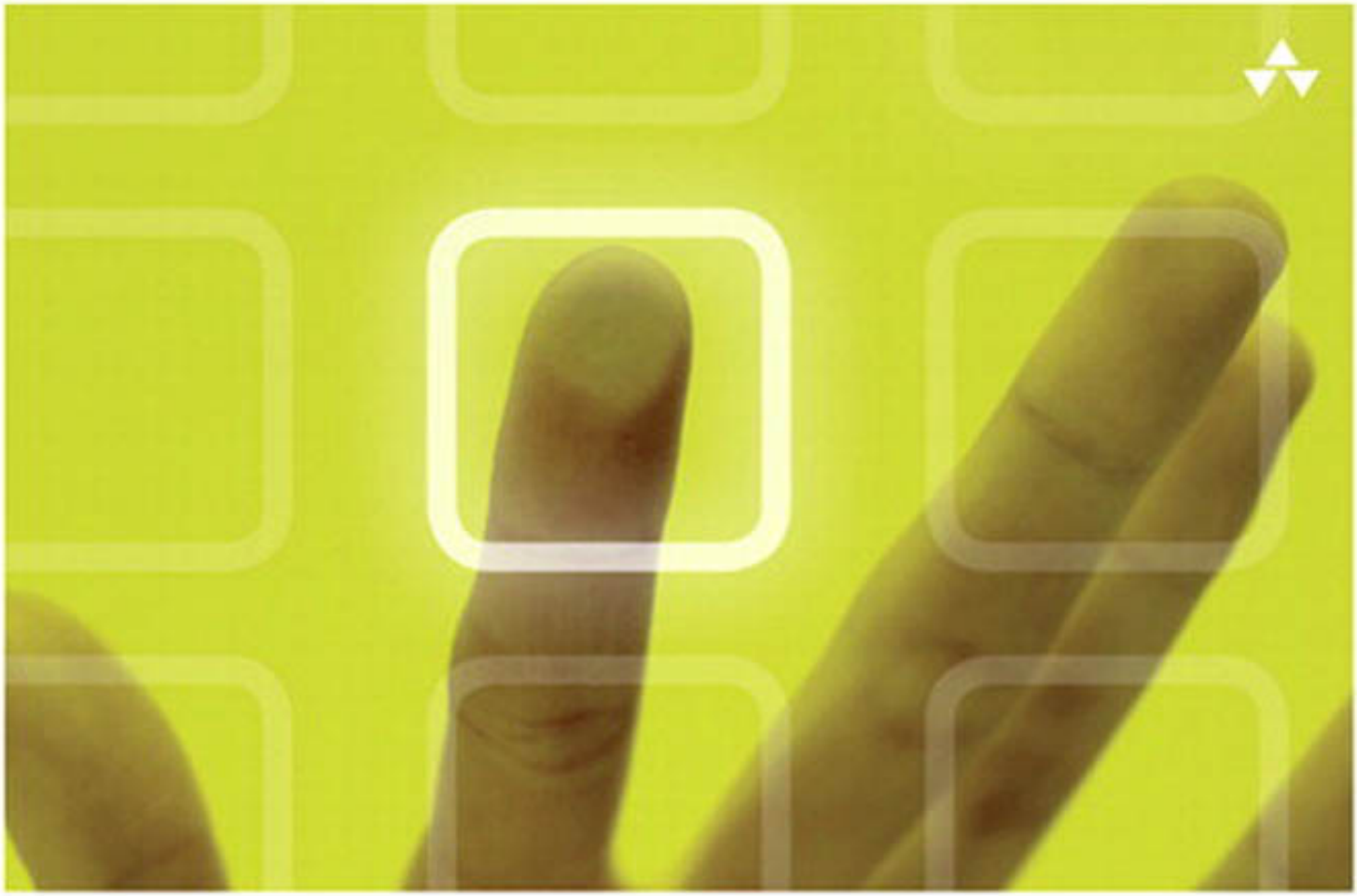

LEARNING Cocos2D

A Hands-On Guide to Building iOS Games with Cocos2D, Box2D, and Chipmunk

ROD STROUGO
RAY WENDERLICH

Praise for *Learning Cocos2D*

"If you're looking to create an iPhone or iPad game, *Learning Cocos2D* should be the first book on your shopping list. Rod and Ray do a phenomenal job of taking you through the entire process from concept to app, clearly explaining both how to do each step as well as why you're dong it."

—Jeff LaMarche, Principal, MartianCraft, LLC, and coauthor of *Beginning iPhone Development* (Apress, 2009)

"This book provides an excellent introduction to iOS 2D game development. Beyond that, the book also provides one of the best introductions to Box2D available. I am truly impressed with the detail and depth of Box2D coverage."

—Erin Catto, creator of Box2D

"Warning: reading this book will make you *need* to write a game! *Learning Cocos2D* is a great fast-forward into writing the next hit game for iOS—definitely a must for the aspiring indie iOS game developer (regardless of experience level)! Thanks, Rod and Ray, for letting me skip the learning curve; you've really saved my bacon!"

—Eric Hayes, Principle Engineer, Brewmium LLC (and Indie iOS Developer)

"*Learning Cocos2D* is an outstanding read, and I highly recommend it to any iOS developer wanting to get into game development with Cocos2D. This book gave me the knowledge and confidence I needed to write an iOS game without having to be a math and OpenGL whiz."

—Kirby Turner, White Peak Software, Inc.

"*Learning Cocos2D* is both an entertaining and informative book; it covers everything you need to know about creating games using Cocos2D."

—Fahim Farook, RookSoft (rooksoft.co.nz)

"This is the premiere book on Cocos2D! After reading this book you will have a firm grasp of the framework, and you will be able to create a few different types of games. Rod and Ray get you quickly up to speed with the basics in the first group of chapters. The later chapters cover the more advanced features, such as parallax scrolling, CocosDenshion, Box2D, Chipmunk, particle systems, and Apple Game Center. The authors' writing style is descriptive, concise, and fun to read. This book is a must have!"

—Nick Waynik, iOS Developer

This page intentionally left blank

Learning Cocos2D

Addison-Wesley Learning Series

Visit informit.com/learningseries for a complete list of available publications.

The Addison-Wesley Learning Series is a collection of hands-on programming guides that help you quickly learn a new technology or language so you can apply what you've learned right away.

Each title comes with sample code for the application or applications built in the text. This code is fully annotated and can be reused in your own projects with no strings attached. Many chapters end with a series of exercises to encourage you to reexamine what you have just learned, and to tweak or adjust the code as a way of learning.

Titles in this series take a simple approach: they get you going right away and leave you with the ability to walk off and build your own application and apply the language or technology to whatever you are working on.

Learning Cocos2D

A Hands-On Guide to Building iOS Games with Cocos2D, Box2D, and Chipmunk

Rod Strougo
Ray Wenderlich

Addison-Wesley

Upper Saddle River, NJ · Boston · Indianapolis · San Francisco
New York · Toronto · Montreal · London · Munich · Paris · Madrid
Capetown · Sydney · Tokyo · Singapore · Mexico City

Library of Congress Cataloging-in-Publication Data

Strougo, Rod, 1976-
Learning Cocos2D : a hands-on guide to building iOS games with Cocos2D, Box2D, and Chipmunk / Rod Strougo, Ray Wenderlich.
p. cm.
Includes index.
ISBN-13: 978-0-321-73562-1 (pbk. : alk. paper)
ISBN-10: 0-321-73562-5 (pbk. : alk. paper)
1. iPhone (Smartphone)—Programming. 2. iPad (Computer)—Programming. 3. Computer games—Programming. I. Wenderlich, Ray, 1980- II. Title.
QA76.8.I64S87 2011
794.8'1526—dc23

2011014419

ISBN-13: 978-0-321-73562-1
ISBN-10: 0-321-73562-5
Text printed in the United States on recycled paper at RR Donnelley in Crawfordsville, Indiana.
First printing, July 2011

Editor-in-Chief
Mark Taub

Acquisitions Editor
Chuck Toporek

Managing Editor
John Fuller

Project Editor
Anna Popick

Copy Editor
Carol Lallier

Indexer
Jack Lewis

Proofreader
Lori Newhouse

Editorial Assistant
Olivia Basegio

Cover Designer
Chuti Prasertsith

Compositor
The CIP Group

❖

Dedicated to my wife, Agata.
—Rod

Dedicated to my wife, Vicki.
—Ray

❖

This page intentionally left blank

Contents at a Glance

Contents

Preface

So you want to be a game developer?

Developing games for the iPhone or iPad can be a lot of fun. It is one of the few things we can do to feel like a kid again. Everyone, it seems, has an idea for a game, and what better platform to develop for than the iPhone and iPad?

What stops most people from actually developing a game, though, is that game development covers a wide swath of computer science skills—graphics, audio, networking—and at times it can seem like you are drinking from a fire hose. When you are first getting started, becoming comfortable with Objective-C can seem like a huge task, especially if you start to look at things like OpenGL ES, OpenAL, and other lower-level APIs for your game.

Writing a game for the iPhone and iPad does not have to be that difficult—and it isn't. To help simplify the task of building 2D games, look no further than Cocos2D.

You no longer have to deal with low-level OpenGL programming APIs to make games for the iPhone, and you don't need to be a math or physics expert. There's a much faster and easier way—use a free and popular open source game programming framework called Cocos2D. Cocos2D is extremely fun and easy to use, and with it you can skip the low-level details and focus on what makes your game different and special!

This book teaches you how to use Cocos2D to make your own games, taking you step by step through the process of making an actual game that's on the App Store right now! The game you build in this book is called *Space Viking* and is the story of a kick-ass Viking transported to an alien planet. In the process of making the game, you get hands-on experience with all of the most important elements in Cocos2D and see how everything fits together to make a complete game.

> **Download the Game!**
>
> You can download *Space Vikings* from the App Store: *http://itunes.apple.com/us/app/space-vikings/id400657526mt=8*. The game is free, so go ahead and download it, start playing around with it, and see if you're good enough to get all of the achievements!

Think of this book as an epic-length tutorial, showing you how you can make a real game with Cocos2D from the bottom up. You'll be coding along with the book, and we explain things step by step. By the time you've finished reading and working

through this book, you'll have made a complete game. Best of all, you'll have the confidence and knowledge it takes to make your own.

Each chapter describes in detail a specific component within the game along with the technology required to support it, be it a tile map editor or some effect we're creating with Cocos2D, Box2D, or Chipmunk. Once an introduction to the functionality and technology is complete, the chapter provides details on how the component has been implemented within *Space Viking*. This combination of theory and real-world implementation helps to fill the void left by other game-development books.

What Is Cocos2D?

Cocos2D *(www.cocos2d-iphone.org)* is an open source Objective-C framework for making 2D games for the iOS and Mac OS X, which includes developing for the iPhone, iPod touch, the iPad, and the Mac. Cocos2D can either be included as a library to your project in Xcode or automatically added when you create a new game using the included Cocos2D templates.

Cocos2D uses OpenGL ES for graphics rendering, giving you all of the speed and performance of the graphics processor (GPU) on your device. Cocos2D includes a host of other features and capabilities, which you'll learn more about as you work through the tutorial in this book.

Cocos2D started life as a Python framework for doing 2D games. In late 2008, it was ported to the iPhone and rewritten in Objective-C. There are now additional ports of Cocos2D to Ruby, Java (Android), and even Mono (C#/.NET).

> **Note**
>
> Cocos2D has an active and vibrant community of contributors and supporters. The Cocos2D forums *(www.cocos2d-iphone.org/forum)* are very active and an excellent resource for learning and troubleshooting as well as keeping up to date on the latest developments of Cocos2D.

Why You Should Use Cocos2D

Cocos2D lets you focus on your core game instead of on low-level APIs. The App Store marketplace is very fluid and evolves rapidly. Prototyping and developing your game quickly is crucial for success in the App Store, and Cocos2D is the best tool for helping you quickly develop your game without getting bogged down trying to learn OpenGL ES or OpenAL.

Cocos2D also includes a host of utility classes such as the `TextureCache`, which automatically caches your graphics, providing for faster and smoother gameplay. `TextureCache` operates in the background and is one of the many functions of Cocos2D that you don't even have to know how to use; it functions transparently to

you. Other useful utilities include font rendering, sprite sheets, a robust sound system, and many more.

Cocos2D is a great prototyping tool. You can quickly make a game in as little as an hour (or however long it takes you to read Chapter 2). You are reading this book because you want to make games for the iPhone and iPad, and using Cocos2D is the quickest way to get there—bar none.

Cocos2D Key Features

Still unsure if Cocos2D is right for you? Well, check out some of these amazing features of Cocos2D that can make developing your next game a lot easier.

Actions

Actions are one of the most powerful features in Cocos2D. Actions allow you to move, scale, and manipulate sprites and other objects with ease. As an example, to smoothly move a space cargo ship across the screen 400 pixels to the right in 5 seconds, all the code you need is:

```
CCAction *moveAction = [CCMoveBy actionWithDuration:5.0f
                                 position:CGPointMake(400.0f,0.0f)];
[spaceCargoShipSprite runAction:moveAction];
```

That's it; just two lines of code! Figure P.1 illustrates the `moveAction` on the space cargo ship.

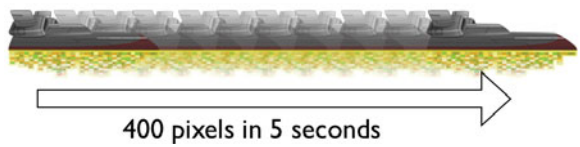

Figure P.1 Illustrating the effect of the moveAction on the Space Cargo Ship sprite

There are many kinds of built-in actions in Cocos2D: rotate, scale, jump, blink, fade, tint, animation, and more. You can also chain actions together and call custom callbacks for neat effects with very little code.

Built-In Font Support

Cocos2D makes it very easy to deal with text, which is important for games in menu systems, score displays, debugging, and more. Cocos2D includes support for embedded TrueType fonts and also a fast bitmap font-rendering system, so you can display text to the screen with just a few lines of code.

An Extensive Effects Library

Cocos2D includes a powerful particle system that makes it easy to add cool effects such as smoke, fire, rain, and snow to your games. Also, Cocos2D includes built-in effects, such as flip and fading, to transition between screens in your game.

Great for TileMap Games

Cocos2D includes built-in support for tile-mapped games, which is great when you have a large game world made up of small reusable images. Cocos2D also makes it easy to move the camera around to implement scrolling backgrounds or levels. Finally, there is support for parallax scrolling, which gives your game the illusion of 3D depth and perspective.

Audio/Sound Support

The sound engine included with Cocos2D allows for easy use of the power of OpenAL without having to dive into the lower level APIs. With Cocos2D's sound engine, you can play background music or sound effects with just a single line of code!

Two Powerful Physics Engines

Also bundled with Cocos2D are two powerful physics engines, Box2D and Chipmunk, both of which are fantastic for games. You can add a whole new level of realism to your games and create entire new gameplay types by using game physics—without having to be a math guru.

Important Concepts

Before we get started, it's important to make sure you're familiar with some important concepts about Cocos2D and game programming in general.

Sprite

You will see the term *sprite* used often in game development. A sprite is an image that can be moved independently of other images on the screen. A sprite could be the player character, an enemy, or a larger image used in the background. In practice, sprites are made from your PNG or PVRTC image files. Once loaded in memory, a sprite is converted into a texture used by the iPhone GPU to render onscreen.

Singleton

A *singleton* is a special kind of Objective-C class, which can have only one instance. An example of this is an iPhone app's Application Delegate class, or the Director class in Cocos2D. When you call a singleton instance in your code, you always get back the one instance of this class, regardless of which class called it.

OpenGL ES

OpenGL ES is a mobile version (ES stands for *Embedded Systems*) of the Open Graphics Language (OpenGL). It is the closest you can get on the iPhone or iPad to sending zeros and ones to the GPU. OpenGL ES is the fastest way to render graphics on the iPhone or iPad, and due to its origin, it is a low-level API. If you are new to game development, OpenGL ES can have a steep learning curve, but luckily you don't need to know OpenGL ES to use Cocos2D.

The two versions of OpenGL ES supported on the iPhone and iPad are 1.1 and 2.0. There are plans in the Cocos2D roadmap to support OpenGL ES 2.0, although currently only version 1.1 is supported.

Languages and Screen Resolutions

Cocos2D is written in Objective-C, the same language as Cocoa Touch and the majority of the Apple iOS APIs. In Objective-C it is important to understand some basic memory-management techniques, as it is a good foundation for you to become an efficient game developer on the iOS platform. Cocos2D supports all of the native resolutions on the iOS devices, from the original iPhone to the iPad to the retina display on the iPhone 4.

2D versus 3D

You first learn to walk before you can run. The same is true for game development; you have to learn how to make 2D games before diving into the deeper concepts of 3D games. There are some 3D effects and transitions in Cocos2D, such as a 3D wave effect and an orbit camera move; however, most of the functionality is geared toward 2D games and graphics.

Cocos2D is designed for 2D games (hence the 2D in the name), as are the tutorials and examples in this book. If you want to make 3D games, you should look into different frameworks, such as Unity, the Unreal Engine, or direct OpenGL.

The Game behind the Book: Space Viking

This book takes you through the process of creating a full-featured Cocos2D-based game for the iPhone and iPad. The game you build in this book is called *Space Viking*. If you want to try *Space Viking* now, you can download a free version of the game from the App Store *(http://itunes.apple.com/us/app/id400657526)* and install it on your iPhone, iPod touch, or iPad.

Of course, if you are more patient, you can build the game yourself and load it onto your device after working through the chapters in this book. There is no greater learning experience than having the ability to test a game as you're building it. Not only can you learn how to build a game, but you can also go back and tweak the code a bit to change things around to see what sort of effect something has on the gameplay. Good things come to those who wait.

This book teaches you how to use all of the features and capabilities of Cocos2D, but more important, how to apply them to a real game. By the time you are done, you will have the knowledge and experience needed to get your own game in the App Store. The concepts you learn from building *Space Viking* apply to a variety of games from action to puzzle.

Space Viking's Story

Every game starts in the depths of your imagination, with a character and storyline that gets transformed into a game. This is the story of *Space Viking*.

In the future, the descendants of Earth are forced into colonizing planets outside our own solar system. In order to create hospitable environments, huge interplanetary machines extract giant chunks of ice from Northern Europe and Greenland and send it across the galaxy to these planets. Unbeknown to the scientists, one of these chunks contains Ole the Viking, who eons ago fell into an icy river on his way home from defeating barbarian tribes. Encased in an icy tomb for centuries, Ole awakens thousands of years later—and light years from home—after being warmed by an alien sun, as shown in Figure P.2.

Figure P.2 Ole awakens on the alien planet

You get to play as Ole the Viking and battle the aliens on this strange world in hopes of finding a way to return Ole to his native land and time.

You control Ole's movement to the right and left by using the thumb joystick on the left side of the screen. On the right side are buttons for jumping and attacking. Ole starts out with only his fists. In later levels Ole finds his trusty mallet, and you use the accelerometer to control him in the physics levels.

Space Viking is an action and adventure game, with the emphasis on *action*. The goal was to create a real game from the ground up so you could learn not only Cocos2D but also how to use it in a real full-featured game. The idea for the game came from

concept art that Eric Stevens, a graphic artist and fellow game devotee, developed earlier when we were discussing game ideas to make next.

Space Viking consists of a number of levels, each of which demonstrates a specific area of Cocos2D or gameplay type. For example, the first level is a side-scrolling beat 'em up, and the fourth level is a mine cart racing level that shows off the game physics found in Box2D and Chipmunk. Our hope is that you can reuse parts of *Space Viking* to make your own game once you've finished this book! That's right: you can freely reuse the code in this book to build your own game.

Organization of This Book

The goal of this book is to teach you about game development using Cocos2D as you build *Space Viking* (and learn more about the quest and story of Ole the Viking). You start with a simple level and some basic game mechanics and work your way up to creating levels with physics and particle systems and finally to a complete game by the end of the book.

First you learn the basics of Cocos2D and build a small level with basic running and jumping movements for Ole. Part II shows you how to add animations, actions, effects, and even text to *Space Viking*. Part III takes the game further, adding more levels and scenes, sounds, and scrolling to the gameplay. In Part IV realism is brought into the game with the Box2D and Chipmunk physics engines. Finally in Part V, you learn how to add a particle system, add high scores, connect to social networks, and debug and optimize *Space Viking* to round out some best practices for the games you will build in the future.

There are 17 chapters and one appendix in the book, each dealing with a specific area of creating *Space Viking*.

- **Part I: Getting Started with Cocos2D**

  Learn how to get Cocos2D installed and start using it to create *Space Viking*. Learn how to add animations and movements to Ole and his enemies.

  - **Chapter 1: Hello, Cocos2D**

    This chapter covers how to install Cocos2D framework and templates in Xcode and some companion tools that make developing games easier. These tools are freely available and facilitate the creation of the elements used by Cocos2D.

  - **Chapter 2: Hello, Space Viking**

    Here you create the basic *Space Viking* game, which you build upon throughout the book. You start out with just a basic Cocos2D template and add the hero (Ole the Viking) to the scene. In the second part of this chapter, you add the methods to handle the touch inputs, including moving Ole around and making him jump.

- **Chapter 3: Introduction to Cocos2D Animations and Actions**

  In this chapter, you learn how to make the game look much more realistic by adding animations to Ole as he moves around the scene.

- **Chapter 4: Simple Collision Detection and the First Enemy**

  In this chapter, you learn how to implement simple collision detection and add the first enemy to your *Space Viking* game, so Ole can start to fight his way off the planet!

- **Part II: More Enemies and More Fun**

  Learn how to create more complex enemies for Ole to battle and in the process learn about Cocos2D actions and effects. Finish up with a live, onscreen debugging system using Cocos2D text capabilities.

  - **Chapter 5: More Actions, Effects, and Cocos2D Scheduler**

    Actions are a key concept in Cocos2D—they are an easy way to move objects around, make them grow or disappear, and much more. In this chapter, you put them in practice by adding power-ups and weapons to the level, and you learn some other important Cocos2D capabilities, such as effects and the scheduler.

  - **Chapter 6: Text, Fonts, and the Written Word**

    Most games have text in them at some point, and *Space Viking* is no exception. In this chapter, you learn how to add text to your games using the different methods available in Cocos2D.

- **Part III: From Level to Game**

  Learn how to expand the *Space Viking* level into a full game by adding menus, sound, and scrolling.

  - **Chapter 7: Main Menu, Level Completed, and Credits Scenes**

    Almost all games have more than one screen (or "scene," as it's called in Cocos2D); there's usually a main menu, main game scene, level completed, and credits scene at the very least. In this chapter, you learn how to create multiple scenes by implementing them in *Space Viking*!

  - **Chapter 8: Pump Up the Volume!**

    Adding sound effects and music to a game can make a huge difference. Cocos2D makes it really easy with the CocosDenshion sound engine, so in this chapter you give it a try!

  - **Chapter 9: When the World Gets Bigger: Adding Scrolling**

    A lot of games have a bigger world than can fit on one screen, so the world needs to scroll as the player moves through it. This can be tricky to get right, so this chapter shows you how by converting the beat-'em-up into a side-scroller, using Cocos2D tile maps for improved performance.

- **Part IV: Physics Engines**

  With the Box2D and Chipmunk physics engines that come with Cocos2D, you can add some amazing effects to your games, such as gravity, realistic collisions, and even ragdoll effects! In these chapters you get a chance to add some physics-based levels to *Space Viking*, from simple to advanced!

  - **Chapter 10: Basic Game Physics: Adding Realism with Box2D**

    Just as Cocos2D makes it easy to make games for the iPhone without knowing low-level OpenGL details, Box2D makes it easy to add physics to your game objects without having to be a math expert. In this chapter, you learn how to get started with Box2D by making a fun puzzle game where objects move according to gravity.

  - **Chapter 11: Intermediate Game Physics: Modeling, Racing, and Leaping**

    This chapter shows you some of the really neat stuff you can do with Box2D by making the start of a side-scrolling cart-racing game. In the process, you learn how to model arbitrary shapes, add joints to restrict movement of physics bodies, and much more!

  - **Chapter 12: Advanced Game Physics: Even Better than the Real Thing**

    In this chapter, you make the cart-racing level even more amazing by adding spikes to dodge and an epic boss fight at the end. You learn more about joints, how to detect collisions, and how to add enemy logic as well.

  - **Chapter 13: The Chipmunk Physics Engine (No Alvin Required)**

    The second physics engine that comes with Cocos2D, called Chipmunk, is similar to Box2D. This chapter shows you how to use Chipmunk, compares it to Box2D, and gives you hands-on practice by making a Metroid-style escape level.

- **Part V: Particle Systems, Game Center, and Performance**

  Learn how to quickly create and add particle systems to your games, how to integrate with Apple's Game Center for online leaderboards and achievements, and some performance tips and tricks to keep your game running fast.

  - **Chapter 14: Particle Systems: Creating Fire, Snow, Ice, and More**

    Using Cocos2D's particle system, you can add some amazing special effects to your game—extremely easily! In this chapter, you learn how to use particle systems to add some special effects to *Space Viking*, such as ship exhaust.

  - **Chapter 15: Achievements and Leaderboards with Game Center**

    With Apple's Game Center, you can easily add achievements and leaderboards to your games, which makes things more fun for players and also might help you sell more copies! This chapter covers how to set things up in *Space Viking*, step by step.

- **Chapter 16: Performance Optimizations**

  In this chapter, you learn how to tackle some of the most common challenges and issues you will face in optimizing and getting the most out of your Cocos2D game. You get hands-on experience debugging the most common performance issues and applying solutions.

- **Chapter 17: Conclusion**

  This final chapter recaps what you learned and describes where you can go next: into 3D, using Cocos2D on other platforms such as Android, and more advanced game-development topics.

- **Appendix: Principal Classes of Cocos2D**

  The Appendix provides an overview of the main classes you will be using and interacting with in Cocos2D.

By the time you've finished reading this book, you'll have practical experience making an awesome game from scratch! You can then take the concepts you've learned (and even some of the code!) and use it to turn your own game into a reality.

Audience for This Book

The audience for this book includes developers who are put off by game-making because they anticipate a long and complex learning curve. Many developers want to write games but don't know where to start with game development or the Cocos2D framework. This book is a hands-on guide, which takes you from the very beginning of using Cocos2D to applying the advanced physics concepts in Box2D and Chipmunk.

This book is targeted to developers interested in creating games for iOS devices, including the iPhone, iPad, and iPod touch. The book assumes a basic understanding of Objective-C, Cocoa Touch, and the Xcode tools. You are not expected to know any lower-level APIs (Core Audio, OpenGL ES, etc.), as these are used internally by Cocos2D.

Who This Book Is For

If you are already developing applications for the iPhone of other platform but want to make a move from utility applications to games, then this book is for you. It builds on the development knowledge you already have and leads you into game development by describing the terminology, technology, and tools required as well as providing real-world implementation examples.

Who This Book Isn't For

If you already have a grasp of the workflow required to create a game or you have a firm game idea that you know will require OpenGL ES for 3D graphics, then this is not the book for you.

It is expected that before you read this book you are already familiar with Objective-C, C, Xcode, and Interface Builder. While the implementations described in this book have been kept as simple as possible, and the use of C is limited, a firm foundation in these languages is required.

The following books can help provide you with the grounding you need to work through this book:

- *Cocoa Programming for Mac OS X, Third Edition,* by Aaron Hillegass (Addison-Wesley, 2008)
- *Learning Objective-C 2.0* by Robert Clair (Addison-Wesley, 2011)
- *Programming in Objective-C 2.0* by Stephen G. Kochan (Addison-Wesley, 2009)
- *Cocoa Design Patterns* by Erik M. Buck and Donald A. Yacktman (Addison-Wesley, 2009)
- *The iPhone Developer's Cookbook, Second Edition,* by Erica Sadun (Addison-Wesley, 2010)
- *Core Animation: Simplified Animation Techniques for Mac and iPhone Development* by Marcus Zarra and Matt Long (Addison-Wesley, 2010)
- *iPhone Programming: The Big Nerd Ranch Guide* by Aaron Hillegass and Joe Conway (Big Nerd Ranch, Inc., 2010)
- *Learning iOS Game Programming: A Hands-On Guide to Building Your First iPhone Game* by Michael Daley (Addison-Wesley, 2011)

These books, along with other resources you'll find on the web, will help you learn more about how to program for the Mac and iPhone, giving you a deeper knowledge about the Objective-C language and the Cocoa frameworks.

Source Code, Tutorial Videos, and Forums

Access to information is not limited only to the book. The complete, fully commented source code for *Space Viking* is also included, along with video tutorials (available at *http://cocos2Dbook.com*) that take you visually through the concepts of each chapter.

There is plenty of code to review throughout the book, along with exercises for you to try out, so it is assumed you have access to the Apple developer tools such as Xcode and the iPhone SDK. Both of these can be downloaded from the Apple iPhone Dev Center: *http://developer.apple.com/iphone*.

If you want to work with your fellow students as you work through the book, feel free to check out the book's forums at *http://cocos2dbook.com/forums/*.

This page intentionally left blank

Acknowledgments

This book would not have been possible without the hard work, support, and kindness of the following people:

- First of all, thanks to our editor, Chuck Toporek, and his assistant, Olivia Basegio. Chuck patiently helped and encouraged us during the entire process (even though we are both first-time authors!) and has managed all of the work it takes to convert a simple Word document into the actual book you're holding today. Olivia was extremely helpful through the entire process of keeping everyone coordinated and the tech reviews coming in. Thanks again to both of you in making this book a reality!
- Another person at Addison-Wesley whom we want to thank is Chuti Prasertsith, who designed the cover for the book.
- A huge thanks to the lead developer and coordinator of Cocos2D, Ricardo Quesada (also known as Riq), along with the other Cocos2D contributors, such as Steve Oldmeadow and many others. Without Riq and his team's hard work and dedication to making Cocos2D into the amazing framework and community that it is today, this book just wouldn't exist. Also, we believe that Cocos2D has made a huge positive difference in many people's lives by enabling them to accomplish a lifelong dream—to make their own games. Riq maintains Cocos2D as his full-time job, so if you'd like to make a donation to thank him for his hard work, you can do so at *www.cocos2d-iphone.org/store*. Riq also sells source code for his game *Sapus Tongue* and a great physics editor called Level-SVG. You can find out more about both at *www.sapusmedia.com*.
- Also, thank you to Erin Catto (the lead developer of Box2D) and Scott Lembcke (the lead developer of Chipmunk) for their work on their amazing physics libraries. Similarly to Riq's work on Cocos2D, Erin's and Scott's work has enabled countless programmers to create cool physics-based games quickly and easily. Erin and Scott are extremely dedicated to supporting their libraries and community, and even kindly donated their time in reviewing the physics chapters of this book. If you'd like to donate to Erin or Scott for their hard work on their libraries, you can do so by following the links at *www.box2d.org* and *http://code.google.com/p/chipmunk-physics*.
- A big thanks to Steve Oldmeadow, the lead developer of CocosDenshion, the sound engine behind Cocos2D. Steve provided assistance and time in reviewing

the chapter on audio. Steve's work has allowed many game developers to quickly and easily add music and sound effects to their games.

- Eric Stevens is an American fine artist who moonlights as a game illustrator. Years of good times and bad music contributed to the initial concept of *Space Viking*. Eric worked closely with us to bring Ole and everything you see in *Space Viking* to life. Eric maintains an illustration site at *http://imagedesk.org*, and you can see his paintings at several galleries in the Southwest and at *http://ericstevensart.com*.
- Mike Weiser is the musician who made the rocking soundtrack and sound effects for *Space Viking*. We think the music made a huge difference in *Space Viking* and really set the tone we were hoping for. A special thanks to Andrew Peplinski for the Viking grunts and Rulon Brown for conducting the choir that you hear in the beginning of the game. Mike has made music for lots of popular iOS games, and you can check him out at *www.mikeweisermusic.com*.
- A huge thanks to our technical reviewers: Farim Farook, Marc Hebert, Mark Hurley, Mike Leonardi, and Nick Waynik. These guys did a great job catching all of our boneheaded mistakes and giving us some great advice on how to make each chapter the best it could be. Thank you so much, guys!

Each of us also has some personal "thank yous" to make.

From Rod Strougo

I thank my wife and family for being ever patient while I was working on this book. There were countless evenings when I was hidden away in my office writing, editing, coding. Without Agata's support and understanding, there is no way this book could exist. Our older son, Alexander, was two and a half during the writing of this book, and he helped beta test *Space Viking*, while Anton was born as I was finishing the last chapters. Thank you for all the encouragement, love, and support, Agata.

I would also like to thank Ray for stepping in and writing the Box2D, Chipmunk, and Game Center chapters. Ray did a fantastic job on in-depth coverage of Box2D and Chipmunk, while adding some fun levels to *Space Viking*.

From Ray Wenderlich

First of all, a huge thank you to my wife and best friend, Vicki Wenderlich, for her constant support, encouragement, and advice throughout this entire process. Without her, I wouldn't be making iOS apps today, and they definitely wouldn't look as good! Also, thank you to my amazing family. You believed in me through the ups and downs of being an indie iOS developer and supported me the entire way. Thank you so much!

Finally, I thank all of the readers and supporters of my iOS tutorial blog at *www.raywenderlich.com*. Without your interest, encouragement, and support, I wouldn't have been as motivated to keep writing all the tutorials and might have never had the opportunity to write this book. Thank you so much for making this possible, and I hope you enjoy this book!

This page intentionally left blank

About the Authors

Rod Strougo is the founder and lead developer of the studio Prop Group at *www.prop.gr*. Rod's journey in physics and games started way back with an Apple][, writing games in Basic. From the early passion in games, Rod's career moved to enterprise software development, spending 10 years writing software for IBM and recently for a large telecom company. These days Rod enjoys helping others get started on their paths to making games. Originally from Rio de Janeiro, Brazil, Rod lives in Atlanta, Georgia, with his wife and sons.

Ray Wenderlich is an iPhone developer and gamer and the founder of Razeware, LLC. Ray is passionate about both making apps and teaching others the techniques to make them. He has written a bunch of tutorials about iOS development, available at *www.raywenderlich.com*.

This page intentionally left blank

4

Simple Collision Detection and the First Enemy

In the previous chapter you learned the basics of Cocos2D animations and actions. You also started building a flexible framework for Space Viking. *In this chapter you go further and create the first enemy for Ole to do battle with. In the process you learn how to implement a simple system for collision detection and the artificial intelligence brain of the enemies in* Space Viking.

There is a significant amount of code necessary in this chapter to drive the behavior of Ole and the RadarDish. Take your time understanding how these classes work, as they are the foundation and models for the rest of the classes in Space Viking.

Ready to defeat the aliens?

Creating the Radar Dish and Viking Classes

From just a `CCSprite` to a fully animated character, Ole the Viking takes the plunge from simple to advanced from here on out. In this section you create the `RadarDish` and `Viking` classes to encapsulate the logic needed by each, including all of the animations. The `RadarDish` class is worth a close look, as all of the enemy characters in *Space Viking* are modeled after it.

Creating the RadarDish Class

In this first scene, there is a suspicious radar dish on the right side of the screen. It scans for foreign creatures such as Ole. Ole needs to find a way to destroy the radar dish before it alerts the enemy robots of his presence. Fortunately, Ole knows two ways to deal with such problems: his left and right fists. Create the new `RadarDish` class in Xcode by following these steps:

1. In Xcode, right-click on the *EnemyObjects* group.
2. Select **New File**, choose the **Cocoa Touch category** under iOS and **Objective-C class** as the file type, and click **Next.**

3. For the Subclass field, enter *GameCharacter* and click **Next**.
4. Enter *RadarDish* for the filename and click **Finish**.

Open the *RadarDish.h* header file and change the contents to match the code in Listing 4.1.

Listing 4.1 **RadarDish.h header file**

```
//  RadarDish.h
//  SpaceViking
//
#import <Foundation/Foundation.h>
#import "GameCharacter.h"

@interface RadarDish : GameCharacter {
    CCAnimation *tiltingAnim;
    CCAnimation *transmittingAnim;
    CCAnimation *takingAHitAnim;
    CCAnimation *blowingUpAnim;
    GameCharacter *vikingCharacter;

}
@property (nonatomic, retain) CCAnimation *tiltingAnim;
@property (nonatomic, retain) CCAnimation *transmittingAnim;
@property (nonatomic, retain) CCAnimation *takingAHitAnim;
@property (nonatomic, retain) CCAnimation *blowingUpAnim;

@end
```

Looking at Listing 4.1 you can see that the `RadarDish` class inherits from the `GameCharacter` class and that it defines four `CCAnimation` instance variables. There is also an instance variable to hold a pointer back to the `Viking` character.

Why the vikingCharacter Variable Is of Type GameCharacter and Not of Type Viking Class

If you look carefully at Listing 4.1, you will notice that the `vikingCharacter` instance variable is of type `GameCharacter` and not of type `Viking`. This is because the `RadarDish` class needs access only to the methods defined in `GameCharacter` and not to the full `Viking` class.

Having an instance variable of type `GameCharacter` here allows for the `RadarDish` class to not have to know anything further about the `Viking` object except that it is a `GameCharacter`. You are free to add features to the `Viking` class without fear that it will break any functionality in `RadarDish`. If you were to change the main character in a future version of *Space Viking*, the code would still

function fine, since that new main character class too would, presumably, be derived from the `GameCharacter` class.

Listings 4.2, 4.3, and 4.4 show the contents of the *RadarDish.m* implementation file. The `changeState` and `updateStateWithDelta` time methods are crucial to understand, as they are the most basic versions of what you will find in all of the characters in *Space Viking*. While reading this code, keep in mind that the `RadarDish` is a simple enemy that never moves or attacks the `Viking`. The `RadarDish` does take damage from the `Viking`, eventually blowing up by moving to a dead state. Listing 4.2 covers the top portion of the *RadarDish.m* implementation file, including the `changeState` method. Open the *RadarDish.m* implementation file and replace the code so that it matches the contents in Listings 4.2, 4.3, and 4.4.

Listing 4.2 **RadarDish.m implementation file (top portion)**

```
//  RadarDish.m
//  SpaceViking
#import "RadarDish.h"

@implementation RadarDish
@synthesize tiltingAnim;
@synthesize transmittingAnim;
@synthesize takingAHitAnim;
@synthesize blowingUpAnim;

- (void) dealloc{
    [tiltingAnim release];
    [transmittingAnim release];
    [takingAHitAnim release];
    [blowingUpAnim release];
    [super dealloc];
}

-(void)changeState:(CharacterStates)newState {
    [self stopAllActions];
    id action = nil;
    [self setCharacterState:newState];

    switch (newState) {
        case kStateSpawning:
            CCLOG(@"RadarDish->Starting the Spawning Animation");
            action = [CCAnimate actionWithAnimation:tiltingAnim
                                restoreOriginalFrame:NO];
            break;
```

```
        case kStateIdle:
            CCLOG(@"RadarDish->Changing State to Idle");
            action = [CCAnimate actionWithAnimation:transmittingAnim
                                restoreOriginalFrame:NO];
            break;

        case kStateTakingDamage:
            CCLOG(@"RadarDish->Changing State to TakingDamage");
            characterHealth =
             characterHealth - [vikingCharacter getWeaponDamage];
            if (characterHealth <= 0.0f) {
                [self changeState:kStateDead];
            } else {
                action = [CCAnimate actionWithAnimation:takingAHitAnim
                                    restoreOriginalFrame:NO];
            }
            break;

        case kStateDead:
            CCLOG(@"RadarDish->Changing State to Dead");
            action = [CCAnimate actionWithAnimation:blowingUpAnim
                                restoreOriginalFrame:NO];
            break;

        default:
            CCLOG(@"Unhandled state %d in RadarDish", newState);
            break;
    }
    if (action != nil) {
        [self runAction:action];
    }
}
```

The `changeState` method is called when the `RadarDish` needs to transition between states. In the beginning of this chapter you were introduced to state machines, and the `changeState` method is what allows for transitions to different states in the miniscule "brain" of the `RadarDish`. The `RadarDish` brain can exist in one of four states: spawning, idle, taking damage, or dead. In the listings that follow, you will see that the `RadarDish` is initialized in the spawning state when it is created, and then through the `updateStateWithDeltaTime` method it will move through the four states.

When the `updateStateWithDeltaTime` determines that the `RadarDish` needs to change its state, the `changeState` method is called. Looking at Listing 4.2, you can recap what the switch state is doing as follows:

- **Spawning** (`kStateSpawning`)

  Starts up the `RadarDish` with the tilting animation, which is the dish moving up and down.
- **Idle** (`kStateIdle`)

  Runs the transmitting animation, which is the `RadarDish` blinking.
- **Taking Damage** (`kStateTakingDamage`)

  Runs the taking damage animation, showing a hit to the `RadarDish`. The `RadarDish` health is reduced according to the type of weapon being used against it.
- **Dead** (`kStateDead`)

  The `RadarDish` plays a death animation of it blowing up. This state occurs once the `RadarDish` health is at or below zero.

The next section of the `RadarDish` implementation file is covered in Listing 4.3, showing the `updateStateWithDeltaTime` method.

Listing 4.3 **RadarDish.m implementation file (middle portion)**

```
-(void)updateStateWithDeltaTime:(ccTime)deltaTime
andListOfGameObjects:(CCArray*)listOfGameObjects {
    if (characterState == kStateDead)
        return;                                                    // 1

    vikingCharacter =
    (GameCharacter*)[[self parent]
      getChildByTag:kVikingSpriteTagValue];                        // 2


    CGRect vikingBoudingBox =
            [vikingCharacter adjustedBoundingBox];                 // 3
    CharacterStates vikingState = [vikingCharacter
                                   characterState];                // 4

    // Calculate if the Viking is attacking and nearby
    if ((vikingState == kStateAttacking) &&
        (CGRectIntersectsRect([self adjustedBoundingBox],
vikingBoudingBox))) {                                              // 5
        if (characterState != kStateTakingDamage) {
            // If RadarDish is NOT already taking Damage
            [self changeState:kStateTakingDamage];
            return;
        }
    }
```

```
    if (([self numberOfRunningActions] == 0) &&
        (characterState != kStateDead)) {
        CCLOG(@"Going to Idle");
        [self changeState:kStateIdle];                          // 6
        return;
    }
}
```

Now let's examine the numbered lines of the code:

1. Checks if the `RadarDish` is already dead. If it is, this method is short-circuited and returned. If the `RadarDish` is dead, there is nothing to update.
2. Gets the `Viking` character object from the `RadarDish` parent. All of *Space Viking*'s objects are children of the scene `SpriteBatchNode`, referred to here as the `parent`. The `Viking` in particular was added to the `SpriteBatchNode` with a particular `tag`, referred to by the constant `kVikingSpriteTagValue`. By obtaining a reference to the `Viking` object, the `RadarDish` can determine if the `Viking` is nearby and attacking the `RadarDish`. (Listing 4.3 contains the code that sets up the `kVikingSpriteTagValue` constant.)
3. Gets the `Viking` character's adjusted bounding box.
4. Gets the `Viking` character's state.
5. Determines if the `Viking` is nearby and attacking. If the adjusted bounding boxes for the `Viking` and the `RadarDish` overlap, and the `Viking` is in his attack phase, the `RadarDish` can be certain that the `Viking` is attacking it. The call to `changeState:kStateTakingDamage` will alter the `RadarDish` animation to reflect the attack and reduce the `RadarDish` character's health.
6. Resets the transmission animation on the `RadarDish`. If the `RadarDish` is not currently playing an animation, and it is not dead, it is reset to idle so that the transmission animation can restart.

The last part of the *RadarDish.m* implementation file is the longest but least complicated. There is an `initAnimations` method, which sets up all of the `RadarDish` animations, and an `init` method that initializes the `RadarDish` and sets up the starting values for the instance variables. Add the contents of Listing 4.4 to your *RadarDish.m* implementation file.

Listing 4.4 **RadarDish.m implementation file (bottom portion)**

```
-(void)initAnimations {
    [self setTiltingAnim:
     [self loadPlistForAnimationWithName:@"tiltingAnim"
      andClassName:NSStringFromClass([self class])]];
```

```
    [self setTransmittingAnim:
     [self loadPlistForAnimationWithName:@"transmittingAnim"
      andClassName:NSStringFromClass([self class])]];

    [self setTakingAHitAnim:
     [self loadPlistForAnimationWithName:@"takingAHitAnim"
      andClassName:NSStringFromClass([self class])]];

    [self setBlowingUpAnim:
     [self loadPlistForAnimationWithName:@"blowingUpAnim"
      andClassName:NSStringFromClass([self class])]];
}
-(id) init {
    if( (self=[super init]) ) {
        CCLOG(@"### RadarDish initialized");
        [self initAnimations];                                  // 1
        characterHealth = 100.0f;                               // 2
        gameObjectType = kEnemyTypeRadarDish;                   // 3
        [self changeState:kStateSpawning];                      // 4
    }
    return self;
}
@end
```

The `initAnimations` method calls the `loadPlistForAnimationWithName` method you declared in the `GameObject` class. The name of the animation to load is passed along with the class name. Note the convenience method `NSStringFromClass` is used to get an `NSString` from the class name, in this case `RadarDish`. The class name is used to find the correct plist file for the object, since the plist files have a name corresponding to the class. The following occurs in the `init` method:

1. Calls the `initAnimations` method, which sets up all of the animations for the `RadarDish`. The frame's coordinates and textures were already loaded and cached by Cocos2D when the texture atlas files (*scene1atlas.png* and *scene1atlas.plist*) were loaded by the `GameplayLayer` class.
2. Sets the initial health of the `RadarDish` to a value of 100.
3. Sets the `RadarDish` to be a `Game Object` of type `kEnemyTypeRadarDish`.
4. Initializes the state of the `RadarDish` to spawning. Looking back at Listing 4.2, you can see that this starts the tilting animation, which is followed by the transmitting animation when the `RadarDish` moves from spawning to an idle state.

There is a little more work left before you can have this chapter's game running on your device. You need to add the `Viking` class and make some changes to the `GameplayLayer` class. It is important to understand how the `updateStateWithDeltaTime` and the `changeState` methods in `RadarDish` control the state of the

AI brain. These same two methods are used to drive the brain of all of the other game characters, including Ole the Viking.

Creating the Viking Class

In the previous chapter, Ole the Viking was nothing more than a `CCSprite`. In this chapter you pull him out into his own class complete with animations and a state machine to transition him through his various states. If the `Viking` class code starts to look daunting, refer back to the `RadarDish` class: the Viking is simply a game character like the `RadarDish`, albeit with more functionality. Create the new `Viking` class in Xcode by:

1. In Xcode, right-click on the *GameObjects* group.
2. Select **Add > New File**, choose the **Cocoa Touch category** under iOS and **Objective-C class** as the file type, and click **Next.**
3. For the Subclass field, enter *GameCharacter* and click **Next.**
4. Enter *Viking* for the filename and click **Save**.

Open the *Viking.h* header file and change the contents to match the code in Listing 4.5.

Listing 4.5 **Viking.h header file**

```
//  Viking.h
//  SpaceViking
#import <Foundation/Foundation.h>
#import "GameCharacter.h"
#import "SneakyButton.h"
#import "SneakyJoystick.h"
typedef enum {
    kLeftHook,
    kRightHook
} LastPunchType;

@interface Viking : GameCharacter  {
    LastPunchType myLastPunch;
    BOOL isCarryingMallet;
    CCSpriteFrame *standingFrame;

    // Standing, breathing, and walking
    CCAnimation *breathingAnim;
    CCAnimation *breathingMalletAnim;
    CCAnimation *walkingAnim;
    CCAnimation *walkingMalletAnim;
```

```
    // Crouching, standing up, and Jumping
    CCAnimation *crouchingAnim;
    CCAnimation *crouchingMalletAnim;
    CCAnimation *standingUpAnim;
    CCAnimation *standingUpMalletAnim;
    CCAnimation *jumpingAnim;
    CCAnimation *jumpingMalletAnim;
    CCAnimation *afterJumpingAnim;
    CCAnimation *afterJumpingMalletAnim;

    // Punching
    CCAnimation *rightPunchAnim;
    CCAnimation *leftPunchAnim;
    CCAnimation *malletPunchAnim;

    // Taking Damage and Death
    CCAnimation *phaserShockAnim;
    CCAnimation *deathAnim;

    SneakyJoystick *joystick;
    SneakyButton *jumpButton ;
    SneakyButton *attackButton;

    float millisecondsStayingIdle;
}
// Standing, Breathing, Walking
@property (nonatomic, retain) CCAnimation *breathingAnim;
@property (nonatomic, retain) CCAnimation *breathingMalletAnim;
@property (nonatomic, retain) CCAnimation *walkingAnim;
@property (nonatomic, retain) CCAnimation *walkingMalletAnim;

// Crouching, Standing Up, Jumping
@property (nonatomic, retain) CCAnimation *crouchingAnim;
@property (nonatomic, retain) CCAnimation *crouchingMalletAnim;
@property (nonatomic, retain) CCAnimation *standingUpAnim;
@property (nonatomic, retain) CCAnimation *standingUpMalletAnim;
@property (nonatomic, retain) CCAnimation *jumpingAnim;
@property (nonatomic, retain) CCAnimation *jumpingMalletAnim;
@property (nonatomic, retain) CCAnimation *afterJumpingAnim;
@property (nonatomic, retain) CCAnimation *afterJumpingMalletAnim;

// Punching
@property (nonatomic, retain) CCAnimation *rightPunchAnim;
@property (nonatomic, retain) CCAnimation *leftPunchAnim;
@property (nonatomic, retain) CCAnimation *malletPunchAnim;
```

```
// Taking Damage and Death
@property (nonatomic, retain) CCAnimation *phaserShockAnim;
@property (nonatomic, retain) CCAnimation *deathAnim;

@property (nonatomic,assign) SneakyJoystick *joystick;
@property (nonatomic,assign) SneakyButton *jumpButton;
@property (nonatomic,assign) SneakyButton *attackButton;
@end
```

Listing 4.5 shows the large number of animations that are possible with the `Viking` character as well as instance variables to point to the onscreen joystick and button controls.

The key items to note are the `typedef enumerator` for the left and right punches, an instance variable to store what the last punch thrown was, and a `float` to keep track of how long the player has been idle. The code for the `Viking` implementation file is a bit on the lengthy side, hence it is broken up into four Listings, 4.6 through 4.9. Open the *Viking.m* implementation file and replace the code so that it matches the contents in Listings 4.6, 4.7, 4.8, and 4.9.

Listing 4.6 **Viking.m implementation file (part 1 of 4)**

```
//  Viking.m
//  SpaceViking
#import "Viking.h"

@implementation Viking
@synthesize joystick;
@synthesize jumpButton ;
@synthesize attackButton;

// Standing, Breathing, Walking
@synthesize breathingAnim;
@synthesize breathingMalletAnim;
@synthesize walkingAnim;
@synthesize walkingMalletAnim;
// Crouching, Standing Up, Jumping
@synthesize crouchingAnim;
@synthesize crouchingMalletAnim;
@synthesize standingUpAnim;
@synthesize standingUpMalletAnim;
@synthesize jumpingAnim;
@synthesize jumpingMalletAnim;
@synthesize afterJumpingAnim;
@synthesize afterJumpingMalletAnim;
// Punching
@synthesize rightPunchAnim;
```

```
@synthesize leftPunchAnim;
@synthesize malletPunchAnim;
// Taking Damage and Death
@synthesize phaserShockAnim;
@synthesize deathAnim;

- (void) dealloc {
    joystick = nil;
    jumpButton = nil;
    attackButton = nil;
    [breathingAnim release];
    [breathingMalletAnim release];
    [walkingAnim release];
    [walkingMalletAnim release];
    [crouchingAnim release];
    [crouchingMalletAnim release];
    [standingUpAnim release];
    [standingUpMalletAnim release];
    [jumpingAnim release];
    [jumpingMalletAnim release];
    [afterJumpingAnim release];
    [afterJumpingMalletAnim release];
    [rightPunchAnim release];
    [leftPunchAnim release];
    [malletPunchAnim release];
    [phaserShockAnim release];
    [deathAnim release];

    [super dealloc];
}

-(BOOL)isCarryingWeapon {
    return isCarryingMallet;
}

-(int)getWeaponDamage {
    if (isCarryingMallet) {
        return kVikingMalletDamage;
    }
    return kVikingFistDamage;
}
-(void)applyJoystick:(SneakyJoystick *)aJoystick forTimeDelta:(float)
deltaTime
{
    CGPoint scaledVelocity = ccpMult(aJoystick.velocity, 128.0f);
    CGPoint oldPosition = [self position];
    CGPoint newPosition =
```

```
    ccp(oldPosition.x +
        scaledVelocity.x * deltaTime,
        oldPosition.y);                                          // 1
    [self setPosition:newPosition];                              // 2

    if (oldPosition.x > newPosition.x) {
        self.flipX = YES;                                        // 3
    } else {
        self.flipX = NO;
    }
}

-(void)checkAndClampSpritePosition {
    if (self.characterState != kStateJumping) {
        if ([self position].y > 110.0f)
            [self setPosition:ccp([self position].x,110.0f)];
    }
    [super checkAndClampSpritePosition];
}
```

At the beginning of the *Viking.m* implementation file is the `dealloc` method. Far wiser Objective-C developers than this author have commented on the benefits of having your `dealloc` method up top and near your `synthesize` statements. The idea behind this move is to make sure you are deallocating any and all instance variables, therefore avoiding one of the main causes of memory leaks in Objective-C code.

Following the `dealloc` method, you have the `isCarryingWeapon` method, but since it is self-explanatory, move on to the `applyJoystick` method. This method is similar to the one back in Chapter 2, "Hello, Space Viking," Listing 2.10, but it has been modified to deal only with Ole's movement and removes the handling for the jump or attack buttons. The first change to `applyJoystick` is the creation of the `oldPosition` variable to track the `Viking`'s position before it is moved. Looking at the `applyJoystick` method in Listing 4.6, take a note of the following key lines:

1. Sets the new position based on the velocity of the joystick, but only in the x-axis. The `y` position stays constant, making it so Ole only walks to the left or right, and not up or down.
2. Moves the `Viking` to the new position.
3. Compares the old position with the new position, flipping the `Viking` horizontally if needed. If you look closely at the `Viking` images, he is facing to the right by default. If this method determines that the old position is to the right of the new position, Ole is moving to the left, and his pixels have to be flipped horizontally. If you don't flip Ole horizontally, he will look like he is trying to do the moonwalk when you move him to the left. It is a cool effect but not useful for your `Viking`.

Cocos2D has two built-in functions you will make use of frequently: `flipX` and `flipY`. These functions flip the pixels of a texture along the x- or y-axis, allowing you to display a mirror image of your graphics without having to have left- and right-facing copies of each image for each character. Figure 4.1 shows the effect of `flipX` on the `Viking` texture. This is a really handy feature to have, since it helps reduce the size of your application, and it keeps you from having to create images for every possible state.

Figure 4.1 Effects of the flipX function on the Viking texture or graphic

The next section of the *Viking.m* implementation file covers the `changeState` method. As you learned with the `RadarDish` class, the `changeState` method is used to transition the character from one state to another and to start the appropriate animations for each state. Copy the contents of Listing 4.7 into your *Viking.m* class.

Listing 4.7 **Viking.m implementation file (part 2 of 4)**

```
#pragma mark -
-(void)changeState:(CharacterStates)newState {
    [self stopAllActions];
    id action = nil;
    id movementAction = nil;
    CGPoint newPosition;
    [self setCharacterState:newState];

    switch (newState) {
        case kStateIdle:
            if (isCarryingMallet) {
                [self setDisplayFrame:[[CCSpriteFrameCache
                      sharedSpriteFrameCache]
                      spriteFrameByName:@"sv_mallet_1.png"]];
            } else {
                [self setDisplayFrame:[[CCSpriteFrameCache
                      sharedSpriteFrameCache]
                      spriteFrameByName:@"sv_anim_1.png"]];
            }
            break;
```

```
case kStateWalking:
    if (isCarryingMallet) {
        action =
            [CCAnimate actionWithAnimation:walkingMalletAnim
                       restoreOriginalFrame:NO];
    } else {
        action =
            [CCAnimate actionWithAnimation:walkingAnim
                       restoreOriginalFrame:NO];
    }
    break;


case kStateCrouching:
    if (isCarryingMallet) {
        action =
            [CCAnimate actionWithAnimation:crouchingMalletAnim
                       restoreOriginalFrame:NO];
    } else {
        action =
            [CCAnimate actionWithAnimation:crouchingAnim
                       restoreOriginalFrame:NO];
    }
    break;

case kStateStandingUp:
    if (isCarryingMallet) {
        action =
            [CCAnimate actionWithAnimation:standingUpMalletAnim
                       restoreOriginalFrame:NO];

    } else {
        action =
            [CCAnimate actionWithAnimation:standingUpAnim
                       restoreOriginalFrame:NO];
    }
    break;

case kStateBreathing:
    if (isCarryingMallet) {
        action =
            [CCAnimate actionWithAnimation:breathingMalletAnim
                       restoreOriginalFrame:YES];
    } else {
        action =
            [CCAnimate actionWithAnimation:breathingAnim
                       restoreOriginalFrame:YES];
    }
    break;
```

```
case kStateJumping:
    newPosition = ccp(screenSize.width * 0.2f, 0.0f);
    if ([self flipX] == YES) {
        newPosition = ccp(newPosition.x * -1.0f, 0.0f);
    }
    movementAction = [CCJumpBy actionWithDuration:0.5f
                                         position:newPosition
                                           height:160.0f
                                            jumps:1];

    if (isCarryingMallet) {
        // Viking Jumping animation with the Mallet
        action = [CCSequence actions:
                  [CCAnimate
                      actionWithAnimation:crouchingMalletAnim
                      restoreOriginalFrame:NO],
                  [CCSpawn actions:
                   [CCAnimate
                      actionWithAnimation:jumpingMalletAnim
                      restoreOriginalFrame:YES],
                   movementAction,
                   nil],
                  [CCAnimate
                     actionWithAnimation:afterJumpingMalletAnim
                     restoreOriginalFrame:NO],
                  nil];
    } else {
        // Viking Jumping animation without the Mallet
        action = [CCSequence actions:
                  [CCAnimate
                      actionWithAnimation:crouchingAnim
                      restoreOriginalFrame:NO],
                  [CCSpawn actions:
                   [CCAnimate
                       actionWithAnimation:jumpingAnim
                       restoreOriginalFrame:YES],
                   movementAction,
                   nil],
                  [CCAnimate
                      actionWithAnimation:afterJumpingAnim
                      restoreOriginalFrame:NO],
                  nil];
    }
    break;

case kStateAttacking:
    if (isCarryingMallet == YES) {
```

```
                action = [CCAnimate
                            actionWithAnimation:malletPunchAnim
                            restoreOriginalFrame:YES];
            } else {
                if (kLeftHook == myLastPunch) {
                    // Execute a right hook
                    myLastPunch = kRightHook;
                    action = [CCAnimate
                                actionWithAnimation:rightPunchAnim
                                restoreOriginalFrame:NO];
                } else {
                    // Execute a left hook
                    myLastPunch = kLeftHook;
                    action = [CCAnimate
                                actionWithAnimation:leftPunchAnim
                                restoreOriginalFrame:NO];
                }
            }
            break;

        case kStateTakingDamage:
            self.characterHealth = self.characterHealth - 10.0f;
            action = [CCAnimate
                        actionWithAnimation:phaserShockAnim
                        restoreOriginalFrame:YES];
            break;

        case kStateDead:
            action = [CCAnimate
                        actionWithAnimation:deathAnim
                        restoreOriginalFrame:NO];
            break;

        default:
            break;
    }
    if (action != nil) {
        [self runAction:action];
    }
}
```

The first part of the `changeState` method stops any running actions, including animations. Any running actions would be a part of a previous state of the `Viking` and would no longer be valid. Following the first line, the `Viking` state is set to the new state value, and a `switch` statement is used to carry out the animations for the new state. A few items are important to note:

1. Method variables cannot be declared inside a `switch` statement, as they would be out of scope as soon as the code exited the `switch` statement. Your `id action` variable is declared above the `switch` statement but initialized inside the `switch` branches.
2. Most of the states have two animations: one for the `Viking` with the `Mallet` and one without. The `isCarryingMallet` Boolean instance variable is key in determining which animation to play.
3. An action in Cocos2D can be made up of other actions in that it can be a compound action. The `switch` branch taken when the `Viking` state is `kStateJumping` has a compound action made up of `CCSequence`, `CCAnimate`, `CCSpawn`, and `CCJumpBy` actions. The `CCJumpBy` action provides the parabolic movement for Ole the Viking, while the `CCAnimate` actions play the crouching, jumping, and landing animations. The `CCSpawn` action allows for more than one action to be started at the same time, in this case the `CCJumpBy` and `CCAnimate` animation action of Ole jumping. The `CCSequence` action ties it all together by making Ole crouch down, then jump, and finally land on his feet in sequence.
4. Taking a closer look at the `kStateTakingDamage switch` branch, you can see that after the animation completes, Ole reverts back to the frame that was displaying before the animation started. In this state transition, the `CCAnimate` action has the `restoreOriginalFrame` set to `YES`. The end effect of `restoreOriginalFrame` is that Ole will animate receiving a hit, and then return to looking as he did before the hit took place.

The first line of Listing 4.7 might be rather odd-looking: `#pragma mark`. The `pragma mark` serves as a formatting guide to Xcode and is not seen by the compiler. After the words `#pragma mark` you can place any text you would like displayed in the Xcode pulldown for this file. If you have just a hyphen (-), Xcode will create a separate section for that portion of the file. Using `pragma mark` can make your code easier to navigate. Figure 4.2 shows the effects of the `pragma mark` statements in the completed *Viking.m* file.

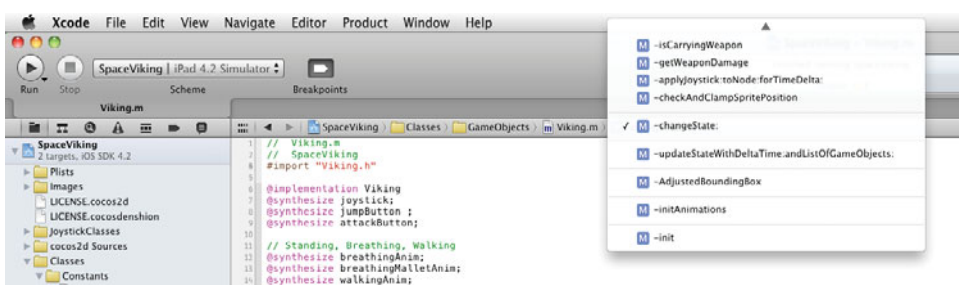

Figure 4.2 The effect of the pragma mark statements in the Xcode pulldown menus

The next section of the *Viking.m* file covers the `updateStateWithDeltaTime` and the `adjustedBoundingBox` methods. Copy the contents of Listing 4.8 into your *Viking.m* file immediately following the `changeState` method.

Listing 4.8 **Viking.m implementation file (part 3 of 4)**

```
#pragma mark -
-(void)updateStateWithDeltaTime:(ccTime)deltaTime
andListOfGameObjects:(CCArray*)listOfGameObjects {
    if (self.characterState == kStateDead)
        return; // Nothing to do if the Viking is dead

    if ((self.characterState == kStateTakingDamage) &&
       ([self numberOfRunningActions] > 0))
        return; // Currently playing the taking damage animation

    // Check for collisions
    // Change this to keep the object count from querying it each time
    CGRect myBoundingBox = [self adjustedBoundingBox];
    for (GameCharacter *character in listOfGameObjects) {
        // This is Ole the Viking himself
        // No need to check collision with one's self
        if ([character tag] == kVikingSpriteTagValue)
            continue;

        CGRect characterBox = [character adjustedBoundingBox];
        if (CGRectIntersectsRect(myBoundingBox, characterBox)) {
            // Remove the PhaserBullet from the scene
            if ([character gameObjectType] == kEnemyTypePhaser) {
                [self changeState:kStateTakingDamage];
                [character changeState:kStateDead];
            } else if ([character gameObjectType] ==
                       kPowerUpTypeMallet) {
                // Update the frame to indicate Viking is
                // carrying the mallet
                isCarryingMallet = YES;
                [self changeState:kStateIdle];
                // Remove the Mallet from the scene
                [character changeState:kStateDead];
            } else if ([character gameObjectType] ==
                        kPowerUpTypeHealth) {
                [self setCharacterHealth:100.0f];
                // Remove the health power-up from the scene
                [character changeState:kStateDead];
            }
        }
    }
```

```
    [self checkAndClampSpritePosition];
    if ((self.characterState == kStateIdle) ||
        (self.characterState == kStateWalking) ||
        (self.characterState == kStateCrouching) ||
        (self.characterState == kStateStandingUp) ||
        (self.characterState == kStateBreathing)) {

        if (jumpButton.active) {
            [self changeState:kStateJumping];
        } else if (attackButton.active) {
            [self changeState:kStateAttacking];
        } else if ((joystick.velocity.x == 0.0f) &&
                   (joystick.velocity.y == 0.0f)) {
            if (self.characterState == kStateCrouching)
                [self changeState:kStateStandingUp];
        } else if (joystick.velocity.y < -0.45f) {
            if (self.characterState != kStateCrouching)
                [self changeState:kStateCrouching];
        } else if (joystick.velocity.x != 0.0f) { // dpad moving
            if (self.characterState != kStateWalking)
                [self changeState:kStateWalking];
            [self applyJoystick:joystick
                  forTimeDelta:deltaTime];
        }
    }

    if ([self numberOfRunningActions] == 0) {
        // Not playing an animation
        if (self.characterHealth <= 0.0f) {
            [self changeState:kStateDead];
        } else if (self.characterState == kStateIdle) {
            millisecondsStayingIdle = millisecondsStayingIdle +
                                      deltaTime;
            if (millisecondsStayingIdle > kVikingIdleTimer) {
                [self changeState:kStateBreathing];
            }
        } else if ((self.characterState != kStateCrouching) &&
                   (self.characterState != kStateIdle)){
            millisecondsStayingIdle = 0.0f;
            [self changeState:kStateIdle];
        }
    }
}

#pragma mark -
-(CGRect)adjustedBoundingBox {
    // Adjust the bouding box to the size of the sprite
    // without the transparent space
```

```
    CGRect vikingBoundingBox = [self boundingBox];
    float xOffset;
    float xCropAmount = vikingBoundingBox.size.width * 0.5482f;
    float yCropAmount = vikingBoundingBox.size.height * 0.095f;

    if ([self flipX] == NO) {
        // Viking is facing to the rigth, back is on the left
        xOffset = vikingBoundingBox.size.width * 0.1566f;
    } else {
        // Viking is facing to the left; back is facing right
        xOffset = vikingBoundingBox.size.width * 0.4217f;
    }
    vikingBoundingBox =
    CGRectMake(vikingBoundingBox.origin.x + xOffset,
               vikingBoundingBox.origin.y,
               vikingBoundingBox.size.width - xCropAmount,
               vikingBoundingBox.size.height - yCropAmount);

    if (characterState == kStateCrouching) {
        // Shrink the bounding box to 56% of height
        // 88 pixels on top on iPad
        vikingBoundingBox = CGRectMake(vikingBoundingBox.origin.x,
        vikingBoundingBox.origin.y,
        vikingBoundingBox.size.width,
        vikingBoundingBox.size.height * 0.56f);
    }

    return vikingBoundingBox;
}
```

In the same manner as the `RadarDish` `updateStateWithDeltaMethod` worked, this method also returns immediately if the `Viking` is dead. There is no need to update a dead `Viking` because he won't be going anywhere.

If the `Viking` is in the middle of playing, the taking damage animation is played. This method again short-circuits and returns. The taking damage animation is blocking in that the player cannot do anything else while Ole the Viking is being shocked.

If the `Viking` is not taking damage or is dead, then the next step is to check what objects are coming in contact with the `Viking`. If there are objects in contact with the `Viking`, he checks to see if they are:

- Phaser: Changes the Viking state to taking damage.
- Mallet power-up: Gives Ole the Viking the mallet, a fearsome weapon.
- Health power-up: Ole's health is restored back to 100.

After checking for contacts, often called *collisions*, a quick call is made to the `checkAndClampSpritePosition` method to ensure that the `Viking` sprite stays within the boundaries of the screen.

The next `if` statement block checks the state of the joystick, jump, and attack buttons and changes the state of the `Viking` to reflect which controls are being pressed. The `if` statement executes only if the `Viking` is not currently carrying out a blocking animation, such as jumping.

Lastly the `Viking` class reaches a section of the `updateStateWithDeltaTime` method that handles what happens when there are no animations currently running. Cocos2D has a convenience method on `CCNodes` that reports back the number of actions running against a particular `CCNode` object. If you recall from the beginning of this chapter, all animations have to be run by a `CCAnimate` action. Once the animation for a state completes, the `numberOfRunningActions` will return zero for the `Viking`, and this block of code will reset the `Viking`'s state.

If the health is zero or less, the `Viking` will move into the dead state. Otherwise, if `Viking` is idle, a counter is incremented indicating how many seconds the player has been idle. Once that counter reaches a set limit, the `Viking` will play a heavy breathing animation. Finally, if the Viking is not already idle or crouching, he will move back into the idle state.

> **Note**
>
> The breathing animation is just a little bonus move to try to get the player to focus back on the game. If the joystick has been idle for more than 3 seconds, the `Viking` will let out a few deep breaths as if to say "Come on! I have aliens to fight here, let's get going!"

After the `updateStateWithDeltaTime` method, there is the `adjustedBoundingBox` method you declared inside the `GameObject` class. In Chapter 3, "Introduction to Cocos2D Animations and Actions," Figure 3.6 illustrated the transparent space in the `Viking` texture between the actual `Viking` and the edges of the image/texture. This method compensates for the transparent pixels by returning an adjusted bounding box that does not include the transparent pixels. The `flipX` parameter is used to determine which side the `Viking` is facing, as fewer pixels are trimmed off the back of the `Viking` image than the front.

The last part of the *Viking.m* implementation file sets up the animations inside the `initAnimations` method and the instance variables inside the `init` method. Once more, copy the contents of Listing 4.9 into your *Viking.m* implementation file immediately following the end of the `adjustedBoundingBox` method.

Listing 4.9 **Viking.m implementation file (part 4 of 4)**

```
#pragma mark -
-(void)initAnimations {

    [self setBreathingAnim:[self loadPlistForAnimationWithName:
@"breathingAnim" andClassName:NSStringFromClass([self class])]];

    [self setBreathingMalletAnim:[self loadPlistForAnimationWithName:
@"breathingMalletAnim" andClassName:NSStringFromClass([self class])]];
```

```
    [self setWalkingAnim:[self loadPlistForAnimationWithName:
@"walkingAnim" andClassName:NSStringFromClass([self class])]];

    [self setWalkingMalletAnim:[self loadPlistForAnimationWithName:
@"walkingMalletAnim" andClassName:NSStringFromClass([self class])]];

    [self setCrouchingAnim:[self loadPlistForAnimationWithName:
@"crouchingAnim" andClassName:NSStringFromClass([self class])]];

    [self setCrouchingMalletAnim:[self loadPlistForAnimationWithName:
@"crouchingMalletAnim" andClassName:NSStringFromClass([self class])]];

    [self setStandingUpAnim:[self loadPlistForAnimationWithName:
@"standingUpAnim" andClassName:NSStringFromClass([self class])]];

    [self setStandingUpMalletAnim:[self loadPlistForAnimationWithName:
@"standingUpMalletAnim" andClassName:NSStringFromClass([self class])]];

    [self setJumpingAnim:[self loadPlistForAnimationWithName:
@"jumpingAnim" andClassName:NSStringFromClass([self class])]];

    [self setJumpingMalletAnim:[self loadPlistForAnimationWithName:
@"jumpingMalletAnim" andClassName:NSStringFromClass([self class])]];

    [self setAfterJumpingAnim:[self loadPlistForAnimationWithName:
@"afterJumpingAnim" andClassName:NSStringFromClass([self class])]];

    [self setAfterJumpingMalletAnim:[self loadPlistForAnimationWithName:
@"afterJumpingMalletAnim" andClassName:NSStringFromClass([self class])]];

    // Punches
    [self setRightPunchAnim:[self loadPlistForAnimationWithName:
@"rightPunchAnim" andClassName:NSStringFromClass([self class])]];

    [self setLeftPunchAnim:[self loadPlistForAnimationWithName:
@"leftPunchAnim" andClassName:NSStringFromClass([self class])]];

    [self setMalletPunchAnim:[self loadPlistForAnimationWithName:
@"malletPunchAnim" andClassName:NSStringFromClass([self class])]];

    // Taking Damage and Death
    [self setPhaserShockAnim:[self loadPlistForAnimationWithName:
@"phaserShockAnim" andClassName:NSStringFromClass([self class])]];

    [self setDeathAnim:[self loadPlistForAnimationWithName:
@"vikingDeathAnim" andClassName:NSStringFromClass([self class])]];

}
```

```
#pragma mark -
-(id) init {
    if( (self=[super init]) ) {
        joystick = nil;
        jumpButton = nil;
        attackButton = nil;
        self.gameObjectType = kVikingType;
        myLastPunch = kRightHook;
        millisecondsStayingIdle = 0.0f;
        isCarryingMallet = NO;
        [self initAnimations];

    }
    return self;
}
@end
```

The `initAnimation` method, while quite long, is very basic in that it only initializes all of the `Viking` animations based on the display frames already loaded from the *scene1atlas.plist* file in the `GameplayLayer` class. The `init` method sets up the instance variables to their starting values.

Final Steps

The final step for this chapter is to make some changes to the `GameplayLayer` class so it loads the `RadarDish` and `Viking` onto the layer. Once these changes are made to the `GameplayLayer` files, you will have a working and playable version of *Space Viking* in your hands.

The GameplayLayer Class

The `GameplayLayer` class has a few changes to the header file. There is an additional `import` for the *CommonProtocols.h* file and the `vikingSprite` has been removed; instead there is a `CCSpriteBatchNode` called `sceneSpriteBatchNode`. Move your *GameplayLayer.h* and *GameplayLayer.m* files into the *Layers Group* folder in Xcode and ensure that your *GameplayLayer.h* header file has the same contents as Listing 4.10.

Listing 4.10 **GameplayLayer.h header file**

```
//  GameplayLayer.h
//  SpaceViking

#import <Foundation/Foundation.h>
#import "cocos2d.h"
#import "SneakyJoystick.h"
```

```
#import "SneakyButton.h"
#import "SneakyButtonSkinnedBase.h"
#import "SneakyJoystickSkinnedBase.h"
#import "Constants.h"
#import "CommonProtocols.h"
#import "RadarDish.h"
#import "Viking.h"

@interface GameplayLayer : CCLayer <GameplayLayerDelegate> {
    CCSprite *vikingSprite;
    SneakyJoystick *leftJoystick;
    SneakyButton *jumpButton;
    SneakyButton *attackButton;
    CCSpriteBatchNode *sceneSpriteBatchNode;
}

@end
```

The `initJoystickAndButtons` method of `GameplayLayer` stays the same as in Chapter 3. The rest of the `GameplayLayer` class requires changes to use the new `CCSpriteBatchNode` instance. Listings 4.11, 4.12, 4.13, and 4.14 cover the code for *GameplayLayer.m*. Replace the code in your *GameplayLayer.m* implementation file with the code in the next four listings.

Listing 4.11 **GameplayLayer.m implementation file (part 1 of 4)**

```
//  GameplayLayer.m
//  SpaceViking

#import "GameplayLayer.h"
@implementation GameplayLayer
- (void) dealloc {
    [leftJoystick release];
    [jumpButton release];
    [attackButton release];
    [super dealloc];
}

-(void)initJoystickAndButtons {
    CGSize screenSize = [CCDirector sharedDirector].winSize;        // 1
    // 2
    CGRect joystickBaseDimensions = CGRectMake(0, 0, 128.0f, 128.0f);
    CGRect jumpButtonDimensions = CGRectMake(0, 0, 64.0f, 64.0f);
    CGRect attackButtonDimensions = CGRectMake(0, 0, 64.0f, 64.0f);
    // 3
    CGPoint joystickBasePosition;
```

```
CGPoint jumpButtonPosition;
CGPoint attackButtonPosition;
// 4
if (UI_USER_INTERFACE_IDIOM() == UIUserInterfaceIdiomPad) {
    // The device is an iPad running iPhone 3.2 or later.
    CCLOG(@"Positioning Joystick and Buttons for iPad");
    joystickBasePosition = ccp(screenSize.width*0.0625f,
                               screenSize.height*0.052f);

    jumpButtonPosition = ccp(screenSize.width*0.946f,
                             screenSize.height*0.052f);

    attackButtonPosition = ccp(screenSize.width*0.947f,
                               screenSize.height*0.169f);
} else {
    // The device is an iPhone or iPod touch.
   CCLOG(@"Positioning Joystick and Buttons for iPhone");

    joystickBasePosition = ccp(screenSize.width*0.07f,
                               screenSize.height*0.11f);

    jumpButtonPosition = ccp(screenSize.width*0.93f,
                             screenSize.height*0.11f);

    attackButtonPosition = ccp(screenSize.width*0.93f,
                               screenSize.height*0.35f);
}

SneakyJoystickSkinnedBase *joystickBase =
[[[SneakyJoystickSkinnedBase alloc] init] autorelease];
joystickBase.position = joystickBasePosition;
joystickBase.backgroundSprite =
[CCSprite spriteWithFile:@"dpadDown.png"];
joystickBase.thumbSprite =
[CCSprite spriteWithFile:@"joystickDown.png"];
joystickBase.joystick = [[SneakyJoystick alloc]
                         initWithRect:joystickBaseDimensions];
leftJoystick = [joystickBase.joystick retain];
[self addChild:joystickBase];

SneakyButtonSkinnedBase *jumpButtonBase =
[[[SneakyButtonSkinnedBase alloc] init] autorelease];
jumpButtonBase.position = jumpButtonPosition;
jumpButtonBase.defaultSprite =
[CCSprite spriteWithFile:@"jumpUp.png"];
jumpButtonBase.activatedSprite =
[CCSprite spriteWithFile:@"jumpDown.png"];
```

```
    jumpButtonBase.pressSprite =
    [CCSprite spriteWithFile:@"jumpDown.png"];
    jumpButtonBase.button = [[SneakyButton alloc]
                             initWithRect:jumpButtonDimensions];
    jumpButton = [jumpButtonBase.button retain];
    jumpButton.isToggleable = NO;
    [self addChild:jumpButtonBase];

    SneakyButtonSkinnedBase *attackButtonBase = [[[SneakyButtonSkinnedBase
alloc] init] autorelease];
    attackButtonBase.position = attackButtonPosition;
    attackButtonBase.defaultSprite = [CCSprite spriteWithFile:
@"handUp.png"];
    attackButtonBase.activatedSprite = [CCSprite
spriteWithFile:@"handDown.png"];
    attackButtonBase.pressSprite = [CCSprite spriteWithFile:
@"handDown.png"];
    attackButtonBase.button = [[SneakyButton alloc] initWithRect:
attackButtonDimensions];
    attackButton = [attackButtonBase.button retain];
    attackButton.isToggleable = NO;
    [self addChild:attackButtonBase];
}
```

The `initJoystick` method remains unchanged from previous chapters. The directional pad (DPad) as well as the jump and attack buttons are set up and added to the `GameplayLayer`. The high `z` values ensure that the joystick controls appear on top of all the other graphical elements in the `GameplayLayer`.

Listing 4.12 **GameplayLayer.m implementation file (part 2 of 4)**

```
#pragma mark -
#pragma mark Update Method
-(void) update:(ccTime)deltaTime {
    CCArray *listOfGameObjects =
            [sceneSpriteBatchNode children];                    // 1
    for (GameCharacter *tempChar in listOfGameObjects) {        // 2
        [tempChar updateStateWithDeltaTime:deltaTime andListOfGameObjects:
                    listOfGameObjects];                         // 3
    }
}
```

The `update` method is the run loop for the entire `GameplayLayer`. The `CCSpriteBatchNode` object contains a list of all of the `CCSprites` for which it will handle the rendering, batching their OpenGL ES draw calls. The `update` method does the following:

1. Gets the list of all of the children CCSprites rendered by the CCSpriteBatchNode. In *Space Viking* this is a list of all of the GameCharacters, including the Viking and his enemies.
2. Iterates through each of the Game Characters, calls their updateStateWithDeltaTime method, and passes a pointer to the list of all Game Characters. If you look back at the updateStateWithDeltaTime code in *Viking.m*, you can see the list of Game Characters used to check for power-ups and phaser blasts. Power-ups and aliens with phaser beams are covered in the next chapter.
3. Calls the updateStateWithDeltaTime method on each of the Game Characters. This call allows for all of the characters to update their individual states to determine if they are colliding with any other objects in the game.

The next section of code in *GameplayLayer.m* (Listing 4.13) contains the methods for creating the enemies and a placeholder for creating the phaser blast.

Listing 4.13 **GameplayLayer.m implementation file (part 3 of 4)**

```
#pragma mark -
-(void)createObjectOfType:(GameObjectType)objectType
              withHealth:(int)initialHealth
              atLocation:(CGPoint)spawnLocation
              withZValue:(int)ZValue {

    if (objectType == kEnemyTypeRadarDish) {
        CCLOG(@"Creating the Radar Enemy");
        RadarDish *radarDish = [[RadarDish alloc] initWithSpriteFrameName:
@"radar_1.png"];
        [radarDish setCharacterHealth:initialHealth];
        [radarDish setPosition:spawnLocation];
        [sceneSpriteBatchNode addChild:radarDish
                               z:ZValue
                               tag:kRadarDishTagValue];
        [radarDish release];
    }

}

-(void)createPhaserWithDirection:(PhaserDirection)phaserDirection
andPosition:(CGPoint)spawnPosition {
    CCLOG(@"Placeholder for Chapter 5, see below");
    return;
}
```

The createObjectOfType method sets up the RadarDish object using the CCSpriteBatchNode and adds it to the layer. This method is expanded upon in

Chapter 5, "More Actions, Effects, and Cocos2D Scheduler," to include the other enemies in the *Space Viking* world.

The last code listing for *GameplayLayer.m* covers the `init` method. Copy the contents of Listing 4.14 into your *GameplayLayer.m* file.

Listing 4.14 **GameplayLayer.m implementation file (part 4 of 4)**

```
-(id)init {
    self = [super init];
    if (self != nil) {
        CGSize screenSize = [CCDirector sharedDirector].winSize;
        // enable touches
        self.isTouchEnabled = YES;

        srandom(time(NULL)); // Seeds the random number generator

    if (UI_USER_INTERFACE_IDIOM() == UIUserInterfaceIdiomPad) {
        [[CCSpriteFrameCache sharedSpriteFrameCache]
          addSpriteFramesWithFile:@"scene1atlas.plist"];           // 1
        sceneSpriteBatchNode =
         [CCSpriteBatchNode batchNodeWithFile:@"scene1atlas.png"]; // 2
    } else {
        [[CCSpriteFrameCache sharedSpriteFrameCache]
          addSpriteFramesWithFile:@"scene1atlasiPhone.plist"];     // 1
        sceneSpriteBatchNode =
         [CCSpriteBatchNode
          batchNodeWithFile:@"scene1atlasiPhone.png"];             // 2
    }
        [self addChild:sceneSpriteBatchNode z:0];                  // 3
        [self initJoystickAndButtons];                             // 4
        Viking *viking = [[Viking alloc]
            initWithSpriteFrame:[[CCSpriteFrameCache
                sharedSpriteFrameCache]
                spriteFrameByName:@"sv_anim_1.png"]];              // 5
        [viking setJoystick:leftJoystick];
        [viking setJumpButton:jumpButton];
        [viking setAttackButton:attackButton];
        [viking setPosition:ccp(screenSize.width * 0.35f,
                                screenSize.height * 0.14f)];
        [viking setCharacterHealth:100];

        [sceneSpriteBatchNode
            addChild:viking
                   z:kVikingSpriteZValue
                 tag:kVikingSpriteTagValue];                       // 6
```

```
        [self createObjectOfType:kEnemyTypeRadarDish
                      withHealth:100
                      atLocation:ccp(screenSize.width * 0.878f,
                                     screenSize.height * 0.13f)
                      withZValue:10];                                  // 7


        [self scheduleUpdate];                                         // 8
    }
    return self;
}
@end
```

Some key lines have been added since Chapter 2; they support the use of the `CCSpriteBatchNode` class and texture atlas:

1. Adds all of the frame dimensions specified in *scene1atlas.plist* to the Cocos2D Sprite Frame Cache. This will allow any `CCSprite` to be created by referencing one of the frames/images in the texture atlas. This line is also key in loading up the animations, since they reference `spriteFrames` loaded by the `CCSpriteFrameCache` here.
2. Initializes the `CCSpriteBatchNode` with the texture atlas image. The image *scene1atlas.png* becomes the master texture used by all of the `CCSprites` under the `CCSpriteBatchNode`. In *Space Viking* these are all of the `GameObjects` in the game, from the `Viking` to the `Mallet` power-up and the enemies.
3. Adds the `CCSpriteBatchNode` to the layer so it and all of its children (the `GameObjects`) are rendered onscreen.
4. Initializes the Joystick DPad and buttons.
5. Creates the `Viking` character using the already cached sprite frame of the `Viking` standing.
6. Adds the `Viking` to the `CCSpriteBatchNode`. The `CCSpriteBatchNode` does all of the rendering for the `GameObjects`. Therefore, the objects have to be added to the `CCSpriteBatchNode` and *not* to the layer. It is important to remember that the objects drawn from the texture atlas are added to the `CCSpriteBatchNode` and only the `CCSpriteBatchNode` is added to the `CCLayer`.
7. Adds the `RadarDish` to the `CCSpriteBatchNode`. The `RadarDish` health is set to 100 and the location as 87% of the screen width to the right (900 pixels from the left of the screen on the iPad) and 13% of the screen height (100 pixels from the bottom).

   The percentages are used instead of hard point values so that the same game will work on the iPhone, iPhone 4, and iPad. Although the screen width and height

ratios between the iPhones and iPad are a little different, they are close enough to work for the placement of objects in *Space Viking*.

8. Sets up a scheduler call that will fire the `update` method in *GameplayLayer.m* on every frame.

Now that you have added code to handle the `RadarDish`, the `Viking`, and the texture atlas, it is time to test out *Space Viking*. If you select **Run** from Xcode, you should see the *Space Viking* game in the iPad Simulator, as shown in Figure 4.3.

Figure 4.3 Space Viking with the RadarDish in place

Summary

If you made it through, great work—you've gotten a simple Cocos2D game working, and you've learned a lot in the process! You learned about texture atlases, actions, and animations. You utilized the texture atlas you created in the previous chapter to render all of the `GameObjects` in *Space Viking*. You created the enemy `RadarDish` and gave Ole the power to go over there and destroy it to bits. In the process you learned how to implement a simple state machine brain (AI) for the `RadarDish` and for the `Viking`. You have also set up the groundwork for *Space Viking* to have multiple enemies onscreen at once, each with its own AI state machines. The `CCArray` of objects you pass in `GameplayLayer` to each character on the `updateStateWithDeltaTime`

call will allow for the enemy objects to send messages to each other and even coordinate attacks against the `Viking`.

Since you just wrote so much code, you might want to take a few moments to examine the code in more detail and make sure you understand how it all fits together. It's important to make sure you understand how things work so far, since you'll be building more on top of what you've built here in the rest of the chapters.

In the next chapter, you will dive deeper into Cocos2D actions, learn to use some of the built-in effects, and add more enemies to *Space Viking*. When you are ready, turn the page and learn how to add a mean alien robot that shoots phaser beams.

Challenges

1. Try changing the `RadarDish` animation delay on the `takingAHitAnim` to `1.0f` seconds instead of `0.2f` in the *RadarDish.plist* file. What happens when you click **Run** and Ole attacks the `RadarDish`?
2. How would you add another instance of the `RadarDish` on the left side of the screen facing in the opposite direction?

> **Hint**
>
> You can use the `CCFlipX` action to flip the `RadarDish` pixels horizontally.

3. How would you detect when the `RadarDish` object is destroyed and alert the player that the level is complete?

> **Hint**
>
> You can extract the `RadarDish` object from the `sceneSpriteBatchNode` by using the unique `tag` assigned to the `RadarDish`.

This page intentionally left blank

Index

Symbols and Numbers

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z