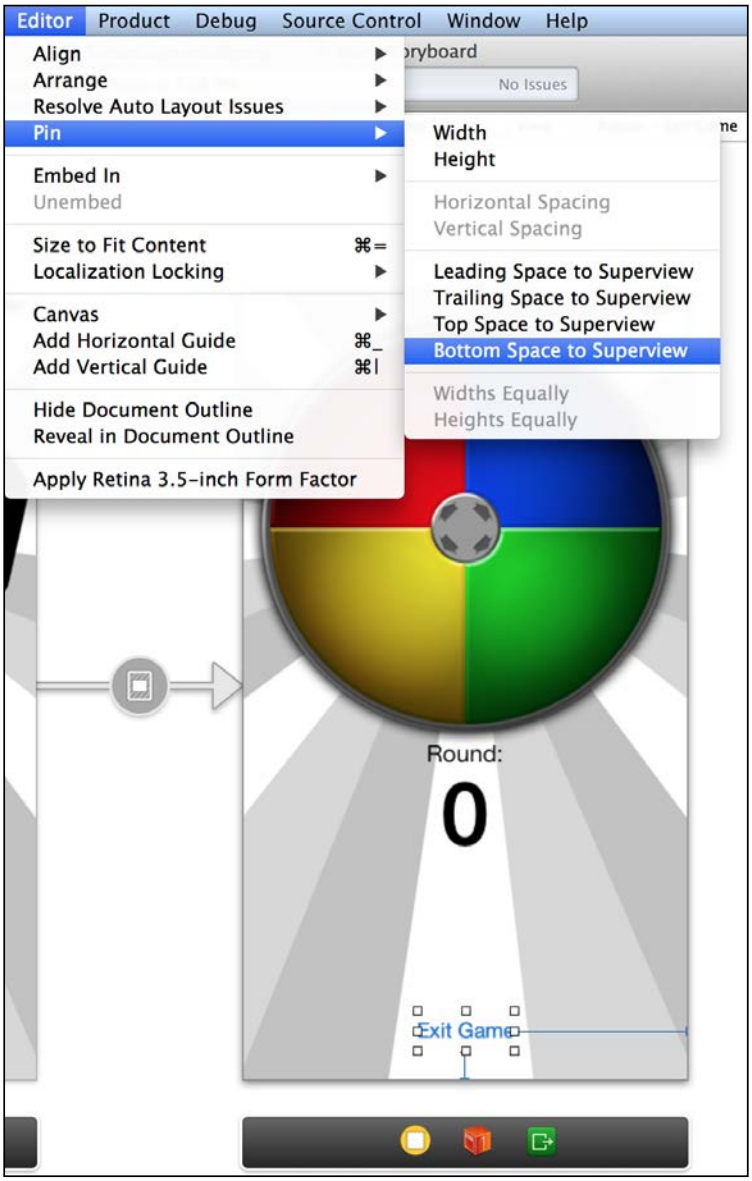


Link all of your new invisible buttons to the `triggerButtonSound` method by pressing the *Ctrl* key and clicking-and-dragging each button to the new method in the header. While we are at it, let's create a similar action for our exit button, **exitGame**, so that we can clean up a bit when we exit back to the main menu.

One last thing before we move on. As I mentioned earlier, do not change the screen size now. Since we are building for the iPhone 3G/3GS, 4, and 5, the screen size changes. The iPhone 3 and 4 are slightly smaller when you are looking at the aspect ratios. If we set up our game for the iPhone 5 and someone on the iPhone 4 were to play it, the exit and play buttons would disappear below the screen area.

www.it-ebooks.info

Close your assistant editor if you need more room to work with your storyboard. What we need to do is pin the buttons to the bottom of the screen. Select your **exit** button and navigate to **Editor | Pin | Bottom Space to Superview**. Now, reselect your button and navigate to **Editor | Pin | Trailing Space to Superview** from your menu. This will tie your button to the bottom of your screen so that it will always be 8 points above the bottom of your screen and 145 points from the right of the screen no matter what the screen size. Do the same with your **Play Game** button.



www.it-ebooks.info

Now that our environment is all set up, let's talk about the code involved. Since this is your first time playing audio, we are going to take the easiest route. Most importantly, we are going to connect to Apple's AV Foundation Class. AV Foundation Class is Apple's framework that provides essential services for working with audiovisual media files. We connect to AV Foundation Class by adding in the `#import <AVFoundation/AVFoundation.h>` statement right below our `#import <UIKit/UIKit.h>` statement in our `SimonGameViewController` header file. We are going to make our `SimonGameViewController` header file a delegate of the `AVAudioPlayer` class. According to the Apple documentation:

> "*A delegate is an object that acts on behalf of, or in coordination with, another object when that object encounters an event in a program.*"

For us, this means that this will give us access to a few extra methods in the `AVAudioPlayer` class for controlling and responding to events related to audio files.

We are going to have a few properties for controlling our game. I am setting up our `gameState` variable as a string since I am only checking it in a couple of places; this is just another way of keeping track of a game state. Our `roundNumber` variable is a simple integer that we will increment when a player gets a correct answer and reset it when a player misses an answer. The `guessList` array is the list of colors that a player needs to guess in order to pass each round. We will add a new color to the list after each round and when a player misses, we will empty it out and start over. The `playerListTimer` is a timer that gets called each time we want a sound to play but will wait for a second before playing the next sound. Once we have all of these changes, our header file should look similar to the following code snippet:

```
#import <UIKit/UIKit.h>
#import <AVFoundation/AVFoundation.h>

@interface SimonGameViewController : UIViewController
<AVAudioPlayerDelegate>

@property NSMutableArray *guessList;
@property NSInteger roundNumber;
@property NSString *gameState;
@property NSTimer *playListTimer;
@property (strong, nonatomic) IBOutlet UILabel *guessLabel;
@property (strong, nonatomic) IBOutlet UIImageView
*highlightImageView;

- (IBAction)exitGame:(id)sender;
- (IBAction) triggerButtonSound:(id)sender;

@end
```


www.it-ebooks.info

In the next set of code, we are going to set up and play the background music for our game. In order to play an audio file, you need only two lines of code. The first line defines a path to your file with a callback for errors and the second line tells the file to play:

```
backgroundMusic = [[AVAudioPlayer alloc] initWithContentsOfURL:
[[NSBundle mainBundle] URLForResource:@"MyFileName" withExtension:@".
mp3"]; error:nil];
[backgroundMusic play];
```

Since we are not just creating an audio player, let's break down our implementation file for our game view controller. We start with importing our `SimonGameViewController` header file and define `@implementation` for our `SimonGameViewController` class. We follow it up with four class variables. A new one for us is the `AVAudioPlayer` type that is available once we've linked our class to the `AVFoundation` class in our header file. We are only defining two sound layers for now, one for our sound effects (`audioEffect`) and one for our background music (`backgroundMusic`). If we were going to have a lot of different audio files playing, it would be good to add each of the audio files into an array and tell them to play from the array, but it is really not necessary for our simple game:

```
#import "SimonGameViewController.h"

@implementation SimonGameViewController
AVAudioPlayer *audioEffect;
AVAudioPlayer *backgroundMusic;
```

Let's keep track of the current sound number and the current guess number as follows:

```
int currentSoundNo;
int guessNumber;

- (id)initWithNibName:(NSString *)nibNameOrNil bundle:(NSBundle *)
nibBundleOrNil
{
  self = [super initWithNibName:nibNameOrNil
                bundle:nibBundleOrNil];
  if (self)
  {
    // Custom initialization
  }
  return self;
}
```


www.it-ebooks.info

We are going to initialize our game state, create `NSMutableArray` for our guesses, and call our `newGame` method, as shown in the following code:

```
- (void)viewDidLoad
{
  [super viewDidLoad];
  self.gameState = @"Loading";
  self.guessList = [[NSMutableArray alloc]init];

  [self newGame];
}
```

Each time we start our game over, we reset `roundNumber` and `guessNumber`, clear out the `guessList` by re-initializing it, and we add a guess to the list. Then, we tell our music to start playing. We will also call the `newGame` method from `viewDidLoad` so the game gets set up when we first enter the game screen:

```
-(void) newGame
{
  self.roundNumber = 1;
  guessNumber = 0;

  [self.guessList removeAllObjects];
  [self addAGuess];
  [self playMusic: @"music"];
}
```

Here is how we tell our MP3 background music to start playing. For our background music player, I passed in a filename as `musicName`. I want to make sure we have not already started the background music file, so we check to make sure our variable is not set to `nil`. Next, we set a path to our file as a `NSURL` string. We initialize our sound file like we explained in the preceding code. We send `prepareToPlay` to preload our sound file. `AVAudioPlayer` lets us control several aspects of our audio files, including the number of times it loops as well as volume levels. We are not setting them here, but you can also set the panning and playback rate for our files. By setting our number of loops to `-1`, we are telling it to loop indefinitely. Now we just tell it to play:

```
-(void)playMusic: (NSString *)musicName
{
  //play an MP3 music file
  if(backgroundMusic == nil)
  {
    NSURL *musicFile = [[NSBundle mainBundle]
      URLForResource:musicName withExtension: @"mp3"];
    backgroundMusic = [[AVAudioPlayer alloc]
       initWithContentsOfURL:musicFile error:nil];
    [backgroundMusic prepareToPlay];
```


www.it-ebooks.info

```
      backgroundMusic.numberOfLoops = -1;
      backgroundMusic.volume = 0.25;

      [backgroundMusic play];
    }
  }
```

I will change the game state here to `GameTurn`, since it is the game's turn to add a guess to the file. Since this is the first sound that we will add to the playlist, I reset some of the variables. The sound we choose will be sent as an integer between one and four, so we pick a random number and check the mod against four and add one to the remainder. We create `NSNumber` from the random number so that we can add it to `NSMutableArray`. We set the text of the guess label to the round number. Then, we call our `playSoundList` method after one second:

```
-(void) addAGuess
{
  self.gameState = @"GameTurn";
  currentSoundNo = 0;
  guessNumber = 0;
  int randEnemy = (arc4random() % 4)+1;
  NSNumber *soundNo = [NSNumber numberWithInt:randEnemy];
  [self.guessList addObject: soundNo];
  self.guessLabel.text = [NSString stringWithFormat:@"%ld", (long)
self.roundNumber];
   self.playListTimer = [NSTimer scheduledTimerWithTimeInterval:1
                           target:self
                           selector:@selector(playSoundList)
                           userInfo:nil
                           repeats:NO];

}
```

We start the `playSoundList` method by getting the current sound number (between one and four) from `guessList` and we call our `playSound` method with a long integer. We increment our `currentSoundNumber` variable, then we check to see if it is larger than `guessList`. We really have not done anything with recursion until now. If `currentSoundNo` is *not* larger, we are going to keep calling the `playSoundList` method until we run out of sounds from `guessList`. If it is larger, we have run out of sounds which the system is supposed to play, so we make sure our image is reset to empty after half a second:

```
-(void)playSoundList
{
  NSNumber *guessNo= self.guessList[currentSoundNo];
  [self playSound: (long)[guessNo integerValue]];
```


www.it-ebooks.info

```
    currentSoundNo ++;
    if(currentSoundNo < self.guessList.count)
    {
      self.playListTimer = [NSTimer scheduledTimerWithTimeInterval:1
                              target:self
                              selector:@selector(playSoundList)
                              userInfo:nil
                              repeats:NO];
    }
    else
    {
      self.gameState = @"Play";
      [NSTimer scheduledTimerWithTimeInterval:.5
                              target:self
                              selector:@selector(hideButtonImage)
                              userInfo:nil
                              repeats:NO];
    }
  }
```

Our `playSound` method is called by both our game's code and when a player pushes a button. It takes a number and assigns a color based on the number in our `switch` statement. We made sure to name our highlight images the same name that we named our sound files. So, once we know `soundName`, we use that same name to change our highlight image to one of our color images and we call our `playSoundEffect` method using the same name:

```
-(void) playSound: (long) tag
{
  NSString *soundName = NULL;

  switch (tag)
  {
    case 1:
      soundName = @"red";
      break;
    case 2:
      soundName = @"blue";
      break;
    case 3:
      soundName = @"yellow";
      break;
    case 4:
      soundName = @"green";
      break;
```


www.it-ebooks.info

```
      default:
        soundName = @"error";
        break;
    }
    [self.highlightImageView setImage:[UIImage imageNamed:
              [NSString stringWithFormat:@"%@.png", soundName]]];
    [self playSoundEffect: soundName];
  }
```

Playing our sound effects is very similar to how we are playing our music file, but we are going to keep overwriting our `audioEffect` variable with a new sound file each time we call it. We are also not passing a file type so you can see that it is not necessary. We also set the number of loops to zero since we don't want it to loop but only play once. If we set our audio files up in an array, we could preload all of the files like this and then just call the audio variable to play the sound like the last line `[myAudioVariable play]`, but this could also cause an echo effect, which I didn't want with this game:

```
-(void)playSoundEffect: (NSString *)effectName
{
  NSString *path  = [[NSBundle mainBundle] pathForResource: effectName
ofType:@"wav"];
  NSURL *pathURL = [NSURL fileURLWithPath : path];
  audioEffect = [[AVAudioPlayer alloc] initWithContentsOfURL:pathURL
error:nil];
  audioEffect.numberOfLoops = 0;
  [audioEffect prepareToPlay];
  [audioEffect play];
}
```

When exiting the game, we need to make sure we stop the timer and all of the playing audio. The audio stop code is closer to the end of the file as follows:

```
- (IBAction)exitGame:(id)sender
{
  [self.playListTimer invalidate];
  self.playListTimer = nil;
  [self stopAllAudio];
}
```

Whenever one of our color buttons is pushed, it comes here with its ID. We are using the button's ID to pick up it's tag number that we set up earlier. We compare the tag number with the current guess number of `guessList`. If they match, we know that the player picked the right color in the right order. If they didn't choose correctly, we play the error sound and start a new game.

www.it-ebooks.info

You will see that we set up a 0.5 second timer to turn off the highlight button. We do this so that the highlight looks lit up when the player pushes the button. If we turned it off without the timer, the player wouldn't feel like they pushed the button.

Finally, we check to see whether the user guessed the last item in the list; if they did, we move on to the next round as follows:

```
- (IBAction) triggerButtonSound:(id)sender
{
  if( ![ self.gameState isEqualToString: @"Play"] ) return;
  //  long tagNo = (long)[sender tag];

  if([self.guessList[guessNumber]intValue] != (long)[sender tag])
  {
    self.roundNumber = 0;
    [self playSound: -1];
    [self newGame];
    return;
  }
  else
  {
    //player guessed correctly
    [self playSound: [sender tag]];
    [NSTimer scheduledTimerWithTimeInterval:.5
                            target:self
                            selector:@selector(hideButtonImage)
                            userInfo:nil
                            repeats:NO];
    guessNumber++;
    if (guessNumber == self.roundNumber)
    {
      //you have guessed all the right answers for this round
      self.roundNumber++;
      [self addAGuess];
    }
  }
}
-(void)hideButtonImage
{
  [self.highlightImageView setImage:[UIImage
    imageNamed:@"empty.png"]];
}
```


www.it-ebooks.info

See, I told you it was here. These three methods send a stop call to each of our audio variables and set them to nil. The call to stop audio is simply `[audioVariable stop]` as shown in the following code:

```
-(void)stopAllAudio
{
  [self stopMusic];
  [self stopSound];
}

-(void)stopMusic
{
  [audioEffect stop];
  audioEffect = nil;
}

-(void)stopSound
{
  [backgroundMusic stop];
  backgroundMusic = nil;
}

- (void)didReceiveMemoryWarning
{
  [super didReceiveMemoryWarning];
  // Dispose of any resources that can be recreated.
}
```

Here is our exit game action that is tied to the exit button. We want to make sure we clean out all of the audio if the player exits the game, so the next time they start up, they come in clean:

```
- (IBAction)ExitGame:(id)sender
{
  [self stopAllAudio];
}
```

We mentioned earlier that we set up our class as a delegate of the `AVAudioPlayer` class. These next two methods are from that class. Whenever we receive a phone call on our iPhone or our game gets interrupted for another reason, these methods get called. We can use these to maintain control over our game's audio when we get interrupted outside our game's control. This is very important for the control freak in all of us.

```
(void)audioPlayerBeginInterruption:(AVAudioPlayer *)player
{
/* Audio Session is interrupted. The player will be paused here */
```


www.it-ebooks.info

```
}
- (void) audioPlayerEndInterruption:(AVAudioPlayer *)player
                        withOptions:(NSUInteger)flags
{
  if (flags == AVAudioSessionInterruptionOptionShouldResume)
  {
    [self playMusic: @"music"];
  }
}

@end
```

Recording audio

The `AVAudioRecorder` class is very similar and is part of the `AVFoundation` class as well. Here are the differences: you would use the `prepareToRecord` and `record` methods instead of `prepareToPlay` and `play`. This is not really used in games very often but it is still very important to know:

```
NSURL *audioRecordingURL = [self audioRecordingPath];
self.audioRecorder = [[AVAudioRecorder alloc]
                      initWithURL:audioRecordingURL
                      settings:[self
                        audioRecordingSettings]
                      error:nil];
```

Summary

In this chapter, we learned the basics of playing and controlling audio files. With what you learned here, you should be able to add audio to just about any audio application. We also went over all of the information you would need in order to create the audio files so you can play them back on an iPhone. We also discussed recursion, delegates, and using timers with images to get a better feel for button presses.

In the next chapter, we will cover how the users can interact with their iPhone and how to put that data to use in our games. We will also discuss touch data, motion data, tilting, and gestures as we apply them to a simple putt-putt golf game.

www.it-ebooks.info

7
User Interactivity – Mini Golf

In this chapter, we are going to learn about user interaction and handling events passed from the iPhone's screen sent from the event information to your game. This includes when a user touches, taps, or drags their finger on their screen or moves, shakes, or tilts their device. The following are the points we will touch upon in this chapter:

- Using user input and touch events
- Using gestures in iOS apps
- Touch start, move, and end
- Other mechanisms

Using user input and touch events

A **touch event** occurs each time a user touches the screen, drags, or releases the screen, and also during an interruption. Touch events begin with a user touching the screen. The touches are all handled by the `UIResponder` class, which then causes a `UIEvent` object to be generated, which then passes your code to a `UITouch` object for each finger touching the screen. For most of the code you work on, you will be concerned about the basic information. Where did the user start to touch the screen? Did they drag their finger across the screen? And, where did they let go? You will also want to know if the touch event got interrupted or canceled by another event. All of these situations can be handled in your view controller code using the following four methods (You can probably figure out what each one does.):

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

www.it-ebooks.info

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

These methods are structured pretty simply, and we will show you how to implement these a little later in the chapter. When a touch event occurs on an object, let's say a button, that object will maintain the touch event until it has ended. So, if you touch a button and drag your finger outside the button, the touch-ended event is sent to the button as a *touch up outside* event. This is very important to know when you are setting up touches for your code. If you have a set of touch events associated with your background and another set associated with a button in the foreground, you need to understand that if `touchesBegan` begins on the button and `touchesEnded` ends on the background, `touchesEnded` is still passed to your button.

In this chapter, we will be creating a simple Mini Golf game. We start with the ball in a standard start position on the screen. The user can touch anywhere on the screen and drag their finger in any direction and release. The ball will move in the direction the player dragged their finger, and the longer the distance from their original position, the more powerful the shot will be. In the Mini Golf game, we put the touch events on the background and not the ball for one simple reason. When the user's ball stops close to the edge of the screen, we still want them to be able to drag a long distance for more power.

Using gestures in iOS apps

There are special types of touch events known as **gestures**. Gestures are used all over in iOS apps, but most notable is their use in maps. The simplest gesture would be tap, which is used for selecting items; however, gestures such as pinch and rotate are used for scaling and, well, rotating.

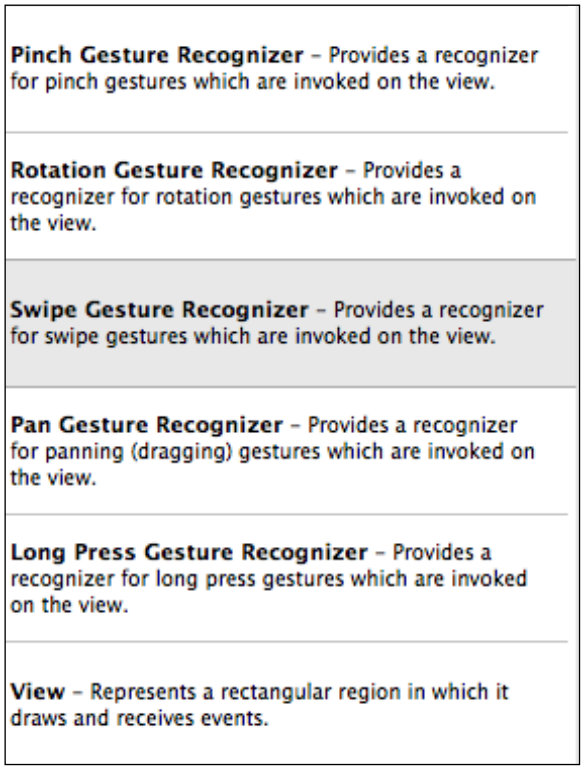
The most common gestures have been put together in the **Gesture Recognizer** classes from Apple, and they work on all iOS devices. These gestures are tap, pinch, rotate, swipe, pan, and long press. You will see all of the following items listed in your **Object Library**:

- **Tap**: This is a simple touch and release on the screen. A tap may also include multiple taps.
- **Pinch**: This is a gesture involving a two-finger touch and dragging of your fingers together or apart.

www.it-ebooks.info

- **Rotate**: This is a gesture involving a two finger-touch and rotation of your fingers in a circle.
- **Swipe**: This gesture involves holding a touch down and then dragging to a release point.
- **Pan**: This gesture involves holding a touch and dragging. A pan does not have an end point like the swipe but instead recognizes direction.
- **Long press**: This gesture involves a simple touch and hold for a prespecified minimum amount of time.

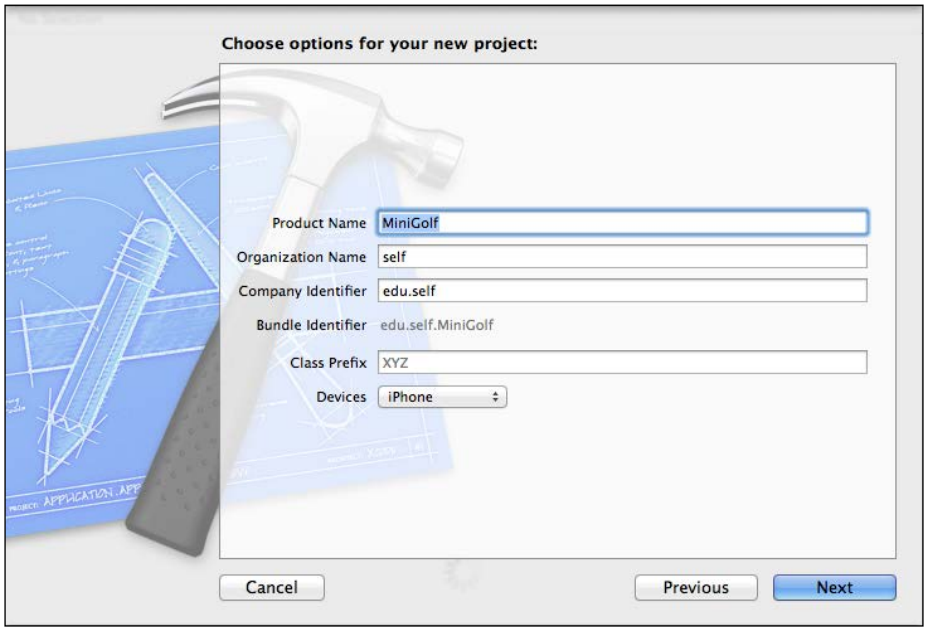
The following screenshot displays the **Object Library**:



www.it-ebooks.info

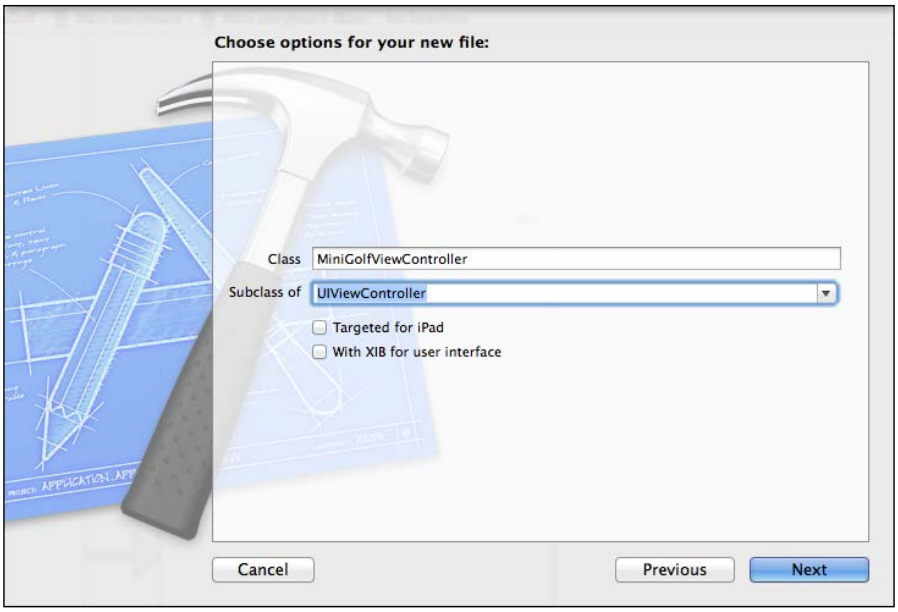
Let's get started on our new game as we show you how to integrate a gesture into your game:

1. We'll start by making our standard **Single View Application** project. Let's call this one `MiniGolf` and save it to a new directory, as shown in the following screenshot:

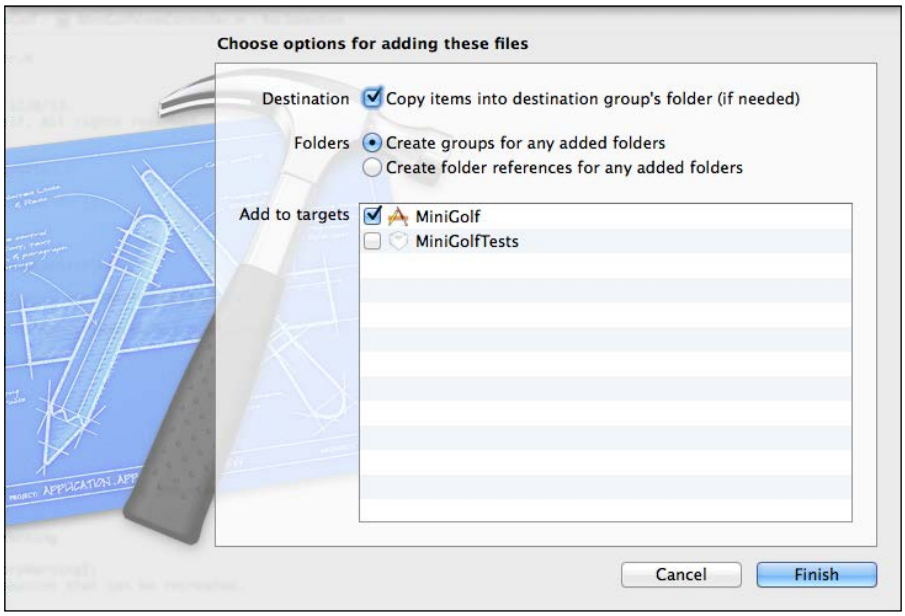


2. Once your project has been created, select it and uncheck **Landscape Left** and **Landscape Right** in your **Deployment Info**.
3. Create a new Objective-C class file for our game view controller with a subclass of `UIViewController`, which we will call `MiniGolfViewController` so that we don't confuse it with our other game view controllers:

www.it-ebooks.info

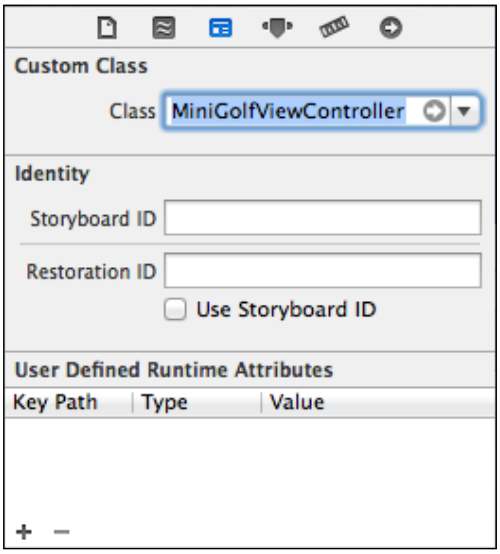


4. Next, you will need to drag it into the **Resources** folder for the Mini Golf game:



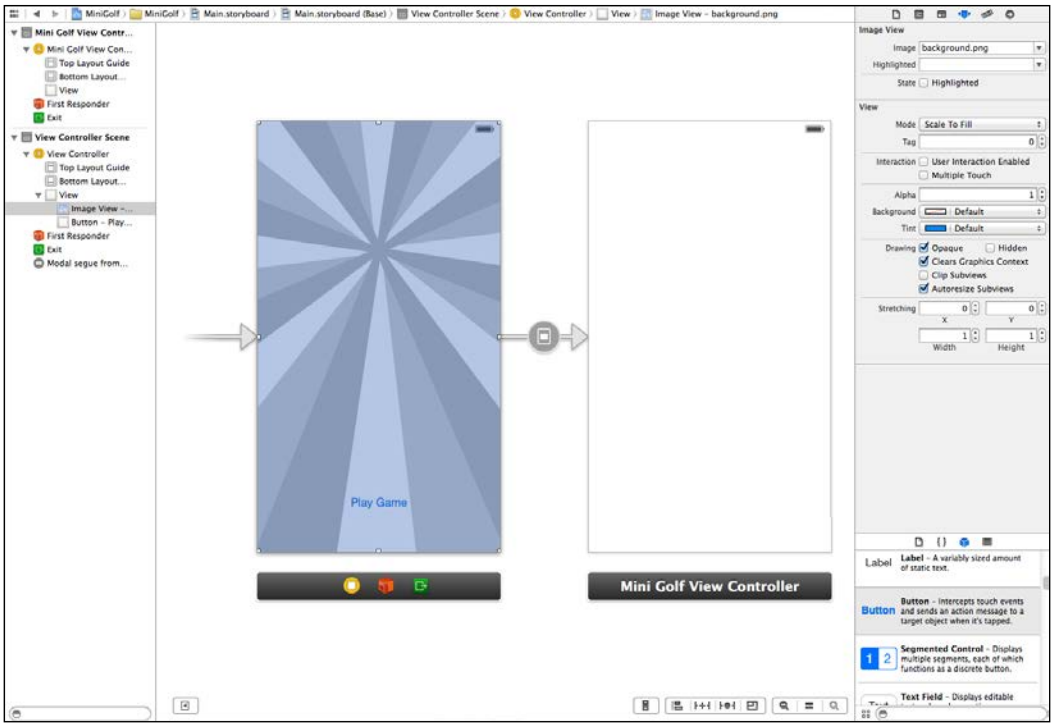
www.it-ebooks.info

5. Select your **Main.storyboard**, and let's get our initial menu set up.
6. Drag in a new **View Controller** object from your **Objects Library**. We are only going to be working with two screens for this game.
7. For the new **View Controller object**, let's set the custom class to your new **MiniGolfViewController**:



8. Let's add in a background and a button to segue into our new **Mini Golf View Controller** scene.
9. Drag out an **Image View** object from our **Object Library** and set it to full screen.
10. Drag out a **Button** object and name it `Play Game`.
11. Press *control*, click on the **Play Game** button, and drag over to the **Mini Golf View Controller** scene to create a modal segue, as shown in the next screenshot. We wouldn't normally do this with a game, but let's use gestures to create a segue from the main menu to start the game.

www.it-ebooks.info



12. In the `ViewController.m` file, add in your unwind code:

```
- (IBAction)unwindToThisView:(UIStoryboardSegue *)unwindSegue
{
}
```

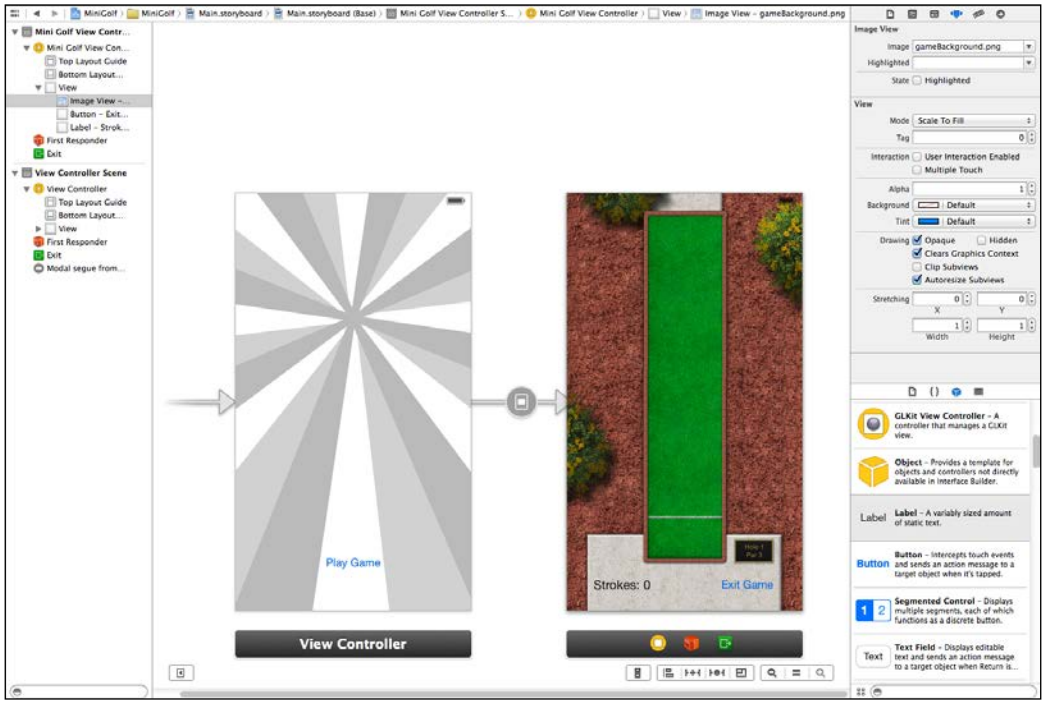
In your `ViewController.h` file, add in the corresponding action declaration:

```
- (IBAction)unwindToThisView:(UIStoryboardSegue *)unwindSegue;
```

13. In the **Mini Golf View Controller**, let's drag out an **Image View** object, set it to full screen, and set the image to `gameBackground.png`.

www.it-ebooks.info

14. Drag out a **Button** object and call it `Exit Game` and a **Label** object and set **Strokes** to `0` in the text area. It should look similar to the following screenshot:



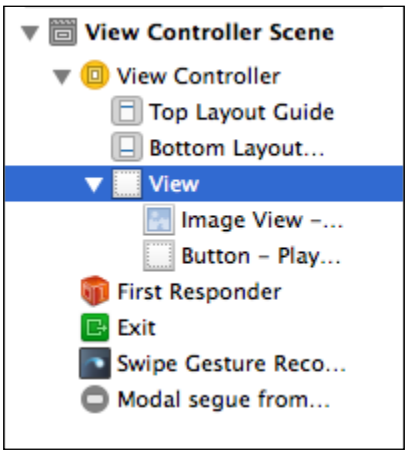
15. Press *control* and click-and-drag your **Exit Game** button to the **Exit unwind segue** button, as shown in the following screenshot:

www.it-ebooks.info

If you are planning on running this game in multiple screen sizes, you should pin your images and buttons in place, as we explained in the previous chapter. If you are just running your code in the iPhone 4 simulator, you should be fine.

That should just about do it for the usual prep work. Now, let's get into some gestures. If you run your game right now, you should be able to go into your game area and exit back out. But what if we wanted to set it up so that when you touch the screen and swipe right, you would use a different segue to enter the gameplay area? The actions performed on the gestures are as follows:

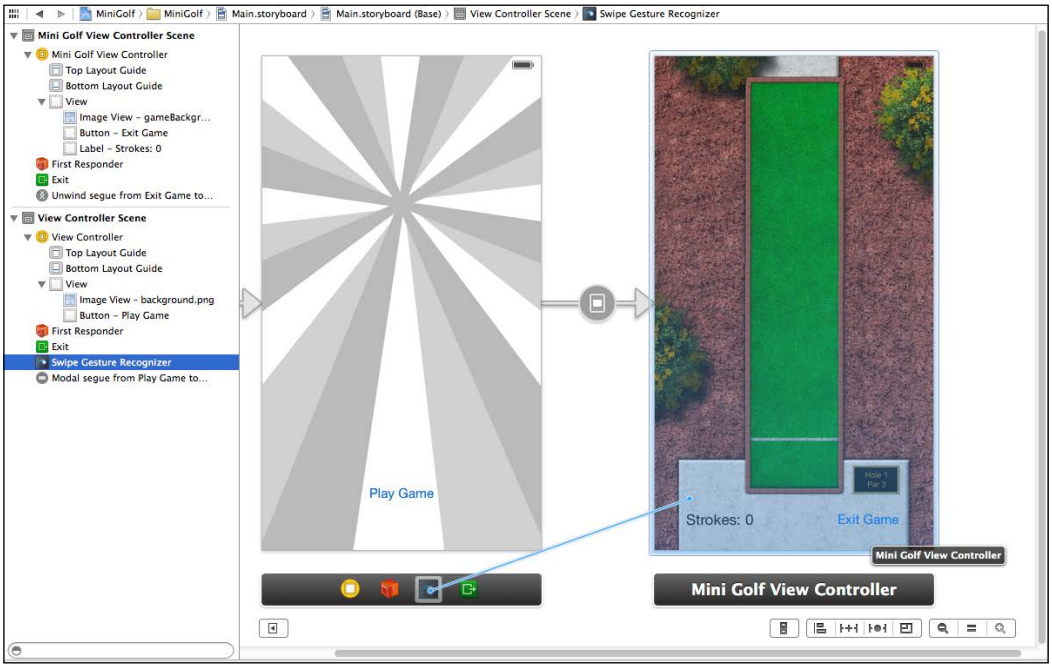
1. Select your view and drag out **Swipe Gesture Recognizer** from your **Objects Library** onto your view:



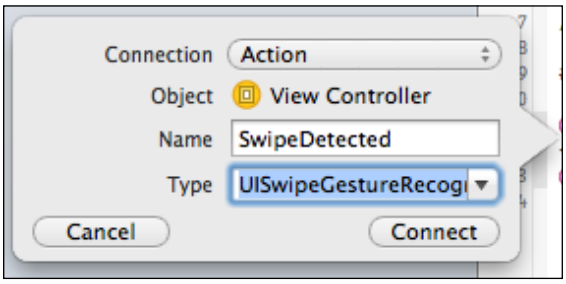
2. Press *control* and drag your **Swipe Gesture Recognizer** to your **Mini Golf View Controller** just like you did with your **Play Game** button.
3. Choose a modal segue and you are done.

www.it-ebooks.info

For each of the different types of gestures that are offered in the **Object Library**, there are unique settings in the attributes inspector. For the swipe gesture, you can choose the swipe direction and the number of touches required to run. For example, you could set up a two-finger left swipe just by changing a few settings, as shown in the following screenshot:



You could just as easily press *control* and click-and-drag from your **Swipe Gesture Recognizer** to your header file to add `IBAction` associated with your swipe; this way, you can control what happens when you swipe. You will need to set **Type** to `UISwipeGestureRecognizer`:



www.it-ebooks.info

Once you have done this, a new `IBAction` function will be added to both your header file and your implementation file:

```
- (IBAction)SwipeDetected:(UISwipeGestureRecognizer *)sender;
and
- (IBAction)SwipeDetected:(UISwipeGestureRecognizer *)sender
{
  //Code for when a user swipes
}
```

This works the same way for each of the gestures in the **Object Library**.

Using touch start, move, and end

Let's continue with the gameplay area. For the actual game, we are going to need a few things, such as a background with the play area, a score or stroke counter, a ball, and a cup for the ball to fall into. We want the player to be able to touch and drag in the direction they choose for the ball, which will also determine the speed based on the distance the player drags their finger. We will need to have an area in which the ball will be contained and whenever the ball hits a wall, we want it to bounce back.

We are going to have to add four methods to manage our touches. Let's take a quick look at these before we get into the game code. When a user first places their finger down, `touchesBegan` sends a message with data for a set of touches. We can then pull the location on the screen for the touch and store it in a variable as the start dragging position. When the user drags their finger around, it is tracked in the `touchesMoved` method. I am going to use the moved method to track the player's end touch position. When the user lifts their finger off the screen, we will use the`touchesEnded` method to calculate the ball direction along with the final distance between touch start and touch end for our ball's velocity. If the touch gets interrupted or canceled, the `touchesCancelled` method will get called. We will use this to release the touch information. The aforementioned methods are shown as follows:

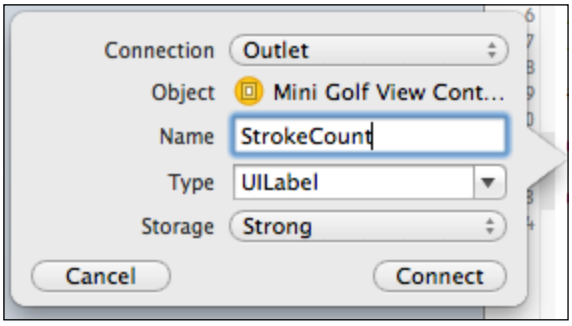
```
- (void)touchesBegan:(NSSet*)touches withEvent:(UIEvent*) event
{
}
- (void)touchesMoved:(NSSet*)touches withEvent:(UIEvent*) event
{
}
```


www.it-ebooks.info

```
- (void)touchesEnded:(NSSet*)touches withEvent:(UIEvent*) event
{
}
- (void)touchesCancelled:(NSSet*)touches withEvent:(UIEvent*)event
{
}
```

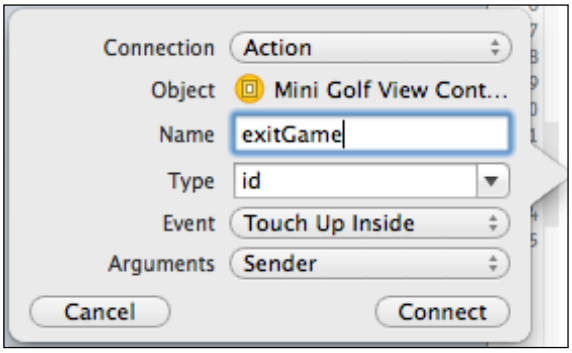
Lets get started:

1. As we already put our **Exit** button and **Stroke Counter** label in the game, let's start by linking them to the code.
2. Press *control* and drag from the **Label** field to our `MiniGolfViewController.h` header file and create an outlet, which we will call `StrokeCount`:



Now, it's time to create a link with the **Exit** button, so we can clean up our code when we leave the game and go back to the menu:

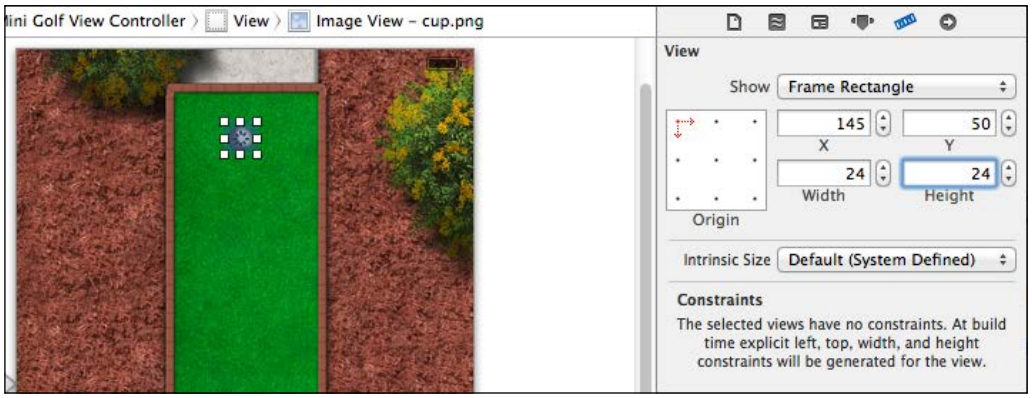
1. Press *control* and click-and-drag from the **Exit Game** button to the `MiniGolfViewController.h` header file.
2. Set the connection type to **Action** and name it **exitGame**, and set **Type** to **id**.



www.it-ebooks.info

We need to add our cup image to the game area as well:

1. Drag out an **Image** object and set the image to the `cup.png` file from your images group.
2. Set the image frame as follows in the size inspector: **X** to `145`, **Y** to `50`, **Width** to `24`, and **Height** to `24`. This is shown in the following screenshot:



Initializing our code

Now, we can start to add the code. Let's start by adding in a few more properties to our header file for our game state, player's ball, and the final cup's position. The following is how our `MiniGolfViewController.h` file should end up like:

```
#import <UIKit/UIKit.h>

@interface MiniGolfViewController : UIViewController
@property NSString *gameState;
@property UIImage *playerBall;
@property UIImageView *playerBallView;
@property CGRect ballRect;
@property CGRect cupRect;
@property (strong) NSTimer *updateTimer;
@property (strong, nonatomic) IBOutlet UILabel *StrokeCount;
- (IBAction)exitGame:(id)sender;
@end
```

These are all the property types that we have worked with previously. We will discuss these more as we work our way through the rest of the implementation code.

www.it-ebooks.info

We start our implementation code by adding some variables. `courseBounds` is a rectangular area where the ball is constrained to the hole you are playing on (not to be confused with the cup that the ball will fall into). `strokeCounter` is your score for this hole, and `ballVelocity` is, well… your ball's velocity or speed. `touchStartPos` and `touchEndPos` are the starting and ending positions for your player's touches, respectively. The aforementioned variables are shown in the following code:

```
#import "TouchEventsViewController.h"

@implementation TouchEventsViewController
CGRect courseBounds;
CGPoint ballDirection;
float ballVelocity;
int strokeCounter;
CGPoint touchStartPos;
CGPoint touchEndPos;


- (id)initWithNibName:(NSString *)nibNameOrNil bundle:(NSBundle *)
nibBundleOrNil
{
    self = [super initWithNibName:nibNameOrNil bundle:nibBundleOrNil];
    if (self)
    {
        // Custom initialization
    }
    return self;
}
```

In our `viewDidLoad` method, we have to set our initial `gameState` variable. As you will see throughout the code, we use a game state mostly to keep our users from hitting the ball while it is moving, as shown in the following code:

```
- (void)viewDidLoad
{
    [super viewDidLoad];
  self.gameState = @"Loading";
```

We then add our ball's image view, image, and position. Set the ball's position based on the bottom of the screen, just in case we are using an iPhone with a smaller screen, and wrap it up by adding the ball view to the main view:

```
  //Assign a ball image to a variable
  self.playerBall = [UIImage imageNamed:@"ball.png"];
```


www.it-ebooks.info

```
  //Create a ball to move around
  self.playerBallView =[[UIImageView alloc]
                  initWithImage: self.playerBall];
  CGFloat screenheight = [UIScreen mainScreen].bounds.size.height;
  self.ballRect = CGRectMake(160,screenheight - 100, 24,24);
  self.playerBallView.frame = self.ballRect;
  [self.view addSubview: self.playerBallView];
```

This is where we initialize a few variables needed throughout the code. The `cupRect` variable is the center point of the cup. This is when the ball intersects the point in which we drop the ball into the cup and end the round. The `CourseBounds` variable is a rectangle that defines the area that the ball is allowed to roll. If the ball intersects with the sides of this rectangle, the ball will bounce back into play. The last three variables are initializing variables used for gameplay. `ballDirection` is stationary to start so the *x* and *y* directions are set to `0`. The `ballVelocity` variable is set to `0`, as the ball is not moving and therefore has no speed. `strokeCounter` is the number of times you have hit the ball and is set to `0` on initialization. We throw that into our `StrokeCount` label on the screen, and then we start the timer that updates every thirtieth of a second. We wrap up the method by setting `gameState` to `Waiting`, as we are waiting for the player to touch the screen. These actions can be performed using the following code snippet:

```
  self.cupRect = CGRectMake(152, 57, 10,10);
  courseBounds = CGRectMake(110,30, 100,470);
  ballDirection = CGPointMake(0, 0);
  ballVelocity = 0;
  strokeCounter = 0;

  self.StrokeCount.text = [NSString stringWithFormat:@"Stroke: %d",
strokeCounter];
  self.updateTimer = [NSTimer scheduledTimerWithTimeInterval:.03
                        target:self
                        selector:@selector(update)
                        userInfo:nil
                        repeats:YES];
  self.gameState = @"Waiting";
}

- (void)didReceiveMemoryWarning
{
  [super didReceiveMemoryWarning];
    // Dispose of any resources that can be recreated.
}
```


www.it-ebooks.info

Touch events

The following is the description of our touch code. We start when the user first touches their device and the `touchesBegan` method gets called. We check `gameState` and then we pick up our touches data and place it into our touch variable. Once we have the `UITouch` object, we can get the position that the user first pressed. We store that in our `touchStartPos` variable. We use `locationInView` to get the touch location in relation to `touch.view`. This will be the start position of an imaginary line that we will use to calculate the direction and distance multipliers for our ball. The code is as follows:

```
- (void) touchesBegan:(NSSet*)touches withEvent:(UIEvent*) event
{
  if([self.gameState isEqual: @"Playing"])
    return;
  // do something with the touches that just landed on the view
  UITouch *touch = [touches anyObject];
  touchStartPos = [touch locationInView:touch.view];
}
```

While the player is dragging their finger around on the iPhone's glass, we will track the finger's position. The `touchesMoved` method is sent every time the player's finger moves its position on the glass. We pick up that new position the same way we pick up the player's touch start position. The only difference is that `touchEndPos` is being updated multiple times, as shown in the following code snippet:

```
- (void) touchesMoved:(NSSet*)touches withEvent:(UIEvent*) event
{
  // do something with the touches that moved on the view
  UITouch *touch = [touches anyObject];
  touchEndPos = [touch locationInView:touch.view];
}
```

Finally, when the user lifts their finger off the glass, the `touchesEnded` method gets invoked. As we already have the start and end positions for the touch events, we don't need to pick it up again. We just process the information we have. We get the distance based on the positions using the Pythagorean Theorem. The distance is the square root of your x positions squared plus your y positions squared. Ta-da! We have now proven everyone wrong who has ever said you would never use geometry in the real world. The `pow` and `sqrt` functions are automatically included in Objective-C in the `math.h` library.

If the player didn't just tap the screen, we change the game state, increment the number of strokes, and set the ball's velocity and direction. This data is all going to be picked up in our update method, which we will discuss next. The `touchesEnded` method is shown as follows:

www.it-ebooks.info

```
- (void) touchesEnded:(NSSet*)touches withEvent:(UIEvent*) event
{
  if([self.gameState isEqual: @"Playing"])
    return;
  // do something with the touches that ended
  CGFloat distance = sqrt(pow((touchEndPos.x - touchStartPos.x), 2.0)
+ pow((touchEndPos.y - touchStartPos.y), 2.0));
  if(distance > 0)
  {
     self.gameState = @"Playing";
    strokeCounter++;
    self.StrokeCount.text = [NSString stringWithFormat:@"Stroke: %d",
strokeCounter];
    ballVelocity = distance * 0.01;
    ballDirection = CGPointMake((touchEndPos.x - touchStartPos.x) *
0.01, (touchEndPos.y - touchStartPos.y) * 0.01);
  }
}
```

Oh, but first, in the event our touch data gets canceled, we are going to clear the data so that the ball doesn't move and the player doesn't lose a stroke just because they received a call or were inadvertently interrupted:

```
- (void)touchesCancelled:(NSSet*)touches withEvent:(UIEvent*)event {
  // do something with the touches that were cancelled
  touchStartPos = CGPointMake(0,0);
  touchEndPos = CGPointMake(0,0);
  ballVelocity = 0;
  self.gameState = @"Waiting";
}
```

Updating the screen

This is the `update` method where our ball is moved around the screen, and we check to see if our ball has hit the side of the `courseBounds` area. We start by making sure `gameState` is set to `Playing`, so we are not wasting frame updates. We pick up `ballDirection` and check it against the sides of the bounding box. If the ball hits on the left- or right-hand side, we adjust the ball's *x* direction to be inverted; and if the ball hits the top or bottom, we invert the *y* direction. We do this by getting the absolute value of the ball's *x* and *y* directions as a float (`fabsf`) and invert the number. The updating of the screen is shown in the following code snippet:

```
//Call this method to move a ball
-(void)update
{
```


www.it-ebooks.info

```
    if(![self.gameState isEqual: @"Playing"])
      return;
    if (self.ballRect.origin.x <= courseBounds.origin.x)
    {
      //hit left
      ballDirection.x = fabsf(ballDirection.x);
    }
    else if ((self.ballRect.origin.x + self.ballRect.size.width) >=
         (courseBounds.origin.x+courseBounds.size.width))
    {
      //hit right
      ballDirection.x = -fabsf(ballDirection.x);
    }
    if (self.ballRect.origin.y <= courseBounds.origin.y)
    {
      //hit top
      ballDirection.y = fabsf(ballDirection.y);
    }
    else if (self.ballRect.origin.y + self.ballRect.size.height >=
  (courseBounds.size.height+courseBounds.origin.y))
    {
      //hit bottom
      ballDirection.y = -fabsf(ballDirection.y);
    }
```

Now that we know the ball's direction, we can adjust the ball's position based on the new direction, velocity, and position. As long as the ball has a velocity, we reduce the velocity to give the ball the effect of slowing down using the following code snippet:

```
    //Update the balls position
    self.ballRect = CGRectOffset(self.ballRect,
          ballDirection.x * ballVelocity,
          ballDirection.y * ballVelocity);
    self.ballRect.origin.x += ballDirection.x * ballVelocity;
    self.ballRect.origin.y += ballDirection.y * ballVelocity;
    //if the ball is moving slow it down over time
    if(ballVelocity > 0)
    {
      ballVelocity -= .05;
    }
```


www.it-ebooks.info

```
    //if the ball stops, change the game state to waiting
    if(ballVelocity <= 0)
    {
      self.gameState = @"Waiting";
      ballVelocity = 0;
    }
    //if the ball intersects the cup, set the state to done.
    if(CGRectIntersectsRect(self.ballRect, self.cupRect))
    {
      self.gameState = @"Done";
      self.ballRect = CGRectMake(150, 55, 12, 12);
      ballVelocity = 0;
    }
    self.playerBallView.frame = self.ballRect;
  }
```

When the user hits the **Exit** button, let's clear out all of our variables:

```
  - (IBAction)exitGame:(id)sender
  {
    [self.updateTimer invalidate];
    self.updateTimer = nil;
    self.ballRect = CGRectMake(0,0, 24,24);
    [self.playerBallView removeFromSuperview];
    self.playerBallView = nil;
    ballDirection = CGPointMake(0, 0);
    ballVelocity = 0;
  }
```

That's it for our touch methods and our Mini Golf game. It's a pretty simple scene to set up and get working once you know how to pick up the player's interaction.

Some things you can try on your own are adding multiple holes for the user to play through; usually eight or sixteen holes are what the player is looking for. Maybe you could add in a swipe to help guide the ball as it is moving. Add in a rotate or zoom for more user interactivity during the game. Ultimately, it's all up to you now; your imagination is your only limitation.

Using other mechanisms

iPhone can detect motion events whenever the user shakes, moves, or tilts their device. iPhone has built-in hardware called an **accelerometer** and a **gyroscope**, which detect these types of events and pass the information back to your code.

www.it-ebooks.info

Shake

Let's start with the shake event. When you want to detect a shake, you will need to designate your object as the first responder. To become the first responder, you will need to add the following code to your object:

```
-(BOOL)canBecomeFirstResponder
{
    return YES;
}

-(void) viewDidAppear:(BOOL)animated
{
    [self becomeFirstResponder];
}
```

Once you have your responder set up, you need to add a method to handle the event. There are three motion methods that you can use: `motionEnded:withEvent:`, `motionEnded:withEvent:`, and `motionEnded:withEvent:`. These are similar to the touch events we talked about earlier. To detect the shake, you will need to check `UIEventSubType` that is sent in with the method:

```
-(void)motionEnded:(UIEventSubtype)motion withEvent:(UIEvent *)event
{
  if (motion == UIEventSubtypeMotionShake)
  {
    // User was shaking the device.
    // Send a log message to tell the user to stop it.
    NSLog(@"STOP SHAKING ME");
  }
}
```

Orientation

For tilting, if you just want to detect whether the orientation on your phone has changed, you can use the methods from the `UIDevice` class to detect them. This is just the general orientation information and does not give you tilted vector information. You start by telling your object that you want to turn on notifications for the orientations. Then, you will need to tell your object where to send the orientation notifications. For our example, we can do this in the `viewDidLoad` method and send the notifications to the `orientationChanged` method. For this example, we send a message if the iOS device is tilted and in portrait mode. The orientation can be demonstrated using the following code snippet:

www.it-ebooks.info

```
- (void)viewDidLoad
{
   [super viewDidLoad];
  [[UIDevice currentDevice]
          beginGeneratingDeviceOrientationNotifications];
  [[NSNotificationCenter defaultCenter] addObserver:self
               selector:@selector(orientationChanged:)
               name:UIDeviceOrientationDidChangeNotification
               object:nil];
}

- (void)orientationChanged:(NSNotification *)notification
{
  // Respond to changes in device orientation
  UIDeviceOrientation orientation = [[UIDevice currentDevice]
orientation];
  if( orientation == UIDeviceOrientationPortrait)
    NSLog(@"TILTED: Portrait");
  else
    NSLog(@"TILTED: NOT portrait");

}
```

Other orientations are `UIDeviceOrientationUnknown`, `UIDeviceOrientationPortrait`, `UIDeviceOrientationPortraitUpsideDown`, `UIDeviceOrientationLandscapeLeft`, `UIDeviceOrientationLandscapeRight`, `UIDeviceOrientationFaceUp`, and `UIDeviceOrientationFaceDown`. Landscape left is when the phone is tilted on its side with the home button on the left. Landscape right is when the phone is tilted on its side with the home button on the right.

Another important point to note is when using anything that uses the accelerometer, you will need to turn it off when you are done because it tends to use a lot of battery life. The code for this is as follows:

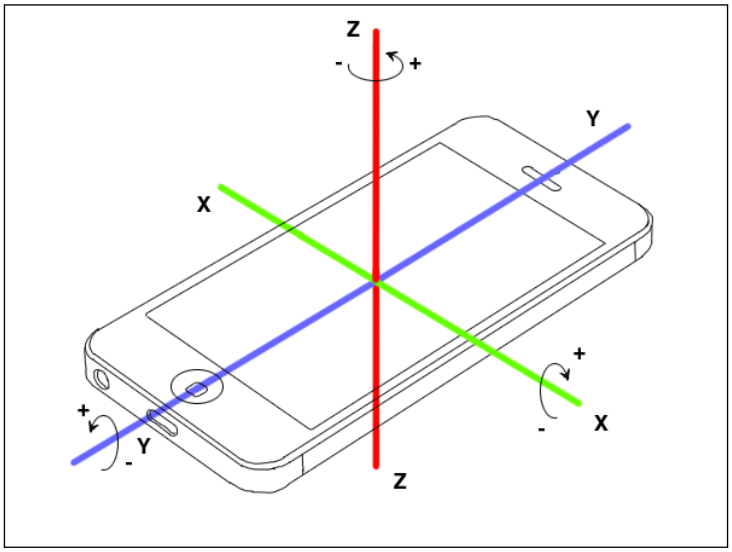
```
-(void) viewDidDisappear
{
  // Request to stop receiving accelerometer events and turn off
accelerometer
  [[NSNotificationCenter defaultCenter] removeObserver:self];
  [[UIDevice currentDevice]
            endGeneratingDeviceOrientationNotifications];
}
```


www.it-ebooks.info

Tilt

Now, we are going to dig into trickier code. One very important point to note is that you will not be able to detect any accelerometer or gyroscope data in the iOS simulator. For the following code, you will need to connect an actual iPhone and select it as your scheme. When you run your code, it will run off of your phone and pass `NSLog` data back to Xcode and display it in your debug area.

If we want to pick up data from a device that is being tilted, we are going to have to hook into a new framework called the Core Motion framework. The Core Motion framework lets you access raw accelerometer and gyroscope data. At this level, the accelerometer data captures the acceleration data along each axes and the Gyro Data captures the rate of rotation around each of the spatial axes. The following diagram shows the different axes in relation to your phone along with how the rotations work. Using this diagram, if you tilt your phone on its side, you would be rotating along its *y* axis.



Let's take a look at the code that is required to pick up the tilt information. Make sure you import the Core Motion framework at the top of your header file:

```
#import <CoreMotion/CoreMotion.h>
```

Inside your implementation file, we will need a variable to store the motion manager:

```
CMMotionManager *mManager;
```


www.it-ebooks.info

In our `viewDidLoad` method, let's initialize the motion manager and put it into our `mManager` variable. Then, we can make sure the device we are using has a gyroscope available to us. If so, the first thing we are going to want to do is set up the interval rate. I have mine set up for 1/30th of a second. Depending on the granularity you desire for your game, you can take this higher or lower. If you only want to check this once a second, you could just set the update interval to `1.0` if you wish. As game developers, we would not want our frame rates to drop below 30 frames per second if at all possible, and when you drop below 24, your eyes will start to see the frames in animation. We would prefer 60 frames per second; however, at that frame rate, it could drain your battery.

We are going to use the pushing approach for our data, which means each interval we will be running in the code will be provided in curly braces or a **queue**. In order to do that, we are going to use `startGyroUpdatesToQueue:withHandler:` for our queue. So, for our sample code, we are going to get the rotation rate and assign it to the `rot` variable and then send it to your debug area every thirtieth of a second:

```
- (void)viewDidLoad
{
    [super viewDidLoad];
  mManager = [[CMMotionManager alloc] init];

  if (mManager.gyroAvailable)
  {
    mManager.gyroUpdateInterval = .03;
    [mManager startGyroUpdatesToQueue:
            [NSOperationQueue currentQueue]
             withHandler: ^(CMGyroData *gyroData, NSError *error){
      CMRotationRate rot = gyroData.rotationRate;
      NSLog(@"rotation rate=[%f, %f, %f]", rot.x, rot.y, rot.z);
     }];
  }
  else
  {
    NSLog(@"No gyroscope on device.");
  }
}
```

Don't forget to stop your updates when you are done:

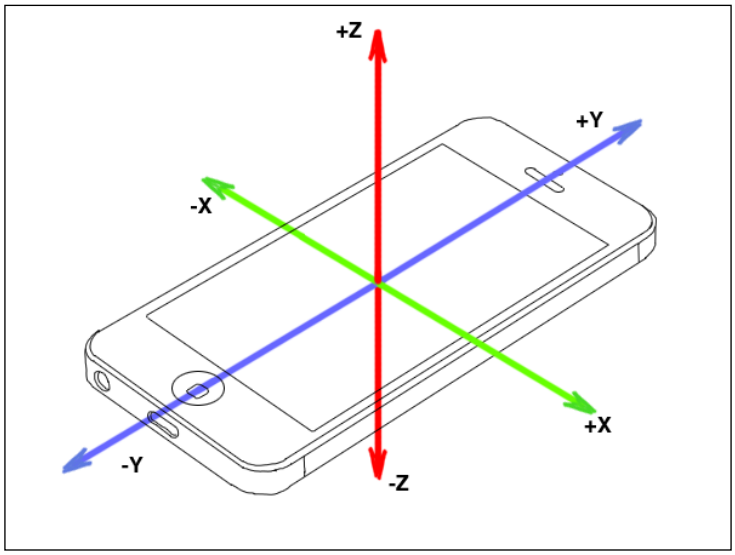
```
- (void)stopUpdates
{
```


www.it-ebooks.info

```
    if ([mManager isGyroActive])
    {
      [mManager stopGyroUpdates];
    }
  }
```

Motion

For our motion code, we are going to handle everything similar to the way we ran our gyroscope code, but for motion, we will be measuring motion in different directions as the following diagram shows:



Make sure you import the Core Motion framework at the top of your header file and connect your iPhone. Once again, this code will not work with the simulator.

Inside your implementation file, we will need a variable to store the motion manager:

```
CMMotionManager *mManager;
```

In our `viewDidLoad` method, let's initialize the motion manager and put it into our `mManager` variable. Then, we can make sure the device we are using has an accelerometer available. Once again, we are going to want to set up the interval rate.

We are going to use `startAccelerometerUpdatesToQueue:withHandler:` for our queue. So, for our sample code, we are going to get the acceleration data and assign it to the `accData` variable and then send it to your debug area every thirtieth of a second:

www.it-ebooks.info

```
- (void)viewDidLoad
{
    [super viewDidLoad];
  mManager = [[CMMotionManager alloc] init];

  if ([mManager isAccelerometerAvailable] == YES)
  {
    mManager.accelerometerUpdateInterval = .03;

    [mManager startAccelerometerUpdatesToQueue:[NSOperationQueue
mainQueue] withHandler:^(CMAccelerometerData *accelerometerData,
NSError *error) {
      CMAcceleration accData = accelerometerData.acceleration;
      NSLog(@"rotation rate = [%f, %f, %f]", accData.x, accData.y,
accData.z);
    }];
  }
  else
  {
    NSLog(@"No accelerometer available.");
  }
}
```

Don't forget to stop your accelerometer:

```
- (void)stopUpdates
{
  if ([mManager isAccelerometerActive])
  {
    [mManager stopAccelerometerUpdates];
  }
}
```

Summary

In this chapter, we learned how the user can interact with their iPhone and how to put that data to use in our games. We discussed different ways we can pick up touch data, motion data, tilting, and gestures. We took everything we learned about touches and gestures and applied them to a simple putt-putt golf game.

In *Chapter 8*, *The Galaxy Game*, we will use everything we have learned previously in order to create your final game similar to Galaga. You will also be introduced to planning, resource management, design and flow, creating game screens, and much much more.

www.it-ebooks.info

www.it-ebooks.info

8
The Galaxy Game

In this chapter, you will learn about resource management as well as some basics for planning what is being used, what is to be used, and how these resources are stored. This chapter will be the accumulation of what you have learned thus far in order to create your final game, similar to **Galaxy**, a single-shooter, spaceship game. The following is what we are going to cover in this chapter:

- Planning the game
- Considering design and flow
- Focusing on resource management
- Creating game screens
- Building the user interface
- Setting up the gameplay environment
- Creating and controlling the player
- Creating and controlling the enemies
- Collision detection
- Adding sounds

Planning the game

With any app you are creating, planning what you are going to create is key to the rest of your app. As you are planning, keep in mind that a lot of this will probably change before you even finish, but you should provide as much detail as possible when putting your docs together. We will take a look at the concept document, design document, and the technical document.

www.it-ebooks.info

Creating the game concept document

Sometimes called a **pitch doc**, **flyer**, or **one-sheet**, the **game concept document** is a document that gets the basic game idea across. It should only be one page and should include the name, a description of the game, the characters, and some basic concept art or pencil sketches. The game concept document is often used to pitch the idea of the game to others (usually publishers, the development team, or investors). Once you have the idea sold and are ready to get into the details, you need to put together the game design document and the technical design document.

Creating the game design document

A **game design document** is, quite simply, a document where you put all of your ideas for your app. This may include information about the characters in your game, level designs, colors used, flow charts, and anything else you feel will be needed in order to complete your game. It is best to think of this as a document you would hand someone else if they were to build your game. The following is what a basic outline for a game design doc might look like:

- Game story
- Gameplay:
    - Game mechanics
    - Core gameplay
    - Characters
    - Game elements
    - Game physics
    - Artificial intelligence
    - User interface
    - In-game UI
- Art and video:
    - Art goals
    - 2D art and animation
    - UI screens
    - Buttons
    - Gameplay elements
    - Special effects
    - Cinematics

www.it-ebooks.info

- Sound and music:
    - Overall goals
    - Sound effects
    - Music

As you can see, the game design doc is more about the way the game looks and plays rather than the actual technical information about how the game will work. The game design doc is more about organizing your thoughts and getting your ideas on paper than actually making it work. Once you have your ideas on paper, you can get to the technical design document, which helps you to figure out how your app will work.

Creating the technical design document

The **technical design document** is a basic layout of how you want your code to work. The technical document should explain what your game objects are and how they interact with both the player and the rest of the game code. Again, build this with the understanding that everything can, and most likely will, change by the end of your game development process. The following is an example of a basic outline of how your technical design document might look:

- Game
    - Platforms
    - Objects
    - Variables
    - External libraries
    - External data and storage
    - Physics
    - AI
    - Multiuser
- User interface
    - UI
    - In-game UI
    - Game UI flow chart
- Art and video
    - Graphics specific code
    - Artist

www.it-ebooks.info

- Audio
    - Audio specific code
    - Audio instructions
- Code flow charts

I realize that this may seem like a lot of work, but in the end, these documents end up saving an enormous amount of time, especially when working with a team of people or any external developers.

Considering the design and flow

When designing your game, consider this important notion: *People hate to wait*. When someone wants to play a game or use an app, they do not want to click through 30 screens, watch 30 minutes of video, or read through pages of text just to play a game. A game should flow smoothly from launch to the end of the game. Most games have a pretty simple flow chart for the game screens. Consider the following:

- **Splash screens**: Publishers and third-party software companies usually require a splash screen. It is also a great space to let your users know that you actually made the game. Usually, each image or animation stays up for about three to five seconds, but the user should be able to instantly click past each screen.
- **Game splash video or screens**: Some games have an intro video or animation to introduce the game. Again, the player should be able to click past this. Either way, it shouldn't be more than 30 seconds for video and 10 seconds for an image.
- **Main menu**: At this point, the user has to make a decision on what they want to do. If you have anything in regard to your app that you need to let the player choose, this is where it should happen. You do not want to force the user to stop their game; therefore, this is the best place to answer any of their questions or any decision making that will affect their gaming experience. At the bare minimum, from this screen, the player should be able to click to:
    - Start the game / Restore a previous game
    - Change the game options
    - Get help
    - Get any stats or high scores

www.it-ebooks.info

- **Game Screen**: This is the area where any gameplay would exist. All of your game controls, pause buttons, in-game help, and exit buttons should be readily available to the player on this screen.
- **End game splash**: Some games have an extra splash screen when the user beats the game.

The way your game flows is different from the way the user interface flows. Once the user is in the game, you want to make sure they will have fun playing. It is best to try and make sure that any user can start the game and make it past the first level with a firm understanding of how to play the game at the end of that level. From there, make sure your game increases in difficulty slowly in order for users to get better over time. In your Simon Says game, this was an easy task, as the levels merely added one new item each time a level incremented.

Focusing on resource management

Managing all of the assets in your game can get daunting as your app size increases. Early on in your app development, take the time to set up a folder structure for your files. Make sure you mention your folder structure in your technical design document. This will help you and others keep track of what is available in different sections of your game. Also, name your files so that you can easily see what files belong in what areas of your game. The following is a sample file structure for your `Resources` directory of game assets—art:

- Intro
  - 00_SplashScreen01.png
  - 00_SplashScreen02.png
  - 00_SplashScreen03.png
  - 00_MainMenuGUI.png
  - 00_ButtonArt
- Level1.atlas
  - 01_Player1.png
  - 01_Enemy01.png
- Level2.atlas
  - 02_Enemy01.png
  - 02_Enemy02.png

www.it-ebooks.info

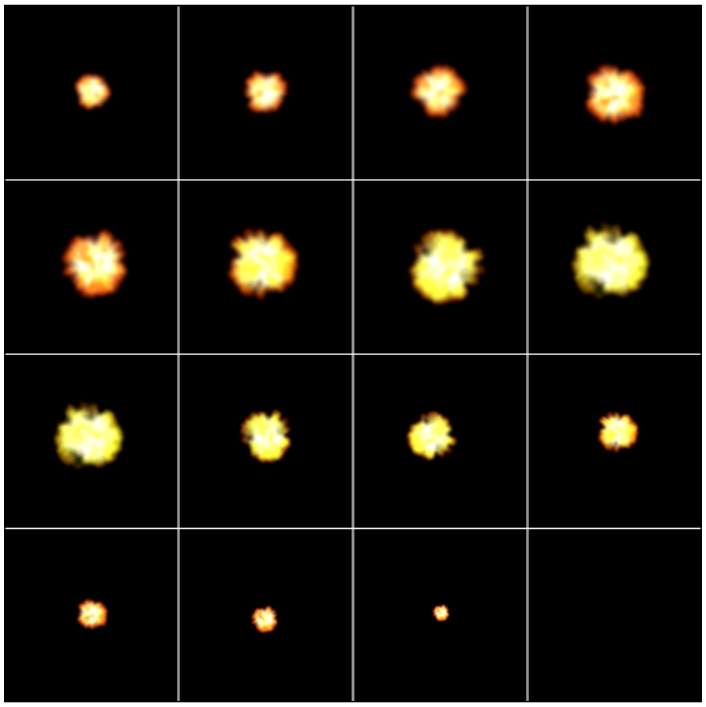
The sample file structure for the audio of the game is as follows:

- Intro
  - SplashSound01.mp3
  - 00_Publisher01.mp3
  - 00_GamerSound.mp3
- Level 1
  - 01_shipThrust.mp3

The sample file structure for the video of the game is as follows:

- Intro

We will be showing you an example of the **atlas directory** in the Galaxy game example in this chapter. Apple uses the directories with the `.atlas` extension to compile all of the images into a single image, similar to how sprite sheets look. The following screenshot is an example of a sprite sheet for an explosion in space. Each frame of the image would be loaded up and played back in succession to show the explosion:

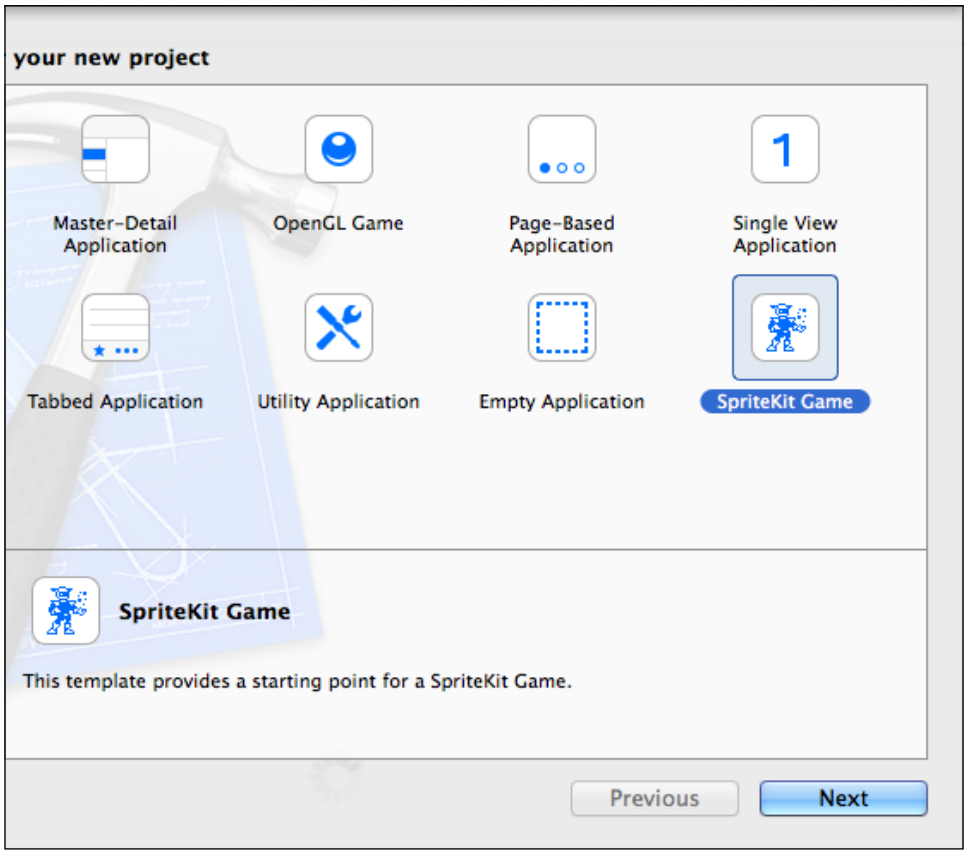


www.it-ebooks.info

Creating game screens

We are going to change things up a bit for this game. In the previous chapters, we built our projects using a standard single view application. Now that we have a good understanding of how to build a basic application, we are going to build a 2D game using the **SpriteKit Game** template for our project:

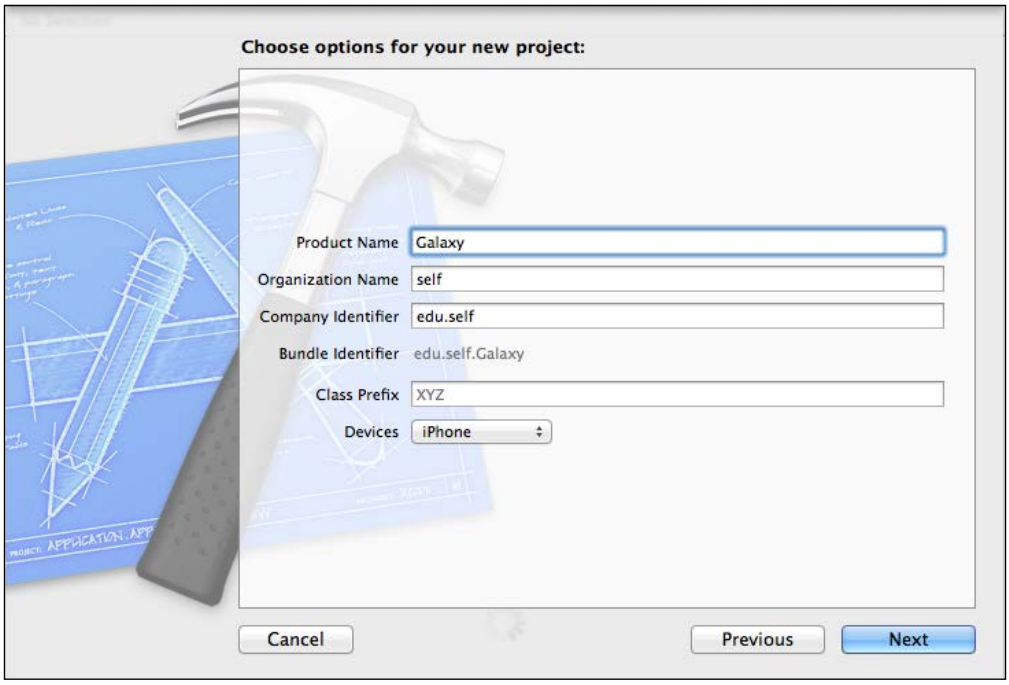
1. Open Xcode and navigate to **File** | **New** | **Project...**.
2. Instead of choosing **Single View Application**, choose **SpriteKit Game** from the iOS application templates:



3. Go ahead and name your project as you would normally.
4. Choose iPhone as your device.

www.it-ebooks.info

5. Set your **Product Name** to `Galaxy`, which is what we are calling our next game, as shown in the following screenshot:



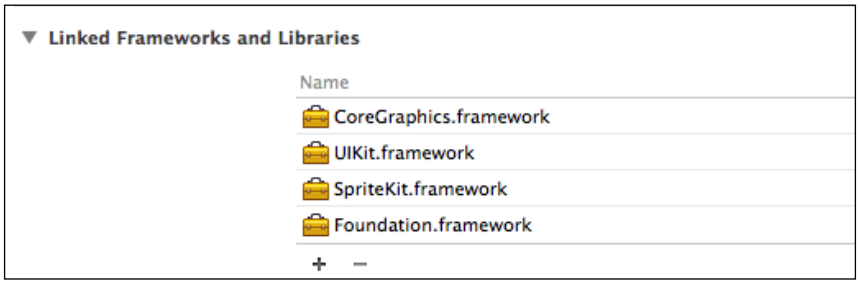
As you feel more and more comfortable with Xcode, feel free to name your projects as you like; however, make sure you understand that any name change will affect the names of the classes we are calling.

Now that the project is created, let's set up our device orientation:

1. Select **Galaxy** in the project navigator.
2. Check the box for **Portrait**.
3. Leave the rest of the orientations unchecked.
4. Set your version number to `1.0`.
5. Set your build number to `1.0`.

www.it-ebooks.info

Your build number is used when submitting your app to the App Store, so you want to keep this low when you first start your development. If you look at the **Linked Frameworks and Libraries** section of your project editor, you can see that a new framework has been added to the list. The **SpriteKit.framework** is the set of code that we are going to use for controlling our sprites in this chapter:



If you build and run your project at this point, you would see a blue screen with **Hello, World!** and the words **1 node 60.0 fps**. This shows that we have created one node object (the `Hello, World!` text label) and the environment is running at 60 frames per second. If you touch the screen, a second node with a spinning space ship will be created, as shown in the following screenshot:

www.it-ebooks.info

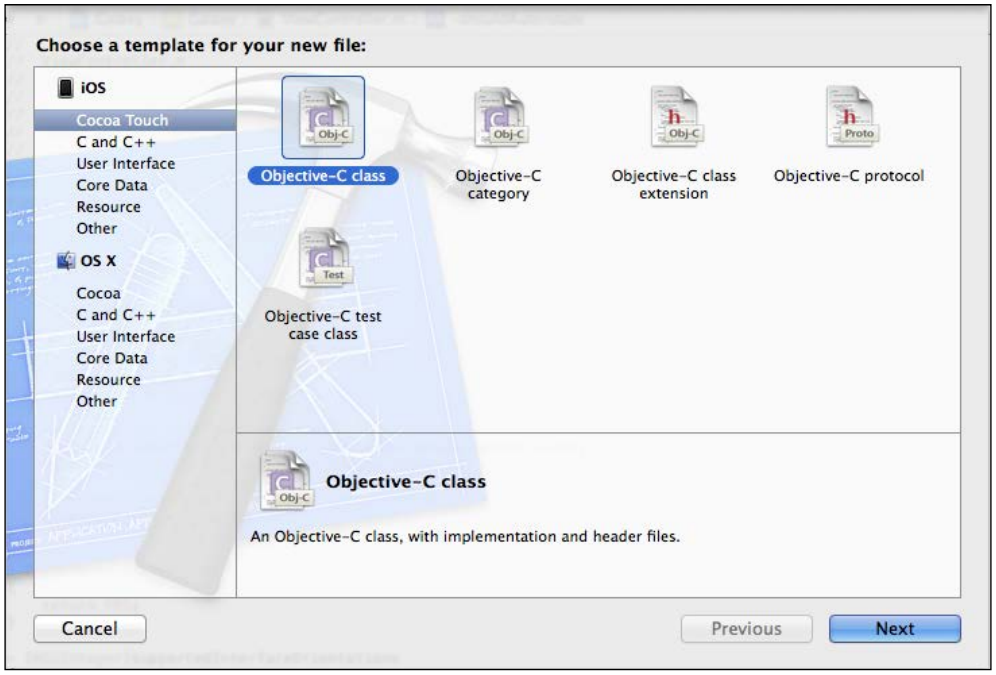
Each time you touch the screen, another spinning spaceship will be added to your screen and the node count will increment.

Building the user interface

Let's create view controllers for our main UI as we have done in previous chapters using our storyboards and a **View Controller** class.

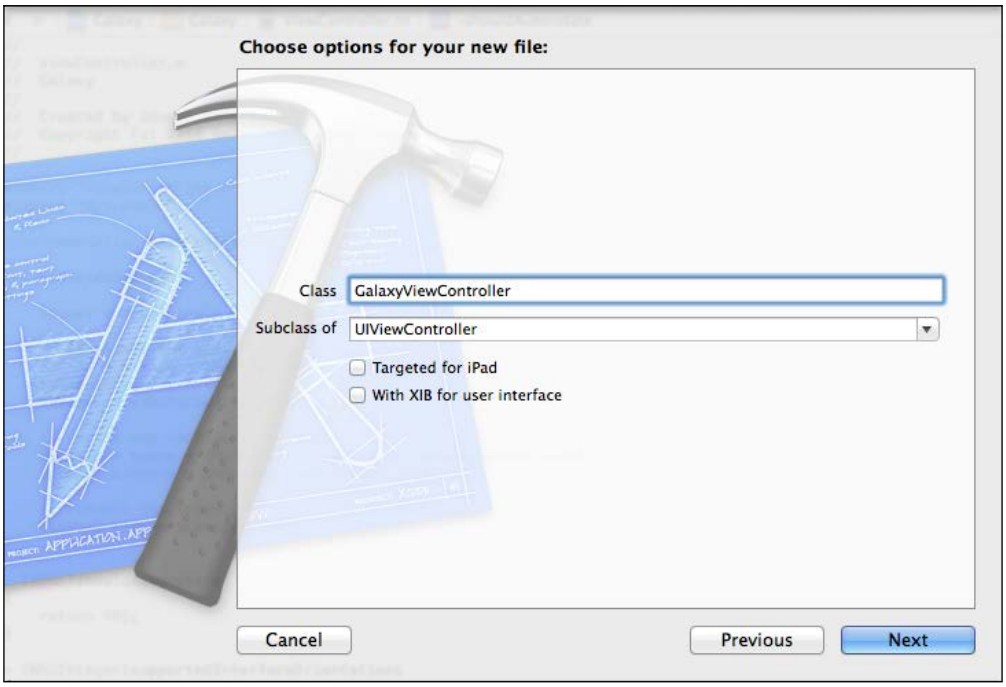
The following are the steps to create view controllers using a **View Controller** class:

1. Start by creating a new **View Controller** class.
2. Go to **File | New | File…** and create a new **Objective-C class**:



www.it-ebooks.info

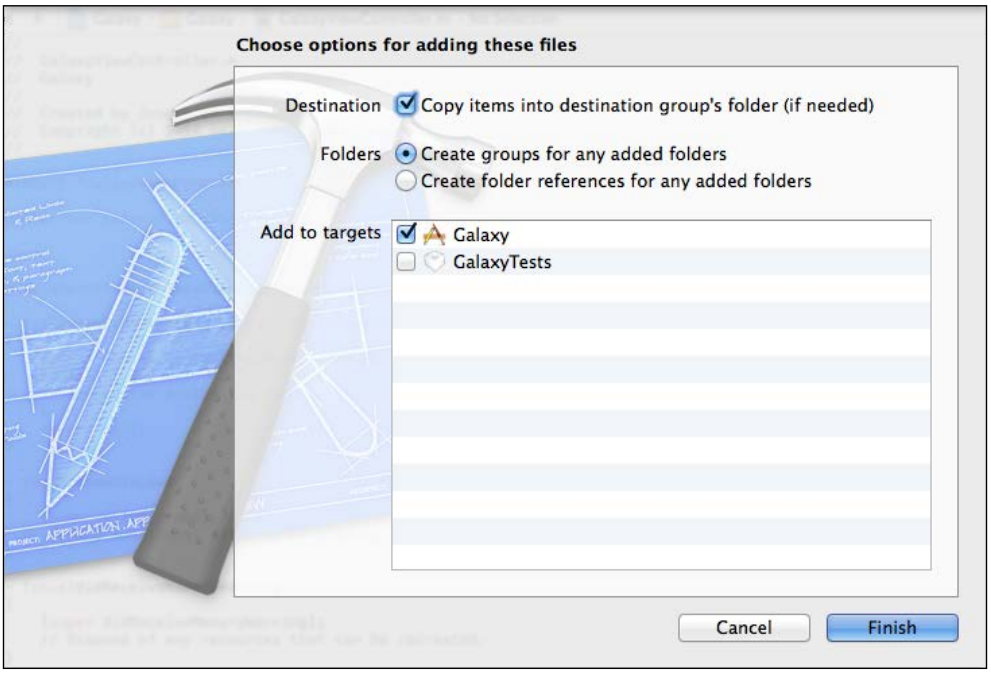
3. Call it `GalaxyViewController` with **Subclass** as `UIViewController`:



4. Next, we will drag it into the **Resources** folder for the Galaxy game. This will include your art and some audio files for the game.
5. Make sure the **Copy items into destination group's folder (if needed) checkbox** is checked.
6. Select **Create groups for any added folders**.

www.it-ebooks.info

7. Select **Finish**.



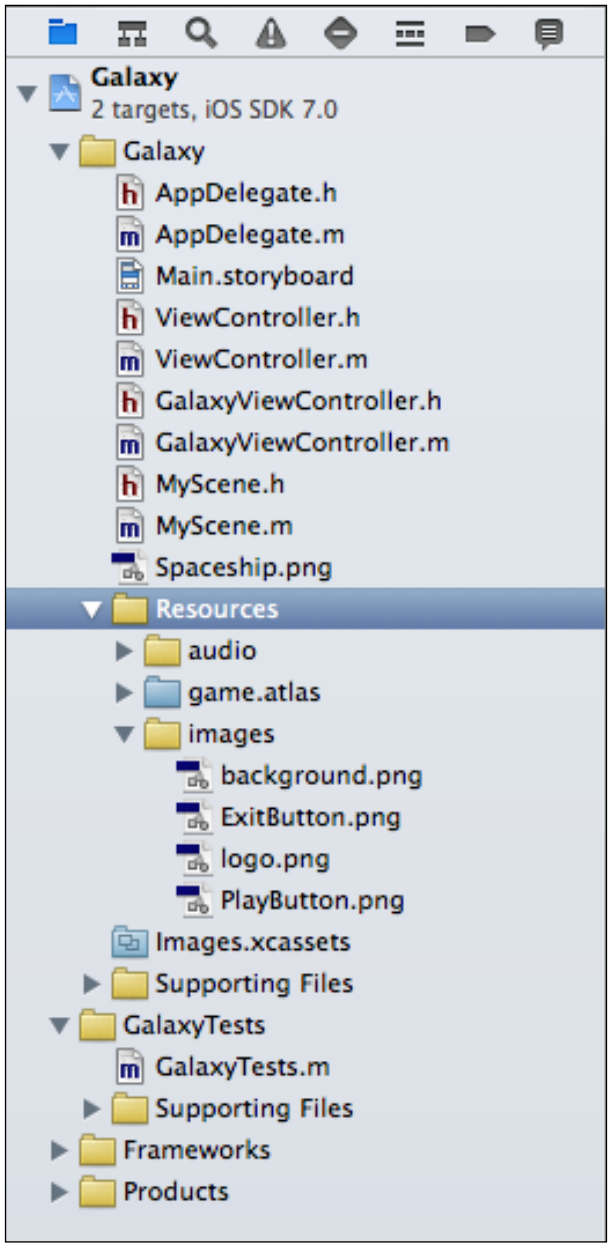
If you open the **Resources** directory, you will see a new directory type in your **Resources** folder with an `.atlas` extension, which is a new way of dynamically creating an **Atlas Texture**. If you drop images in here, when you compile Xcode, they will automatically create grouped images from all of the files similar to a sprite sheet. This is a fantastic way of increasing performance for your game by drawing multiple images with a single draw call.

> **Draw calls**
>
> The number of draw calls is a count of how many materials are being drawn on the screen at a time. Every draw call you have is an item for your device to draw and calculate. The more draw calls you add, the more it has to calculate, and in turn takes longer to accomplish each task.

It's important to note that every little thing you attempt to do to avoid your game from slowing down is important. Things such as atlas texture and keeping track of nodes may seem like little pieces, but if you keep an eye on things like these while you are building your game, it will keep you from having to go back and optimize your code later.

www.it-ebooks.info

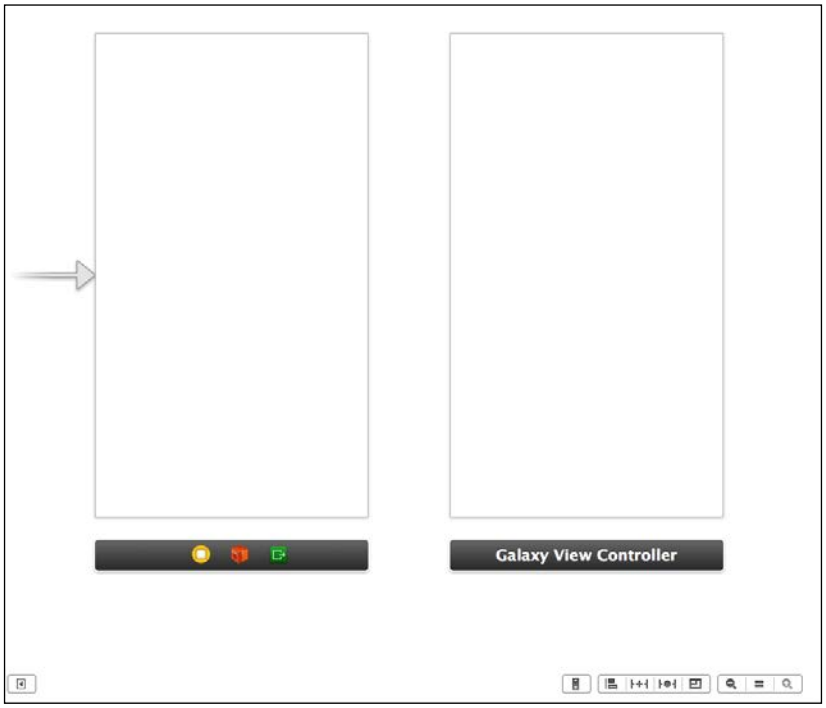
The following screenshot displays the **Resources** directory with a folder with the `.atlas` extension:



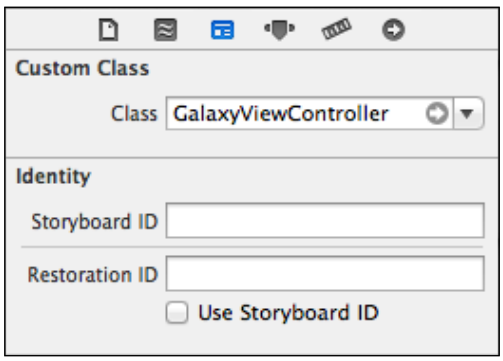
www.it-ebooks.info

The following are the steps to create view controllers using storyboards:

1. In **Main.storyboard**, create a new view controller by dragging one out from your **Object Library**:

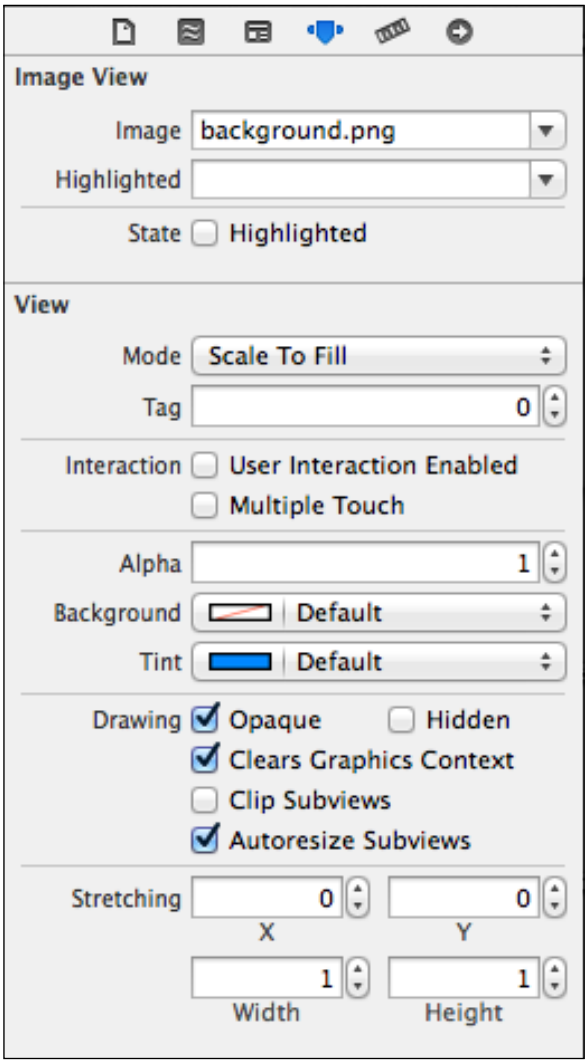


2. Select your new view controller.
3. Set the class for your controller to **GalaxyViewController** in the **Custom Class** section of your identity inspector:



www.it-ebooks.info

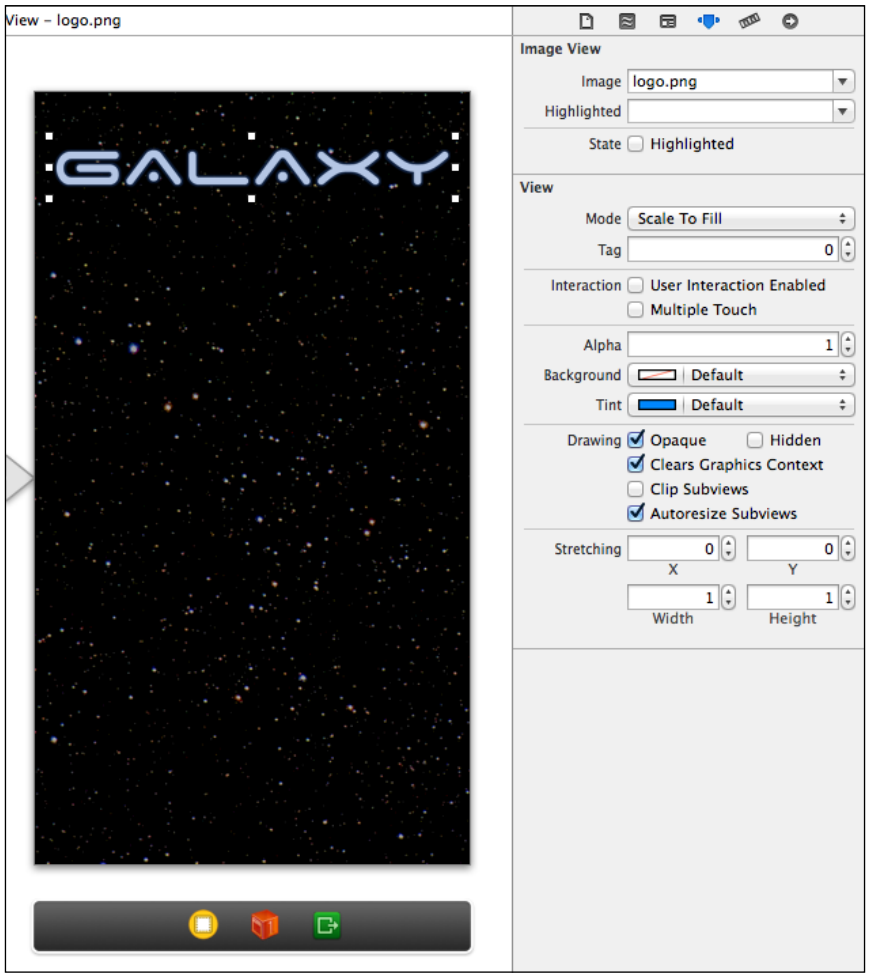
4. In your initial view controller, create a new **Image View** and set the image to the **background.png** image in the attributes inspector:



5. Now, do the same for the game's **logo.png** graphic.

www.it-ebooks.info

6. Scale both of these images to fit the screen properly:



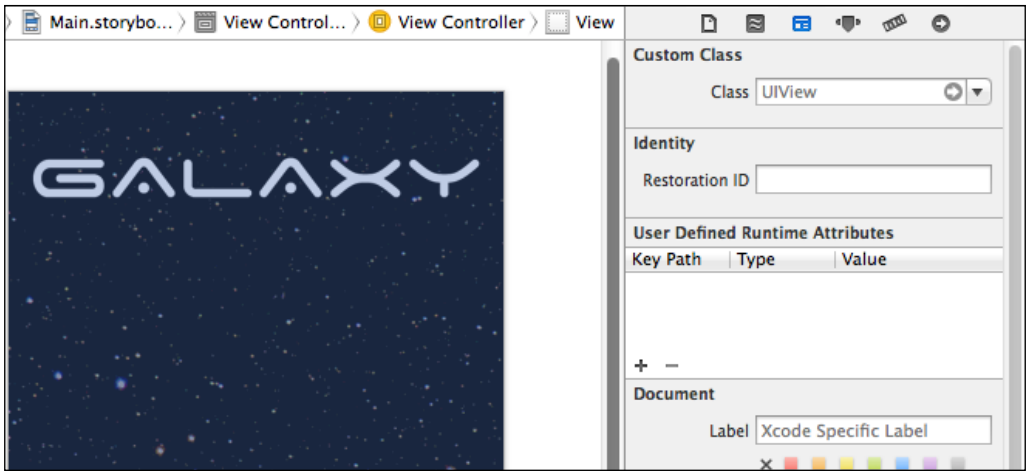
7. Now, we are going to create a new button by dragging out a **Button** object from our library.
8. This time we are going to set the image property to **PlayButton.png** so that we have a custom graphic for our button rather than just a standard text button. This is an easy and useful way of creating custom GUI elements for games. Xcode handles all of the button code and saves you a lot of time.

www.it-ebooks.info

9. Press *control*, click-and-drag the button, and link it to your **Galaxy View Controller** so that it automatically opens your new view controller when the user clicks on it:

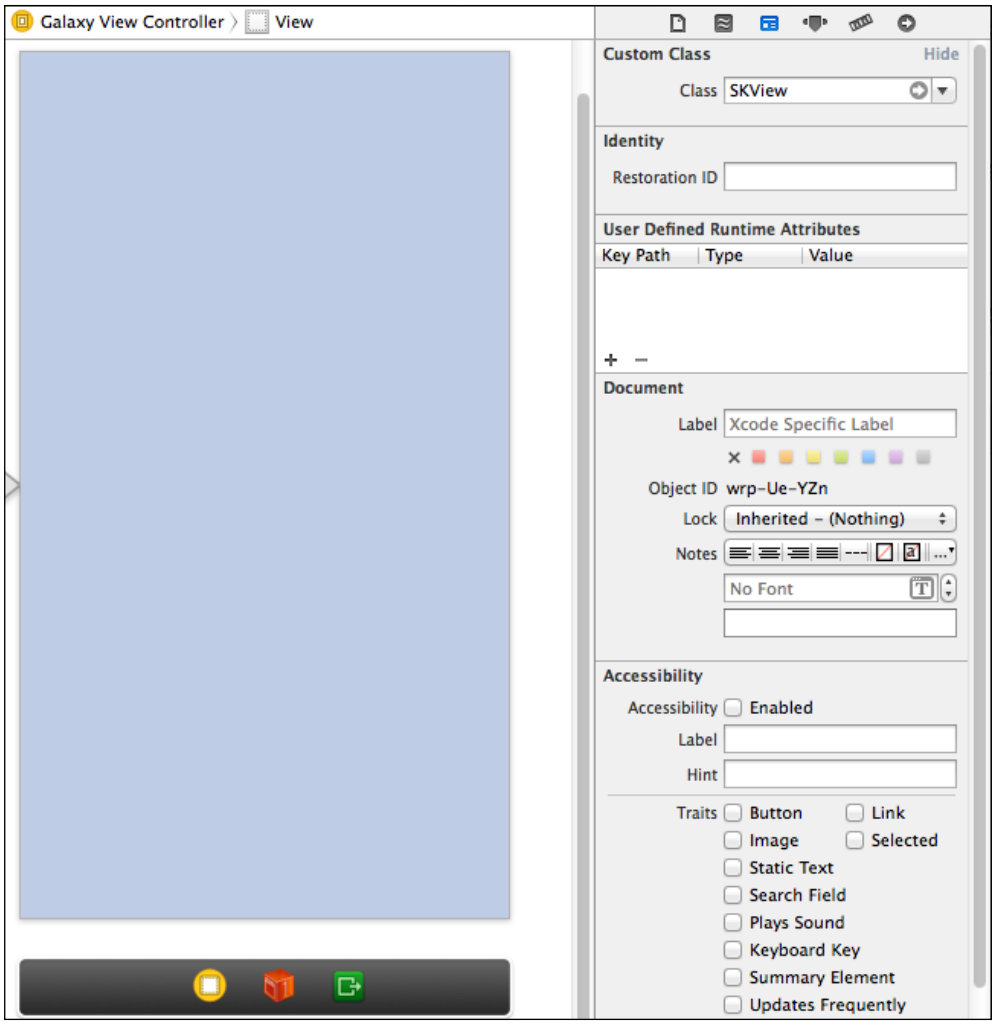
www.it-ebooks.info

10. Since we created our project using Xcode's SpriteKit Game template, our first view is linked with the sprite kit's view or SKView. We don't want the SKView to load up just yet; therefore, we need to change the class of the view controller's **View** to **UIView** in the identity inspector's **Custom Class**:



11. We do want the **SKView** to load up once we are on our view of **GameViewController** though. With the view of **Galaxy View Controller** selected, change the custom class in the identity inspector to **SKView**:

www.it-ebooks.info



12. In your `ViewController.h` file, we are going to add a new unwind segue to the code and remove `#import <SpriteKit/SpriteKit.h>` from the top of the code. We are still going to need this in our code later but not until `gameViewController` is loaded. The resulting `ViewController.h` file is as follows:

```
@interface ViewController : UIViewController
- (IBAction)unwindToThisView:(UIStoryboardSegue *)unwindSegue;
@end
```


www.it-ebooks.info

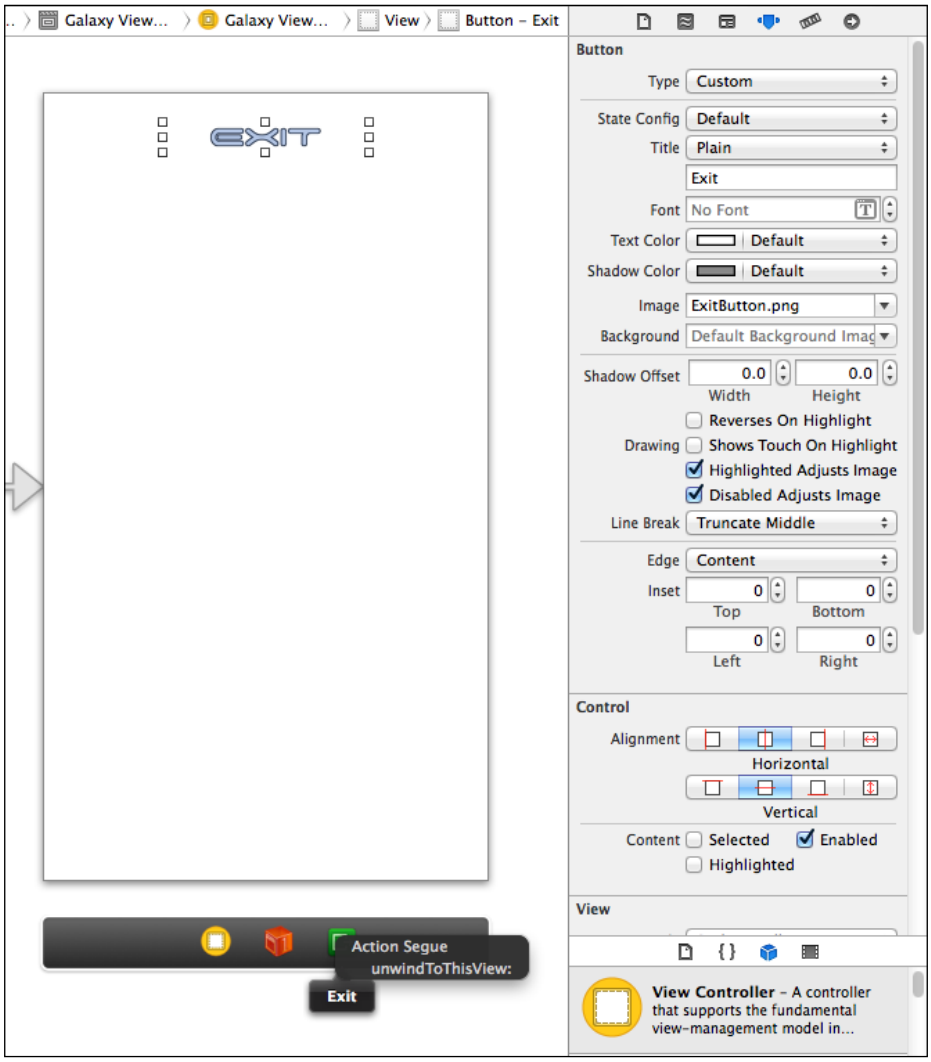
13. Let's clean out our `viewController.m` file so that it doesn't have all of the SKView code, and add our `unwindToThisView` method as well:

```
#import "ViewController.h"

@implementation ViewController

- (void)viewDidLoad
{
    [super viewDidLoad];
}

- (BOOL)shouldAutorotate
{
    return YES;
}

- (void)didReceiveMemoryWarning
{
    [super didReceiveMemoryWarning];
    // Release any cached data, images, etc that aren't in use.
}

- (IBAction)unwindToThisView:(UIStoryboardSegue *)unwindSegue
{
}

@end
```

14. Let's add another button to the game screen with graphics for the button area to exit the game.
15. Drag out a **Button** object from your library and set its image to **ExitButton.png**.
16. Press *control* and click-and-drag down to the exit icon.

www.it-ebooks.info

17. Set the action segue to **unwindToThisView** so that our exit button unwinds our game back to the main menu:



www.it-ebooks.info

As we removed all of our code for the `SKView` class from the **View Controller**, we are going to have to set up our **Galaxy View Controller** to take over. We will have to start by linking the sprite kit framework into our `GalaxyViewVontroller.h` file. Just add `#import <SpriteKit/SpriteKit.h>` directly below `#import <UIKit/UIKit.h>` at the beginning of the header file. That's all we will need to do to the header. We are going to add a few of the lines from the `ViewController` implementation file into the `GalaxyViewController.m` file as well. We will need to link to the `MyScene.h` file for our sprite kit, and we will need to load the scene in the `viewDidLoad` method. We are creating a sprite kit scene using the `self.view` screen size and using the sprite kit's view to display the scene. The majority of our game code for this game is going to be located within the `MyScene` file. Once we create the new scene, the `-(id)initWithSize:(CGSize)size` code in `MyScene` is called. The code for this is as follows:

```
#import "GalaxyViewController.h"
#import "MyScene.h"

@implementation GalaxyViewController

- (void)viewDidLoad
{
  [super viewDidLoad];

  // Configure the view.
  SKView * skView = (SKView *)self.view;

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

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

- (BOOL)shouldAutorotate
{
  return YES;
}

- (void)didReceiveMemoryWarning
{
  [super didReceiveMemoryWarning];
  // Release any cached data, images, etc that aren't in use.
}
@end
```


www.it-ebooks.info

Setting up the gameplay environment

As we mentioned earlier in the chapter, the sprite kit is focused around creating sprite nodes. For each item in a sprite kit game, a node will be created for the type. Let's start with our background. In order to create a background image node, we will first need to create a `SKSpriteNode`, assign it to a variable, and tell it which image to use. A sprite node is considered to be a textured node but can be initialized with a texture, a named image as we are doing here, or just as a colored square. The code for sprite node creation is as follows:

```
SKSpriteNode *background = [SKSpriteNode
       spriteNodeWithImageNamed:@"background"];
```

There are currently seven different types of node classes associated with the `SKNode` class that each have their own set of properties, which are as follows:

- `SKSpriteNode`: This class is for textured sprites for images
- `SKVideoNode`: This class is for playing digital video
- `SKLabelNode`: This class is a text node
- `SKShapeNode`: This class is a shape-based node that uses Core Graphics
- `SKEmitterNode`: This class is a particle emitter node
- `SKCropNode`: This class is a masking node
- `SKEffectNode`: This class is a Core Image filter node

Once our sprite node is created, we can adjust the properties on it. For our background, we will only need to give it a new position. To position the image from its center point to the center of the screen, we are using `CGRectGetMidX` and `Y` in order to get the center of the scene's frame. The code for this is as follows:

```
background.position = (CGPoint) {
  CGRectGetMidX(self.frame),
  CGRectGetMidY(self.frame)
};
```

Once our sprite node is set up, we need to add it to the screen:

```
[self addChild: background];
```

Let's add something a little more complex. We are going to set up a player score area. We will start by adding a class variable for the player's score:

```
int playerScore = 0;
```


www.it-ebooks.info

We are going to add a `SKLabelNode` property to our header file, as we will need to update our score on the screen later in the code:

```
@property SKLabelNode *scoreLabel;
```

Next, we assign a `SKLabelNode` property with a predefined font. For our game, I used the **System Font** but most fonts will work. Following along in our code, we set the font size, alignment, and then the position of the label. The last property we set is the text that will show up in the field. You will see this same code called later when we destroy an enemy and increment our score. Once we have the label set up the way we want it to appear, we add it to the scene like we did with the background image. The `SKLabelNode` property is as follows:

```
self.scoreLabel = [SKLabelNode labelNodeWithFontNamed:@"System"];
self.scoreLabel.fontSize = 15;
self.scoreLabel.horizontalAlignmentMode =
SKLabelHorizontalAlignmentModeLeft;
self.scoreLabel.position=CGPointMake(10,CGRectGetMaxY(self.frame)-50);
self.scoreLabel.text = [NSString stringWithFormat: @"%d",playerScore];
 [self addChild: self.scoreLabel];
```

Creating and controlling the player

For a game like this, I thought it would be more fun to set it up with the accelerometer for the controller and have a touch to fire. As we want to use the accelerometer, we will need to add the CoreMotion to our header file and the motion manager property.

```
#import <AVFoundation/AVFoundation.h>
@property CMMotionManager *mManager;
```

We will add our motion manager to our file similar to the way we talked about it in the previous chapter. For now, we just want to initialize it, start the updates, and make sure it is available:

```
//add an accelerometer for moving the player
self.mManager = [[CMMotionManager alloc] init];
if (self.mManager.accelerometerAvailable) {
 [self.mManager startAccelerometerUpdates];
} else {
NSLog(@"No Accelerometer");
}
```


www.it-ebooks.info

With the sprite kit, we have access to a 2D physics engine. This is really handy for our collisions and bounding boxes. Let's take a look at the player creation code. We start the player's code off by creating a bounding physics box around the scene's frame in order to keep our player within the screen's bounding rectangle:

```
self.physicsBody = [SKPhysicsBody
   bodyWithEdgeLoopFromRect: self.frame];
```

We are going to want to reference our player and their bullets later in the code, so let's start by creating a couple of properties for them. The player is going to be a sprite node, and we are going to store our bullet objects in a mutable array, as shown:

```
@property SKSpriteNode *playerShip;
@property NSMutableArray *playerBullets;
```

For our player, we are going to create a standard sprite node using our ship's image file and set its position. Once this is done, we are going to create a simple rectangle shape around the player's image for the physics body. The physics body is a node that stores the physics properties for the node in which it is attached. Since we defined a rectangle shape as the physics shape, our ship will be treated as though it is shaped like a box rather than the shape of the ship. I will often use rectangles as they are usually the easiest for me to visualize. We could have just as easily used a circle if we wanted to calculate it faster or a polygon if we wanted more accuracy. Once the physics body is attached, we can define how the physics will react to the rest of the environment. Dynamic defines that the object is moved by physics, `affectedBygravity` says that the object is subject to the environment's gravitational force, and lastly, we give our ship a mass so our ship moves a bit more realistically when we tilt the screen. The code to add the player's ship node is as follows:

```
// Add the players ship node
self.playerShip = [SKSpriteNode spriteNodeWithImageNamed:@"ship.png"];
self.playerShip.position = CGPointMake(CGRectGetMidX(self.frame),
CGRectGetMinY(self.frame) + self.playerShip.frame.size.height);
self.playerShip.physicsBody = [SKPhysicsBody
bodyWithRectangleOfSize: self.playerShip.frame.size];
self.playerShip.physicsBody.dynamic = TRUE;
self.playerShip.physicsBody.affectedByGravity = FALSE;
self.playerShip.physicsBody.mass = 0.02;
```


www.it-ebooks.info

The sprite kit has a special method called `Update` that it calls before each frame. This is where you would put any code that you want to be called for each frame. This is where we are calling the `shipUpdates` method, which checks the accelerometer data and whether the phone tilted on its *x* axis will move the player's ship:

```
-(void)shipUpdates{
  CMAccelerometerData* data = self.mManager.accelerometerData;
  if (fabs(data.acceleration.x) > 0.2) {
    [self.playerShip.physicsBody applyForce:
CGVectorMake(40.0 * data.acceleration.x, 0.0)];
  }
}
```

Now, let's create a whole bunch of bullets for the player to shoot. We will start by defining how many bullets we want to create, and because we are going to loop the bullets our player is using, we are going to define which bullet was the last one shot. We will do this with integer class variables:

```
int maxBullets = 5;
int currentBullet=0;
```

We will create our bullets as sprite nodes, position them, and then hide them until we need them; add them to the mutable array we created earlier and then add them to the scene:

```
//create your enemies and an array to hold our enemy objects
self.playerBullets = [[NSMutableArray alloc]
initWithCapacity:maxBullets];
for (int i = 0; i < enemyCount; ++i) {
SKSpriteNode *bullet = [SKSpriteNode
spriteNodeWithImageNamed: @"bullet02.png"];
bullet.position = self.playerShip.position;
bullet.hidden = TRUE;
[self.playerBullets addObject:bullet];
[self addChild:bullet];
}
```


www.it-ebooks.info

Later in the code, we are picking up whenever the player touches the screen, which is called `TouchesBegan`. Whenever they do, we fire a bullet for the player. As soon as that happens, we pull a bullet from our array, move it to the players position, unhide it, and then we start some sprite actions. This is the first time we have talked about actions, but basically, you can assign an action to a sprite and then trigger the action. For our bullet, we create an action that tells it to move from the player's ship position to the top of the screen within one second. We have created another action that runs a block of code when the previous action is finished. For the bullet, we clear out any actions that remain on the bullet and then hide it. We put those two actions in a sequence that we will run whenever the bullet is fired. Lastly, we run the action, or we tell it to execute it. The resultant code after including `TouchesBegan` is as follows:

```
-(void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event {
  /* Called when a touch begins */
  for (UITouch *touch in touches) {
// fire
    SKSpriteNode *playerBullet = [self.playerBullets
        objectAtIndex:currentBullet];
    currentBullet++;
    if(currentBullet >= self.playerBullets.count) currentBullet = 0;
    playerBullet.position = self.playerShip.position;
    playerBullet.hidden = FALSE;

    SKAction *firebulletAction = [SKAction moveTo:
        CGPointMake(self.playerShip.position.x,
self.frame.size.height) duration:1];
    SKAction *endBulletAction = [SKAction runBlock:^{
      [playerBullet removeAllActions];
      playerBullet.hidden = TRUE;
    }];

    SKAction *firebulletAndDestroy = [SKAction
        sequence:@[firebulletAction, endBulletAction]];
    [playerBullet runAction:firebulletAndDestroy withKey:@"Fire"];
  }
}
```


www.it-ebooks.info

Because we are using the tilt functionality, let's connect your iOS device to your computer and change your active scheme to your connected device:

Run your code on your device. Try tilting your device to see how your ship moves:

Creating and controlling the enemies

The last part of the initialization code sets up our enemies. Add in the enemy's property in our header file and then we can move forward:

```
@property NSMutableArray *enemies;
```


www.it-ebooks.info

We set up our enemies in a mutable array as a way to store them as objects. We are going to create them similar to the way that we created our bullets, but we are going to start them out in a position along the top of the screen, as they will animate down rather than up. For a little variety, I tend to create three different enemies and populate them randomly by name. Once the node is created, line the enemies up horizontally, set them to visible, add them to the mutable array, and then add them to the scene. They are controlled later in the code in the `enemyUpdates` method. The code for creating your enemies and to hold the objects is as follows:

```
//create your enemies and an array to hold our enemy objects
self.enemies = [[NSMutableArray alloc] initWithCapacity:enemyCount];
for (int i = 0; i < enemyCount; ++i) {
NSString *enemyName = [NSString stringWithFormat:
@"enemy0%ld.png",(random() % 3)+1];
SKSpriteNode *enemy = [SKSpriteNode
spriteNodeWithImageNamed: enemyName];
enemy.position = CGPointMake(
enemy.size.width + (i*enemy.size.width),
400);
enemy.hidden = FALSE;
[self.enemies addObject:enemy];
[self addChild:enemy];
}
```

Collision detection

Here is a question for you. What do we do when two of our objects collide? What we want to happen when a bullet hits a ship is we want the ship to be removed from the scene, increment our score, and play a sound effect. One piece of code that is being called in our update is our collision code. This code simply takes our list of enemies and checks to see if any of our bullets have hit them. We created a `for` loop that picks up each of our enemy objects in the enemy array and then runs through another for loop that gets each of our bullet objects in the bullet array. Then, we use `intersectsNode` to see if they are intersecting each other. If they are, we hide them and play an explosion sound. The `collisionDetection` method is as follows:

```
-(void)collisionDetection{
  //cycle through each of the enemeies then each of the bullets to see
if they intersected.
  for(SKSpriteNode *enemy in self.enemies){
    if(enemy.hidden == TRUE) continue;
    for(SKSpriteNode *bullet in self.playerBullets){
      if(bullet.hidden ==  TRUE) continue;
      if([bullet intersectsNode:enemy]){
```


www.it-ebooks.info

```
            bullet.hidden = TRUE;
            enemy.hidden = TRUE;
            playerScore++;
            self.scoreLabel.text = [NSString stringWithFormat:
@"%d", playerScore];
            NSString *soundName = @"impact01";
            [self playSoundEffect: soundName];
            continue;
          }
        }
      }
    }
```

Adding sounds

Last but not least, we want to add a little sound to our game. Our sound code should look familiar if you completed the Simon Says game. We simply used the same sound effect method to play our sounds. The code to add sounds is as follows:

```
-(void)playSoundEffect: (NSString *)effectName{
  NSString *path  = [[NSBundle mainBundle]
pathForResource: effectName ofType:@"wav"];
  NSURL *pathURL = [NSURL fileURLWithPath : path];
  audioEffect = [[AVAudioPlayer alloc]
    initWithContentsOfURL:pathURL error:nil];
  audioEffect.numberOfLoops = 0;
  [audioEffect prepareToPlay];
  [audioEffect play];
}
```

Whenever we need to play a sound, we just set up a string and call our method. Whenever we start a new game, the startup sound is called. Each time a player fires a weapon, it gets called, and if we hit an enemy, it gets called. The code for this is as follows:

```
NSString *soundName = @"impact01";
[self playSoundEffect: soundName];
```


www.it-ebooks.info

Let's run our code and see how it plays:

Summary

In this chapter, we started off by discussing how we plan out a game and some of the documentation that is required in order to start building a game. We dug into the sprite kit for creating sprites and objects for your game. We took a look at how physics work on sprites using the 2D physics body node and applied it to the player's ship using the accelerometer built into iPhone. We explored how actions are set up and run for our bullets and how collision detection works with our nodes.

In our next chapter, we are going to discuss how to prepare and release your game in Apple's App Store. We will also discuss some of the pitfalls to avoid in order to make sure you can get your app out in a timely fashion.

www.it-ebooks.info

www.it-ebooks.info

9
Releasing Your Game

Now that you have created your own game, what do you do with it? This chapter will give you everything you need in order to release your game in Apple's App Store. At this point in your development, if you want to release an app to the general public, you will need to upgrade your Apple account to the paid iOS Developers Program.

Here's what we will cover in this chapter in order to release your game:

- Preparing your app for distribution
- Selecting your identity
- Deployment information
- Selecting app icons and launch images
- Beta testing and ad hoc distribution
- Avoiding rejection from the App Store

Preparing your app for distribution

The first thing we are going to do is prepare your app for distribution. This is a relatively easy process; however, if any of it is skipped, then the app will get kicked back from Apple and you will need to fix it later. Perform the following steps to prepare your app for distribution:

1. In your project navigator, select your project.
2. In your target/project list, select your app. You should see the general app information similar to what is in the following image.
3. Make sure you test your app again whenever you change anything in your general settings.

www.it-ebooks.info

Selecting your identity

The identity section of your app is your company and game information area. This will be used in helping to identify your app in the Apple Store.

The bundle identifier

The **bundle identifier** will identify your app specifically in the Apple Store. Apple uses this ID to differentiate your apps from any of the extended services they offer, such as in-app purchases and the game center. Your bundle identifier should look like a DNS in reverse, similar to a website's URL. If we had a company named *myCompany*, our bundle identifier would look something similar to `com.mycompany.SimonSays`, and the app name would be automatically inserted.

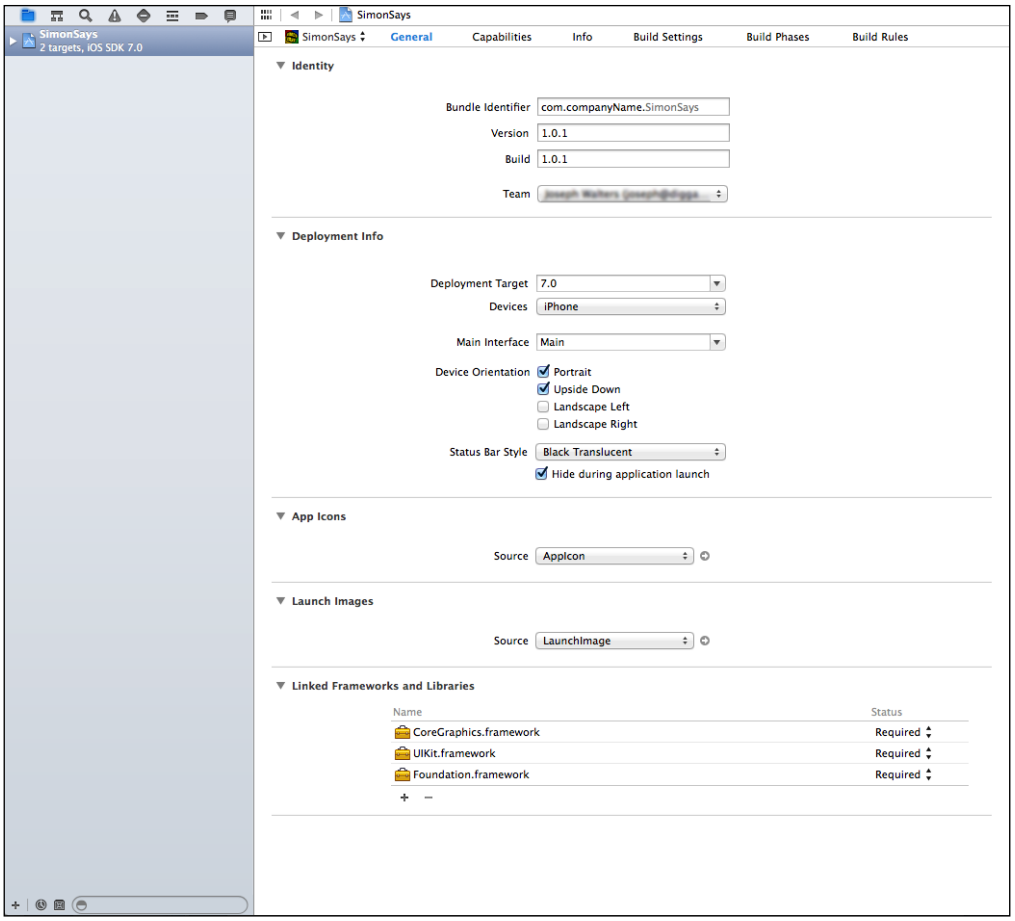
The version number and build string

Next, enter the version number of your app. This is a standard numbering scheme for software development. It consists of three numbers separated by two periods as follows:

- The first number is your major revision number.
- The second is the minor revision number.
- The third is for any maintenance updates that you release. This is the number that will show up with your app in the Apple Store.

The build string is for updating your app for distribution in the Apple Store. Whenever your build string changes, iTunes will recognize it as a change and will sync the app to the device. The build string can be a combination of any numbers or letters as long as it changes when you want your app to sync. I try and keep it similar to my version number so that I can recognize my changes with specific apps.

www.it-ebooks.info

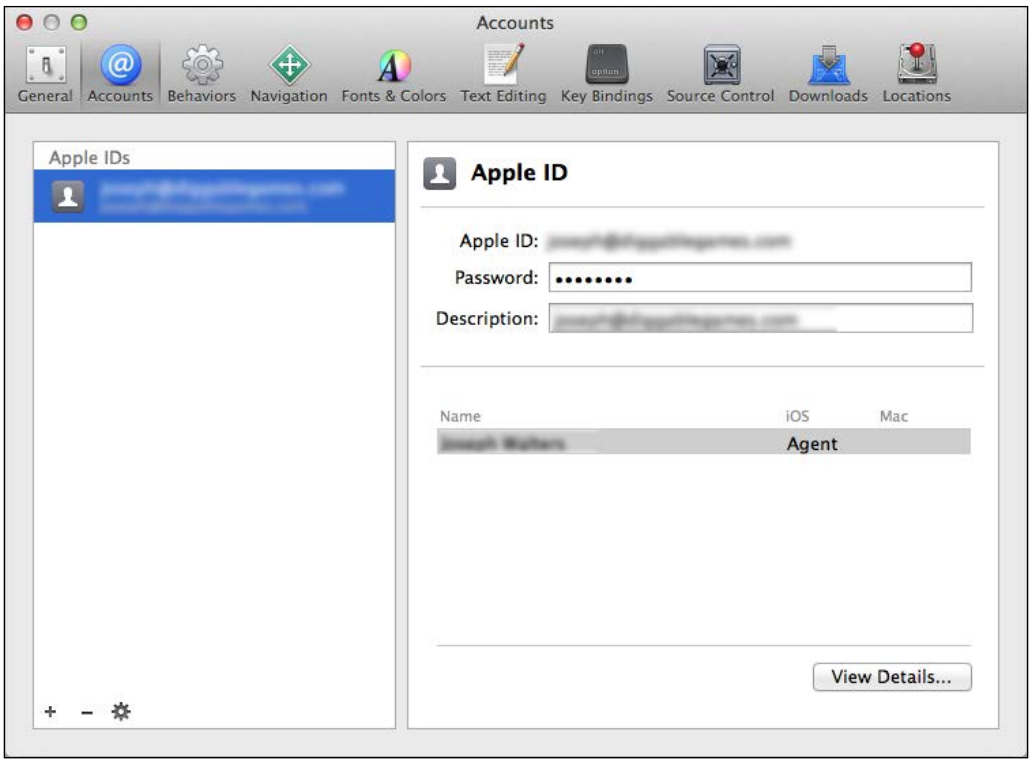


Team

You will need to assign a team provisioning profile to your app. You can do this from the **Team** pop-up menu.

www.it-ebooks.info

If a name or team does not appear in the pop-up list, you will need to add an account for your team. You can add an account by choosing **Preferences** from your Xcode menu, and then choose the **Accounts** tab. In the bottom-left corner, click on the **+** button, choose **Apple ID**, and enter the Apple ID that you used for your developer program. While you are here, click on the **View Details** button and then click on the **Refresh** button in the bottom-left corner to refresh your provisioning profiles. If it requests, you should allow Xcode to request a distribution certificate on your behalf. If you are not the account administrator, your certificate will need to be approved by the account administrator:



The deployment information

The deployment section is the information needed for your app to play properly on different devices.

www.it-ebooks.info

The deployment target

The deployment target defines the lowest iOS version number in which your app will work. The good news is that most iOS users are very good at keeping their devices up to date. Please note that by supporting older devices and operating systems, you increase the number of users who can use your app. Also, you should always test on as many operating systems as possible before releasing your app.

Devices

This is where you determine which devices you choose for your app to play. We have been developing exclusively for the iPhone, but if you so desire to release your app on the iPad as well, you may set that here. If you choose iPhone, your app will run on all iPhone and iTouch devices. If you choose iPad, then your app will be set to run on all iPad and iPad mini devices.

The device orientation

Device orientation determines the time your app will update its orientation if the user turns it. For the Simon Says game, we did not set up a landscape (horizontal) format for the app; so, we are just going to turn on the portrait and upside down orientation.

The status bar style

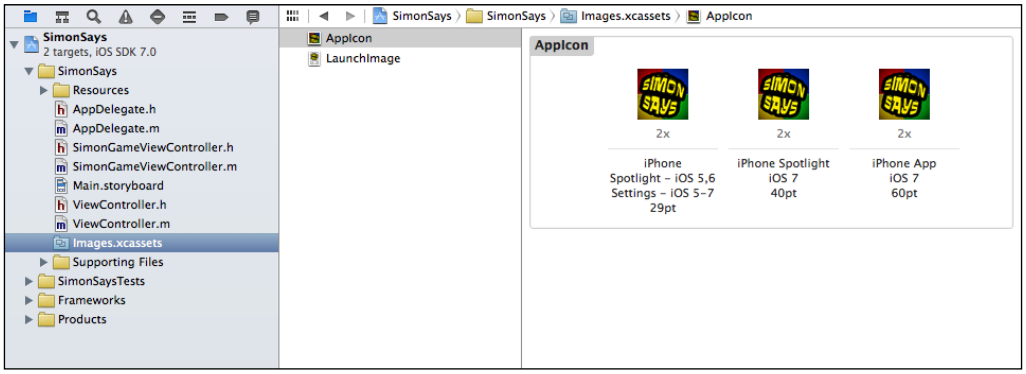
The status bar is the bar at the top of the iPhone screen that shows your carrier, connection, time, and battery info. This pop-up list allows you to choose the type of bar your app will show. The **Hide during application launch** checkbox determines if it will display while your app is starting up.

Selecting app icons and launch images

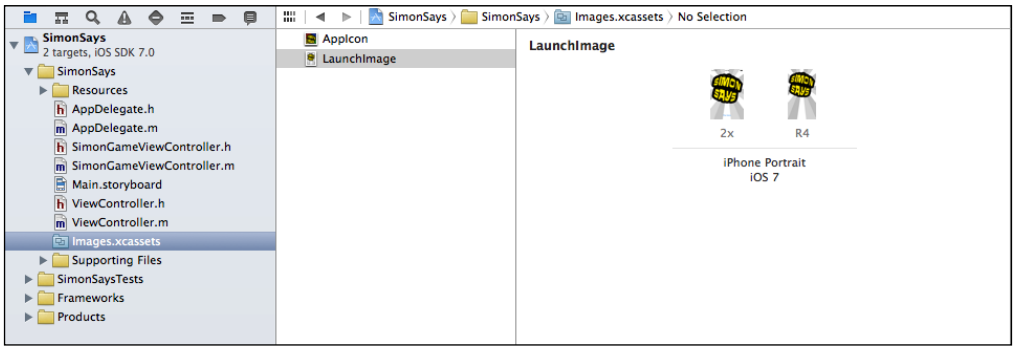
The app icon is the icon that shows up on the iPhone for your app. The launch image is an image that loads up while your app is starting up and stays up until your app is ready to be used. You can use asset catalogs to bundle these icons and images in your app and keep track of them or you can use individual image files. We have supplied images for the **SimonSays** app and will explain how to use the asset catalogs. All images for asset catalogs should be saved as `.png` files.

www.it-ebooks.info

When we first created our project, an asset catalog was created and added into our project named `Images.xcassets`. If you select that item from your project navigator, you will see items for **AppIcon** and **LaunchImage**; select **AppIcon** and drag the supplied icons into the three different slots. This will set the icons for each of the different operating systems' resolutions and spotlight searches as well. You do not need to round the corners for your icons; Apple takes care of that for you:



Next, we will do the same for the launch images. We have supplied screen captures for the two different screen resolutions for iOS 7:



If you decide that you do not want to use asset catalogs for your images, choose **Don't use Asset Catalogs** from the pop-up list and click on the folder icon to choose the images for each of the items listed:

www.it-ebooks.info

App Icons

Source: Use Asset Catalog

	Kind	Dimensions	Resource
App	iPhone Retina	120x120	No image with correct dimensions found
Spotlight	iPhone Retina	80x80	No image with correct dimensions found
Settings	iPhone Retina	58x58	No image with correct dimensions found

Linked frameworks and libraries

Any additional frameworks or external libraries needed for your app will need to be linked here. By default, Xcode links the libraries that we need automatically; however, if you are making references to any other external files, you will need to add them here.

Beta testing and ad hoc distribution

You have created your app and you have tested it on your devices, but now you need to test outside your devices. In order to do that, you will need to invite people to beta test your app. This will give you a better idea of how your app will work in the real world.

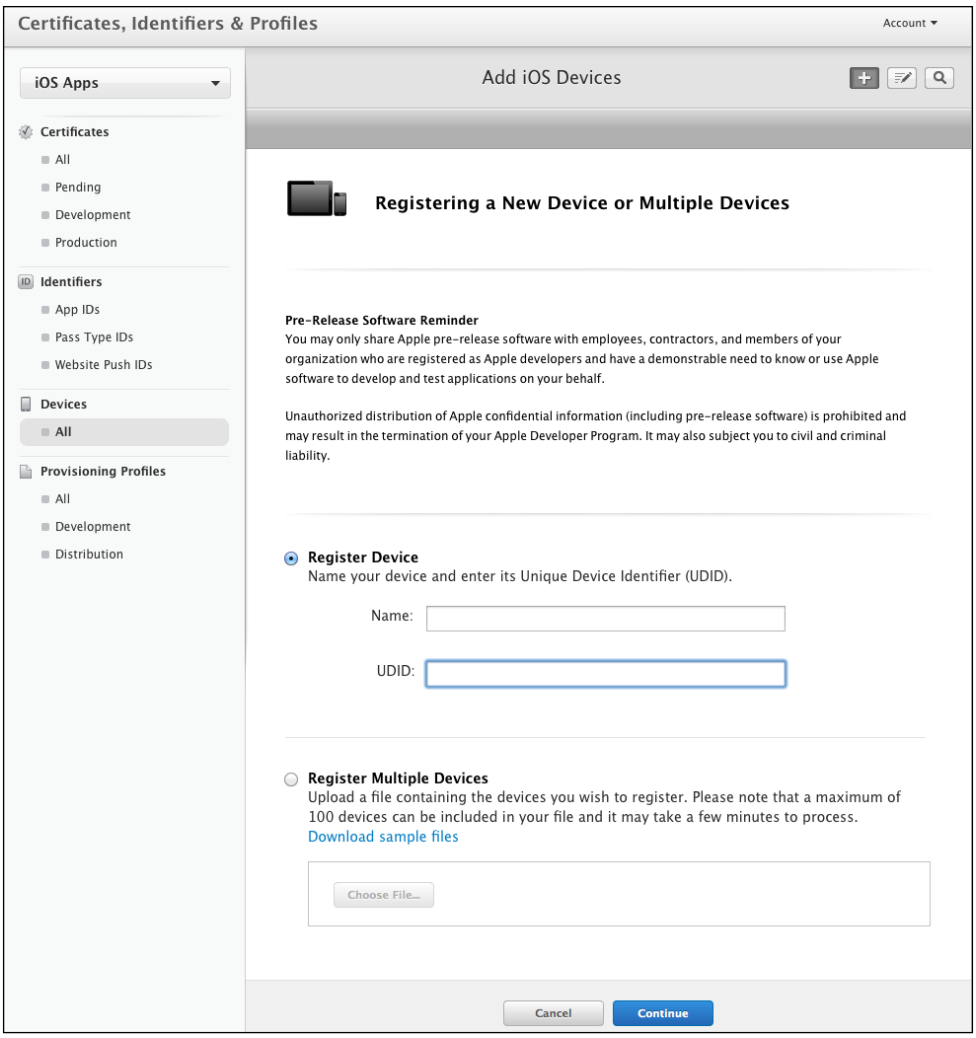
Register all test devices

For any user who would like to test your app, you will need to register their device on the Apple developer website in the **Certificates, Identifiers & Profiles** section under **Devices**. In order to do this, you will need your users to supply the UDID for their iOS device. You can get the UDID of a device from Xcode by navigating to **Organizer**, selecting the device under the **Devices** tab, and copying the identifier (it is the same thing as the UDID). In iTunes, if you click on **Serial Number** on the summary page for your device, the serial number will change to the UDID and you can copy it from there. Once you have the ID, paste it into the website along with a unique name for the device.

www.it-ebooks.info

Another option is to use a beta software distribution service such as TestFlight (`https://www.testflightapp.com`). TestFlight makes distributing your software to multiple beta testers a lot easier by helping you set up teams, users, and their devices. Since the users accept an invitation with their device information, you don't have to go through the trouble of keeping track of all of this yourself. Once the tester accepts, you can just copy and paste their UDID from TestFlight to the developer's site.

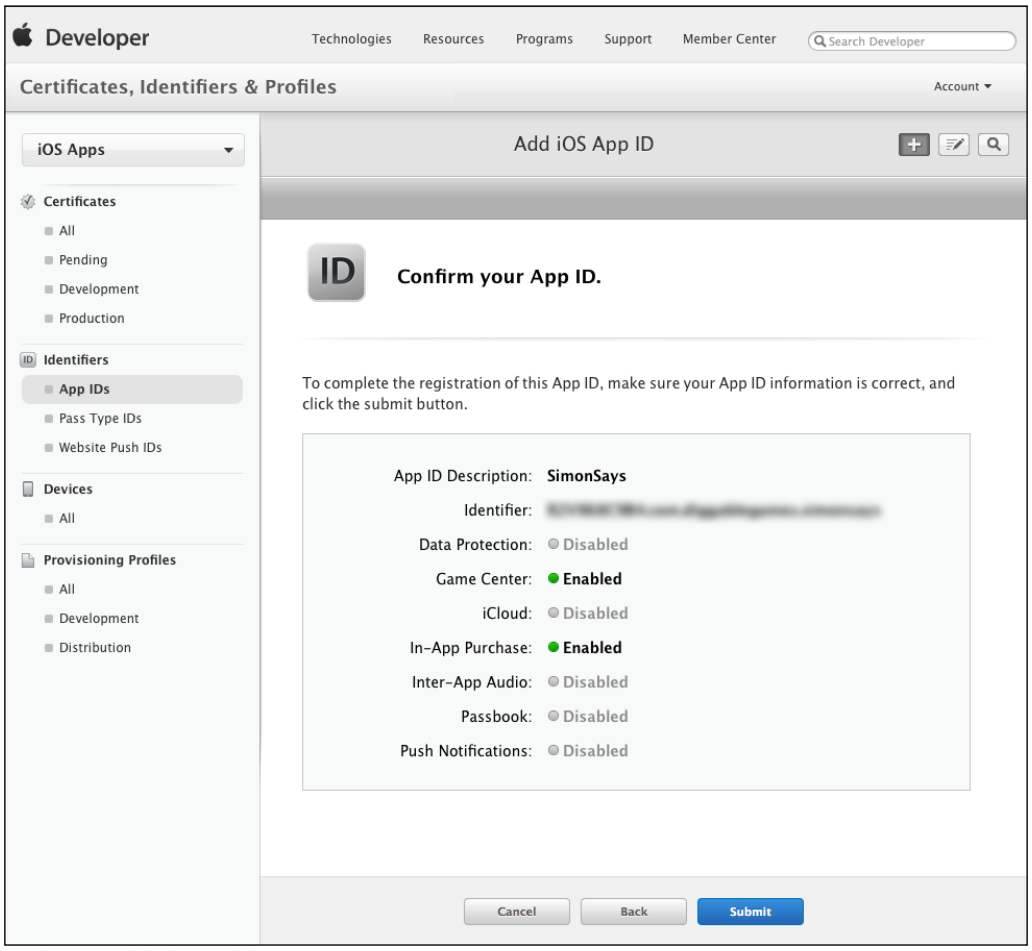
Once this is complete, you should refresh your provisioning profiles, as we mentioned earlier in the chapter. The following screenshot shows how to register a new device:



www.it-ebooks.info

App ID

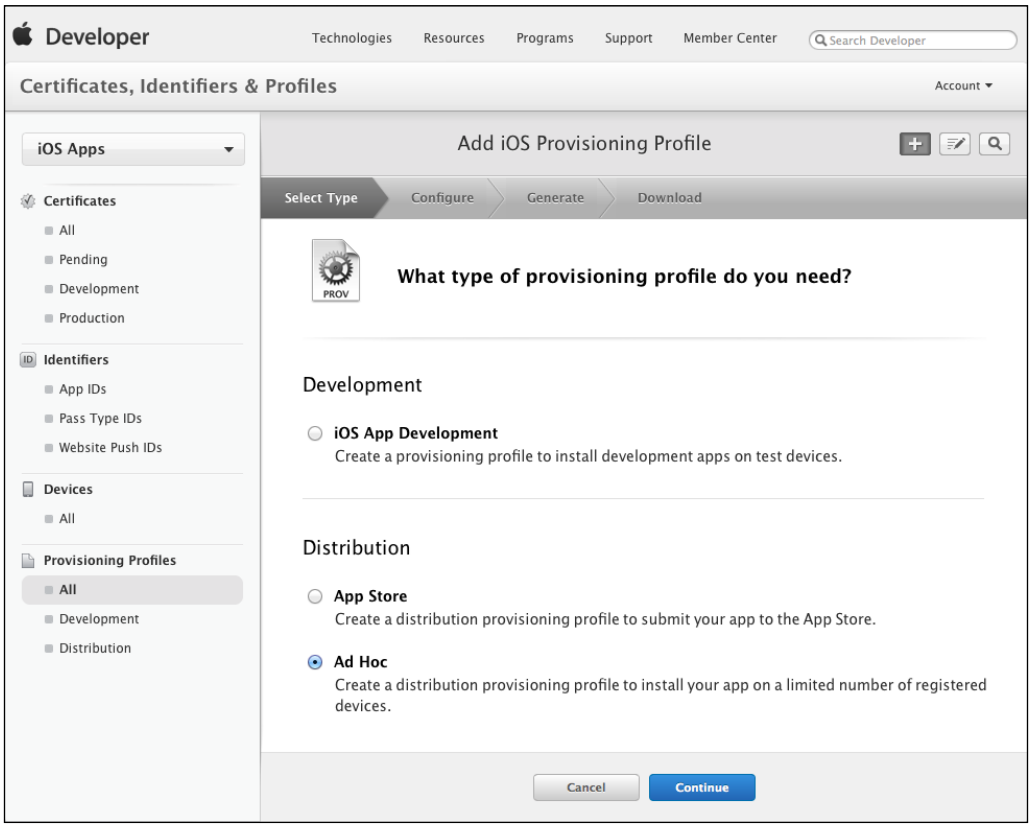
You will need to create an App ID for your new app. Under **Identifiers** in the **Certificates, Identifiers & Profiles** section of the developer site, select **App IDs**. Click on the **+** button to add **Register an App ID**. Give it a name such as `SimonSays`, select **App ID Prefix** from the pop-up list, put the bundle ID we created earlier into the field, then click on **Continue** and **Submit** to create your App ID:



www.it-ebooks.info

Ad hoc provisioning profile

Now, we will create the provisioning profile for distributing apps to your test users. Select **Provisioning Profiles** from the **Certificates, Identifiers & Profiles** section. Click on the **+** button to add an iOS provisioning profile and choose **Ad Hoc** as the distribution type and click on **Continue**. Select **SimonSays App ID** from the pop-up list and click on **Continue**. Select **iOS Distribution Certificate** and click on **Continue**. Select all of the devices in the list for which you would like to use your app and then click on **Continue**. Give it a profile name and click on **Generate** as shown in the following screenshot:



Whenever you add or change a profile, it is a good idea to update your provisioning profiles in Xcode. In Xcode, navigate to **Xcode | Preferences... | Accounts** and click on the **Details** button associated with your Apple ID. Click on the **Refresh** button in the bottom-left-hand corner, and then click on **Done**.

www.it-ebooks.info

Archive and create your app package

Choose **Archive** under the **Product** menu in Xcode. This will create an archive of your app and open the **Archive** section of your organizer window in Xcode.

Click on the **Distribute** button and choose the radio button for **Save for Enterprise or Ad Hoc Deployment** and click on **Next**. Choose your provisioning profile from the pop-up list and export the `.ipa` file.

Installing on test devices

To install your ad hoc app on a test device, you need to connect the device to a Mac or Windows machine that runs iTunes. Drag your `.ipa` package file to the iTunes application list; this will install the app into the local iTunes library. Select the device from the device list in iTunes and select the **Apps** tab from the list at the top of the screen and then sort your apps by name. Find your app in the list of apps and click on the **Install** button next to the app. Finally, click on the **Sync** button in the bottom-right-hand corner of iTunes.

iTunes Connect app

When you are ready to distribute your app through iTunes, you will have to add a new app into your iTunes Connect account for your game. Log in to `itunesconnect.apple.com` and open the **Manage Your App** section. You will need to provide all of the information to match up with your app when you click on the **Add New App** button. There is a lot of information in here that is describing your app for iTunes. You should take the time to provide as accurate and as much information as possible in this section, since this is how users will discover your app when they use the search function in the App Store. Keep in mind that the screenshots you provide here will also be the first impression that a user will have of your app. When the status in iTunes Connect reads **Waiting for Upload**, you can proceed. Rebuild your app for distribution in Xcode using your distribution certificate instead of the development certificate. Under the Xcode menu, navigate to **Open Developer Tool | Application Loader**. Follow the onscreen instructions for submitting your app. It should be as simple as logging into your account and clicking on the **Deliver Your App** button. Once that is done, it is just a matter of waiting for Apple to approve your app. Keep checking your app's status in iTunes Connect to make sure your app has not been rejected for some reason. If it has, just follow the instructions provided by Apple to resolve the issue.

www.it-ebooks.info

Avoiding rejection from the App Store

Generally, your app should not get rejected as long as you supplied all of the correct information to iTunes Connect. However, it has been known to happen. For a detailed list of reasons for getting rejected, read the App Store Review Guidelines that are available in the iOS Dev Center. Here are a few of the more common reasons your app may get rejected:

- **Information about your app**: It is very important that when you are submitting your app and your app's information, that you be as truthful as possible. Don't try and submit a game by falsely presenting it, for example, as a kid's app. No doubt, it will get bounced back to you.
- **Art, screenshots, and icons**: Make sure you supply all of your art files in the format and size that Apple requests. This has been known to change, so make sure you double-check all of your data with Apple's latest docs rather than trusting that it is the same as it has always been. Also, make sure you own all the art, trademarks, and logos used in your app.
- **Crashes and load times**: I am sure it goes without saying, but if your app crashes, it will get rejected. It will also get rejected if your app takes too long to load. If your app is taking more than ten seconds to load, something is probably wrong with your code.
- **Bypassing apple code. ent system**: If you bypass the Apple Store's payment process in any way, you will most likely get rejected. This includes linking to a website and requesting a payment through the site rather than making a payment through the Apple Store. This has been tried in several different ways and it never ends well for the developer.
- **Copying applesting a payment**: Apple has a basic set of apps that they have built into iOS. If you try and build something similar to one of Apple's apps, it will most likely be rejected. So, if you build an app to make phone calls, to play locally stored music, or to e-mail a client, it will most likely get bounced back to you.

Summary

In this chapter, we introduced you to the Apple Store and iTunes Connect's submission process; we prepared your app for distribution as well. At this point, you should be able to create an app, its icons, and screenshots, and submit them to the Apple Store with little or no issues.

In the next chapter, we will cover developing an app for the iPad, using Apple's Game Center, in-app purchasing, saving space with iCloud, and much more.

www.it-ebooks.info

10
Conclusion

You did it! Congratulations! You can create and sell apps for gamers like you. By purchasing this book, becoming an iOS developer, and following the steps in each chapter, you have learned about the following:

- Setting up and installing Xcode by setting up a developer account with Apple
- Programing in Objective-C by developing a Hello World! iPhone app, as well as gaining a general understanding of how most programming languages are structured
- How to use CoreGraphics, Foundation, and UIKit frameworks by creating a simple matching game
- How to work with graphics, artificial intelligence, and game logic by creating a Space Invaders game; specifically, how to display a background image, program aliens to drop bombs, and store information about your invaders in objects, properties, and arrays
- Working with audio (sound effects, a background music loop, and so on) by developing an audio-based Simon memory game
- Programming moving graphics using touch events (buttons, gestures, and multitouch interactivity) and creating a Mini Golf game
- And then, stepping it up a notch by creating a Galaga-type game using what you've previously learned, as well as learning about resource management
- And ultimately, learning how to sell your completed app in the App Store

Although we did not cover all of the technologies that Apple offers for building apps and games, with what you have learned in this book, you should be able to create just about any type of app. Apple supplies several tools for developing different types of apps for the iPhone. If you would like to expand your knowledge base for iOS devices, I recommend looking into the following areas.

www.it-ebooks.info

Developing an app for iPad

The iPad is just a short leap from iPhone. Anything you can do on the iPhone you can do on the iPad (except maybe make a telephone call) with some adjustments to your scaling. Take some time to learn about the different sizes and capabilities on the iPad, and you will be surprised how much of what you have learned is applied there.

Using Apple's game center

Using Apple's game center can be useful if you want to integrate social gaming features into your game. Apple provides all the tools necessary to track and share high scores, leaderboards, achievements, challenges, multiplayer matchmaking, and real-time multiuser support. Best of all, this information is all stored on Apple's servers, so you don't need to worry about taking a hit on your servers.

In-app purchasing

A lot of developers are using in-app purchasing as a way of distributing their games for free or very cheap and letting their players make purchases after they get their app. This model lets you charge a basic price and your players get to decide how they want to expand your game, whether through purchasing new levels, additional characters, expanding their weapons catalog, and so on. Even though in-app purchasing is mostly used for expanding games, you could easily use it for subscription services, premium assets, or even expand the functionality of your app.

Implementing iAd

Apple built iAds as a way to implement advertising in games. This helped expand the free-to-play apps market. With apps and games receiving income from advertising, there was no need to charge players to use the apps. Apple expanded this concept with the ability for developers to cross-promote their apps in other apps. Apple even provides iAd Producer, a visual advertising editor, to help you make ads more effective and visually appealing.

The social framework

In iOS 6, Apple added the ability to communicate with social networks in a much easier manner. This lets you communicate with Twitter, Facebook, and Sina Weibo (a microblogging network in China). The framework gives you a template for creating HTTP requests and even gives you a generalized interface for posting requests.

www.it-ebooks.info

Core Location and Map Kit

You can add map and location capabilities into your app, using Core Location and Map Kit. By using Core Location, you can get your current GPS location, and with Map Kit, you can display a map and then pan, zoom, and show your current location.

Saving space with iCloud

Do you need a way to store data across devices and don't want to use valuable storage space on your servers? iCloud is the solution. iCloud's storage APIs are easy to use and very effective. iCloud will store your users' documents and key value data and make it available to them wirelessly. You can push documents to a user's device automatically and update the documents whenever your user changes them across devices.

Exploring game engines

If you would like to take a look at more of what you can do with games, I highly recommend looking into some of the game engines available for iPhone.

If you really like programming 2D games for iPhone and would really like to take it to the next level, I recommend digging into the Cocos2D game engine. Cocos2D is a free, open source game engine under the MIT license. You can make some strong games by taking what you have learned here and applying it to the capabilities of Cocos. Cocos2D has a multiplatform version called Cocos2D-X, which you can download from `http://www.cocos2d-x.org`. This will allow you to build for multiple platforms, including iOS devices, Android, Windows Mobile, Windows, Mac, and Linux. There is also an iPhone-specific variant that you can download, Cocos2D, for free at `http://www.cocos2d-iphone.org`.

Personally, I am a big fan of Unity Technologies' gaming engine. You can create very complex 2D and 3D games in an easy-to-manipulate environment. You then have the capabilities to export to other devices including directly to Xcode. With your new-found Xcode knowledge, you can manipulate some of what is exported. The Unity engine will also let you port to the same platforms as Cocos2D-X, but you can also port to PlayStation platforms, Xbox platforms, Wii, and Blackberry. You can download Unity at `http://unity3d.com`; both their free version and their pro version are available here. With Unity, you have a few more capabilities that other engines do not provide, such as being able to program in C#, Boo, or JavaScript. You can also write your own plugins using native code libraries that use C, C++, or Objective-C and access them within the Unity environment with C#, JavaScript, or Boo. Unity has their own asset store where you can download and purchase plugins, art, effects, and code for extending your games.

www.it-ebooks.info

One last engine I would like to mention is the Unreal Engine. The Unreal Engine has been in the games industry the longest, developed by the game development team at Epic Games. It does have a blueprint visual scripting environment, but if you want to modify any code, you will have to use C++ and dig into the source. Unreal has its marketplace, which is similar to Unity's asset store, which allows you to purchase additions to the engine and assets for your game. You can download the full source code engine at `https://www.unrealengine.com` with the subscription plan.

Each of these engines offers its own advantages and disadvantages from graphics quality to pricing. There are new engines coming out every day that offer different advantages. It is often easier to figure out what your project needs and then decide if you want to use a preexisting game engine or program everything by yourself. By using a game engine, you free yourself up to work on your game design and less on core programming. In turn, you will always end up giving up something else in another area, such as file or memory size.

Summary

Well, that's the end of our book, but it should be the beginning for you as an Apple developer. Thank you for purchasing this book and allowing us to guide you. We enjoyed creating this book and hope you enjoyed yourself too.

Now that you have been given everything you need to get a leg up on developing games for iPhone, go and make something cool!

www.it-ebooks.info

Index

A

accelerometer 201
Activity viewer 19
ad hoc app
  installing, on test device 251
ad hoc distribution 247
ad hoc provisioning profile
  creating, for apps distribution 250
Advanced Audio Coding (AAC) 162
APIs (Application Programming Interface) 85
app
  App ID, creating for 249
  developing, for iPad 254
  preparing, for distribution 241
app icons
  selecting 245, 246
App ID
  creating, for app 249
Apple
  game center, using 254
  URL, for additional development tools 13
  URL, for user registration in developer community 8
Apple developer
  becoming 7, 8
  registering as 8, 9
Apple ID
  creating 9
app package
  creating 251
app rejection
  reasons 252
apps distribution
  ad hoc provisioning profile, creating for 250
App Store
  rejection, avoiding from 252
arguments 45
arithmetic operators 35
arrays 31
assignment operator 32
atlas directory example 214
Atlas Texture 220
attributes inspector 25
audio files
  data format 161, 162
  file format 161, 162
Audio Interchange File Format (AIFF) 162

B

beta testing 247
bindings inspector 25
bit rate 162
block comment 34
boolean (bool) data type 31
breakpoint navigator 21
break statement 43
build string 242
bundle identifier 242

C

character (char) data type 30
C language 29
class 47
class method 47
Cocoa Touch 85

www.it-ebooks.info

Cocos2D
URL, for downloading 255
Cocos2D-X
about 255
URL, for downloading 255
code
debugging 153-158
collision detection, Space Invader game 151
collision detection, Galaxy game 237
comments 33, 34
comparison operators 35
compound operators 37
connections inspector 25
constant 33
Content window 21
continue statement 43
controller objects 63
control statements
about 38
if…else if statement 39
if…else statement 38
if statement 38
switch statement 39, 40
Core Animation 85
Core Audio 85
Core Audio Format (CAF) 162
Core Data 85
Core Data model editor 23
Core Location 255

D

data types 30
data types, Objective-C
NSArray 53, 54
NSMutableArray 54
NSMutableSet 52
NSMutableString 50, 51
NSNumber 51
NSSet 52
NSString 49, 50
date (date) data type 31
Debugger area, Xcode workspace 23
debug navigator 21
decrement operator 37
delegate 174
deployment information
about 244
deployment target 245
device orientation 245
devices 245
status bar style 245
deployment target 245
device orientation 245
distribution
app, preparing for 241
double (double) data type 31
do while loop 42, 43

E

Editor area, Content window
Core Data model editor 23
interface builder 23
mapping model editor 23
project editor 23
property list editor 23
rich text editor 23
script editor 23
scripting dictionary editor 23
source editor 23
Editor area, Xcode workspace
about 22
Jump Bar 22
Editor buttons 19
effects inspector 25
enumerator (enum) data type 146

F

file inspector 25
Filter bar 22
float (float) data type 31
flyer. *See* **game concept document**
for loop 41
Foundation Kit Framework 85
frame rates 152
frameworks 85
functions 44

www.it-ebooks.info

G

Galaxy game
about 209
collision detection 237
enemies, controlling 237
enemies, creating 237
flow charts, for game screens 212
focusing, on resource management 213, 214
gameplay environment, setting up 231, 232
planning 209
player, controlling 232-236
player, creating 232-236
screens, creating 215-218
sounds, adding 238, 239
user interface, building 218-230
game concept document
creating 210
game design document
basic outline 210, 211
creating 210, 211
game engines
exploring 255
game logic 137
gameplay environment
setting up, for Galaxy game 231, 232
game scene
building 86-104
game screens
creating, for Galaxy game 215-218
game state
learning 140-149
gestures
about 184
integrating, into Mini Golf game 186-193
long press 185
pan 185
pinch 184
rotate 185
swipe 185
tap 184
using, in iOS apps 184
Git
URL 17
graphics
about 105
adding, programmatically to scene 106-111
gyroscope 201

H

header file 83
Hello World! program
for iPhone app 54-58

I

iAd
implementing 254
iCloud
space, saving with 255
identity
selecting 242
identity inspector 25
identity section
about 242
build string 242
bundle identifier 242
team provisioning profile 243, 244
version number 242
if…else if statement 39
if…else statement 38
if statement 38
implementation file 58-62
in-app purchasing 254
increment operator 37
Inspector selector bar
about 25
file inspector 25
Quick help 25
installation, Xcode 13
instance method 47
integer (int) data type 31
Integrated Development Environment (IDE) 14

www.it-ebooks.info

interface 58
interface builder 23
iOS apps
gestures, using in 184
iOS Dev Center
URL 9
iOS developer
becoming 10
registering as 10-12
iOS simulator
about 26
accessing 26
simulated hardware, accessing 26, 27
iPad
app, developing for 254
iPhone
basics, for game development 65
iPhone models
screen sizes 106
issue navigator 21
iTunes Connect app 251

L

launch images
selecting 245, 246
Launchpad 14
Library pane
code snippets 26
file templates 26
media files 26
objects 26
linked frameworks 247
logical operators 36
log navigator 21
long press gesture 185
loops
about 41
do while loop 42, 43
for loop 41
while loop 42

M

Map Kit 255
mapping model editor 23
memory game
scene, building 86-104
memory game code
overview 80-84
methods 47
Mini Golf game
about 184
code, initializing 195, 196
cup image, adding 195
gestures, integrating into 186-193
link, creating to Exit button 194
methods, adding for touch management 193-199
motion, demonstrating 206
orientation, demonstrating 202, 203
screen, updating 199-201
shake event, demonstrating 202
tilt, demonstrating 204, 205
modal item 74
model objects 62
model-view-controller
about 62
controller objects 63
model objects 62
view objects 63
motion
demonstrating, in Mini Golf game 206
mouse clicks
for touch simulations 27

N

Navigation area, Xcode workspace
about 20
Content window 20, 21
Filter bar 20, 22
Navigation selector 20
Navigation selector bar
about 20
breakpoint navigator 21
debug navigator 21
issue navigator 21
log navigator 21
project navigator 20
search navigator 21
symbol navigator 20
NSArray data type 53, 54
NSLog 96
NSMutableArray data type 54

www.it-ebooks.info

NSMutableSet data type 52
NSMutableString data type 50, 51
NSNumber data type 51
NSSet data type 52
NSString data type 49, 50
numeric data types
double (double) 31
float (float) 31
integer (int) 31

O

Objective-C
about 29, 30
data types 49
object-oriented programming (OOP)
about 46
class 47
methods 47
objects 47
pointers 48
objects 47
one-sheet doc. *See* **game concept document**
OPENSTEP operating system 30
operators
about 35
arithmetic 35
comparison 35
compound 37
decrement 37
increment 37
logical 36
orientations 202, 203

P

pan gesture 185
pinch gesture 184
pitch doc. *See* **game concept document**
pixel 105
plan, for Galaxy game
game concept document, creating 210
game design document, creating 210, 211
technical design document,
creating 211, 212
playback file formats
AAC 162
ALAC 162
HE-AAC 162
iLBC 163
IMA4 163
Linear PCM 163
MP3 163
u-law/a-law 163
pointers 48
programming basics
about 30
basic data types 30
comments 33
constants 33
control statements 38
functions 44
loops 41
operators 35
variables 32
variable scope 45
programming, with Objective-C
implementation file 58-62
interface 58
project editor 23
project navigator 20
property list editor 23

Q

queue 205
Quick help 25

R

recording file formats
AAC 163
ALAC 163
iLBC 163
IMA4 163
linear PCM 163
u-law/a-law 163
rejection
avoiding, from App Store 252
resource management, Galaxy game
focusing on 213, 214
rich text editor 23
rotate gesture 185
Run button 19

www.it-ebooks.info

S

sample rate 162
scenes
about 65, 70
controlling, code used 70
creating 70
Scheme pop-up menu 19
screens 105
screens, Galaxy game
End game splash 213
Game Screen 213
Game splash video or screens 212
Main menu 212
Splash screens 212
script editor 23
scripting dictionary editor 23
search navigator 21
segue 73
shake event 202
Simon Says game
about 163
code, involved 174
exit game action 181
functioning 163
game play area, creating 168-170
game state, initializing 176
game state, modifying 177, 178
implementation file, breaking down for game controller 175
invisible buttons, creating 171, 172
labels, adding to layouts 171
main menu area, setting up 167
music, playing 176
options, selecting for file 165
options, selecting for file addition 166
options, selecting for project 164
properties, for controlling game 174
sound effects, playing 179
stop call, sending to audio 181
Single View Application template
overview 66-69
size inspector 25
SmallTalk 29
social framework 254
Software Development Kit (SDK) 7
sounds
adding, to Galaxy game 238, 239
source editor 23
space
saving, with iCloud 255
Space Invaders-style game
about 105
collision detection 151
enemies bombs, creating 130-135
enemies, creating 117-124
frame rates 152
game logic, understanding 137, 138
game state, learning 140-149
graphics, adding to scene 106-111
ship sprite, moving 112-116
sprite sheet, using 124-130
state machines, understanding 138-140
timers 152
sprite 112
SpriteKit.framework 217
SpriteKit Game template 215
sprite sheet
about 124
using 124-130
state machines 138-140
Stop button 19
storyboards
about 70
drag-and-drop, performing for objects 70-80
string (NSString) data type 30
swipe gesture 185
switch statement 39, 40
symbol navigator 20

T

tap gesture 184
team provisioning profile 243, 244
technical design document
basic outline 211, 212
creating 211, 212
templates 80
test device
ad hoc app, installing on 251
registering 247

www.it-ebooks.info

TestFlight
URL 248
text data types
character (char) 30
string (NSString) 30
tilt
demonstrating, in Mini Golf game 204, 205
timers 152
toolbar, Xcode workspace
about 18
Activity viewer 19
Editor buttons 19
Run button 19
Scheme pop-up menu 19
Stop button 19
View buttons 19
touchesBegan method 193
touchesCancelled method 193
touchesEnded method 193
touchesMoved method 193
touch event 183, 184
touch up outside event 184
type casting 32

U

UIEvent object 183
UIKit 85
UIResponder class 183
UITouch object 183
Unity
URL, for downloading 255
Unreal Engine
about 256
URL, for downloading 256
user interface
building, for Galaxy game 218-230
Utility area, Xcode workspace
about 24
Inspector selector bar 25

V

variables 30, 32
variable scope 45
version number, app 242
View buttons 19
view controllers 70
view objects 63
views 65

W

WAV 162
while loop 42

X

Xcode 82
about 153
downloading 13
installing 13
opening 14-17
removing 13
URL, for downloading from Macintosh App Store 13
Xcode IDE 14
Xcode workspace
about 18
Debugger area 23
Editor area 22
Navigation area 20
toolbar 18, 19
Utility area 24

www.it-ebooks.info

www.it-ebooks.info

[PACKT] PUBLISHING

Thank you for buying

Learning Objective-C by Developing iPhone Games

About Packt Publishing

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

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

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

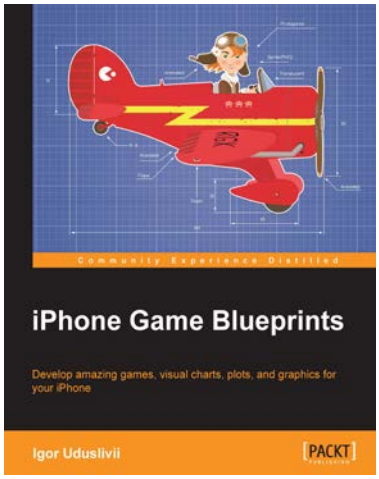
Writing for Packt

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

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

www.it-ebooks.info

[PACKT]
PUBLISHING

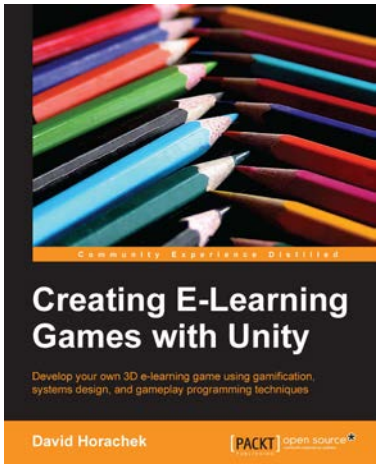


iPhone Game Blueprints

ISBN: 978-1-84969-026-3 Paperback: 358 pages

Develop amazing games, visual charts, plots, and graphics for your iPhone

1. Seven step-by-step game projects for you to build.
2. Cover all aspects from graphics to game ergonomics.
3. Tips and tricks for all of your iPhone programming.



Creating E-Learning Games with Unity

ISBN: 978-1-84969-342-4 Paperback: 246 pages

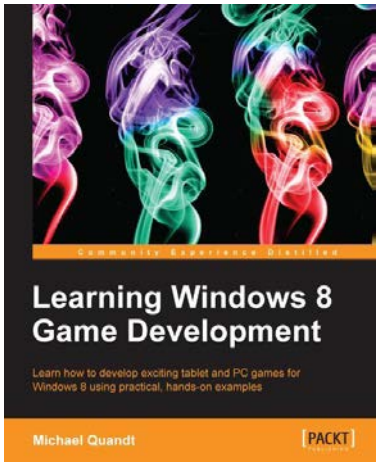
Develop your own 3D e-learning game using gamification, systems design, and gameplay programming techniques

1. Develop a game framework for a 3D e-learning game.
2. Program dynamic interactive actors and objects to populate your game world.
3. An easy-to-follow guide along with an extensive source code to support and guide readers through the concepts in the book.

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

www.it-ebooks.info

[PACKT]
PUBLISHING

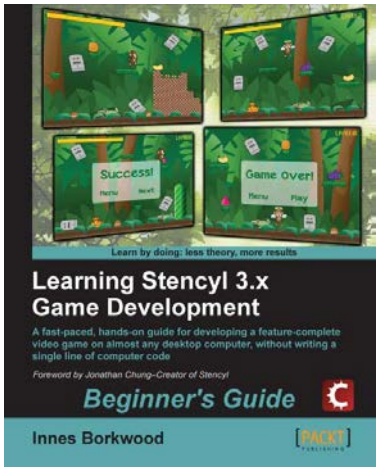


Learning Windows 8 Game Development

ISBN: 978-1-84969-744-6 Paperback: 244 pages

Learn how to develop exciting tablet and PC games for Windows 8 using practical, hands-on examples

1. Use cutting-edge technologies like DirectX to make awesome games.
2. Discover tools that will make game development easier.
3. Bring your game to the latest touch-enabled PCs and tablets.



Learning Stencyl 3.x Game Development Beginner's Guide

ISBN: 978-1-84969-596-1 Paperback: 336 pages

A fast-paced, hands-on guide for developing a feature-complete video game on almost any desktop computer, without writing a single line of computer code

1. Learn important skills that will enable you to quickly create exciting video games, without the complexity of traditional programming languages.
2. Find out how to maximize potential profits through licensing, paid-sponsorship, and in-game advertising.
3. Explore numerous step-by-step tutorials that will guide you through the essential features of Stencyl's powerful game-development toolkit.

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

www.it-ebooks.info