

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 doing 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)

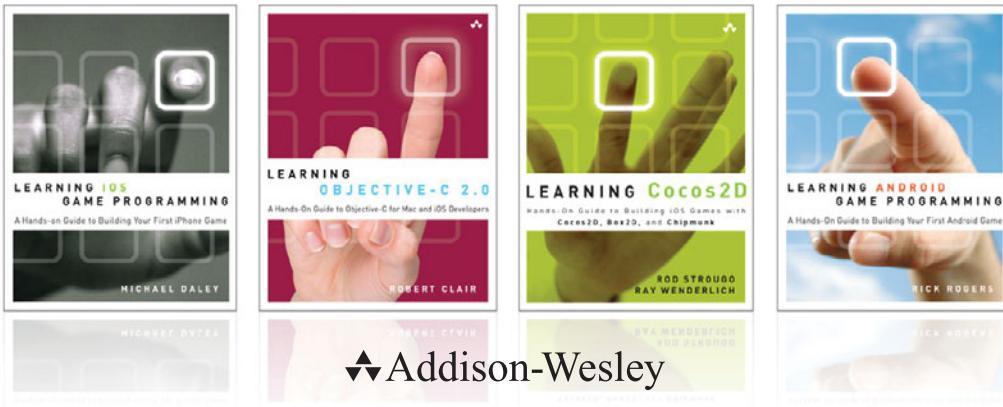
“This is the premiere book on Cocos2D! After reading this book you will have a firm grasp of the framework, and you will be able to create a few different types of games. Rod and Ray get you quickly up to speed with the basics in the first group of chapters. The later chapters cover the more advanced 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



▼ Addison-Wesley

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.

▼ Addison-Wesley

informIT.com

Safari®
Books Online

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

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and the publisher was aware of a trademark claim, the designations have been printed with initial capital letters or in all capitals.

The authors and publisher have taken care in the preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

The publisher offers excellent discounts on this book when ordered in quantity for bulk purchases or special sales, which may include electronic versions and/or custom covers and content particular to your business, training goals, marketing focus, and branding interests. For more information, please contact:

U.S. Corporate and Government Sales
(800) 382-3419
corpsales@pearsontechgroup.com

For sales outside the United States please contact:

International Sales
international@pearson.com

Visit us on the Web: informati.com/aw

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

Copyright © 2012 Pearson Education, Inc.

All rights reserved. Printed in the United States of America. This publication is protected by copyright, and permission must be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. For information regarding permissions, write to:

Pearson Education, Inc.
Rights and Contracts Department
501 Boylston Street, Suite 900
Boston, MA 02116
Fax: (617) 671-3447

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

Preface **xxi**

Acknowledgments **xxxiii**

About the Authors **xxxvii**

I Getting Started with Cocos2D	1
1 Hello, Cocos2D	3
2 Hello, Space Viking	23
3 Introduction to Cocos2D Animations and Actions	57
4 Simple Collision Detection and the First Enemy	83
II More Enemies and More Fun	115
5 More Actions, Effects, and Cocos2D Scheduler	117
6 Text, Fonts, and the Written Word	151
III From Level to Game	167
7 Main Menu, Level Completed, and Credits Scenes	169
8 Pump Up the Volume!	197
9 When the World Gets Bigger: Adding Scrolling	231
IV Physics Engines	277
10 Basic Game Physics: Adding Realism with Box2D	279
11 Intermediate Game Physics: Modeling, Racing, and Leaping	333
12 Advanced Game Physics: Even Better than the Real Thing	375
13 The Chipmunk Physics Engine (No Alvin Required)	419

V Particle Systems, Game Center, and Performance 479

- 14 Particle Systems: Creating Fire, Snow, Ice, and More **481**
- 15 Achievements and Leaderboards with Game Center **495**
- 16 Performance Optimizations **545**
- 17 Conclusion **565**
- A Principal Classes of Cocos2D **569**
- Index **571**

Contents

Preface	xxi
Acknowledgments	xxxiii
About the Authors	xxxvii

I Getting Started with Cocos2D	1
1 Hello, Cocos2D	3
Downloading and Installing Cocos2D	4
Downloading Cocos2D	4
Installing the Cocos2D Templates	5
Creating Your First Cocos2D HelloWorld	6
Inspecting the Cocos2D Templates	6
Building the Cocos2D HelloWorld Project	7
Taking HelloWorld Further	9
Adding Movement	10
For the More Curious: Understanding the Cocos2D HelloWorld	11
Scenes and Nodes	11
From the Beginning	14
Looking Further into the Cocos2D Source Code	18
Getting CCHelloWorld on Your iPhone or iPad	20
Letting Xcode Do Everything for You	20
Building for Your iPhone or iPad	21
Summary	22
Challenges	22
2 Hello, Space Viking	23
Creating the SpaceViking Project	23
Creating the Space Viking Classes	24
Creating the Background Layer	26
The Gameplay Layer: Adding Ole the Viking to the Game	29
The GameScene Class: Connecting the Layers in a Scene	31
Creating the GameScene	32

Commanding the Cocos2D Director	34
Adding Movement	35
Importing the Joystick Classes	35
Adding the Joystick and Buttons	36
Applying Joystick Movements to Ole the Viking	40
Texture Atlases	44
Technical Details of Textures and Texture Atlases	45
Creating the Scene 1 Texture Atlas	48
Adding the Scene 1 Texture Atlas to Space Viking	51
For the More Curious: Testing Out CCSpriteBatchNode	52
Fixing Slow Performance on iPhone 3G and Older Devices	53
Summary	54
Challenges	54
3 Introduction to Cocos2D Animations and Actions	57
Animations in Cocos2D	57
Space Viking Design Basics	62
Actions and Animation Basics in Cocos2D	66
Using Property List Files to Store Animation Data	67
Organization, Constants, and Common Protocols	69
Creating the Constants File	71
Common Protocols File	72
The GameObject and GameCharacter Classes	74
Creating the GameObject	74
Creating the GameCharacter Class	80
Summary	82
Challenges	82
4 Simple Collision Detection and the First Enemy	83
Creating the Radar Dish and Viking Classes	83
Creating the RadarDish Class	83
Creating the Viking Class	90
Final Steps	105
The GameplayLayer Class	105

Summary	112
Challenges	113

II More Enemies and More Fun 115

5 More Actions, Effects, and Cocos2D	
Scheduler	117
Power-Ups	118
Mallet Power-Up	118
Health Power-Up	120
Space Cargo Ship	122
Enemy Robot	125
Creating the Enemy Robot	126
Adding the PhaserBullet	137
GameplayLayer and Viking Updates	141
Running Space Viking	144
For the More Curious: Effects in Cocos2D	145
Effects for Fun in Space Viking	146
Running the EffectsTest	148
Returning Sprites and Objects Back to Normal	149
Summary	149
Exercises and Challenges	149
6 Text, Fonts, and the Written Word 151	
CCLabelTTF	151
Adding a Start Banner to Space Viking	152
Understanding Anchor Points and Alignment	153
CCLabelBMFont	155
Using Glyph Designer	156
Using the Hiero Font Builder Tool	156
Using CCLabelBMFont Class	159
For the More Curious: Live Debugging	160
Updating EnemyRobot	160
Updating GameplayLayer	163
Other Uses for Text Debugging	164
Summary	165
Challenges	165

III From Level to Game 167**7 Main Menu, Level Completed, and Credits****Scenes 169**

Scenes in Cocos2D 169

Introducing the GameManager 170

Creating the GameManager 172

Menus in Cocos2D 179

Scene Organization and Images 180

Adding Images and Fonts for the Menus 181

Creating the Main Menu 182

Creating the MainMenuScene 182

MainMenuLayer class 183

Additional Menus and GameplayLayer 190

Importing the Intro, LevelComplete, Credits, and

Options Scenes and Layers 190

GameplayLayer 190

Changes to SpaceVikingAppDelegate 192

For the More Curious: The IntroLayer and LevelComplete

Classes 193

LevelCompleteLayer Class 194

Summary 195

Challenges 195

8 Pump Up the Volume! 197

Introducing CocosDenshion 197

Importing and Setting Up the Audio Filenames 198

Adding the Audio Files to Space Viking 198

Audio Constants 198

Synchronous versus Asynchronous Loading
of Audio 201

Loading Audio Synchronously 201

Loading Audio Asynchronously 203

Adding Audio to GameManager 204

Adding the soundEngine to GameObjects 215

Adding Sounds to RadarDish and

SpaceCargoShip 216

Adding Sounds to EnemyRobot 219

Adding Sound Effects to Ole the Viking	222
Adding the Sound Method Calls in changeState for Ole	226
Adding Music to the Menu Screen	228
Adding Music to Gameplay	228
Adding Music to the MainMenu	228
For the More Curious: If You Need More Audio Control	229
Summary	230
Challenges	230
9 When the World Gets Bigger: Adding Scrolling	231
Adding the Logic for a Larger World	232
Common Scrolling Problems	234
Creating a Larger World	235
Creating the Second Game Scene	236
Creating the Scrolling Layer	242
Scrolling with Parallax Layers	250
Scrolling to Infinity	252
Creating the Scrolling Layer	254
Creating the Platform Scene	263
Tile Maps	265
Installing the Tiled Tool	266
Creating the Tile Map	267
Cocos2D Compressed TiledMap Class	271
Adding a TileMap to a ParallaxNode	272
Summary	276
Challenges	276
IV Physics Engines	277
10 Basic Game Physics: Adding Realism with Box2D	279
Getting Started	279
Mad Dreams of the Dead	281
Creating a New Scene	282

Adding Box2D Files to Your Project	284
Box2D Units	288
Hello, Box2D!	289
Creating a Box2D Object	292
Box2D Debug Drawing	295
Putting It All Together	296
Creating Ground	299
Basic Box2D Interaction and Decoration	302
Dragging Objects	304
Mass, Density, Friction, and Restitution	309
Decorating Your Box2D Bodies with Sprites	313
Making a Box2D Puzzle Game	320
Ramping It Up	324
Summary	332
Challenges	332
11 Intermediate Game Physics: Modeling, Racing, and Leaping	333
Getting Started	334
Adding the Resource Files	334
Creating a Basic Box2D Scene	335
Creating a Cart with Box2D	346
Creating Custom Shapes with Box2D	346
Using Vertex Helper	348
Adding Wheels with Box2D Revolute Joints	352
Making the Cart Move and Jump	356
Making the Cart Move with the Accelerometer	356
Making It Scrollable	359
Forces and Impulses	368
Fixing the Tipping	368
Making the Cart Jump	369
More Responsive Direction Switching	373
Summary	374
Challenges	374

12 Advanced Game Physics: Even Better than the Real Thing 375

Joints and Ragdolls: Bringing Ole Back into Action	376
Restricting Revolute Joints	376
Using Prismatic Joints	378
How to Create Multiple Bodies and Joints at the Right Spots	378
Adding Ole: The Implementation	380
Adding Obstacles and Bridges	386
Adding a Bridge	386
Adding Spikes	390
An Improved Main Loop	394
The Boss Fight!	396
A Dangerous Digger	405
Finishing Touches: Adding a Cinematic Fight Sequence	411
Summary	417
Challenges	417

13 The Chipmunk Physics Engine (No Alvin Required) 419

What Is Chipmunk?	420
Chipmunk versus Box2D	420
Getting Started with Chipmunk	421
Adding Chipmunk into Your Project	426
Creating a Basic Chipmunk Scene	429
Adding Sprites and Making Them Move	438
Jumping by Directly Setting Velocity	444
Ground Movement by Setting Surface Velocity	445
Detecting Collisions with the Ground	445
Chipmunk Arbiter and Normals	446
Implementation—Collision Detection	446
Implementation—Movement and Jumping	450
Chipmunk and Constraints	455
Revolving Platforms	458
Pivot, Spring, and Normal Platforms	460

The Great Escape!	467
Following Ole	467
Laying Out the Platforms	468
Animating Ole	469
Music and Sound Effects	473
Adding the Background	474
Adding Win/Lose Conditions	476
Summary	477
Challenges	477
 V Particle Systems, Game Center, and Performance 479	
14 Particle Systems: Creating Fire, Snow, Ice, and More	481
Built-In Particle Systems	482
Running the Built-In Particle Systems	482
Making It Snow in the Desert	483
Getting Started with Particle Designer	485
A Quick Tour of Particle Designer	486
Creating and Adding a Particle System to Space Viking	489
Adding the Engine Exhaust to Space Viking	490
Summary	494
Challenges	494
15 Achievements and Leaderboards with Game Center	495
What Is Game Center?	495
Why Use Game Center?	497
Enabling Game Center for Your App	497
Obtain an iOS Developer Program Account	497
Create an App ID for Your App	498
Register Your App in iTunes Connect	501
Enable Game Center Support	505
Game Center Authentication	506
Make Sure Game Center Is Available	506

Try to Authenticate the Player	507
Keep Informed If Authentication Status Changes	508
The Implementation	508
Setting Up Achievements	515
Adding Achievements into iTunes Connect	515
How Achievements Work	517
Implementing Achievements	518
Creating a Game State Class	519
Creating Helper Functions to Load and Save Data	522
Modifying GCHelper to Send Achievements	524
Using GameState and GCHelper in SpaceViking	530
Displaying Achievements within the App	534
Setting Up and Implementing Leaderboards	536
Setting up Leaderboards in iTunes Connect	536
How Leaderboards Work	538
Implementing Leaderboards	539
Displaying Leaderboards in-Game	540
Summary	543
Challenges	543
16 Performance Optimizations	545
CCSprite versus CCSpriteBatchNode	545
Testing the Performance Difference	550
Tips for Textures and Texture Atlases	551
Reusing CCSprites	552
Profiling within Cocos2D	554
Using Instruments to Find Performance Bottlenecks	557
Time Profiler	558
OpenGL Driver Instrument	560
Summary	563
Challenges	563

17 Conclusion 565

Where to Go from Here 567

Android and Beyond 567

Final Thoughts 568

A Principal Classes of Cocos2D 569

Index 571

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.

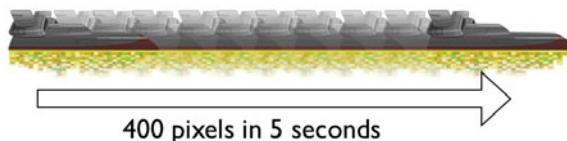


Figure P.1 Illustrating the effect of the moveAction on the Space Cargo Ship sprite

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 game-play. 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 or 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 {
    CCAutomation *tiltingAnim;
    CCAutomation *transmittingAnim;
    CCAutomation *takingAHitAnim;
    CCAutomation *blowingUpAnim;
    GameCharacter *vikingCharacter;
}

@property (nonatomic, retain) CCAutomation *tiltingAnim;
@property (nonatomic, retain) CCAutomation *transmittingAnim;
@property (nonatomic, retain) CCAutomation *takingAHitAnim;
@property (nonatomic, retain) CCAutomation *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(@"RadarDisk->Changing State to Idle");
            action = [CCAnimate actionWithAnimation:transmittingAnim
                           restoreOriginalFrame:NO];
            break;

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

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

```

The `changeState` method is called when the `RadarDisk` 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 minuscule “brain” of the `RadarDisk`. The `RadarDisk` brain can exist in one of four states: spawning, idle, taking damage, or dead. In the listings that follow, you will see that the `RadarDisk` 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 `RadarDisk` 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 numberOfRowsInSection] == 0) &&
    (characterState != kStateDead)) {
    CCLOG(@"Going to Idle");
    [self changeState:kStateIdle];
    return; // 6
}

```

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(@"### RadarDisk initialized");
        [self initAnimations]; // 1
        characterHealth = 100.0f; // 2
        gameObjectType = kEnemyTypeRadarDisk; // 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 `RadarDisk`. 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 `RadarDisk`. 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 `RadarDisk` to a value of 100.
3. Sets the `RadarDisk` to be a Game Object of type `kEnemyTypeRadarDisk`.
4. Initializes the state of the `RadarDisk` 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 `RadarDisk` 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 `RadarDisk` 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 `RadarDisk` class: the `Viking` is simply a game character like the `RadarDisk`, 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
    CCAutomation *breathingAnim;
    CCAutomation *breathingMalletAnim;
    CCAutomation *walkingAnim;
    CCAutomation *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) {           // 3
            self.flipX = YES;
        } 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.

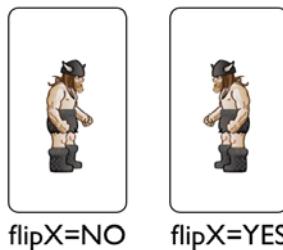


Figure 4.1 Effects of the `flipX` function on the Viking texture or graphic

The next section of the *Viking.m* implementation file covers the `changeState` method. As you learned with the `RadarDisk` 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.

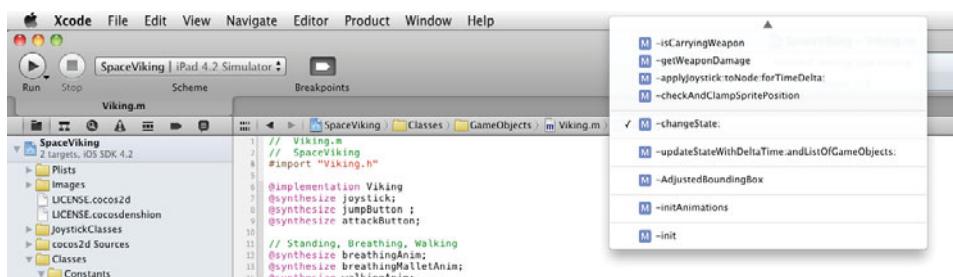


Figure 4.2 The effect of the `pragma mark` statements in the Xcode pulldown menus

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 numberOfRowsInSection] > 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 right, 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/textures. 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 `RadarDisk` 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.userInteractionEnabled = YES;

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

        if (UI_USER_INTERFACE_IDIOM() == UIUserInterfaceIdiomPad) {
            [[CCSpriteFrameCache sharedSpriteFrameCache]
                addSpriteFramesWithFile:@"scenelatlas.plist"]; // 1
            sceneSpriteBatchNode =
                [CCSpriteBatchNode batchNodeWithFile:@"scenelatlas.png"]; // 2
        } else {
            [[CCSpriteFrameCache sharedSpriteFrameCache]
                addSpriteFramesWithFile:@"scenelatlasiPhone.plist"]; // 1
            sceneSpriteBatchNode =
                [CCSpriteBatchNode
                    batchNodeWithFile:@"scenelatlasiPhone.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 `RadarDisk` animation delay on the `takingAHitAnim` to 1.0f seconds instead of 0.2f in the `RadarDisk.plist` file. What happens when you click **Run** and Ole attacks the `RadarDisk`?
2. How would you add another instance of the `RadarDisk` on the left side of the screen facing in the opposite direction?

Hint

You can use the `CCFlipX` action to flip the `RadarDisk` pixels horizontally.

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

Hint

You can extract the `RadarDisk` object from the `sceneSpriteBatchNode` by using the unique `tag` assigned to the `RadarDisk`.

This page intentionally left blank

Index

Symbols and Numbers

? (Ternary operator), 134–135

3D

2D vs., xxv

extensions to Cocos2D, 567

z values in, 33

A

AABB (axis-aligned bounding boxes)

avoiding object overlap, 77

searching for objects in Box2D world,
305–307

Accelerometer

cart movement example, 355–358

commenting out, 319

enabling support for, 302–303

implementing movement in Box2D, 281

implementing movement in Chipmunk,
451, 454–455

Accounts

iOS Developer Program account, 497–498

iPhone Developer account, 20

Sandbox accounts, 514

Achievements

adding to iTunes Connect, 515–517

displaying within apps, 534–536

GameState class and, 519–521

helper function for sending achievement
data, 524–530

helper functions for loading/saving
achievement data, 522–524

how they work, 517–518

implementing, 518

overview of, 515

using GameState and GCHelper classes
in Space Viking, 530–534

Action layer, getting started with Chipmunk,
423–425

Actions. *See also* Animation (CCAnimation)

CCMoves and CCScale actions in Space

Cargo ship, 123

compound, 99

effects packaged as, 145

GameplayLayer class and, 127

key features in Cocos 2D, xxiii

numberOfRunningActions method, 135

overview of, 66–67

space cargo ship and, 125

addChild method, CCParallaxNode,
251–252

addEnemy method, 143–144

addScrollingBackground method, 245

addScrollingBackgroundWithParallax
method, 250–252

addScrollingBackground-
withTileMapInsideParallax
method, 272–275

adjustedBoundingBox method

enemy robot, 137

Ole the Viking, 100–103

adjustLayer method, 245–247

AI (artificial intelligence)

design basics, 65

game logic and, 63

Anchor points

for Game Start banner, 153–154

overview of, 153

for rotation and other effects, 154

Android, Cocos2D-Android, 567

Angular impulses, Box2D

controlling flipping of cart, 368–369

overview of, 368

Animation (CCAnimate)

actions and, 66

approaches to animation, 57

creating actions, 58

delays between frames and frame list, 61

- Animation (`CCAnimation`)
 - actions and, 66–67
 - animating sprites generally, 57–60
 - animating sprites rendered by `CCSpriteBatchNode`, 60–61
 - caching, 62
 - delays between frames and frame list, 61
 - frame rate in, 61
 - overview of, 57
 - storing animation data in plist files, 61, 67–69
- Animation, generally
 - adding background animation that interacts with game, 122
 - of breathing, 103
 - `changeState` method for starting, 95–98
 - helper methods for, 411
 - initiating for `RadarDisk` class, 88–89
 - initiating for `Viking` class, 103–105
 - of Ole the Viking in Chipmunk, 469–473
 - repeating, 67, 120
- App ID, creating, 498–501
- App Store, 497
- Application Delegate (`AppDelegate`)
 - `ApplicationDidFinishLaunching` method, 14–15
 - commanding director to run game scene, 34–35
 - in `HelloWorld` app, 15–18
- `ApplicationDidFinishLaunching` method, `AppDelegate` class, 14–15
- `applyJoystick` method, `Viking` class, 94
- Apps
 - creating App ID, 498–501
 - displaying achievements in, 534–536
 - enabling support in Game Center, 505–506
 - registering in iTunes Connect, 501–505
- Arbiters, collision events and, 446
- `ARCH_OPTIMAL_PARTICLE_SYSTEM`, 482
- artificial intelligence (AI)
 - design basics, 65
 - game logic and, 63
- Assignment operator, combining `if` statement with, 134–135
- Attack buttons, added to `GameplayLayer` class, 108
- Attack phase, `RadarDisk` class and `Viking` class and, 88
- Attack state, enemy robot and, 132–133
- Audio
 - adding audio files, 198
 - additions to game manager header and implementation files, 204–205
 - audio constants, 198–201
 - CocosDenshion sound engine, 197–198
 - getting list of sound effects, 208–211
 - `initAudioAsync` method, 206–207
 - initializing audio manager (`CD AudioManager`), 207–208
 - loading asynchronously, 203–204
 - loading sound effects, 211–213
 - loading synchronously, 201–203
 - loading/unloading audio files, 214–215
 - music added to `GameplayLayer`, 228
 - music added to `MainMenu`, 228–229
 - music and sound effects in Chipmunk, 473–474
 - `playBackgroundTrack`, `stopSoundEffect`, and `playSoundEffect` methods, 213–214
 - setting up audio engine, 205–206
 - `SimpleAudioEngine`, 229–230
 - sound engine in Cocos2D, xxiv
 - sounds added to `EnemyRobot`, 219–222
 - sounds added to game objects, 215–216
 - sounds added to Ole the Viking, 222–228
 - sounds added to `RadarDisk`, 216–217
 - sounds added to `SpaceCargoShip`, 217–219
- Audio constants, 198–201
- Audio engines
 - setting up, 205–206
 - `SimpleAudioEngine`, 229–230
- Audio files
 - adding to Space Viking project, 198
 - loading/unloading, 214–215
- Authentication
 - notification of changes to authentication status, 508–514
 - of players in Game Center, 507–508

`AVAudioPlay`, audio framework for iOS devices, 197

Axis-aligned bounding boxes (AABB)
 avoiding object overlap, 77
 searching for objects in Box2D world, 305–307

B

Background color, Particle Designer controls, 487

Background layer
 adding background music in Chipmunk, 473–474
 adding in Chipmunk, 474–476
 `addScrollingBackground` method, 245
 connecting background and game layers to a scene, 31–32
 creating for Space Viking project, 26–29
 creating wave action in, 146–148
 splitting into static and scrolling layers, 237–239

Background threads
 adding audio asynchronously in, 201
 managing, 204

`begin` events, collision-related events in Chipmunk, 445, 448–449

Bind calls, OpenGL ES, 45, 48

Bit depth, performance tips and, 551

Bitmap fonts, 155, 179

Bodies, Box2D
 `createBodyAtLocation` method, 338
 creating, 292–295
 creating drill sensor for Digger Robot, 401–402
 creating for Ole and connecting with joints, 376
 creating ground body in `PuzzleLayer`, 299–302
 creating multiple bodies and joints, 378–380
 decorating, 313–320
 good and bad ways for placing, 379–380
 `setLinearVelocity` method, 415

Bodies, Chipmunk
 adding, 420
 adding box to Chipmunk space, 431–433

constraints acting on, 457–458
 creating revolving platform, 459–460
 directly setting velocity, 444

Bottlenecks, finding, 557–558

Bounding boxes
 for avoiding object overlap, 77
 `EyesightBoundingBox` method, for enemy robot, 129

Box2D
 bodies. *See* bodies, Box2D
 Chipmunk compared with, 420–421
 source code, 18
 template, 7

Box2D, advanced physics
 adding dangerous methods to Digger, 405–411
 bridges in, 386–389
 creating cinematic fight sequence, 411–416
 creating multiple bodies and joints, 378–380
 joints in, 376
 Ole leaping with ragdoll effect, 381–386
 overview of, 375
 pitting Ole against Digger in fight, 396–405
 prismatic joints, 378
 restricting revolute joints, 376–377
 spike obstacle in, 390–394
 summary and challenges, 417
 variable and fixed rate timestamps in, 394–396

Box2D, basic physics
 adding files to project, 284–288
 creating ground body in `PuzzleLayer`, 299–302
 creating new scene in `PuzzleLayer`, 282–284
 creating objects, 292–295
 creating world, 289–292
 debug drawing, 295–296
 decorating using sprites, 313–320
 dragging objects, 304–309
 getting started with, 279–281
 interaction and decoration in, 302–304
 mass, density, friction, and restitution in, 309–313
 overview of, 279

- Box2D, basic physics (*continued*)

 puzzle game example, 320–324

 ramping up puzzle game, 324–332

 units in, 288–289

 viewing `PuzzleLayer`, 296–298
- Box2D, intermediate physics

 adding resource files, 334–335

 adding wheels to cart using revolute joints, 352–355

 controlling flipping of cart, 368–369

 creating cart scene for, 335–346

 creating custom shapes, 346–348

 forces and impulses, 368

 getting started with, 334

 making cart jump, 369–373

 making cart move using accelerometer, 355–358

 making cart scene scrollable, 358–368

 overview of, 333

 responsive direction switching, 373–374

 Vertex Helper, 348–352
- `Box2DSprite` class, subclasses for Digger Robot, 398
- Bridges, creating in Box2D, 386–389
- Build (z-B), testing build of Space Viking, 33
- Buttons

 adding to Space Viking, 36–40

 connecting button controls to Ole the Viking, 245
- C**
- C++, Box2D written in, 280, 420
- C language, Chipmunk written in, 420
- Caching

 animations, 62, 411

 textures, 17
- Callback functions, collision detection and, 446
- Cargo ship. *See* Space cargo ship
- Cart

 adding wheels using revolute joints, 352–355

 controlling flipping, 368–369

 creating cart scene, 335–346

 creating custom shapes, 346–348

 header and implementation files, 337–338
- making cart jump, 369–373

making cart scene scrollable, 358–368

moving using accelerometer, 355–358
- `categoryBits`, setting object categories, 381–382
- `CCAnimate`. *See* Animation (`CCAnimate`)
- `CCAnimation`. *See* Animation (`CCAnimation`)
- `CCAnimationCache`, 62, 411
- `CCCallFunc` action, 132–133
- `CCDirector`. *See* Director (`CCDirector`)
- `CCFollow` action, 249
- `CCJumpBy` action, 66
- `CCLabel` class. *See* Labels (`CCLabel`)
- `CCLabelIBMFont` class

 overview of, 155

 using, 159
- `CCLabelTTF` class

 adding Game Start banner, 152–153

 anchor points for Game Start banner, 153–154

 fonts, 155

 overview of, 151
- `CCLayer` class. *See* Layers (`CCLayer`)
- `CCLOG` macro, for `NSLOG` method, 41
- `CCMenu` class. *See also* Menus, 179
- `CCMenuAtlasFont`, 179
- `CCMenuItemFont`, 180
- `CCMenuItemImage`, 180
- `CCMenuItemLabel`, 180
- `CCMenuItemSprite`, 180
- `CCMenuItemToggle`, 180
- `CCMoves` action, 123
- `CCNode` class. *See* Nodes (`CCNode`)
- `ccp` macro, shortcut to `CGPointMake` method, 20
- `CCParallaxNode`

 `addChild` method, 251–252

 adding TileMap to, 272–275

 `addScrollingBackgroundWithParallax` method, 250–251
- `CCParticleSystemPoint`, 481–482
- `CCParticleSystemQuad`, 482
- `CCRepeat`, 120
- `CCRepeatForever`, 67, 120
- `CCScale` action, 123
- `CCScenes`. *See* Scenes (`CCScenes`)

`CCSequence`, 67
`CCSpawn`, 67
`CCSprite` (*Sprites*). *See* `CCSprite`
`CCSpriteBatchNode`
 animating sprites rendered by, 60–61
 `GameplayLayer` class and, 111
 performance benefits of, 255, 545–550
 testing use in game layer, 52–53
 using texture atlases and, 44–45
`CCSpriteFrame`, 60
`CCSpriteSheet`, 134–135
`CCTMXTiledMap`, 271
`ccTouchBegan` method, 304–308, 344
`ccTouchEnded` method, 344
`ccTouchesBegan` method, 262
`ccTouchesMoved` method, 308–309, 344
`CCWaves` action, creating wave action in background, 146–148
`CD AudioManager`, initializing, 207–208
`CGSize`, 232–233
`changeState` method
 enemy robot, 129–133
 Ole the Viking, 95–98
 radar dish, 85–86
Characters. *See* Game characters
 (`GameCharacter`)
`checkAnd.ClampSpritePosition` method
 ensuring enemy robot remains within screen boundaries, 134
 ensuring Viking sprite remains within screen boundaries, 102
`gameCharacter` class and, 233–234
Chipmunk
 adding backgrounds, 474–476
 adding music and sound effects, 473–474
 adding sprites, 438–444
 adding to Xcode project, 426–429
 adding win/lose conditions, 476–477
 animating Ole, 469–473
Box2D compared with, 420–421
collision detection in, 445–450
constraints in, 455–458
creating a scene, 430–438
following Ole, 467–468
getting started with, 421–426
implementing velocity of sprite, 444
initializing, 429–430
 laying out platforms, 468–469
 movement and jumping, 450–455
 overview of, 419–420
 pivot, spring, and normal platforms in, 460–466
 revolving platform in, 458–460
 summary and challenges, 477
 surface velocity for ground movement, 445
 template, 7
 viewing source code, 18
Cinematic fight sequence, creating, 411–416
Classes
 converting objects into, 63
 creating for Space Viking project, 24–26
 creating `GameCharacter` class, 80–82
 creating `GameObject` class, 74–80
 of game objects, 64–65
 grouping as organization technique, 70
 importing joystick class for Space Viking, 35–36
 loose coupling, 117–118
 principal classes in Cocos2D, 569–570
Cocos2D-Android, 567
Cocos2D Application template, 7, 284
Cocos2D Box2D Application template, 284
Cocos2D Director. *See* Director
 (`CCDirector`)
Cocos2D, introduction to
 important concept, xiv–xxv
 key features, xxiii–xiv
 what it is, xxii
 why you should use it, xxii–xxiii
Cocos2D-JavaScript, 568
Cocos2D-X, 567–568
CocosDenshion
 importing `SimpleAudioEngine`, 205
 initializing audio manager (`CD AudioManager`), 207–208
 loading audio asynchronously, 203–204
 loading sound effects, 211–213
 sound engine, 197–198
 viewing source code, 18
Collision filters, Box2D, 381–382
Collisions
 checking for, 102
 comparing Box2D with Chipmunk, 421

- Collisions (*continued*)**
- detecting in Chipmunk, 445–450
 - Digger Robot and, 408
 - optimizing collision detection in Chipmunk, 431
- Common protocols.** *See* Protocols
- Compression formats,** 43
- Constants**
- audio, 198–201
 - for static values used in more than one class, 71–72
- Constraints,** in Chipmunk
- compared with joints, 420–421
 - creating pivot platforms and, 462
 - creating spring platforms and, 463–464
 - steps in use of, 456–458
 - types of, 455–456
- ControlLayer,** connecting joystick and button controls to Viking, 245
- Coordinate systems,** converting UIKIT to/from OpenGL ES, 29
- cptArbiter,** collision events and, 446
- cptPolyShapeNew,** 448
- CPRevolvePlatform,** subclass for revolving platform, 458–460
- CPSprite (Sprites).** *See* Sprites (CPSprite)
- CPU utilization,** Time Profiler capturing data related to, 558–560
- CPViking,** animating Ole in Chipmunk, 469–473
- createBodyAtLocation** method, Box2D, 338
- createCartAtLocation** method, Box2D, 344
- createCloud** method, platformScrollingLayer class, 257–258
- createGround** method, carts, 344
- createObjectType** method, adding objects to gameplayLayer class, 141–142
- createPhaserWithDirection** method, 142–143
- createStaticBackground** method, in platformScrollingLayer, 257
- createVikingAndPlatform** method, in platformScrollingLayer, 261–262
- createWheelWithSprite,** 354
- Credits**
- scene types and, 170
 - setting up menus, 190
- Ctrl-z-D (Jump to Definition),** for viewing source code, 18–19
- Cut-scene**
- creating group for, 252–253
 - creating scrolling layer in, 254–262
- D**
- Damage taking state**
- (kStateTakingDamage)
 - Digger Robot and, 407–408
 - for enemy robot, 133
 - spike obstacle and, 391–392
 - taking a hit and being restored to previous state, 99
- Damped rotary spring**
- constraints in Chipmunk, 457
 - creating pivot platform, 462
- Damped spring**
- constraints in Chipmunk, 457
 - creating spring platform, 463
- Database,** loading/saving achievement data to GCDatabase, 522–524
- Dead state (kStateDead)**
- for enemy robot, 133
 - health at zero level, 103
 - state transition in RadarDish class, 87
- dealloc** method, Viking class, 94
- Debug draw**
- in Box2D, 295–296
 - in Chipmunk, 434–436
- Debugging,** creating debug label, 160–165
- Decoration,** Box2D, 302–304
- #define statement**
- setting object categories, 382
 - setting up audio filenames as, 199–200
- Delegate classes,** 118
- deltaTime**, scheduler and, 145
- density** property
- for cart wheels, 354
 - for fixtures, 309–313
- Design basics**
- artificial intelligence and, 65
 - caching and, 62

- classes of game objects, 64–65
 - object-orientation in, 63
 - overview of, 62–63
- Devices, older
 - fixing slow performance, 53–54
 - power-of-two support, 46
- Digger robot
 - adding dangerous methods to, 405–411
 - creating cinematic fight sequence, 411–416
 - pitting Ole against, 396–405
- Direction switching, in Box2D, 373–374
- Directional pad (DPad)
 - added to `GameplayLayer` class, 108
 - initializing, 111
- Director (`CCDirector`)
 - running loops and rendering graphics, 16–18
 - running scenes, 11, 34–35
 - types of, 569
- Directory, adding Chipmunk files to Xcode project, 426–429
- Distance joints
 - in Box2D, 304
 - Chipmunk damped spring compared with, 457
 - Chipmunk pin joint compared with, 456
- Downloading Cocos2D, 4–5
- DPad (directional pad)
 - added to `GameplayLayer` class, 108
 - initializing, 111
- Dragging objects, in Box2D, 304–309
- Drill sensors, creating for Digger Robot, 401–402
- `dropCargo` method, space cargo ship, 125
- `dropWithLowPerformanceItemWithID` method, reusing sprites and, 553–554
- Dynamic bodies, Box2D, 293

- E**
- `EAGLView`, rendering game with, 16
- Effects, 145–149
 - anchor points for, 154
 - comparing Box2D with Chipmunk, 421
 - creating wave action in background, 146–148
 - packaged as actions, 145
 - returning sprites and objects to nonaltered state, 149
 - running `EffectsTest`, 148
 - screen shake, 467–468
 - subtypes of, 146
- Effects library, key features in Cocos 2D, xxiv
- Elasticity, setting for ground in Chipmunk space, 433
- Emitters
 - adding engine exhaust to space cargo ship, 490–494
 - Particle Designer controls for, 487–488
 - in particle systems, 481
- `enableLimit`, restricting revolute joints and, 377
- Enemy characters
 - Digger Robot. *See* Digger robot
 - enemy robot. *See* Enemy robot
 - methods for creating in `GameplayLayer` class, 109
 - `RadarDish` class. *See* RadarDish class
- Enemy robot
 - adding as long as radar dish is not dead, 143–144
 - adding sounds to, 219–222
 - animating, 58–59
 - `changeState` method and, 129–133
 - checking if Viking is attacking, 135
 - header file, 126–127
 - implementation file, 127–137
 - overview of, 125
 - setting up to update debug label, 160–163
 - steps in creation of, 126
 - teleport graphic for, 132
 - texture atlases and robot size, 61
 - `updateStateWithDeltaTime` method for, 133–135
- Engine exhaust effect, adding to space cargo ship, 490–494
- EyesightBoundingBox method, for enemy robot, 129

- F**
- FBO (frame buffer object), 145–146
- Fight sequence, creating, 411–416

Files

- Add New File dialog, 26
- adding Box2D files to project, 334–335
- adding Chipmunk files to project, 426–429
- audio files, 198, 214–215
- constants file for static values used in more than one class, 71–72
- format for fonts, 155
- formats for images, 43
- GLES-Render files, 295–296
- header. *See* Header files
- implementation. *See* Implementation files
- PNG files, 43, 270
- property list. *See* plist files
- TMX files, 270–271
- Fixed rate timestamps
 - game loops and, 434
 - improving main loop and, 394–396
- Fixtures
 - of Box2D bodies, 292–294
 - compared with Chipmunk shapes, 420
 - creating drill sensor for Digger Robot, 401–402
 - properties, 309–313
- `flipX/flipY` functions
 - for mirroring graphic views, 95
 - reversing images, 552
- font file format, 155
- Fonts
 - adding for menus, 181–182
 - built-in support for, xxiii
 - `CCLabelIBMFFont` class, 155, 159
 - `CCLabelTTF` class, 155
 - `CCMenuAtlasFont`, 179
 - `CCMenuItemFont`, 180
 - Hiero Font Builder Tool, 156–159
- Forces, Box2D, 368
- FPS (Frames Per Second), managing frame rate in animation, 16, 61
- Frame buffer object (FBO), 145–146
- friction property
 - fixtures, 309–313
 - setting for cart wheels, 354
 - setting for ground, 433

G

Game Center

- achievements. *See* Achievements
- authenticating players, 507–508
- checking availability of, 506–507
- creating App ID, 498–501
- enabling support for apps, 505–506
- leaderboards. *See* Leaderboards
- notification of changes to authentication status, 508–514
- obtaining iOS Developer Program account, 497–498
- overview of, 495–497
- reasons for using, 497
- registering apps in iTunes Connect, 501–505
- sending scores to, 538
- summary and challenges, 543
- Game characters (`GameCharacter`)
 - `checkAndClampSpritePosition` method, 233–234
 - in class hierarchy, 64
 - creating, 80–82
 - enemy robot inheriting from, 126–127
 - `RadarDisk` class inheriting from, 84–85
 - `Viking` class inheriting from, 90
- Game layers. *See* Layers (`CCLayer`)
- Game logic, behind game objects, 63
- Game manager (`GameManager`)
 - adding last level completed property to, 532–534
 - adding support to `GameplayLayer` class for, 190–192
 - additions for audio to header and implementation files, 204–205
 - changing level width, 234–235
 - connecting to Chipmunk scene with, 425–426
 - creating, 172–179
 - `getDimensionsOfCurrentScene` method, 232–233
 - getting list of sound effects, 208–211
 - header file, 172–173
 - implementation file, 174–177
 - `initAudioAsync` method, 206–207

- initializing audio manager (`CDAudioManager`), 207–208
- `IntroLayer` class and, 193
- `LevelCompleteLayer` class and, 194–195
- loading audio asynchronously, 203–204
- loading sound effects, 211–213
- loading/unloading audio files, 214–215
- overview of, 170–172
- `playBackgroundTrack`, `stopSoundEffect`, and `playSoundEffect` methods, 213–214
- running new cart scene, 345
- setting up audio engine, 205–206
- `SpaceVikingAppDelegate` supporting, 192–193
- switching to win/lose conditions, 476–477
- Game objects (`GameObject`). *See also* Objects
 - adding sound to game objects, 215–216
 - in class hierarchy, 64–65
 - creating, 74–80
 - `Mallet` class inheriting from, 119
- Game physics. *See* Physics engines
- Game Start banner
 - adding, 152–153
 - anchor points for, 153–154
- `GameControlLayer`, as subclass of `CCLayer`, 239–242
- `GameManager` class. *See* Game manager (`GameManager`)
- Gameplay scenes, 170
- `GameplayLayer` class
 - adding music to, 228
 - adding support for game manager, 190–192
 - `addScrollingBackgroundWithParallax` method, 250–252
 - associating debug label with, 163–165
 - header file, 105–106
 - implementation file, 106–111, 138–140
 - importing updates for Viking, 141–144
 - `loadAudio` method, 201–203
 - overview of, 105
- `GameplayScrollingLayer` class
 - `adjustLayer` method, 245–247
 - connecting joystick and button controls to Viking, 245
- subclass of `CCLayer`, 243–245
- `update` method, 247–248
- `GameState` class
 - adding to Space Viking project, 530–534
 - creating to track user achievements, 519–521
- `GCDatabase`, loading/saving achievement data to, 522–524
- `GCHelper`. *See also* Helper methods
 - adding to Space Viking project, 530–534
 - creating helper class for Game Center, 508–510
 - implementing leaderboards, 539–540
 - keeping track of player authentication status, 511–512
 - modifying for sending achievements, 524–530
- `GetWorldPoint` helper method, 379–380
- `GKScore` object, creating, 538
- `GLES-Render` files, 295–296
- `Glyph` Designer, creating font texture atlas, 156
- GPU, checking performance of, 560–563
- Gravity property
 - initializing in Chipmunk space, 431
 - Particle Designer controlling, 488
- Groove joint
 - constraints in Chipmunk, 456
 - creating spring platforms and, 463–464
- Ground
 - creating for Chipmunk space, 432–433
 - detecting collisions with, 445–450
 - setting collision type for, 447–448
 - surface velocity, 445
- `GroundLayer`, of `TileMap`, 269
- `groupIndex` field, 382
- Groups
 - creating for scenes, 236
 - organizing classes by, 70
 - organizing scenes, 180–181

H

- Header files
 - additions for audio to, 204–205
 - cart, 337–338
 - enemy robot, 126–127

- Header files (*continued*)
- game manager, 172–173
 - `GameplayLayer` class, 105–106
 - health, 121
 - Main Menu, 182–183
 - mallet, 118
 - Ole the Viking, 90–92
 - phaser, 138
 - `PlatformScene`, 263–264
 - `PlatformScrollingLayer`, 254–255
 - radar dish, 84
 - space cargo ship, 123
- Health (`Health` class)
- in class hierarchy, 65
 - enemy robot, 134
 - moving into dead state, 103
 - power-up, 120–122
 - restoring Ole's health, 102
- HelloWorld apps
- adding movement to cargo ship, 10–11
 - adding space cargo ship to app, 9–10
 - adding to iPhone or iPad, 20–21
 - `applicationDidFinishLaunching` method in, 14–15
 - building, 7–9
 - Director's role in running game loop and rendering graphics, 16–18
 - Hello, Box2D, 289–292
 - initializing `UIWindow`, 15–16
 - inspecting Cocos2D templates, 6–7
 - scenes and nodes in application template, 11–14
- Helper methods
- for creating animations, 411
 - for loading/saving achievement data, 522–524
 - overview of, 19–20
 - for sending achievement data, 524–530
- Hiero Font Builder Tool, 156–159
- |
- Idle state (`kStateIdle`)
- for enemy robot, 132
 - for radar dish, 87
- `if` statement, combining with assignment operator, 134–135
- Images
- adding for menus, 181–182
 - adding to Space Viking project, 24–26
 - advantages of texture atlases for, 47
 - loading image files, 43
 - performance tips and, 551
- Implementation files
- additions to for audio, 204–205
 - enemy robot, 127–137
 - game manager (`GameManager`), 174–177
 - `GameplayLayer` class, 106–111
 - `gameplayLayer` class, 138–140
 - health, 121
 - `@interface` declaration in, 256
 - mallet, 119–120
 - phaser, 138–140
 - `PlatformScene`, 263–264
 - radar dish, 85–89
 - space cargo ship, 123–125
- Implementation files, `Viking` class
- `changeState` method for animation, 95–98
 - `dealloc` method, 94
 - effect of pragma mark statements in Xcode pulldown menus, 99
 - flipping graphic views, 95
 - `initAnimations` method, 103–105
 - joystick methods, 94–95
 - `updateStateWithDeltaTime` and `adjustedBoundingBox` methods, 100–103
- Importing updates, for Space Viking project, 141–144
- Impulses
- Box2D, 368
 - controlling flipping of cart, 368–369
 - making cart jump, 369–373
 - for responsive direction switching, 373–374
- Infinite scrolling
- creating group for cut-scene, 252–253
 - creating platform scene, 263–265
 - creating scrolling layer in cut-scene, 254–262
 - creating texture atlas for cloud images, 254
 - overview of, 252–253

- Inheritance, class hierarchy and, 64–65
- `init` method
 - for Chipmunk, 429–430
 - in game layer of Space Viking, 41–42
 - for HelloWorld app, 13
 - for PerformanceTestGame, 546–549
 - of `platformScrollingLayer`, 255–256
- `initAnimations` method
 - `Mallet` class, 120
 - `RadarDish` class, 88–89
 - `Viking` class, 103–105
- `InitAudioAsync` method, `GameManager` class, 206–207
- `initJoystick` method, 108
- `InitWithScene4UILayer` method, carts, 344
- Installing Cocos2D templates, 5–6
- Instance variables
 - adding to `CPViking`, 450–451
 - for sprite positions, 43
- Instruments tool
 - for checking GPU, 560–563
 - for finding bottlenecks, 557–558
- Interaction, in Box2D, 302–304
- `@interface` declaration, inside implementation files, 256
- `Intro` class, setting up menus, 190
- `IntroLayer` class, images displayed before game play, 193
- iOS
 - audio framework for, 197
 - fonts available in, 155
 - Game Center app, 495
 - making games in, xxii
- iOS Developer Program account, 497–498
- iPad
 - adding HelloWorld app to, 20–21
 - power-of-two support on older devices, 46
 - running performance test game on, 546, 550
 - running Space Viking on iPad Simulator, 144–145
 - simulator in Particle Designer, 485–486
- iPhone
 - adding HelloWorld app to, 20–21
 - fixing slow performance on older devices, 53–54
 - power-of-two support on older devices, 46
 - simulator in Particle Designer, 485–486
- iPhone Developer account, 20
- iPhone Developer Portal, 497
- iPod, power-of-two support on older devices, 46
- `isCarryingWeapon` method, `Viking` class, 94
- iTunes Connect
 - adding achievements to, 515–517
 - registering apps in, 497, 501–505
 - setting up leaderboards in, 536–538

J

- JavaScript, Cocos2D-JavaScript, 568
- Joints
 - adding wheels using revolute joints, 352–355
 - breaking Ole's body into pieces, 376
 - compared with Chipmunk constraints, 420–421
 - creating multiple bodies and joints, 378–380
 - dragging objects in Box2D, 304
 - motor settings for revolute joint, 385
 - prismatic, 378
 - restricting revolute joints, 376–377
- Joysticks
 - adding, 36–40
 - applying joystick movement, 40–44
 - connecting to Space Viking, 245
 - importing joystick class, 35–36
 - initializing, 111
 - `initJoystick` method, 108
 - `Viking` class methods, 94–95
- JPEG files, 43
- Jump buttons, adding `GameplayLayer` class, 108
- Jump to Definition (Ctrl-z-D), for viewing source code, 18–19
- Jumping, in Chipmunk
 - implementing, 450–455
 - by setting velocity, 444

K

Kinematic bodies, Box2D, 293
kStateDead. *See* Dead state (**kStateDead**)
kStateIdle (idle state)
 for enemy robot, 132
 for radar dish, 87
kStateSpawning (Spawning state)
 for enemy robot, 132
 for radar dish, 87
kStateTaking Damage (Taking damage state), 87
kStateTakingDamage. *See* Damage taking state (**kStateTakingDamage**)

L

Labels (**CCLabel**)
 adding to scenes, 13–14
 CCLabelIBMFont class, 155, 159
 CCLabelTTF class, 151–155
 in layers, 12
Layers (**CCLayer**)
 adding, 29–31
 allocating sprites when layer is initialized, 552
 connecting background and game layers to a scene, 31–32
 creating background layer, 26–29
 GameControlLayer as subclass of, 239–242
 GameplayScrollingLayer as subclass of, 243–245
 principal classes in Cocos2D, 570
 Scene4UILayer as subclass of, 335
 scenes as container for, 12, 33
 z values and, 33
Leaderboards
 displaying, 540–542
 how they work, 538
 implementing, 539–540
 overview of, 536
 setting up in iTunes Connect, 536–538
LevelComplete class
 scene types and, 170
 setting up menus, 190

LevelCompleteLayer class
 achievements and, 530–534
 displaying leaderboards, 540–542
 scenes and, 194–195

Levels

accounting for level width when scrolling, 233–234
 creating in Chipmunk, 432–433
 creating with LevelSVG tool, 380
 getting dimension of current level, 232–233

LevelSVG tool, 380

Linear impulses, Box2D

making cart jump, 369–373
 overview of, 368

LinkTypes, URLs and, 172

loadAudio method, **GameplayLayer** class, 201–203

Loops

Director running, 16–18
 update loop, 279–280, 394–396
 variable and fixed rate timestamps and, 394–396, 434

Loosely coupled classes, 117–118

lowerAngle method, restricting revolute joints, 377

M

Mac OS X
 Cocos2D native support for, 568
 downloading particle system to, 485
 making games in, xxii

Macros, 20

Main Menu (**MainMenu**)
 adding music to, 228–229
 creating, 182–190
 header file for, 182–183
 MainMenuLayer class, 183–190
 scene types and, 169

Mallet (**Mallet** class)
 dropping from space cargo ship, 125
 powering up, 102, 118–120

Manager. *See* Game manager

(**GameManager**)

mass property, fixtures, 309–313

- Mekanimo tool, for working with bodies, 380
- Member variables, for body part sprites, 380–381
- M**
- Memory
- benefits of texture atlases, 48
 - managing memory footprint, 17
 - textures and, 45–47
- Menus
- adding images and fonts for, 181–182
 - in addition to Main Menu, 190
 - classes in, 179–180
 - Main Menu. *See* Main Menu (MainMenu)
 - Options Menu, 170
- Meters, converting points to, 420–421, 430
- Methods, declaring in Objective-C, 117
- Metroid-style platform. *See* Platforms, in Chipmunk
- Motors
- in Chipmunk, 456
 - settings for revolute joint, 385
- Mouse joint
- dragging objects in Box2D, 304–309
 - supporting in Chipmunk, 436–437
- Movement
- adding movement to cargo ship, 10–11
 - adding to Space Viking project, 35
 - implementing in Chipmunk, 450–455
 - jumping, 444
 - surface velocity, 445
- Music. *See also* Audio
- adding in Chipmunk, 473–474
 - adding to GameplayLayer, 228
- N**
- New Group (Option-z-N), 70
- Nodes (CCNode)
- in application templates, 11–14
 - in Cocos2D hierarchy, 12
 - principal classes in Cocos2D, 570
 - tags, 71–72
- Normal platform, creating in Chipmunk, 464–466
- NSCoding protocol, loading/saving achievement data to GCDatabase, 522–524
- NSDictionary objects, storing animation settings in, 67
- NSOperationQueues
- adding audio asynchronously in background thread, 201
 - managing background threads, 204
- NSTimer, scheduler compared with, 145
- numberOfRunningActions, 135
- O**
- Object-Oriented Programming* (Coad and Nicola), 63
- Objective-C framework
- Cocos2D and, xxii, xxv
 - protocols in, 117
- Objects. *See also* Game objects (GameObject)
- adding sound to, 215–216
 - converting into classes, 63
 - creating Box2D, 292–295
 - creating C++, 280
 - creating game objects, 74–80
 - GameObject in class hierarchy, 64–65
 - GKScore object, 538
 - Mallet class inheriting from GameObject class, 119
 - plist files and, 67
 - positioning using anchor points, 153
 - positioning using point system, 82
 - returning to nonaltered state after effects, 146–148
 - update method added to, 443
 - use in design, 63
- Obstacles, creating spikes in Box2D, 390–394
- Offsets, restricting prismatic joints, 379–380
- Ole the Viking. *See also* Viking class
- adding sounds to, 222–228
 - adding subclass for, 440–443
 - adjustedBoundingBox method, 100–103
 - animating in Chipmunk, 469–473
 - breaking body into pieces using joints, 376
 - changeState method, 95–98

- Ole the Viking (*continued*)
 connecting button controls to, 245
 creating cinematic fight sequence, 411–416
 following in Chipmunk, 467–468
 Header files, 90–92
 leaping with ragdoll effect, 381–386
 pitting against Digger in fight, 396–405
 restoring health of, 102
- OpenAL audio framework, for iOS devices, 197
- OpenGL Driver Instrument, for checking GPU, 560–563
- OpenGL ES
 benefits of batching bind calls, 45
 converting UIKIT to OpenGL ES coordinate system, 29
EAGLView and, 16
 FBO (frame buffer object), 145–146
 for graphics rendering in Cocos2D, xxii
 references for, 567
 support in Cocos2D, xxv
- Option-z-N (New Group), 70
- Options Menu
 scene types and, 170
 setting up menus, 190
- OptionsLayer, displaying achievements within apps, 534–536
- Organizing source code
 constants file for static values used in more than one class, 71–72
 grouping classes, 70
 protocols in implementation of class methods, 72–74
- P**
- Parallax scrolling
 adding background to cart layer, 364–368
 defined, 231
 overview of, 250–252
- ParallaxBackgrounds folder, importing, 235–236
- Particle Designer
 application in cinematic fight sequence, 413
 controls, 487–488
 creating particle system, 489–490
- downloading to Mac, 485
 engine exhaust effect, 490–494
 features of, 486–488
 toolbar, 486
- Particle systems
 creating, 489–490
 engine exhaust added to space cargo ship, 490–494
 running built-in system, 482–483
 snow effect, 483–485
 summary and challenges, 494
 terminology related to, 481–482
 tour of Particle Designer, 486–488
- Particles
 controls for, 487–488
 defined, 481
- Performance optimization
 bottlenecks and, 557–558
 capturing CPU utilization data, 558–560
 CCSprite vs. CCSpriteBatchNode, 545–550
 checking GPU, 560–563
 on older devices, 53–54
 overview of, 545
 profiling tool for, 554–557
 reusing sprites, 552–554
 running performance test game, 550
 summary and challenges, 563
 textures and texture atlases and, 551–552
- PerformanceTestGame
 adding profiling tool to, 554–557
 capturing CPU utilization data, 558–560
 checking GPU, 560–563
 init method, 546–549
 opening and running on iPad, 545–546
 reusing sprites, 552–554
 running, 550
 update method for, 549–550
- Phaser (*Phaser* class)
 adding phaser bullet, 137–141
 createPhaserWithDirection method, 142–143
 header file, 138
 implementation file, 138–140
 placeholder for creating phaser blast, 109
 protocols for creating in GameplayLayer class, 127

- shootPhaser method, 129, 132–133
- taking damage from, 102
- Physics Editor**, 380
- Physics engines**
 - advanced. *See* Box2D, advanced physics
 - basic. *See* Box2D, basic physics
 - bundled with Cocos2D, xxiv
 - Chipmunk. *See* Chipmunk
 - intermediate. *See* Box2D, intermediate physics
- Pin joint**, constraints in Chipmunk, 456
- Pivot joint**
 - constraints in Chipmunk, 455
 - creating pivot platform, 462
- Pixels**, in object positioning, 82
- Platforms**, in Chipmunk
 - laying out, 468–469
 - normal platform, 464–466
 - pivot platform, 460–462
 - revolving platform, 458–460
 - spring platform, 463–464
- PlatformScene**
 - header and implementation files, 263–264
 - playScene method, 264
- platformScrollingLayer**
 - ccTouchesBegan method, 262
 - createCloud method, 257–258
 - createStaticBackground method, 257
 - createVikingAndPlatform method, 261–262
 - declarations and init method, 255–256
 - header file, 254–255
 - resetCloudWithNode method, 258–261
 - playbackgroundTrack method, audio, 213–214
 - playScene method, platform scene, 264
 - playSoundEffect method, audio, 213–214
 - plist files
 - phaser bullet effect, 137
 - sound effects in, 198–201
 - storing animation data in, 61, 67–69
 - PNG files
 - file formats for images, 43
 - using in TileMap, 270
 - Point-to-meter (PTM) ratio, 420–421, 430
 - Pointers, C++, 281
- Points**
 - converting to meters, 420–421, 430
 - in object positioning, 82
- postSolve events**, collision events in Chipmunk, 445
- Power-of-two**, textures and, 45–47
- Power-up objects**
 - Health class, 120–122
 - Mallet class, 118–120
 - overview of, 118
 - protocols for creating in GameplayLayer class, 127
- Pragma mark statements**, in Xcode pulldown menus, 99
- preSolve events**, collision events in Chipmunk, 445, 448–449
- Prismatic joints**
 - Chipmunk groove joint compared with, 456
 - offsets for restricting, 379
 - overview of, 378
- Profiling**
 - capturing CPU utilization data, 558–560
 - finding bottlenecks, 557–558
 - for performance optimization, 554–557
- Project setup**
 - background and game layers connected to a scene, 31–32
 - background layer created, 26–29
 - CCSpriteBatchNode in, 52–53
 - classes for, 24–26
 - creating new project, 23–24
 - director running game scene, 34–35
 - fixing slow performance on older devices, 53–54
 - game layer added, 29–31
 - game scene for, 32–33
 - joystick class imported for, 35–36
 - joystick movement in, 40–44
 - joysticks and buttons added, 36–40
 - movement added, 35
 - summary and challenges, 54–55
 - texture atlas added to scene, 48–51
- Property list files.** *See* plist files
- Protocols**
 - common protocol class, 72–74
 - in implementation of class methods, 72–74

Protocols (*continued*)
 in Objective-C, 117
 use with enemy robot, 127

PTM (point-to-meter) ratio, 420–421, 430

Puzzle game example, in Box2D, 320–324

PuzzleLayer
 box created for, 293
 createBoxAtLocation method, 294–295
 debug drawing, 295–296
 decorating bodies using sprites, 313–320
 dragging objects in Box2D, 304–309
 ground body created for, 299–302
 interaction and decoration in, 302–304
 mass, density, friction, and restitution properties, 309–313
 puzzle game example, 320–324
 scene created for, 282–284
 viewing on screen, 296–298
 world created for, 290–292

PVR TC
 compression format, 43
 performance tips for textures, 551

Q

Queries, searching for objects in Box2D world, 305–307

R

RadarDish class
 adding sounds to, 216–217
 changeState method, 86
 in class hierarchy, 64
 header file, 84
 implementation file, 85–89
 inheriting from **GameCharacter** class, 84–85
 initAnimations method, 88–89
 plist files for, 68–69
 steps in creation of, 83–84
 updateStateWithDeltaTime method, 86

Ragdoll effect
 adding action to Ole, 376
 leaping effect and, 381–386

resetCloudWithNode method, **platform-ScrollingLayer**, 258–261

Responsive direction switching, Box2D, 373–374

restitution property, fixtures, 309–313

Revolute joints

- in Box2D, 304
- for bridge, 386–389
- for cart wheels, 352–355
- Chipmunk** pivot joint compared with, 455
- for Digger Robot wheels, 400
- motor settings for, 385
- restricting, 376–377

Revolving platform, creating in Chipmunk, 458–460

Rigid body physics simulation, 292

Robots

- Digger Robot. *See* Digger Robot
- enemy robot. *See* Enemy robot

RockBoulderLayer, of TileMap, 270

RockColumnsLayer, of TileMap, 269

RootViewController, for device orientation, 53–54

Rotary limit joint

- constraints in Chipmunk, 456
- creating pivot platform, 462

Rotation, anchor points for, 154

Running state, Digger Robot, 407

S

Sandbox accounts, in Game Center, 514

Scaling images, performance tips and, 551

Scenes (CCScenes)

- ActionLayer** of Scene4, 339–343
- adding images and fonts, 181–182
- additional menu types, 190
- in application template, 11–14
- background and game layers connected to, 31–32
- basic Box2D scene, 335–346
- basic Chipmunk scene, 429–438
- CCScenes** as principal class in Cocos2D, 570
- changing **SpaceVikingAppDelegate** to support game manager, 192–193

classes in menu system, 179–180
 creating for Space Viking, 32–33
 creating new Chipmunk scene, 421–425
 creating new scene (`PuzzleLayer`), 282–284
 creating second game scene, 236–242
 director running game scene, 34–35
 game manager connected to Chipmunk scene, 425–426
 game manager for switching between, 170–172
`GameplayLayer` class and, 190–192
 group for cut-scene, 253
 group for scene2, 236
`IntroLayer` class and, 193
`LevelCompleteLayer` class, 194–195
 Main Menu, 182–190
 organizing, 180–181
 texture atlases for, 48–51
 types of, 169–170
`UILayer` of `Scene4`, 335–336
`SceneTypes`, 172
 Scheduler, for timed events and call, 145
 Scores, sending to Game Center, 538
 Screen resolution, support in Cocos2D, xxv
 Screen shake effect, 467–468
 Scrolling. *See also* Tile maps
 accounting for level width, 233–234
 background, 271–272
 common problems in, 234–235
 creating scrolling layer, 242–249
 in cut-scene, 254–262
 getting dimension of current level, 232–233
 to infinity, 252–253
 new scene for, 236–242
 overview of, 231
 parallax layers and, 250–252
 in platform scene, 263–265
`ScrollingCloudsBackground` folder, 254
`ScrollingCloudsTextureAtlases` folder, 254
 Selection techniques, three-finger swipe, 27
 separate events, collision events in Chipmunk, 445, 448–449
`SetLinearVelocity` function, for Box2D bodies, 415
`setupDebugDraw` method, cart, 344
`setupWorld` method, cart, 344
 Shapes, Chipmunk
 adding, 420
 box shaped added, 431–433
 converting dynamic shape into static platform, 447–448
 Shapes, custom shapes with Box2D, 346–348
`ShootPhaser` method, 129
 Simple motor, constraints in Chipmunk, 456
`SimpleAudioEngine`, in CocosDenshion, 197, 229–230
 Singletons
 for game manager, 195
 important concepts in Cocos2D, xxiv
 SneakyInput joystick project, 35–36
 Snow effect, creating with particle system, 483–485
 Social gaming network. *See* Game Center
 Sound effects. *See also* Audio
 adding in Chipmunk, 473–474
 getting list of, 208–211
 loading, 211–213
 in plist files, 198
 Sounds folder, 198
 Source code
 availability of, 18–20
 constants file and, 71–72
 grouping classes and, 70
 protocols in implementation of class methods, 72–74
 Space cargo ship (`SpaceCargoShip` class)
 adding sounds to, 217–219
 in class hierarchy, 64
 creating, 122
 engine exhaust effect for, 490–494
 header file, 123
 implementation file, 123–125
 Space Viking
 basic setup. *See* Project setup
 downloading, xxi
 introduction to, xxv–xxvi
 storyline in, xxvi–xxvii
 Spaces, Chipmunk
 box added to, 431–432
 creating, 429–431
 creating a physics world, 420
 creating the level and ground, 432–433

- S**
- SpaceVikingAppDelegate, 192–193
 - Spawning state (`kStateSpawning`)
 - for enemy robot, 132
 - for radar dish, 87
 - Spikes, creating obstacles in Box2D, 390–394
 - Spring platform, creating in Chipmunk, 462–463
 - Sprite Frame Cache, 111
 - SpriteBatchNode, 88
 - Sprites (`CCSprite`)
 - allocating during layer initialization, 552
 - animating sprites rendered by `CCSpriteBatchNode`, 60–61
 - basic animation, 57–60
 - batching (`CCSpriteBatchNode`), 44–45
 - Box2D bodies, 380–381
 - `CCSprite` as principal classes in Cocos2D, 570
 - `CCSprite` vs. `CCSpriteBatchNode`, 545–550
 - containing within screen boundaries, 102
 - decorating bodies, 313–320
 - important concepts in Cocos2D, xxiv
 - in layers, 12
 - listed in `CCSpriteBatchNode` object, 108
 - returning to nonaltered state after effects, 146–148
 - reusing, 552–554
 - Sprite Frame Cache, 111
 - Sprites (`CPSprite`)
 - adding, 438
 - defining body and shape for, 438–440
 - implementing velocity of, 444
 - subclass for Ole, 440–443
 - subclass for revolving platform, 458–460
 - startFire method, 415
 - State transitions. *See also* Animation
 - Ole the Viking, 471–473
 - radar dish, 86–87
 - spike obstacle and, 392–393
 - Viking class and, 95–98
 - visual effects and, 410
 - Static bodies, Box2D, 293
 - StaticBackgroundLayer, splitting background into static and scrolling layers, 237–239
 - stopSoundEffect method, audio, 213–214
 - Surface velocity, for ground movement in Chipmunk, 445
 - switch statement
 - method variables not declared in, 99
 - state transitions and, 98
- T**
- Taking damage state. *See* Damage taking state (`kStateTakingDamage`)
 - Teleport graphic, for enemy robot, 132
 - Templates
 - inspecting, 6–7
 - installing, 5–6
 - SpaceViking based on, 23–24
 - working of scenes and nodes in application template, 11–14
 - Ternary operator (?), combining if statement with assignment operator, 134–135
 - Text. *See also* Labels (`CCLabel`)
 - adding Game Start banner, 152–153
 - anchor points for Game Start banner, 153–154
 - `CCLabelIBMPFont` class, 155, 159
 - `CCLabelTTF` class, 151
 - creating debug label, 160–165
 - font texture atlas for, 156–159
 - fonts, 155
 - Texture atlases
 - `CCSpriteBatchNode` initialized with image from, 111
 - for cloud images, 254
 - combining textures into, 44
 - downloading tiles texture atlas, 266
 - for fonts, 156–159
 - overview of, 44–45
 - performance optimization and, 551–552
 - reasons for using, 48
 - for Space Viking Scene 1, 48–51
 - steps in use of, 53
 - technical details of, 45–47
 - Texture padding, 45–47
 - TexturePacker
 - creating texture atlas for Space Viking Scene 1, 49–51

- texture atlas software, 47
- trial version, 380
- Textures**
 - caching, 17
 - combining into texture atlases, 43
 - flushing unused, 552
 - loading images into RAM and, 43
 - performance optimization and, 551–552
- Tile maps.** *See also* **Scrolling**
 - adding to `ParalaxNode`, 272–275
 - built-in support for, xxiv
 - compressed `TiledMap` class, 271–272
 - creating, 267–268
 - defined, 232
 - installing `Tiled` tool on Mac, 266–267
 - overview of, 265–266
 - three-layered, 268–270
- Tiled tool**
 - creating three-layered tile map, 268–270
 - creating `TileMap` for iPad, 267–268
 - Installing on Mac, 266–267
- Tiles**
 - defined, 231
 - of repeating images, 265–266
- TileSets**, 232
- Time Profiler**, for capturing CPU utilization data, 558–560
- TMX files**, 270–271
- Touch-handling code**, helper methods for, 436–437
- typedef enumerator**
 - getting list of sound effects, 210–211
 - for left and right punches, 92

- U**
- UIFont** class, 155
- UIKit**, converting to OpenGL ES coordinate system, 29
- UILayer**, in Chipmunk, 421–423
- UIViewController**, for device orientation, 53–54
- UIWindow**, initialization of, 15–16
- Units**
 - `Box2D`, 288–289
 - converting points to meters in Chipmunk, 430

- V**
- Variable rate timestamps**, 394–396
- Variables**, method variables not declared in switch statement, 99
- Vectors**, for direction and magnitude, 291
- Velocity**
 - of ground movement in Chipmunk, 445
 - of sprite in Chipmunk, 444
- Vertex Helper**
 - creating vertices for Digger Robot, 399–401
 - creating vertices with, 348–352
 - making cart scene scrollable, 359–362
- Vertices**
 - for `Box2D` shapes, 347–348
 - creating shapes with arbitrary vertices, 448
 - for Digger Robot, 399–401
 - making cart scene scrollable, 359–362
 - Vertex Helper and, 348–352
- View controller**
 - displaying achievements, 534–536
 - displaying leaderboards, 540–542

Viking class. *See also* Ole the Viking
 adding sounds to, 222–228
 applying joystick movement to, 40–44
 changeState method for animations in,
 95–98
 checking to *see* if attacking enemy robot,
 135
 in class hierarchy, 64
`dealloc` method, 94
 effect of pragma mark statements in
 Xcode pulldown menus, 99
 flipping graphic views, 95
 header file, 90–92
`initAnimations` method, 103–105
 joystick methods, 94–95
 pitting Digger against Ole, 396–405
 referencing from `SpriteBatchNode`, 88
 retrieving from `CCSpriteSheet`,
 134–135
 subclass for in Chipmunk, 440–443
`updateStateWithDeltaTime` and
`adjustedBoundingBox` methods,
 100–103
 in Xcode, 90
 Visual effects, state transitions and, 410

W

Walking state
 Digger Robot, 407–408
 enemy robot, 132

Wheels
 adding to cart, 352–355
 creating for Digger Robot, 399–400
 Win/lose conditions, adding in Chipmunk,
 476–477

World, Box2D
 Chipmunk space compared with, 430
 creating, 289–292
 searching for objects in, 305–307

X

Xcode
 Add New File dialog, 26
 build management for iOS devices, 20–21
 Chipmunk files added to Xcode project,
 426–429
 classes as organization technique in, 70
 HelloWorld app, 7–9
 inspecting Cocos2D templates, 6–7
 Instruments tool, 557–558
 location of Cocos2D templates in, 23–24
 pragma mark statements in pull down
 menus, 99
 RadarDish class created with, 83–84
 texture atlases added to, 51

Z

z values, in 3D engines, 33
 Zwoptex, 47–49