

Use cutting-edge tools to create exciting
iPhone and iPad game apps

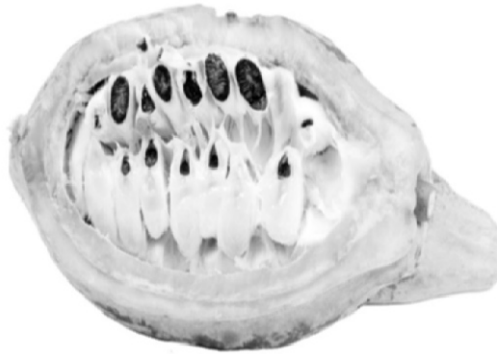


Learn
cocos2d
Game Development
with iOS 5

Steffen Itterheim | Andreas Löw

Apress®

Learn cocos2D Game Development with iOS 5



Steffen Itterheim
Andreas Löw

Apress®

Learn cocos2D Game Development with iOS 5

Copyright © 2011 by Steffen Itterheim and Andreas Löw

All rights reserved. No part of this work may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage or retrieval system, without the prior written permission of the copyright owner and the publisher.

ISBN 978-1-4302-3813-3

ISBN 978-1-4302-3814-0 (eBook)

Trademarked names, logos, and images may appear in this book. Rather than use a trademark symbol with every occurrence of a trademarked name, logo, or image we use the names, logos, and images only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

The use in this publication of trade names, trademarks, service marks, and similar terms, even if they are not identified as such, is not to be taken as an expression of opinion as to whether or not they are subject to proprietary rights.

President and Publisher: Paul Manning

Lead Editor: Steve Anglin

Development Editor: Chris Nelson

Technical Reviewer: Boon Chew

Editorial Board: Steve Anglin, Mark Beckner, Ewan Buckingham, Gary Cornell, Jonathan Gennick, Jonathan Hassell, Michelle Lowman, Matthew Moodie, Duncan Parkes, Jeffrey Pepper, Frank Pohlmann, Douglas Pundick, Ben Renow-Clarke, Dominic Shakeshaft, Matt Wade, Tom Welsh

Coordinating Editor: Kelly Moritz

Copy Editors: Kim Wimpsett

Compositor: MacPS, LLC

Indexer: SPi Global

Artist: SPi Global

Cover Designer: Anna Ishchenko

Distributed to the book trade worldwide by Springer Science+Business Media, LLC., 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com.

For information on translations, please e-mail rights@apress.com, or visit www.apress.com.

Apress and friends of ED books may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Special Bulk Sales—eBook Licensing web page at www.apress.com/info/bulksales.

The information in this book is distributed on an “as is” basis, without warranty. Although every precaution has been taken in the preparation of this work, neither the author(s) nor Apress shall have any liability to any person or entity with respect to any loss or damage caused or alleged to be caused directly or indirectly by the information contained in this work.

The source code for this book is available to readers at www.apress.com and www.learn-cocos2d.com/store/book-learn-cocos2d.

*To Gabi, the one and only space ant.
Sometimes alien, often antsy, always loved.*

(Steffen)

*To Saskia & Renate for making it possible to
spend my time with things I love most.*

(Andreas)

Contents at a Glance

Contents.....	V
About the Authors.....	xiii
About the Technical Reviewer	xiv
Acknowledgments	xv
Preface	xvi
Chapter 1: Introduction	1
Chapter 2: Getting Started.....	15
Chapter 3: Essentials.....	41
Chapter 4: Your First Game	81
Chapter 5: Game Building Blocks	115
Chapter 6: Sprites In-Depth	141
Chapter 7: Scrolling with Joy	169
Chapter 8: Shoot em Up.....	195
Chapter 9: Particle Effects.....	217
Chapter 10: Working with Tilemaps	243
Chapter 11: Isometric Tilemaps	269
Chapter 12: Physics Engines	297
Chapter 13: Pinball Game	321
Chapter 14: Game Center.....	365
Chapter 15: Cocos2d with UIKit Views	401
Chapter 16: Kobold2D Introduction	439
Chapter 17: Out of the Ordinary.....	467
Index	495

Contents

Contents at a Glance	iv
About the Authors	xiii
About the Technical Reviewer	xiv
Acknowledgments	xv
Preface	xvi

Chapter 1: Introduction	1
What's New in the Second Edition?	2
Why Use cocos2d for iOS?	3
It's Free	3
It's Open Source	3
It's Objective, See?	3
It's 2D	4
It's Got Physics	4
It's Less Technical	4
It's Still Programming	5
It's Got a Great Community	5
The Future of the cocos2d-iphone Project	6
Other cocos2d Game Engines	7
This Book Is for You	8
Prerequisites	8
Programming Experience	8
Objective-C	8
What You Will Learn	9
What Beginning iOS Game Developers Will Learn	10
What iOS App Developers Will Learn	10
What Cocos2d Developers Will Learn	11
What's in This Book	11
Chapter 2, Getting Started	11
Chapter 3, Essentials	11
Chapter 4, Your First Game	11
Chapter 5, Game Building Blocks	12

Chapter 6, Sprites In-Depth	12
Chapter 7, Scrolling with Joy	12
Chapter 8, Shoot 'em Up	12
Chapter 9, Particle Effects	12
Chapter 10, Working with Tilemaps	12
Chapter 11, Isometric Tilemaps	12
Chapter 12, Physics Engines	12
Chapter 13, Pinball Game	13
Chapter 14, Game Center	13
Chapter 15, Cocos2d with UIKit Views	13
Chapter 16, Kobold2D Introduction	13
Chapter 17, Conclusion	13
Where to Get the Book's Source Code?	13
Questions and Feedback.....	14
Chapter 2: Getting Started	15
What You Need to Get Started	15
System Requirements.....	15
Register as an iOS Developer	16
Certificates and Provisioning Profiles	16
Download and Install the iOS SDK	17
Download and Install cocos2d	17
The HelloWorld Application	21
Locating the HelloWorld Files	22
Resources	23
Supporting Files.....	23
HelloWorld Classes	24
Memory Management with cocos2d.....	29
Changing the World	32
What Else You Should Know	34
The iOS Devices	34
About Memory Usage.....	36
The iOS Simulator	37
About Logging.....	39
Summary	39
Chapter 3: Essentials.....	41
The cocos2d Scene Graph	41
The CCNode Class Hierarchy	44
CCNode	46
Working with Nodes.....	46
Working with Actions.....	47
Scheduled Messages.....	47
Director, Scenes, and Layers	51
The Director	51
CCScene.....	53
Scenes and Memory	54
Pushing and Popping Scenes	55
CCTransitionScene.....	57

CCLayer	59
CCSprite	64
Anchor Points Demystified	65
Texture Dimensions	65
CCLabelTTF	66
Menus	67
Actions	69
Interval Actions	70
Instant Actions	76
A Note on Singletons in cocos2d	78
Cocos2d Test Cases	80
Summary	80
Chapter 4: Your First Game	81
Step-by-Step Project Setup	82
Adding the Player Sprite	87
Accelerometer Input	91
First Test Run	91
Player Velocity	92
Adding Obstacles	95
Collision Detection	101
Labels and Bitmap Fonts	103
Adding the Score Label	103
Introducing CCLabelBMFont	104
Creating Bitmap Fonts with Glyph Designer	105
Simply Playing Audio	107
Porting to iPad	109
One Universal App or Two Separate Apps?	109
Porting to iPad with Xcode 3	110
Porting to iPad with Xcode 4	111
Summary	113
Chapter 5: Game Building Blocks	115
Working with Multiple Scenes	115
Adding More Scenes	115
Loading Next Paragraph, Please Stand By	118
Working with Multiple Layers	121
How to Best Implement Levels	126
CCLayerColor	128
Subclassing Game Objects from CCSprite	128
Composing Game Objects Using CCSprite	129
Curiously Cool CCNode Classes	134
CCProgressTimer	134
CCParallaxNode	135
CCRibbon	138
CCMotionStreak	139
Summary	140

Chapter 6: Sprites In-Depth	141
Retina Display	142
CCSpriteBatchNode	144
When to Use CCSpriteBatchNode	145
Demo Projects	146
Sprite Animations the Hard Way	152
Animation Helper Category	154
Working with Texture Atlases	156
What Is a Texture Atlas?	156
Introducing TexturePacker	157
Preparing the Project for TexturePacker	158
Creating a Texture Atlas with TexturePacker	159
Using the Texture Atlas with cocos2d	163
Updating the CCAnimation Helper Category	165
All into One and One for All	166
Summary	167
Chapter 7: Scrolling with Joy	169
Advanced Parallax Scrolling	169
Creating the Background As Stripes	169
Re-creating the Background in Code	172
Moving the ParallaxBackground	174
Parallax Speed Factors	175
Scrolling to Infinity and Beyond	178
Fixing the Flicker	180
Repeat, Repeat, Repeat	181
A Virtual Joypad	182
Introducing SneakyInput	183
Integrating SneakyInput	184
Touch Button to Shoot	185
Skinning the Button	187
Controlling the Action	190
Digital Controls	193
Summary	193
Chapter 8: Shoot em Up	195
Adding the BulletCache Class	195
What About Enemies?	199
The Entity Class Hierarchy	201
The EnemyEntity Class	201
The EnemyCache Class	205
The Component Classes	209
Shooting Things	211
A Healthbar for the Boss	213
Summary	215
Chapter 9: Particle Effects	217
Example Particle Effects	217
Creating a Particle Effect the Hard Way	221
Subclassing CCParticleSystem: Point or Quad?	222

CCParticleSystem Properties	224
Particle Designer.....	234
Introducing the Particle Designer	234
Using Particle Designer Effects.....	237
Sharing Particle Effects	239
Shoot 'em Up with Particle Effects	240
Summary	242
Chapter 10: Working with Tilemaps	243
What Is a Tilemap?	243
Preparing Images with TexturePacker.....	247
Tiled (Qt) Map Editor	248
Creating a New Tilemap.....	248
Designing a Tilemap	251
Using Orthogonal Tilemaps with Cocos2d	254
Locating Touched Tiles	257
An Exercise in Optimization and Readability.....	260
Working with the Object Layer.....	260
Drawing the Object Layer Rectangles.....	262
Scrolling the Tilemap.....	265
Summary	266
Chapter 11: Isometric Tilemaps	269
Designing Isometric Tile Graphics	270
Isometric Tilemap Editing with Tiled.....	273
Creating a New Isometric Tilemap.....	273
Creating a New Isometric Tileset.....	276
Laying Down Some Ground Rules.....	276
Isometric Game Programming	278
Loading the Isometric Tilemap in Cocos2d	278
Setup Cocos2d for Isometric Tilemaps	279
Locating an Isometric Tile.....	281
Scrolling the Isometric Tilemap	283
This World Deserves a Better End.....	284
Adding a Movable Player Character.....	287
Adding More Content to the Game	295
Summary	295
Chapter 12: Physics Engines	297
Basic Concepts of Physics Engines	297
Limitations of Physics Engines	298
The Showdown: Box2D vs. Chipmunk	299
Box2D.....	300
The World According to Box2D	301
Restricting Movement to the Screen	302
Converting Points.....	304
Adding Boxes to the Box2D World	305
Connecting Sprites with Bodies.....	306
Collision Detection	308
Joint Venture.....	310

Chipmunk.....	311
Objectified Chipmunk.....	311
Chipmunks in Space	312
Boxing-In the Boxes.....	313
Adding Ticky-Tacky Little Boxes.....	314
Updating the Boxes' Sprites.....	316
A Chipmunk Collision Course	317
Joints for Chipmunks.....	319
Summary	320
Chapter 13: Pinball Game	321
Shapes: Convex and Counterclockwise	322
Working with PhysicsEditor	323
Defining the Plunger Shape	325
Defining the Table Shapes	328
Defining the Flippers.....	331
Defining the Bumper and Ball	332
Save and Publish	333
Programming the Pinball Game	333
The BodyNode Class	333
Creating the Pinball Table.....	338
Box2D Debug Drawing	343
Adding the Ball.....	344
Forcing the Ball to Move	347
Adding the Bumpers	350
The Plunger.....	351
The Flippers	360
Summary	363
Chapter 14: Game Center.....	365
Enabling Game Center	365
Creating Your App in iTunes Connect	366
Setting Up Leaderboards and Achievements.....	367
Creating a Cocos2d Xcode Project.....	367
Configuring the Xcode Project	368
Game Center Setup Summary.....	372
Game Kit Programming.....	373
The GameKitHelper Delegate	373
Checking for Game Center Availability.....	374
Authenticating the Local Player	375
Block Objects	378
Receiving the Local Player's Friend List	380
Leaderboards	382
Achievements	387
Matchmaking	392
Sending and Receiving Data	396
Summary	400

Chapter 15: Cocos2d with UIKit Views	401
What Is Cocoa Touch?.....	401
Using Cocoa Touch and cocos2d Together	402
Why Mix Cocoa Touch with cocos2d?.....	402
Limitations of Mixing Cocoa Touch with cocos2d.....	403
How Is Cocoa Touch Different from cocos2d?	404
Alert: Your First UIKit View in cocos2d.....	405
Embedding UIKit Views in a cocos2d App.....	408
Adding Views in Front of the cocos2d View	408
Skinning the UITextField with a UIImage	410
Adding Views Behind the cocos2d View	412
Adding Views Designed with Interface Builder	419
Orientation Course on Autorotation.....	422
Embedding the cocos2d View in Cocoa Touch Apps	427
Creating a View-Based Application Project with cocos2d	427
Designing the User Interface of the Hybrid App.....	430
Start Your cocos2d Engine.....	432
Stop the cocos2d Engine and Restart It.....	434
Changing Scenes	436
Summary	437
Chapter 16: Kobold2D Introduction	439
Benefits of Using Kobold2D	440
Kobold2D Is Ready to Use	440
Kobold2D Is Free.....	440
Kobold2D Is Easy to Upgrade	440
Kobold2D Provides Lib Service	441
Kobold2D Takes Cross-Platform to Heart	442
The Kobold2D Workspace	442
The Hello Kobold2D Template Project.....	444
The Hello World Project Files	445
How Kobold2D Launches an App	446
The Hello Kobold2D Scene and Layer	450
Running Hello World with iSimulate	454
Doodle Drop for Mac with KKInput.....	455
Entering the Third Dimension with cocos3d	457
Changes to the AppDelegate Class	458
The World According to cocos3d	462
Adding cocos3d to an Existing Kobold2D Project	464
Summary	465
Chapter 17: Out of the Ordinary.....	467
Additional Resources for Learning and Working.....	468
Where to Find Help	468
Source Code Projects to Benefit From	470
Cocos2D Podcast	476
Tools, Tools, Tools	477
Cocos2d Reference Apps	478
The Business of Making Games.....	482

CONTENTS

Working with Publishers482

Finding Freelancers484

Finding Free Art and Audio484

Finding the Tools of the Trade485

Marketing.....485

Engaging Players for More Revenue.....489

Summary494

Index 677

About the Authors



Steffen Itterheim has been a game development enthusiast since the early 1990s. His work in the Doom and Duke Nukem 3D communities landed him his first freelance job as a beta tester for 3D Realms. He has been a professional game developer for more than a decade, having worked most of his career as a game play and tools programmer for Electronic Arts Phenomic. His first contact with cocos2d was in 2009, when he cofounded an aspiring iOS games start-up company called Fun Armada. He loves to teach and enable other game developers so that they can work smarter, not harder. Occasionally you'll find him strolling around in the lush vineyards near his domicile at daytime, and the desert of Nevada at night, collecting bottle caps.



Andreas Löw has been a computer feak since he the age of 10 when he got is first Commodore C16. Teaching himself how to write games, he released his first computer game, Gamma Zone, for Commodore Amiga in 1994, written in pure assembly language. After his diploma in electrical engineering, he worked for Harman International, in the department responsible for developing navigation and infotainment systems with speech recognition for the automotive industry. He invented his own programming language and development tools, which are in use by every car with speech recognition technology around the world.

With the iPhone, he found his way back to his roots and began developing a game called TurtleTrigger. He realized there is a huge demand for good tools in the cocos2d community. With his knowledge in both game and tool development, his products TexturePacker and PhysicsEditor quickly became essential development tools for any cocos2d user.

About the Technical Reviewer



Boon Chew is the managing director for Nanaimo Studio, a game studio based in Seattle and Shanghai that specializes in web and mobile games. He has extensive experience with game development and interactive media, having previously worked for companies such as Vivendi Universal, Amazon, Microsoft, and various game studios and advertising agencies. His passion is in building things and working with great people. You can reach Boon at boon@nanaimostudio.com.

Acknowledgments

This is the part of the book that makes me a little anxious. I don't want to forget anyone who has been instrumental and helpful in creating this book, yet I know I can't mention each and every one of you. If you're not mentioned here, that doesn't mean I'm not thankful for your contribution! Give me a pen, and I'll scribble your name right here in your copy of the book, and I'll sincerely apologize for not having mentioned you here in the first place.

My first thanks go to you, dear reader. Without you, this book wouldn't make any sense. Without knowing that you might read and enjoy this book, and hopefully learn from it, I probably wouldn't even have considered writing it in the first place. I've received valuable insights and requests from my blog readers and other people I've met or mailed during the course of this book. Thank you, all!

My first thanks go to Jack Nutting, who put the idea of writing a book about cocos2d in my head in the first place. I'm grateful that he did not sugarcoat how much work goes into writing a book so that I wasn't unprepared.

Clay Andres I have to thank for being such a kind person, whose input on my chapter proposals were invaluable and to the point. He helped me form the idea of what the book was to become, and he's generally a delightful person to talk to. Clay, I hope that storm did not flood your house.

Many thanks to Kelly Moritz, the coordinating editor, who though incredibly busy always found the time and patience to answer my questions and follow up on my requests. When chaos ensued, she was the one to put everything back in order and made it happen.

Lots and lots of feedback and suggestions I received from Brian MacDonald and Chris Nelson, the development editors for the book, and Boon Chew, the technical reviewer. They made me go to even greater lengths. Brian helped me understand many of the intricacies of writing a book, while Boon pointed out a lot of technical inaccuracies and additional explanations needed. Many thanks to both of you. Chris was a tremendous help for the second edition; he pointed out a lot of the small but crucial improvements. He shall forever be known as CCCC: Code Continuation Character Chris.

Many thanks go to the copy editor, Kim Wimpsett. Without you, the book's text would be rife with syntax errors and compiler warnings, to put it in programmer's terms.

I also wish to thank Bernie Watkins, who managed the Alpha Book feedback and my contracts. Thanks also to Chris Guillebeau for being an outstanding inspirational blogger and role model.

Of course, my friends and family all took some part in writing this book, through both feedback and plain-and-simple patience with putting up with my writing spree. Thank you!

Preface

In May 2009 I made first contact. For the first time in my life, I was subjected to the Mac OS platform and started learning Xcode, Objective-C, and cocos2d. Even for experienced developers like me and my colleagues, it was a struggle. It was then that I realized cocos2d was good, but it lacked documentation, tutorials, and how-to articles—especially when compared with the other technologies I was learning at the time.

Fast-forward a year to May 2010. I had completed four cocos2d projects. My Objective-C and cocos2d had become fluent. It pained me to see how other developers were still struggling with the same basic issues and were falling victim to the same misconceptions that I did about a year earlier. The cocos2d documentation was still severely lacking.

I saw that other cocos2d developers were having great success attracting readers to their blogs by writing tutorials and sharing what they know about cocos2d. To date, most of the cocos2d documentation is actively being created in a decentralized fashion by other developers. I saw a need for a web site to channel all of the information that's spread over so many different web sites.

I created the www.learn-cocos2d.com web site to share what I knew about cocos2d and game development, to write tutorials and FAQs, and to redirect readers interested in cocos2d to all the important sources of information. In turn, I would be selling cocos2d-related products, hoping it might one day bring me close to the ultimate goal of becoming financially independent. I knew I could make the web site a win for everyone.

From day one, the web site was a success—beyond my wildest imaginations. Then, within 24 hours of taking the web site live, Jack Nutting asked me if I had considered writing a cocos2d book. The rest is history, and the result is the book you're reading right now.

I took everything I had in mind for the web site and put it in the book. But that alone would have amounted to maybe a quarter of the book, at most. I hope the four months I spent writing the book full-time paid off by being able to provide an unprecedented level of detail on how cocos2d works and how to work with cocos2d.

I learned a lot in the process, and even more so during months updating the chapters of the second edition. I wish nothing more than for you to learn a great deal about cocos2d and game development from this book.

What I learned from writing about cocos2d is that there's a lot of room for improvement. I strongly believed that the world needed a better cocos2d that's more approachable for beginning game developers. The result of that is Kobold2D, which you'll find an introduction to in Chapter 16 and of course on www.kobold2d.com. Of course, almost all of what you'll learn throughout the book still applies to Kobold2D.

Steffen Itterheim

Introduction

Did you ever imagine yourself writing a computer game and being able to make money selling it? With Apple's iTunes App Store and the accompanying mobile devices iPhone, iPod touch, and iPad, it's now easier than ever. Of course, that doesn't mean it's easy—there's still a lot to learn about game development and programming games. But you are reading this book, so I believe you've already made up your mind to take this journey. And you've chosen one of the most interesting game engines to work with: cocos2d for iOS.

Developers using cocos2d have a huge variety of backgrounds. Some, like me, have been professional game developers for years and even decades. Others are just starting to learn programming for iOS devices or are freshly venturing into the exciting field of game development. Whatever your background might be, I'm sure you'll get something out of this book.

Two things unite all cocos2d developers: we love games, and we love creating and programming them. This book will pay homage to that yet won't forget about the tools that will help ease the development process. Most of all, you'll be making games that matter along the way, and you'll see how this knowledge is applied in real game development.

You see, I get bored by books that spend all their pages teaching me how to make yet another dull Asteroids clone using some specific game-programming API. What's more important, I think, are game programming concepts and tools—the things you take with you even as APIs or your personal programming preferences change. I've amassed hundreds of programming and game development books over 20 years. The books I value the most to this day are those who went beyond the technology and taught me why certain things are designed and programmed the way they are. This book will focus not just on working game code but also on why it works and which trade-offs to consider.

I would like you to learn how to write games that matter—games that are popular on the App Store and relevant to players. I'll walk you through the ideas and technical concepts behind these games in this book and, of course, how cocos2d and Objective-C make these games tick. You'll find that the source code that comes with the book is enriched

with a lot of comments, which should help you navigate and understand all the nooks and crannies of the code.

Learning from someone else's source code with a guide to help focus on what's important is what works best for me whenever I'm learning something new, and I like to think it will work great for you too. And since you can base your own games on the book's source code, I'm looking forward to playing your games in the near future! Don't forget to let me know about them! You can share your projects and ask questions on Cocos2D Central (www.cocos2d-central.com), and you can reach me at steffen@learn-cocos2d.com. You might also want to visit my web site dedicated to learning cocos2d at www.learn-cocos2d.com and you should check out how I'm improving cocos2d with Kobold2D by visiting www.kobold2d.com.

What's New in the Second Edition?

First, I'm proud to have had Andreas Löw as the coauthor for the second edition. Andreas is the developer of the TexturePacker and PhysicsEditor tools, and in particular he went out of his way to update the projects for several chapters with new code and better graphics.

Most importantly, the goal of the second edition was to bring the book up to date with recent developments, one being the final 1.0.1 version of cocos2d as well as compatibility with Xcode 4 and iOS 5. The text, the code, and the figures have all been updated to reflect the new versions of cocos2d, Xcode and the iOS SDK.

Many more changes were made based on reader feedback. Chapter 3 has been overhauled to improve and extend the descriptions of essential cocos2d features. It has also become more visual, with a lot more figures illustrating key concepts and classes. The number of figures in the book has increased throughout.

Over the course of a year, new tools for cocos2d game development have emerged. To reflect the changing tool landscape, the book now refers to TexturePacker in favor of Zwoptex as the leading texture atlas creation tool. Since Löw works full-time on his tools, his customers benefit by receiving frequent updates, new features, and great support. Similarly, PhysicsEditor is used in the second edition in place of VertexHelper since it offers a far better workflow and powerful convenience features. Finally, the second edition introduces you to Glyph Designer, which is essentially the Hiero Bitmap Font tool but with a native Mac OS X user interface and with none of the bugs that plagued Hiero.

The shoot 'em up project first introduced in Chapter 6 and used throughout Chapters 7 to 9 has seen a graphic overhaul. Let's just say it looks a lot better than the programmer's art it used before, courtesy of myself. Likewise, Chapter 13, the pinball physics game, has been improved with new code and improved graphics. Accordingly, the aforementioned chapters have seen some of the more substantial changes.

Last but certainly not least, the second edition adds two entirely new chapters.

Chapter 15 explores the frequently misunderstood and underutilized possibility of mixing cocos2d with regular UIKit views. You'll learn how to add UIKit views to a cocos2d app as well as add cocos2d to an already existing UIKit app. This chapter will make it a lot easier for you to cross the chasm between pure UIKit and pure cocos2d apps, regardless of which way you're going.

Chapter 16 introduces you to Kobold2D (www.kobold2d.com), my idea of making cocos2d a much better game development environment. Kobold2D aims to make commonly performed tasks easy while adding preconfigured libraries such as wax (Lua Scripting), ObjectAL (OpenAL audio), and cocos3d (3D rendering) to the distribution. It also comes with a lot of project templates, many of them based on the projects discussed in this book.

Why Use cocos2d for iOS?

When game developers look for a game engine, they first evaluate their options. I think cocos2d is a great choice for a lot of developers, for many reasons.

It's Free

First, it is free. It doesn't cost you a dime to work with it. You are allowed to create both free and commercial iPhone, iPod, and iPad apps. You can even create Mac OS X apps with it. You don't have to pay royalties. Seriously, no strings attached.

It's Open Source

The next good reason to use cocos2d is that it's open source. This means there's no black box preventing you from learning from the game engine code or making changes to it where necessary. That makes cocos2d both extensible and flexible.

You can download cocos2d from www.cocos2d-iphone.org/download.

It's Objective-C, See?

Furthermore, cocos2d is written in Objective-C, Apple's native programming language for writing iOS apps. It's the same language used by the iOS SDK, which makes it easier to understand Apple's documentation and implement iOS SDK functionality.

A lot of other useful APIs like Facebook Connect and OpenFeint are also written in Objective-C, so it makes it easier to integrate those APIs, too.

NOTE: Learning Objective-C is advised, even if you prefer some other language. I have a strong C++ and C# background, and the Objective-C syntax looked very odd at first glance. I wasn't happy at the prospect of learning a new programming language that was said to be old and outdated. Not surprisingly, I struggled for a while to get the hang of writing code in a programming language that required me to let go of old habits and expectations.

Don't let the thought of programming with Objective-C distract you, though. It does require some getting used to, but it pays off soon enough, if only for the sheer amount of documentation available. I promise you won't look back!

It's 2D

Of course, cocos2d carries the 2D in its name for a reason. It focuses on helping you create 2D games. It's a specialization few other iOS game engines are currently offering.

It does not prevent you from loading and displaying 3D objects. In fact, an entire add-on product aptly named cocos3d has been created as an open source project to add 3D rendering support to cocos2d.

But I have to say that the iOS devices are an ideal platform for great 2D games. The majority of new games released on the iTunes App Store are still 2D-only games even today. 2D games are generally easier to develop, and the algorithms in 2D games are easier to understand and implement. In almost all cases, 2D games are less demanding on the hardware, allowing you to create more vibrant, more detailed graphics.

It's Got Physics

There are also two physics engines you can choose from that are already integrated with cocos2d. On one hand there's Chipmunk, and on the other there's Box2d. Both physics engines superficially differ only in the language they're written in: Chipmunk is written in C, and Box2d is written in C++. The feature set is almost the same. If you're looking for differences, you'll find some, but it requires a good understanding of how physics engines work to base your choice on the differences. In general, you should simply choose the physics engine you think is easier to understand and better documented, and for most developers that tends to be Box2d. Plus, its object-oriented nature makes it a little easier to use with Objective-C.

It's Less Technical

What game developers enjoy most about cocos2d is how it hides the low-level OpenGL ES code. Most of the graphics are drawn using simple sprite classes that are created from image files. In other words, a sprite is a texture that can have scaling, rotation, and color applied to it by simply changing the appropriate Objective-C properties of the

CCSprite class. You don't have to be concerned about how this is implemented using OpenGL ES code, which is a good thing.

At the same time, cocos2d gives you the flexibility to add your own OpenGL ES code at any time for any game object that needs it. And if you're thinking about adding some Cocoa Touch user interface elements, you'll appreciate knowing that these can be mixed in as well.

And cocos2d doesn't just shield you from the Open GL ES intricacies; it also provides high-level abstraction of commonly performed tasks, some of which would otherwise require extensive knowledge of the iOS SDK. But if you do need more low-level access or want to make use of iOS SDK features, cocos2d won't hold you back.

It's Still Programming

In general, you could say that cocos2d makes programming iOS games simpler while still truly requiring excellent programming skills first and foremost. Other iOS game engines such as Unity, iTorque, and Shiva focus their efforts on providing tool sets and workflows to reduce the amount of programming knowledge required. In turn, you give away some technical freedom—and cash too. With cocos2d, you have to put in a little extra effort, but you're as close to the core of game programming as possible without having to actually deal with the core.

It's Got a Great Community

The cocos2d community always has someone quick to answer a question, and developers are generally open to sharing knowledge and information. You can get in touch with the community on the official forum (www.cocos2d-iphone.org/forum) or in my own forum dubbed Cocos2D Central (<http://cocos2d-central.com>).

New tutorials and sample source code are released on an almost daily basis, most of it for free. And you'll find scattered over the Internet plenty of resources to learn from and get inspired by.

Once your game is complete and released on the App Store, you can even promote it on the cocos2d web site. At the very least, you'll get the attention of fellow developers and ideally valuable feedback.

TIP: To stay up to date with what's happening in the cocos2d community, I recommend following cocos2d on Twitter: <http://twitter.com/cocos2d>.

While you're at it, you might want to follow me as well:
<http://twitter.com/gaminghorror>.

Next, enter **cocos2d** in Twitter's search box and then click the `Save this search` link. That way, you can regularly check for new posts about cocos2d with a single click. More often than not, you'll come across useful cocos2d-related information that would otherwise have passed you by. And you'll definitely get to know your fellow developers who are also working with cocos2d.

The Future of the cocos2d-iphone Project

In May 2011, Zynga announced that it hired Ricardo Quesada and Rolando Abarca, key contributors to the cocos2d-iphone project.

Zynga is the developer of the blockbuster social game Farmville. The iPhone version of Farmville was created with cocos2d-iphone and published in June 2010. It's expected that Zynga plans to create more iOS games based on cocos2d-iphone now that Quesada and Abarca work for them.

Quesada is the creator and lead developer of cocos2d-iphone. In fact, he has been running all aspects of cocos2d-iphone since 2008. He managed the web site, he fostered the community, and he is the driving force behind the development and success of the cocos2d-iphone project. Abarca is a contributor to the project, best known for his Ruby wrapper for cocos2d-iphone.

Both developers relocated from Argentina and Chile, respectively, to work for Zynga in San Francisco. At the same time, their previous company, Sapus Media, has stopped selling their two flagship products, Sapus Tongue Source Code and Level SVG, after they had been bought by Zynga.

This means Quesada and Abarca now get a regular paycheck and aren't required to sell, support, and maintain their commercial products to make a living. On the other hand, they will now work primarily for Zynga, and time will have to tell how much of their work will actually find its way into the publicly available open source version of cocos2d-iphone. Currently Ricardo is working on cocos2d 2.0 which will use OpenGL ES 2.0 exclusively.

Although Zynga promises to advance the development of the cocos2d-iphone project and community, there is reason to doubt that the cocos2d-iphone project will continue to be developed at the same rate as in the past. Still, the community can help itself, and

some of them have already started a project dubbed `cocos2d-iphone-extensions`, which provides additional functionality for the `cocos2d-iphone` game engine.

I do not think we have to generally worry about the future of `cocos2d`. The `cocos2d-iphone` project has a strong community and has seen widespread adoption, and even though there are competing game engines, most of them are commercial and proprietary, not free and open source like `cocos2d-iphone`.

Other cocos2d Game Engines

You may have noticed that `cocos2d` ports exist for various platforms, including Windows, JavaScript, and Android. There's even a C++ version of `cocos2d` dubbed `cocos2d-x` that supports multiple mobile platforms, including iOS and Android.

These `cocos2d` ports all share the same name and design philosophy but are written in different languages by different authors and are generally quite different from `cocos2d` for iOS. For example, the Android `cocos2d` port is written in Java, which is the native language when developing for Android devices.

If you're interested in porting your games to other platforms, you should know that the various `cocos2d` game engines differ a lot. Porting your game to Android, for example, isn't an easy task. First there's the language barrier— all your Objective-C code must be rewritten in Java. When that's done, you still need to make a lot of modifications to cope with numerous changes in the `cocos2d` API or possibly unsupported features of the port or the target platform. Finally, every port can have its own kind of bugs, and every platform has its own technical limitations and challenges.

Overall, porting iOS games written with `cocos2d` to other platforms that also have a `cocos2d` game engine entails almost the same effort as rewriting the game for the target platform using some other game engine. This means there's no switch you can flip and it'll work. The similarity of the `cocos2d` engines across various platforms is mostly in name and philosophy. If cross-platform development is your goal, you should take a look at `cocos2d-x`, which has most of the features of `cocos2d-iphone` and is backed financially by China Unicom.

In any case, you should still know about the most popular `cocos2d` game engines. Table 1–1 lists the `cocos2d` game engines that are frequently updated and are stable enough for production use. I did not include `cocos2d` ports in this list that are significantly out of date and haven't been updated for months, if not years.

Table 1–1. *Most Popular cocos2d Game Engine Ports*

Name	Language	Platforms	Web Site
cocos2d-iphone	Objective-C	iOS, Mac OS X	www.cocos2d-iphone.org
cocos2d-x	C++	iOS, Android, Windows	www.cocos2d-x.org
cocos2d-javascript	JavaScript	Web browsers	www.cocos2d-javascript.org
cocos2d-android-1	Java	Android	http://code.google.com/p/cocos2d-android-1
cocos2d	Python	Mac OS, Windows, Linux	www.cocos2d-javascript.org

This Book Is for You

I’d like to imagine you picked this book because its title caught your interest. I suppose you want to make 2D games for iPhone, iPod touch, and iPad, and the game engine of your choice is cocos2d for iOS. Or maybe you don’t care so much about the game engine but you do want to make 2D games for the iOS devices in general. Maybe you’re looking for some in-depth discussion on cocos2d, since you’ve been using it for a while already. Whatever your reasons for choosing this book, I’m sure you’ll get a lot out of it.

Prerequisites

As with every programming book, there are some prerequisites that are nice to have and some that are almost mandatory.

Programming Experience

The only thing that’s mandatory for this book is some degree of programming experience, so let’s get that out of the way first. You should have an understanding of programming concepts such as loops, functions, classes, and so forth. If you have written a computer program before, preferably using an object-oriented programming language, you should be fine.

Still with me? Good.

Objective-C

So, you do have programming experience, but maybe you’ve never written anything in that obscure language called Objective-C.

You don't need to know Objective-C for this book, but it definitely helps to know the language basics. If you are already familiar with at least one other object-oriented programming language, such as C++, C#, or Java, you may be able to pick it up as you go. But to be honest, I found it hard to do that myself even after roughly 15 years of programming experience with C++, C#, and various scripting languages. There are always those small, bothersome questions about tiny things you just don't get right away, and they tend to steal your attention away. In that case, it's handy to have a resource you can refer to whenever there's something you need to understand about Objective-C.

Objective-C may seem scary with its square brackets, and you may have picked up some horror stories about its memory management and how there's no garbage collection on iOS devices. Worry not.

First, Objective-C is just a different set of clothes. It looks unfamiliar, but the underlying programming concepts such as loops, classes, inheritance, and function calls still work in the same way as in other programming languages. The terminology might be different; for example, what Objective-C developers call sending messages is in essence the same as calling a method. As for memory management, let's just say cocos2d makes it as easy for you as possible, and I'll help you understand the very simple and basic rules you can follow.

I had one invaluable Objective-C book to learn from, and I recommend it wholeheartedly as a companion book in case you want to learn more about Objective-C and Xcode. It's called *Learn Objective-C on the Mac* by Mark Dalrymple and Scott Knaster, published by Apress.

There is also Apple's "Introduction to the Objective-C Programming Language," which proved valuable as an online reference. It's available here:
<http://developer.apple.com/mac/library/DOCUMENTATION/Cocoa/Conceptual/ObjectiveC/Introduction/introObjectiveC.html>.

What You Will Learn

I will provide you with a fair share of my game development experiences to show how interactive games are made. I believe that learning to program is not at all about memorizing API methods, yet a lot of game development books I've read over the past two decades follow that "reference handbook" approach. But that's what the API documentation is for. When I started programming some 20 years ago, I thought I'd never learn to program just by looking at a huge stack of compiler reference handbooks and manuals. Back at that time, compiler manuals were still printed and, obviously, didn't come with online versions. The World Wide Web was still in its infancy. So, all that information was stacked some 15 inches high on my desk, and it seemed very daunting to try to learn all of this.

Today, I still don't recall most methods and APIs from memory, and I keep forgetting about those I used to know. I look them up time and time again. After 20 years of programming, I do know what's really important to learn: the concepts. Good

programming concepts and best practices stick around for a long time, and they help with programming in any language. Learning concepts is done best by understanding the rationale behind the choices that were made in designing, structuring, and writing the source code. That's what I'll focus on the most.

What Beginning iOS Game Developers Will Learn

I'll also ease you into the most important aspects of cocos2d. I'll focus on the kind of classes, methods, and concepts that you should be able to recall from memory just because they are so fundamental to programming with cocos2d.

You'll also learn about the essential tools supporting or being supported by cocos2d. Without these tools, you'd be only half the cocos2d programmer you can be. You'll use tools like TexturePacker and ParticleDesigner to create games that will be increasingly complex and challenging to develop. Because of the scope of this book, these games will not be complete and polished games, nor will I be able to discuss every line of code. Instead, I'll annotate the code with many helpful comments so that it's easy to follow and understand.

I leave it up to you to improve on these skeleton game projects, and I'm excited to see your results. I think giving you multiple starting points to base your own work on works better than walking you through the typical Asteroids games over the course of the whole book.

I chose the game projects for this book based on popularity on the App Store and relevance for game developers, who often inquire about how to solve the specific problems that these games present. For example, the line-drawing game genre is a huge favorite among cocos2d game developers, yet line-drawing games require you to overcome deceptively complex challenges.

I've also seen a fair bit of other developers' cocos2d code and followed the discussions on code design, structure, and style. I'll base my code samples on a framework that relies on composition over inheritance and will explain why this is preferable. One other frequent question that has to do with code design is how different objects should communicate with each other. There are interesting pros and cons for each approach to code design and structure, and I want to convey these concepts because they help you write more stable code with fewer bugs and better performance.

What iOS App Developers Will Learn

So, you are an iOS app developer, and you've worked with the iOS SDK before? Perfect. Then you'll be most interested in how making games works in a world without Interface Builder. In fact, there are other tools you'll be using. They may not be as shiny as Apple's tools, but they'll be useful nonetheless.

The programming considerations will change, too. You don't normally send and receive a lot of events in game programming, and you let a larger number of objects decide what to do with an event. For performance reasons and to reduce user input latency,

game engine systems often work more closely connected with each other. A lot of work is done in loops and update methods, which are called at every frame or at specific points in time. While a user interface-driven application spends most of the time waiting for a user's input, a game keeps pushing a lot of data and pixels behind the scenes, even when the player is not doing anything. So, there's a lot more going on, and game code tends to be more streamlined and efficient because of concerns for performance.

What Cocos2d Developers Will Learn

You're already familiar with cocos2d? You may be wondering whether you can learn anything new from this book. I say you will. Maybe you need to skip the first chapters, but you'll definitely get hooked by the games' sample source code supplied with the book. You'll learn how I structure my code and the rationale behind it. You'll probably find inspiration reading about the various games and how I implemented them. There's also a good number of tips you'll benefit from.

Most importantly, this book isn't written by some geek you've never heard of and never will hear from again, with no e-mail address or web site where to post your follow-up questions. Instead, it's written by a geek you may not have heard of but who will definitely be around. I'm actively engaged with the cocos2d community at my www.learn-cocos2d.com blog, where I'll basically keep writing this book.

What's in This Book

Here's a brief overview of the chapters in this book. The second edition of the book features two entirely new chapters: Chapter 15 discusses integration of UIKit views with cocos2d, and Chapter 16 introduces Kobold2D, my own take on a cocos2d development environment with additional convenience features.

Chapter 2, Getting Started

I'll cover setting up cocos2d for development, installing project templates, and creating the first "Hello World" project. You'll learn about cocos2d basics, such as scenes and nodes.

Chapter 3, Essentials

I'll explain the essential cocos2d classes that you'll need most often, such as sprites, transitions, and actions. And you'll learn how to use them, of course.

Chapter 4, Your First Game

Enemies drop from the top, and you have to avoid them by tilting your device. This will be our first simple game using accelerometer controls.

Chapter 5, Game Building Blocks

Now prepare yourself for a bigger game, one that requires a better code structure. You'll learn how scenes and nodes are layered and the various ways that game objects can exchange information.

Chapter 6, Sprites In-Depth

You'll learn what a texture atlas is and why we'll be using it for our next game and how to create a texture atlas with the TexturePacker tool.

Chapter 7, Scrolling with Joy

With the Texture Atlas ready, you'll learn how to implement a parallax scrolling shooter game, controlled by touch input.

Chapter 8, Shoot em Up

Without enemies, our shooter wouldn't have much to shoot at, right? I'll show you how to add game-play code to spawn, move, hit, and animate the enemy hordes.

Chapter 9, Particle Effects

By using the ParticleDesigner tool, you'll add some particle effects to the side-scrolling game.

Chapter 10, Working with Tilemaps

Infinitely jumping upward, you'll apply what you've learned from the side-scrolling game in portrait mode to create another popular iOS game genre.

Chapter 11, Isometric Tilemaps

Since cocos2d supports the TMX file format, you'll take a look at how to create tile-based games using the Tiled editor.

Chapter 12, Physics Engines

Directing where things go with the move of your fingertips— you'll learn here how that's done.

Chapter 13, Pinball Game

This is a primer on using the Chipmunk and Box2d physics engines and the crazy things you can do with them.

Chapter 14, Game Center

This time, you'll use real physics for a gravity-defying, planet-bouncing, ball-shooter in space. It's not going to be realistic, but it's going to have real physics. It's a conundrum, maybe, but fun in any case.

Chapter 15, Cocos2d with UIKit Views

This chapter goes into depth on how to mix and match cocos2d with regular Cocoa Touch, particularly UIKit views. You'll learn how to add UIKit views to a cocos2d game or, conversely, how to make use of cocos2d in an existing UIKit app.

Chapter 16, Kobold2D Introduction

Kobold2D is my take on improving the cocos2d game engine by adding popular libraries and combining them into a single, ready-to-use package. In this chapter, you'll learn how to set up new Kobold2D projects and the role of Lua scripting in Kobold2D, and you'll get a primer on 3D game development with cocos3d.

Chapter 17, Conclusion

This is where the book ends. Worry not, your journey won't. You'll get inspiration on where to go from here.

Where to Get the Book's Source Code?

One of the most frequently asked questions following the release of the first edition of this book was about where to get the book's source code. I've added this little section to answer this question.

You can get the book's source code on the Apress web site under Source Code/Downloads if you follow this link: www.apress.com/9781430233039. Alternatively, you can download the source code in the Downloads section on Cocos2D Central: cocos2d-central.com/files/file/2-source-code.

Of course, you can always type the code directly from the book if you prefer.

Questions and Feedback

I do hope I get the right mixture of easing you into cocos2d and iOS game development while challenging you with advanced game-programming concepts.

If at any time I fail and leave you wondering, please feel free to ask your questions on Cocos2D Central (www.cocos2d-central.com). I'll also continue to post frequently about cocos2d news and developments on the Learn Cocos2D companion web site for this book at www.learn-cocos2d.com. Your feedback is always welcome!

Getting Started

I will get you up to speed and developing cocos2d games as quickly as possible. By the end of this chapter, you'll be able to create new cocos2d projects based on the supplied Xcode project templates. I'll also introduce you to the important bits of knowledge you need to keep in mind during game development. And since it's always been a big source of confusion, I'll explain how memory management works in the context of cocos2d, ideally helping you avoid some of the common pitfalls. At the end of this chapter, you'll have a first cocos2d project based on a project template up and running.

What You Need to Get Started

In this section, I'll quickly walk you through the requirements and necessary steps to get started. Getting registered as an iOS developer and creating the necessary provisioning profiles are both excellently documented by Apple, so I won't re-create that detailed information here.

System Requirements

These are the minimum hardware and software requirements for developing iOS applications:

- Intel-based Mac computer with 1GB RAM
- Mac OS X 10.6 (Snow Leopard) or greater
- Any iOS device

For development, any Intel-based Mac computer suffices. Even the Mac mini and MacBook Air are perfectly fine for developing iOS applications and games. I do recommend having 2GB of RAM installed. It'll make using your computer smoother, especially since game development tools often require much more memory than most other software. You'll be handling a lot of images, audio files, and program code, and you'll probably be running all these tools in parallel.

Note that Mac OS X 10.6 is mandatory for iOS development since the release of the iOS SDK 4 in June 2010. With the release of Mac OS X 10.7 Lion, you can expect that eventually it will become a requirement for iOS development. As a developer with Apple, you're regularly required to be using the latest OS X version.

If you are running an older version of Mac OS X, please consult the Mac OS X Technical Specifications web site (www.apple.com/macosx/specs.html) to learn whether your Mac meets the system requirements and how to purchase and upgrade to the latest Mac OS X version.

Register as an iOS Developer

If you haven't done so yet, you might want to register yourself as an iOS developer with Apple. Access to the iOS Developer Program costs \$99 per year. If you plan to submit Mac OS X apps to the Mac App Store, you will also have to register as a Mac OS X developer, which costs an additional \$99 per year.

As a registered developer, you get access to the iOS SDK, Xcode, and the iOS Developer Portal where you have to set up your development devices and provisioning profiles in order to deploy your app to iOS devices. You also get access to iTunes Connect where you can manage your contracts, manage and submit your apps, and review financial reports. In addition, you'll be offered preview (beta) versions of Apple software. Registered Mac OS X developers will also get free access to the latest Mac OS.

You can register as an iOS developer at <http://developer.apple.com/programs/ios>.

To register as a Mac OS X developer, go to <http://developer.apple.com/programs/mac>.

TIP: You can also get Xcode from the Mac App Store for free. This download includes the iOS SDK and Mac OS X SDK. You'll be able to develop cocos2d apps for iOS and Mac OS X. The catch: you cannot run your iOS apps on an iOS device unless you register as an iOS developer, so you'll be limited to using the iOS Simulator. You will also not be able to submit your app to the iOS App Store or Mac App Store until you sign up as a registered iOS respectively Mac OS developer.

Certificates and Provisioning Profiles

Eventually you'll want to deploy the games you're building onto your iOS device. To do so, you must create an iOS development certificate, register your iOS device, and enable it for development. Finally, you'll create development or distribution provisioning profiles, download them to your computer, and set up each Xcode project to use them.

All of these steps are well explained on the iOS Provisioning Portal. Apple has done an excellent job at documenting these steps on the How To tabs of each section of the Provisioning Portal.

The iOS Provisioning Portal is accessible for registered iOS developers at <http://developer.apple.com/ios/manage/overview/index.action>.

Download and Install the iOS SDK

As a registered iOS developer, you can download the latest iOS SDK from the iOS Dev Center. The download is well over 4GB and will take a while to download and install, so you might want to do it right away.

After the installation of the iOS SDK is complete, you are set with everything you need to develop iOS applications, including the Xcode integrated development environment (IDE). If you've never worked with Xcode before, I suggest you familiarize yourself with it. I recommend *Learn Xcode Tools for Mac OS X and iPhone Development* by Ian Piper (Apress, 2010).

CAUTION: It may be tempting to be at the bleeding edge of iOS SDK development. From time to time, beta versions of the iOS SDK are made available. I recommend not using iOS SDK beta versions unless you have a very, very good reason to do so!

Beta versions can contain bugs, they may be incompatible with the current cocos2d version, and they are under NDA. This means it's hard to find solutions if any issue related to the beta version arises, since no one is allowed to discuss the beta SDK in public.

Moreover, you have to install a beta version of the iOS to your device, and you can't revert to a previous iOS version. Installed apps on your device may be incompatible with the new iOS beta, and they usually aren't updated until the new iOS SDK is officially released. If you rely on any apps to do your work, don't upgrade.

Download and Install cocos2d

The next step is to get cocos2d. You can download it from www.cocos2d-iphone.org/download. Since many new developers continue to have issues with the template installation script, I also provide an installer for cocos2d and cocos3d that runs the template installation script for you. You can download the installer on Cocos2d Central: <http://cocos2d-central.com/files/file/6-installer>.

I recommend downloading and extracting the Stable version of cocos2d. The Unstable version doesn't mean it's going to crash all the time, but it is a beta version. It'll work just fine in general, but it may have some rough edges, untested features, and incompatibilities with third-party tools. Before you consider the Unstable version, please

review the release notes to see whether it contains anything of particular use to you. If not, just stick to the Stable version.

After downloading and extracting cocos2d, you'll have a subfolder named cocos2d-iphone-1.0.1 or similar, depending on the exact version number of cocos2d you downloaded.

Install cocos2d Xcode Project Templates

NOTE: If you used my cocos2d/cocos3d installer, you can skip this section. The Xcode project templates have already been installed by the installer.

Open the Terminal app, which you'll find in the Utilities folder of your Applications folder on your Mac. Or just enter Terminal.app in Spotlight to locate it. The cocos2d Xcode project templates installation procedure is driven by a shell script that you'll have to run from the command-line program Terminal.

First, change to the directory where cocos2d is installed. For example, if your cocos2d version is installed under Documents in the folder cocos2d-iphone-1.0.0, then enter the following:

```
cd ~/Documents/cocos2d-iphone-1.0.0
```

Press Return to change into the cocos2d directory and then enter the following:

```
./install-templates.sh ...f -u
```

Press Return to run the template installation script. If everything goes fine, you should see a number of lines printed on the Terminal window. Most of them will start with `□□ copying.□` If that's the case, the templates should now be installed.

If you get any kind of error, verify that you have changed into the cocos2d-iphone directory with the `cd` command and that the command for the `install-templates.sh` script is correct, including spaces between the command and the `...f` and `...u` options. If that does not help, consider using the cocos2d installer I created to alleviate exactly such dreaded template installation problems. You can download the installer from here:

cocos2d-central.com/files/file/6-installer

Create a cocos2d Application

Now open Xcode and select **File □ New Project**. Under User Templates you should see the cocos2d project templates, as shown in Figure 2–1.

NOTE: The Box2d and Chipmunk application templates will be discussed in Chapter 13. Feel free to try them if you want to have some fun with physics right now.

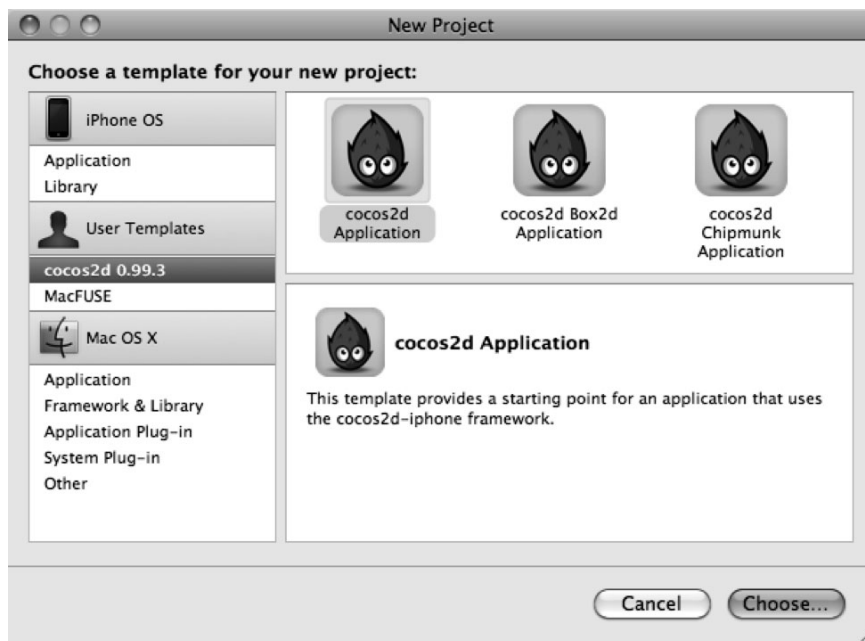


Figure 2–1. *The cocos2d Xcode project templates*

Choose the cocos2d Application template and name it HelloWorld.

TIP: It is good practice not to use space characters in project names. Xcode doesn't mind, but some tools you might use could. It's just a matter of defensively avoiding any potential hiccups.

For a very, very long time, programmers who built operating systems and applications could rely on file names not containing spaces. Even today, after modern operating systems have allowed spaces in file names for at least the past 10 years, there are occasional problems related to spaces and special characters in file names. I always avoid naming anything code-related, whether projects, source files, or resources, with spaces or other special characters. Only numbers, digits, and the minus sign and underscore are guaranteed safe for developers to use in file names.

Xcode will create the project based on the template. An Xcode project window like the one in Figure 2–2 will open.



Figure 2–2. The newly created HelloWorld project in Xcode 4

When you click the **Run** button, the project will build and then run in the iOS Simulator. The result should look like Figure 2–3.



Figure 2-3. Success! The template project works and displays a *Hello World* label running in the iPhone Simulator.

The HelloWorld Application

So here we are with minimal fuss you created a running cocos2d application. Perfect. Say no more. Say no more.

But now you want to know how it works, right? Well, I didn't expect you'd let me off the hook so easily. And something tells me that, however deep I go into the details over the course of the book, you'll want to know more. That's the spirit!

Let's check what's in the HelloWorld code project and see how it all works so you get a rough overview of how things are connected. Feel free to play around with this HelloWorld project. If anything breaks, you can just start over with a new project created from the cocos2d template. Over the course of this book, you'll learn all the details about cocos2d and the settings first mentioned in this chapter.

Locating the HelloWorld Files

First, here's a quick primer in case you've never worked with Xcode before. By default, you'll see a pane called Project Navigator on the left side of the Xcode project window, like the one in Figure 2–4. That's where Xcode keeps all file references, among plenty of other things like targets and executables. Just focus on the groups and files for now that are below the HelloWorld project.

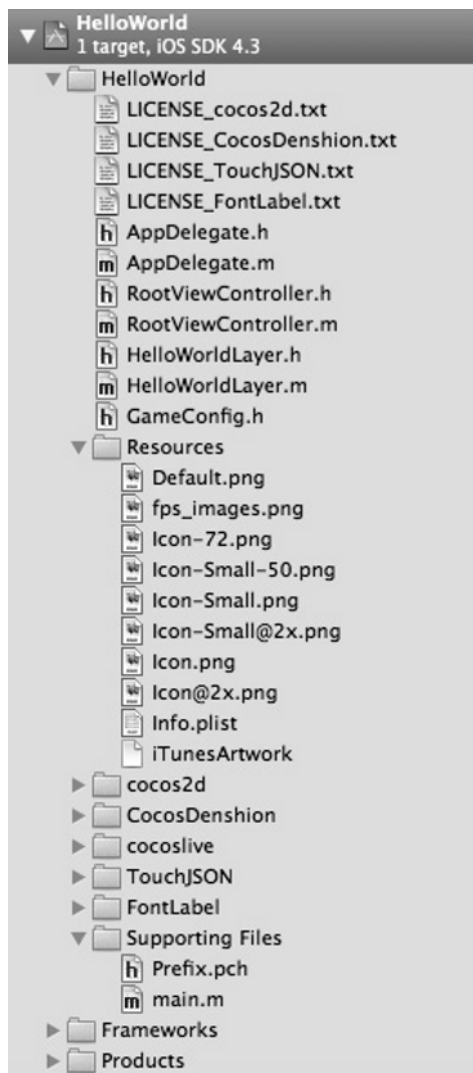


Figure 2–4. Xcode's Project Navigator pane. The expanded groups contain the project files we'll be looking at.

In the group named `cocos2d` you'll find all the files the `cocos2d` game engine consists of. Feel free to explore these files if you're curious, but unless you know what you're

doing, you should not modify them. The same goes with the other cocos2d-related groups; they are the groups that are not expanded in Figure 2–4.

You don't need to know the details of the cocos2d game engine, but it's good to have the source files accessible, especially when it comes to debugging or in case you get curious and want to know how things work under the hood and learn a trick or two.

CAUTION: Xcode's Project Navigator pane looks a lot like folders and files in Finder. Don't mistake what Xcode calls groups for Finder's folders. You can have your files in Xcode arranged in many groups, but in Finder they can and normally will still all be in the same folder. This is why they are not called folders but groups. They allow you to arrange files logically without affecting their physical location on your computer's hard drive.

Resources

In the Resources group you'll find (and later add) all the additional files that aren't source code, such as images and audio files.

The Default.png file is the image that's displayed when iOS is loading your app, and Icon.png is, of course, the app's icon. The fps_images.png file is used by cocos2d to display the framerate; you should not remove or modify it.

Inside the Info.plist file you'll find a number of settings for your application. You'll only need to make changes here when you get close to publishing your app.

Supporting Files

If you're familiar with programming in C or similar languages, you may recognize main.m in the Other Sources group as the starting point of the application. The only other file in this group is the precompiled header file Prefix.pch.

Main.m

Everything that happens between the main function and the HelloWorldAppDelegate class is the behind-the-scenes magic of the iOS SDK, over which you have no control. Since you'll hardly ever need to change main.m, you can safely ignore its contents. Still, it never hurts to peek inside.

To quickly sum up, the main function creates an `NSAutoreleasePool` and then calls `UIApplicationMain` to start the application using `HelloWorldAppDelegate` as the class that implements the `UIApplicationDelegate` protocol.

```
int main(int argc, char *argv[])
{
    NSAutoreleasePool *pool = [NSAutoreleasePool new];
    int retVal = UIApplicationMain(argc, argv, nil, ↵
```

```
        @"HelloWorldAppDelegate");  
    [pool release];  
    return retVal;  
}
```

The only interesting point to take away from this is that every iOS application uses an `NSAutoreleasePool` to help you manage memory. In short, by using the `autorelease` message on objects, you don't have to worry about sending them a `release` message. The autorelease pool ensures that the memory of autorelease objects is eventually released. Don't worry if you don't know what the heck I'm talking about here. I'll introduce you to memory management with `cocos2d` later in this chapter.

You can also learn more about using autorelease pools in Apple's Memory Management Programming Guide at <http://developer.apple.com/library/mac/#documentation/Cocoa/Conceptual/MemoryMgmt/Articles/mmAutoreleasePools.html>.

Precompiled Prefix Header

Just in case you're wondering what the `Prefix.pch` header file is for, it's a tool used to speed up compilation. You are supposed to add the header files of frameworks that never or only rarely change to the prefix header. This causes the framework's code to be compiled in advance and made available to all your classes. Unfortunately, it also has the disadvantage that, if a header added to the prefix header changes, all your code will recompile, which is why you should only add header files that rarely or never change.

For example, the `cocos2d.h` header file is a good candidate to add to the prefix header, as I've done in Listing 2–1. To create a noticeable increase in compilation time, your project would need to be reasonably complex, however, so don't get your stopwatch out just yet. But it's good practice to add `cocos2d.h` as a prefix header right away, if only to never have to write `#import "cocos2d.h"` in any of your source files again.

Listing 2–1. *Adding the `cocos2d.h` Header File to the Prefix Header*

```
#ifdef __OBJC__  
    #import <Foundation/Foundation.h>  
    #import <UIKit/UIKit.h>  
    #import "cocos2d.h"  
#endif
```

Once again, you can refer to Apple's developer documentation if you want to learn more about reducing build times with prefix headers: http://developer.apple.com/mac/library/documentation/DeveloperTools/Conceptual/XcodeBuildSystem/800-Reducing_Build_Times/bs_speed_up_build.html.

HelloWorld Classes

Two classes make up the core of the HelloWorld project. The `AppDelegate` class handles the application's global events and state changes, while the `HelloWorldLayer` class contains all the code that displays the "Hello World" label.

AppDelegate

Every iOS application has one AppDelegate class that implements the UIApplicationDelegate protocol. In our HelloWorld project, it's simply called AppDelegate.

The AppDelegate is a global concept you'll find in every iOS application. It is used to track state changes of the application, and to do this it receives messages from the iOS at certain points in time. For example, it allows you to determine when the user gets an incoming phone call or when the application is low on memory. The very first message your application will receive is the `applicationDidFinishLaunching` method. That's where all the startup code goes and where cocos2d is initialized.

If you want to learn more about the AppDelegate's various methods, what they do, and when these messages are sent by the iOS SDK, you can look it up in Apple's reference documentation on the UIApplicationDelegate protocol at http://developer.apple.com/iphone/library/documentation/uikit/reference/UIApplicationDelegate_Protocol.

NOTE: Since I'm talking about application startup, I might as well talk about application shutdown. You may eventually notice an oddity with the AppDelegate's `dealloc` method. It never gets called! Any breakpoint set in the AppDelegate's `dealloc` method will never be hit!

This is normal behavior. When iOS terminates an application, it simply wipes the memory clean to speed up the shutdown process. That's why any code inside the `dealloc` method of the AppDelegate class is never run. Also, it's bad practice to call `dealloc` manually, so don't try to fix this issue by doing so. If you ever need to run code in your AppDelegate just before the application terminates, do it inside the `applicationWillTerminate` method.

In most cases, there are only three things you might want to change during the cocos2d initialization process:

```
[[CCDirector sharedDirector] ➤  
    setDeviceOrientation:kCCDeviceOrientationLandscapeLeft];  
[[CCDirector sharedDirector] setAnimationInterval:1.0/60];  
[[CCDirector sharedDirector] setDisplayFPS:YES];
```

I'll provide a few details on each of these in the following sections.

Device Orientation

The most important one is setting the device orientation. The HelloWorld application uses a landscape orientation, which means you'll be holding your iOS device sideways. If you change this option from `kCCDeviceOrientationLandscapeLeft` to `kCCDeviceOrientationLandscapeRight`, you'll find that the message "Hello World" is now displayed upside down compared to what it was before.

Here's a list of supported device orientations. Try each of them to see how they change the "Hello World" text label orientation.

- `kCCDeviceOrientationPortrait`
- `kCCDeviceOrientationPortraitUpsideDown`
- `kCCDeviceOrientationLandscapeLeft`
- `kCCDeviceOrientationLandscapeRight`

TIP: You can change the device orientation at a later point in time, even during game play. For example, you can make this a game setting the user can choose. As long as you change from one landscape mode to another or from one portrait mode to another, you don't even need to modify your code. Allowing the user to choose to play the game in any of the two landscape or portrait directions is very straightforward to implement. Since everyone has a different opinion on that matter, it's a good idea to let the user choose between the regular and the upside-down orientation. Please refer to Chapter 15, which explains how to implement autorotation to the current device orientation.

Animation Interval

The animation interval determines how often cocos2d updates the screen. Effectively, this affects the maximum framerate your game can achieve.

The animation interval is not given in frames per second, however. It's the inverse because it determines how frequently cocos2d should update the screen. That's why the parameter is `1.0/60` because 1 divided by 60 results in 0.0167, which is the time in seconds between each call to update the screen. Of course, if your game is complex, it might take the CPU or GPU longer than 0.0167 seconds to display a frame, so there's no guarantee that 60 fps will be achieved throughout. As a matter of fact, it's your responsibility to keep the game running at a high framerate. I'll explain techniques to improve performance throughout the book.

In some cases, it may be preferable to lock the framerate to 30 frames per second. This may be helpful in very complex games where you can't achieve 60 fps consistently and the framerate fluctuates a lot between 30 and 60 fps. In such a case, it's often better to lock the framerate to the lowest common denominator because a lower but steady framerate is perceived as smoother by players than a framerate that tends to fluctuate abruptly, even when the actual average framerate may be higher. Human perception is a tricky thing.

NOTE: You can't render more than 60 frames per second on iOS devices. The device's display is locked to update at 60 frames per second (Hz), and forcing cocos2d to render more frames than 60 per second is, at best, not doing anything. At worst, it can actually slow down your framerate. Stick with the `animationInterval` of 1.0/60 if you want to run cocos2d at the fastest possible rate.

Display FPS

Enabling the FPS display will show a small number at the lower-left corner of the screen. This is your framerate, or the frames per second the screen is updated. Ideally, your game should be running at 60 fps at all times, especially if it's an action or twitch-based game. Some games, such as most puzzle games, can go with a constant 30 fps just fine. The FPS display helps you keep track of the framerate and any hiccups or stutters your game is experiencing.

NOTE: If you need to tweak the responsiveness of the FPS display, you can do so by modifying the `CC_DIRECTOR_FPS_INTERVAL` line in `ccConfig.h`. You'll find this file in the cocos2d Sources/cocos2d group. By default it is set to 0.1, which means the framerate display will be updated ten times per second. If you increase the value, the FPS display will average out over a longer period of time. However, you won't be able to see any sudden, short drops in framerate, which can still be noticeable. Keep that in mind.

HelloWorldLayer

The `HelloWorldLayer` class is where pure cocos2d code does its magic to display the `□Hello World□` label. Before I get into that, you should understand that cocos2d uses a hierarchy of `CCNode` objects to determine what is displayed where.

The base class of all nodes is the `CCNode` class, which contains a position but has no visual representation. It's the parent class for all other node classes, including the two most fundamental ones: `CCScene` and `CCLayer`.

`CCScene` is an abstract concept and does nothing more than to allow proper placement of objects in the scene according to their pixel coordinates. A `CCScene` node is thus always used as the parent object for every cocos2d scene hierarchy. Most of the time you will have only one running scene, except when transitioning from one scene to another.

The `CCLayer` class does very little by itself other than allowing touch and accelerometer input. You'll normally use it as the first class added to the `CCScene`, simply because most games use at least simple touch input.

If you open the HelloWorldLayer.h header file, you'll see that the HelloWorldLayer class is derived from CCLayer. So, where does the CCScene class come into play?

Since CCScene is merely an abstract concept, the default way of setting up a scene has always been to use a static initializer `+(id) scene` in your class. This method creates a regular CCScene object and then adds an instance of the HelloWorldLayer class to the scene. In almost all cases, that's the only place where a CCScene is created and used. The following is a generic example of the `+(id) scene` method:

```
+(id) scene
{
    CCScene *scene = [CCScene node];
    id layer = [HelloWorldLayer node];
    [scene addChild:layer];

    return scene;
}
```

First, a CCScene object is created using the static initializer `+(id) node` of the CCScene class. Next, our HelloWorldLayer class is created using the same `+(id) node` method and then added to the scene. The scene is then returned to the caller.

Moving on to the `...(id) init` method in Listing 2-2, you'll notice something that might seem odd: `self` is assigned the return value from the `init` message sent to the super object in the call to `self = [super init]`. If you come from a C++ background, you'll shudder in pain looking at this. Don't get too upset; it's all right. It simply means that in Objective-C we have to manually call the superclass's `init` method. There is no automatic call to the parent class. And we do have to assign `self` the return value of the `[super init]` message because it might return `nil`.

Listing 2-2. *The `init` Method Creates and Adds the Hello World Label*

```
-(id) init
{
    if ((self = [super init])) {
        // create and initialize a label
        CCLabelTTF* label = [CCLabelTTF labelWithString:@"Hello World"
                                                             fontName:@"Marker Felt"
                                                             fontSize:64];

        // get the window (screen) size from CCDirector
        CGSize size = [[CCDirector sharedDirector] winSize];

        // position the label at the center of the screen
        label.position = CGPointMake(size.width / 2, size.height / 2);

        // add the label as a child to this Layer
        [self addChild:label];
    }
    return self;
}
```

The CCLabelTTF class draws text on the screen using a TrueType font.

If you're deeply concerned by the way Objective-C programmers write the call to `[super init]`, here's an alternative that might ease your mind. It's fundamentally the same, just not what tradition dictates:

```
-(id) init
{
    self = [super init];
    if (self != nil) {
        // do init stuff here
    }
    return self;
}
```

Now let me explain how the label is added to the scene. If you look again at the `init` method in Listing 2-2, you'll see that a `CCLabelTTF` object is created using one of `init`'s static initializer methods. It'll return a new instance of the `CCLabelTTF` class as an autoreleased object. To not have its memory freed after control leaves the `init` method, you have to add the label to `self` as a child using the `[self addChild:label]` message. In between, the label is assigned a position at the center of the screen. Note that whether you assign the position before or after the call to `addChild` doesn't matter.

Memory Management with cocos2d

At this point, I need to talk a bit about memory management and the `autorelease` message. Memory management in the reference-counting Objective-C world is governed by two simple rules:

- If you own (`alloc`, `copy`, or `retain`) an object, you must release or `autorelease` it later.
- If you send `autorelease` to an object, you must not release it.

Normally, when you create an object in Objective-C, you do so by calling `alloc`. By doing this, you become responsible for releasing the object when you don't need it anymore. The following is a typical `alloc/init` and `release` cycle:

```
// allocate a new instance of NSObject
NSObject* myObject = [[NSObject alloc] init];

// do something with myObject here

// release the memory used by myObject
// if you don't release it, the object is leaked
// and the memory used by it is never freed.
[myObject release];
```

Now, with the `autorelease` message and the fact that iOS applications always use an autorelease pool, you can avoid sending the `release` message. Here's the same example rewritten using `autorelease`:

```
// allocate a new instance of NSObject
NSObject* myObject = [[[NSObject alloc] init] autorelease];
```

```
// do something with myObject here
```

```
// no need to call release, in fact you should not send release as it would crash.
```

As you can see, this simplifies memory management somewhat in that you no longer have to remember to send the release message. The autorelease pool takes care of that for you by sending the object a release message at the end of the current frame update if it is no longer referenced. Creating the object gets just a little bit more complex because the autorelease message was added.

Now consider the following code, which illustrates how you'd allocate a CCNode object if you followed the traditional release style:

```
// allocate a new instance of CCNode
CCNode* myNode = [[CCNode alloc] init];
```

```
// do something with myNode
```

```
[myNode release];
```

This is *not* the preferred way to create cocos2d objects. It's much easier to use the static initializer methods, which return an autorelease object. Contrary to what Apple recommends, the use of autorelease is consistently built into the cocos2d engine's design by moving calls like `[[[NSObject alloc] init] autorelease]` to a static method in the class itself. And that's a good thing ☐ do not fight it! It will make your life easier since you don't have to remember which objects need to be released, which is often cause for either crashes due to over-releasing certain objects or memory leaks due to not releasing all objects.

In the case of the CCNode class, the static initializer is `+(id) node`. The following code sends the alloc message to self, which is equivalent to using `[CCNode alloc]` if the code is placed in the CCNode implementation.

```
+(id) node
{
    return [[[self alloc] init] autorelease];
}
```

It's just a little more generic to use self in this case and perfectly legal in case you happen to be a C++ programmer now scratching your head.

Seeing this, we can rewrite the CCNode allocation to use the static initializer, and, quite unsurprisingly, the code is now very short, concise, and tidy. Just the way I like it:

```
// allocate a new instance of CCNode
CCNode* myNode = [CCNode node];
```

```
// do something with myNode
```

That's the beauty of using autorelease objects. You don't need to remember to send them a release message. Each time cocos2d advances to the next frame, the autorelease objects that are no longer in use are released automatically. But there's also

one caveat. If you use this code and at least one frame later you want to access the `myNode` object, it'll be gone. Sending any messages to it will cause an `EXC_BAD_ACCESS` crash.

Simply adding the `CCNode* myNode` variable as a member to your class doesn't mean that the memory used by the object is automatically retained. If you want an autorelease object to stick around into the next and future frames, you do need to retain it and subsequently release it if you don't explicitly add it as a child node.

There's an even better way to use autorelease objects and keep them around without explicitly calling `retain`. Usually you'll create `CCNode` objects and add them to the scene hierarchy by adding the nodes as children to another `CCNode`-derived object. You can even get rid of the member variable if you want to, by relying on `cocos2d` to store the object for you.

```
// creating an autorelease instance of CCNode
-(void) init
{
    myNode = [CCNode node];
    myNode.tag = 123;

    // adding the node as children to self
    [self addChild:myNode];
}

-(void) update:(ccTime)delta
{
    // later access and use the myNode object again
    CCNode* myNode = [self getChildByTag:123];

    // do something with myNode
}
```

The magic is that `addChild` adds the object to a collection, in this case a `CCArray` that's similar to the `NSMutableArray` of the iOS SDK but faster. The `CCArray` and the `NSMutableArray` and any other iOS SDK collection automatically send a `retain` message to any object added to them and send a `release` message to any object removed from the collection. Thus, the object stays around and remains valid and can be accessed at a later time, yet it will automatically be released after it has been removed from the collection.

What you should keep in mind is that managing memory for `cocos2d` objects is best done as I described here. You may run into other developers who say that autorelease is bad or slow and shouldn't be used. Don't give in to them.

NOTE: The Apple developer documentation recommends reducing the number of autorelease objects. Most cocos2d objects, however, are created as autorelease objects. It makes memory management much easier, as I've shown.

If you start using `alloc/init` and `release` for every cocos2d object, you'll get yourself into a lot of pain for little to no benefit. That isn't to say that you'll never use `alloc/init`; it does have its uses and is sometimes even required. But for cocos2d objects, you should rely on using the static autorelease initializers.

Autorelease objects have only one caveat, which is that their memory is in use until the game advances by one frame. This means if you create a lot of throwaway autorelease objects every frame, you might be wasting memory. But that's actually a rare occurrence.

This concludes my quick primer on cocos2d memory management. For a more in-depth discussion of memory management, refer to Apple's Memory Management Programming Guide

(<http://developer.apple.com/iphone/library/documentation/Cocoa/Conceptual/MemoryMgmt/MemoryMgmt.html>).

Changing the World

What good is a template project like HelloWorld if I don't have you tinker with it at least a little? I'll have you change the world by touching it! How's that for a start?

First you'll make two changes to the `init` method to enable touch input and to use a tag value to retrieve the label at a later point. The changes are highlighted in Listing 2-3.

Listing 2-3. *Enabling Touch and Gaining Access to the Label Object*

```
-(id) init
{
    if ((self = [super init])) {
        // create and initialize a label
        CCLabelTTF* label = [CCLabelTTF labelWithString:@"Hello World"
                                                                fontFamily:@"Marker Felt"
                                                                fontSize:64];

        // get the window (screen) size from CCDirector
        CGSize size = [[CCDirector sharedDirector] winSize];

        // position the label at the center of the screen
        label.position = CGPointMake(size.width / 2, size.height / 2);

        // add the label as a child to this Layer
        [self addChild: label];

        // our label needs a tag so we can find it later on
        // you can pick any arbitrary number
        label.tag = 13;
    }
}
```

```

        // must be enabled if you want to receive touch events!
        self.isTouchEnabled = YES;
    }
    return self;
}

```

The label object gets 13 assigned to its tag property. Now why did you do that? I know, I told you to, but I must have had a reason, right? In the previous section, I explained that's how you can later access a child object of your class—you can refer to it by its tag. The tag number is completely arbitrary, other than that it must be a positive number and every object should have its own tag number, so there aren't two with the same number or you couldn't tell which you'd be retrieving.

TIP: Instead of using magic numbers like 13 as tag values, you should get in the habit of defining constants to use with tags. You'll have a hard time remembering what tag number 13 stands for, compared to writing a meaningful variable name like `kTagForLabel`. I'll get to this in Chapter 5.

In addition, `self.isTouchEnabled` is set to YES. This is a property of the `CCLayer` class and tells it that you want to receive touch messages. Only then will the method `ccTouchesBegan` be called:

```

-(void) ccTouchesBegan:(NSSet*)touches withEvent:(UIEvent*)event
{
    CCLabelTTF* label = (CCLabelTTF*)[self getChildByTag:13];
    label.scale = CCRANDOM_0_1();
}

```

By using `[self getChildByTag:13]`, you can access the `CCLabelTTF` object by its tag property, which you assigned in the `init` method. You can then use the label as usual. In this case, we use cocos2d's handy `CCRANDOM_0_1()` macro to change the label's scale property to a value between 0 and 1. This will change the label's size every time you touch the screen.

Since `getChildByTag` will always return the label, we can safely cast it to a (`CCLabelTTF*`) object. However, you should be aware that doing so will crash your game if the retrieved object is not derived from the `CCLabelTTF` class for some reason. This could easily happen if you accidentally give another object the same tag number 13. For that reason, it is good practice to use a defensive programming style and verify that what you're working with is exactly what you expect. Defensive programming uses assertions to verify that assumptions made are true. For this, you should use the `NSAssert` method:

```

-(void) ccTouchesBegan:(NSSet*)touches withEvent:(UIEvent*)event;
{
    CCNode* node = [self getChildByTag:13];

    // defensive programming: verify the returned node is a CCLabelTTF
    NSAssert([node isKindOfClass:[CCLabelTTF class]],
        @"node is not a CCLabelTTF!");
}

```

```
CCLabelTTF* label = (CCLabelTTF*)node;
label.scale = CCRANDOM_0_1();
}
```

In this case, we expect the node returned by `getChildByTag` to be an object derived from `CCLabelTTF`, but we can never be sure, which is why adding an `NSAssert` to verify the fact is helpful in finding errors before they lead to a crash.

Note that this adds two more lines of code, but in terms of performance things remain the same. The call to `NSAssert` is completely removed in Release builds, and the cast `CCLabelTTF* label = (CCLabelTTF*)node;` is what we've done already, just on the same line. Essentially, both versions perform exactly the same, but in the second case you get the benefit of being notified when you didn't get the expected `CCLabelTTF` object, instead of crashing with an `EXC_BAD_ACCESS` error.

What Else You Should Know

Since this is the "Getting Started" chapter, I think it's important to take the opportunity to introduce you to some vital but often overlooked aspects of iOS game development. I want you to be aware of the subtle differences among various iOS devices. In particular, available memory is often incorrectly considered because you can use only a fraction of each device's memory safely.

I also want you to know that the iOS Simulator is a great tool for testing your game, but it can't be used to assess performance, memory usage, and other features. The simulator experience can differ greatly from running your game on an actual iOS device. Don't fall into the trap of making assessments based on your game's behavior in the iOS Simulator. It's only the device that counts.

The iOS Devices

When you develop for iOS devices, you need to take into account their differences. Most independent and hobby game developers can't afford to purchase each slightly different iOS device, of which there are eight at the time of this writing, with roughly two more to be released each year. At the very least, you need to understand that there are important differences.

You might want to refer to Apple's spec sheets to familiarize yourself with the iOS device technical specifications. The following links list the iPhone, iPod touch, and iPad device specifications, respectively:

- <http://support.apple.com/specs/#iphone>
- <http://support.apple.com/specs/#ipodtouch>
- <http://support.apple.com/specs/#ipad>

Table 2–1 summarizes the most important hardware differences that concern game developers. The table lists iPod touch devices by generation since Apple doesn't use suffixes like "3G" for its iPod touch models. This table serves to show that the iOS

devices are not as homogenous as you might expect. For example, it is noteworthy that the second-generation iPod touch has a faster CPU than the second-generation iPhone 3G, but then the fourth-generation iPod touch has only half the memory of the iPhone 4. And the iPad 2 introduces for the first time a dual-core processor.

Table 2–1. *iOS Hardware Differences*

Device	Processor	Graphics	Resolution	Memory (RAM)
iPhone/iPod touch first generation	412 MHz	PowerVR MBX	480×320	128MB
viPhone 3G	412 MHz	PowerVR MBX	480×320	128MB
iPod touch second generation	533 MHz	PowerVR MBX	480×320	128MB
iPhone 3GS/iPod touch third generation	600 MHz	PowerVR SGX535	480×320	256MB
iPad	1 GHz	PowerVR SGX535	1024×768	256MB
iPhone 4	800 MHz	PowerVR SGX535	960×640	512MB
iPod touch fourth generation	800 MHz	PowerVR SGX535	960×640	256MB
iPad 2	2x 900 MHz	PowerVR SGX543	1024×768	512MB

As you can see, with every new generation, iOS devices usually have a faster CPU, a more powerful graphics chip, and increased memory and screen resolution. This trend will continue, with newer devices getting more and more powerful. If you plan to make money from iOS games, keep in mind that the older models still have significant market share, and this changes only very slowly, much slower than the rate at which new devices are released. Even today, if you do not design your game to run on second-generation devices, you are giving up a significant portion of the market!

Usually when game developers look at hardware features, they tend to focus on the CPU speed and graphics chip to assess what's technically possible. However, being mobile devices, the iOS devices until the most recent iPhone 4 are limited mostly by the amount of available RAM.

NOTE: RAM is not to be confused with the flash storage memory where MP3s, videos, apps, and photos are stored, of which even the smallest iOS device has 8GB. Flash storage memory is equivalent to the hard drive on your desktop computer. RAM is the memory your application uses to store code, data, and textures while it’s running.

About Memory Usage

Current iOS devices have 128MB, 256MB, or 512MB of RAM installed. However, that’s not the amount of memory available to apps. iOS uses a big chunk of memory all the time, and this is compounded by iOS multitasking introduced with iOS 4. Each device running iOS 4 or newer may be running various background tasks that use up an undefined additional amount of memory.

Over time, iOS developers have been able to close in on the theoretical maximum amount of RAM an app can use before it’s forcibly closed by the OS. Table 2–2 shows what you can expect to work with. Ideally, you want to keep your memory usage below the number in the Memory Warning Threshold column at all times. This is especially challenging on devices with only 128MB of RAM because there is only 20MB to 25MB of RAM more or less guaranteed to be available for an app. Around that point your app might start receiving Memory Warning notifications. You can ignore Memory Warning Level 1 notifications, but if the app continues to use more memory, you may get a Memory Warning Level 2 message, at which point the OS basically threatens to close your app if you don’t free some memory right now. It’s like your mom threatening not to buy your new computer if you don’t clean up your room right now! Please oblige.

Table 2–2. *Installed Memory Is Not Free Memory*

Installed Memory	Available Memory	Memory Warning Threshold
128MB	35MB to 40MB	20MB to 25MB
256MB	120MB to 150MB	80MB to 90MB
512MB	340MB to 370MB	260MB to 300MB (estimated)

Cocos2d can help you a little with freeing memory by calling the purge methods. By adding the `purgeCachedData` method to the `AppDelegate`’s `applicationDidReceiveMemoryWarning` method, you can have cocos2d try to free up some more unused memory:

```
- (void)applicationDidReceiveMemoryWarning:(UIApplication *)application {
    [[CCTextureCache sharedTextureCache] removeUnusedTextures];
    [[CCDirector sharedDirector] purgeCachedData];
}
```

But when your games get more complex, you may need to implement your own scheme of handling Memory Warning notifications. The problem inherent with Memory Warning

notifications is that they can cause performance glitches as your app frees big chunks of memory. If you rely on the cocos2d mechanisms to do this, it may remove a sprite's texture from memory, but if within a few frames that sprite is needed again, the texture will be reloaded during game play, which is slow. This can create noticeable stutters, so having your own memory management scheme implemented is preferable. You should be able to differentiate between textures that may be needed again soon and those that aren't needed at all right now. Unfortunately, there's no one size fits all solution or I'd provide you with one.

If you are developing on a device with 256MB or 512MB of memory, keep in mind that a great number of iOS devices are models with only 128MB. Unless you plan to limit your game to third-generation and newer iOS devices, you would do well to buy a cheap, used, first- or second-generation device and test your game primarily with that device in order to catch issues of excessive memory consumption early. That's when they are still easy and cheap to fix, especially if they require a change in the game's design. In general, it's advisable to use the device for development that's the weakest one available to you in terms of hardware capabilities. This helps you catch any performance or low-memory issues as early as possible.

NOTE: It's also recommended that you test your app on a multitasking-capable device that has a good number of background tasks running in order to test for a worst-case scenario of background tasks using up additional memory. Multitasking is available only for third-generation and newer devices, which means only devices with 256MB of RAM are allowed to run apps in the background. That's good news, since the 128MB of the first- and second-generation devices is barely enough for a single app. If you design for those devices, and I think you should, you don't have to worry too much about background tasks eating away your app's precious memory.

On devices with 128MB of RAM, you can at most allocate around 35MB to 40MB memory. Keep in mind that this is only a theoretical maximum; the number varies on each device and may even depend on which apps the user had previously used. This is the reason app developers recommend rebooting a device if you are experiencing crashes. It can free up some more memory. The number-one cause for apps to quit unexpectedly is that the device ran out of memory. So, be sure to be wary of your app's memory usage and to frequently reboot your devices when you get weird behavior.

You can measure memory usage using the Instruments application, which is explained in Apple's Instruments User Guide:
<http://developer.apple.com/iphone/library/documentation/DeveloperTools/Conceptual/InstrumentsUserGuide/Introduction/Introduction.html>.

The iOS Simulator

Apple's iOS SDK allows you to run and test iPhone and iPad applications on your Mac with the iOS Simulator. The primary purpose of the iOS Simulator is to let you more rapidly test your application because deployment to an iOS device takes longer and

longer as your game gets bigger and bigger. Games in particular use a lot of images and other assets that need to be transferred, slowing down deployment.

However, there are several caveats to using the iOS Simulator. The following sections reveal what the iOS Simulator does not allow you to do. For all these reasons, it is recommended that you test your game early and often on a device. At least after every major change or near the end of the day, you should run a test on your iOS device to verify that the game behaves exactly as intended.

Can't Assess Performance

The performance of your game running in the iOS Simulator depends entirely on your computer's CPU. The graphics-rendering process does not even use the hardware acceleration capabilities of your Mac's graphics chip. That's why the framerate of your game running in the simulator has no meaning at all. You can't even be sure that comparing the framerate before and after a change will reveal the same results on the device. In extreme cases, the framerate in the simulator may go up even as it goes down on the device. Always do your performance testing on the device, using the Release build configuration.

Can't Assess Memory Usage

The iOS Simulator is able to use all the memory available on your computer, so there's a lot more memory available on the simulator than on the device. This means you won't get Memory Warning notifications and your game will run fine on the iOS Simulator, but you may be in for a shock (a crash) when you try the game for the first time on an iOS device.

You can, however, assess how much memory is currently used by your game using the iOS Simulator.

Can't Use All iOS Device Features

Some features, such as device orientation, can be simulated using menu items or keyboard shortcuts, but this comes nowhere close to the experience of a real device. And certain hardware features, such as multitouch input, accelerometer, vibration, or obtaining location information can't be tested at all on the iOS Simulator because your computer's hardware can't simulate these features. No, it doesn't help to shake your Mac or touch its screen. Try it if you don't believe me.

TIP: The iSimulate app (www.vimov.com/isimulate) is an invaluable development tool, as it allows an iOS device to send accelerometer, GPS, compass and multi-touch events to an app running in the iOS Simulator.

Runtime Behavior Can Differ

From time to time you may encounter nasty cases where a game runs just fine on the iOS Simulator but crashes on the device or the game slows down for no apparent reason. There may also be graphical glitches that appear only on the iOS Simulator or only on the device. If in doubt and before delving into a prolonged quest to figure out what's wrong, always try running your game on the device if you're having trouble on the iOS Simulator, or vice versa. Sometimes, the problem may just go away, but if not, you may get a hint about what's going on.

About Logging

By default, an Xcode project will have two build configurations named Debug and Release. The main difference between the two is that only in Debug builds are certain functions like `CCLOG` compiled. This is typically controlled by preprocessor macros like `DEBUG` and `COCOS2D_DEBUG`, which control the level of debugging code built into the app. That's the single most important factor when it comes to performance variations between Debug and Release builds.

NOTE: The `CCLOG` macro wraps Apple's `NSLog` method so that `CCLOG` is compiled only in Debug builds but omitted in Release builds. I recommend using `CCLOG` in place of `NSLog` because logging is for your eyes only. `NSLog` can slow down your published game because it will be run even in Release builds!

One `NSLog` or `CCLOG` in the wrong place can spam the Debugger Console window with logging messages, causing slowdowns and lag. Logging is very slow, and a continuous stream of log messages printed to the Debugger Console can drag your game's performance down to a crawl. If you suspect your game's performance to be particularly slow in Debug builds, always check the Debugger Console for excessive logging activities. From the **Run** menu in Xcode, select **Console** to show the Debugger Console window.

The exclusion of logging and typically better code optimization settings are the main reason you should only use Release builds to test your game's performance.

Summary

Wow, that was a lot for a "Getting Started" chapter! In the first part of this chapter you learned to download and set up all the necessary tools to the point where you had your first cocos2d template project running.

I then walked you through the workings of the template project to get you up to speed with how an iOS cocos2d application works in principle and somewhat in detail as well. I do have a pet peeve about proper memory management, which is why I also included

those details. I think it's important because it's easy to misunderstand or even completely ignore memory management, and then you might be building your game on a very crumbly foundation.

I did manage to sneak in a short "do it yourself" section to at least show you how touch input is done in cocos2d and how cocos2d objects are stored and retrieved.

Finally, I thought it important to give you the details about the various iOS devices and what you can expect in terms of available memory. I also discussed the simulator and how it differs for testing your game compared with testing it on a device.

In the next chapter, you'll learn all the essential features of cocos2d, which will bring you closer to making a complete cocos2d game.

Essentials

This chapter will introduce you to the essential building blocks of the cocos2d game engine. You'll be using most of these classes in every game you create, so understanding what's available and how these classes work together will help you write better games. Armed with this knowledge, you'll find it a lot easier to start working with cocos2d.

Accompanying this chapter is an Xcode project named Essentials that includes everything I discuss here plus additional examples. The source code is full of comments, so you can read it as if it were an appendix to the book.

We'll start with a high-level overview of the cocos2d game engine architecture. Every game engine is different in the way game objects are managed and presented on the screen. It's best to begin with an understanding of what the individual elements are and how they fit together.

The cocos2d Scene Graph

Sometimes called a *scene hierarchy*, the scene graph is a hierarchy of every cocos2d node that's currently active.

A cocos2d node is any object that is derived from the CCNode class. Most nodes like CCSprite and CCLabelTTF are displayed on the screen, but a few have no visual representation, including CCNode as well as CCScene and CCLayer. They are no less important; to the contrary, they frequently cause confusion among new cocos2d developers.

I'll go into more detail on those classes and explain what they are used for in the upcoming sections. For now, I'll focus on the high-level concepts, and it's sufficient for you to know that CCSprite displays a texture on the screen, CCLabelTTF prints arbitrary text, and CCNode, CCScene, and CCLayer are used mainly to group nodes together.

Figure 3–1 depicts the shoot 'em up game that you'll create starting in Chapter 6 and illustrates what is not immediately noticeable to players of the game but very important to understand for game developers.



Figure 3–1. *A shoot 'em up game*

The scene in Figure 3–1 is entirely made of just `CCSprite` objects. At least that's what you can see. What you can't see is how the `CCScene` and `CCLayer` classes are used to group and order the various sprites: several background layers, the player's ship, the enemies, the bullets, and the virtual joypad and button. To illustrate the layering of this scene's elements, I created an exploded view drawing of that scene in Figure 3–2.

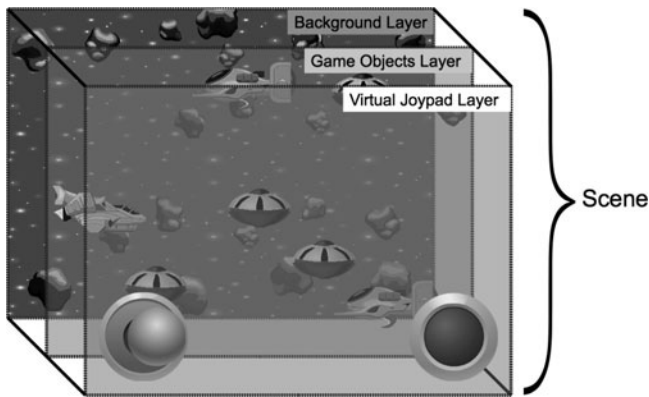


Figure 3–2. *An exploded view illustrating a typical cocos2d game scene layout*

In this particular case, three layers help maintain the proper draw order of the sprites in relation to each other: the background layer, the game objects layer, and the virtual joypad layer. Using multiple layers in a scene is also helpful if you want to hide an entire layer's nodes, move the layer and thus all the nodes it contains, or reorder the layer so that its nodes are drawn above or behind another layer's nodes. You can even rotate and scale a layer, which would rotate and scale all the nodes contained in that layer. This makes using layers a powerful concept.

In this respect, the game scene is modeled using layers just like you edit an image in an image-editing program such as Photoshop, Seashore, or Gimp. However, the nodes (brush strokes) in each layer aren't static and instead remain individual elements.

The actual `Scene` object is just the container for all the layers (the actual image so to speak), just like a layer is a container for other nodes. Each of these nodes can run logic,

the scene, the layers, the individual nodes, the sprites, the labels, and so on, depending on how you organize your code.

Any node can have any other node as a child, and every node in the hierarchy—except for the scene itself—has one parent object, which is the object the node is a child of. If you remove the node from the scene graph or you haven't added it yet, it will not have a parent object. Note that this parent-child relationship of nodes is not to be confused with inheritance in object-oriented programming. In other words, the parent node is not the node's super class!

This treelike hierarchy of nodes that you can create is what is called the *scene graph*, and throughout this book I'll sometimes refer to it as the *node hierarchy*. For those familiar with programming design patterns, you'll recognize this hierarchical structure as the Composite design pattern. Figure 3–3 shows you what the simplified node hierarchy of this scene looks like as a tree structure.

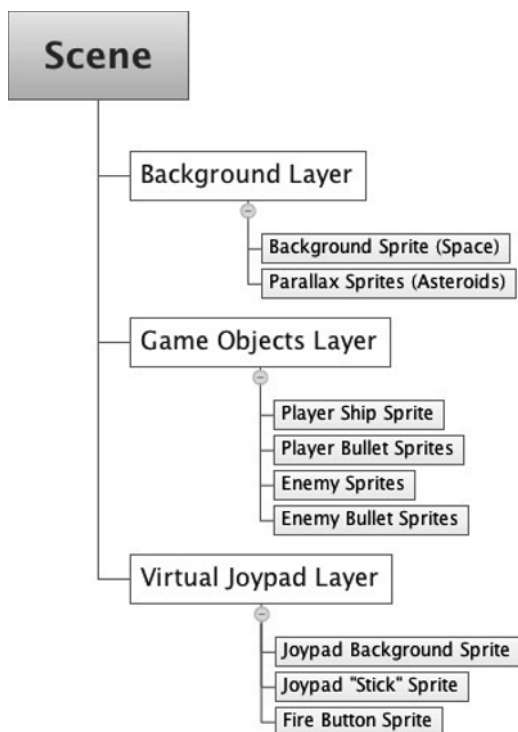


Figure 3–3. The node hierarchy of the scene illustrated in Figure 3.2

Note that this scene/layer/nodes structure is not enforced by cocos2d, except that the scene graph always has a `CCScene` class object as its root node. But other than that, you can use any `CCNode` class instead of `CCLayer` to create your layers.

In fact, I generally prefer to use the plain `CCNode` class over `CCLayer` for layering and grouping objects; in most cases, the `CCLayer` class adds unnecessary overhead because it is able to receive touch and accelerometer input on Mac OS X keyboard and mouse

input. Strip away support for input handling, and the CCLayer class is practically just a CCNode class. The same goes for CCScene, which is also just an abstract concept in order to enforce a common root node class. But otherwise a CCScene class is virtually the same as a CCNode.

CAUTION: The nodes in a cocos2d node hierarchy are positioned relative to their parent nodes. The child nodes inherit certain properties from their parents, such as `scale` and `rotation` but not `color` and `opacity`. This may be confusing the first time you experience this.

For example, if the parent of a CCLabelTTF node is a nondrawing node like the CCNode, CCScene, or CCLayer nodes and they themselves are the only children of other nondrawing nodes, the position of the label will be relative to the view's lower-left corner. So, all is well and as expected. But if you were to add another CCLabelTTF as a child to this label, the position of the child label will be relative to the parent label.

You would have reason to expect the child label to be centered on the parent label's position. Alas, this is not the case, as you can see in Figure 3.4. You'll find that the child label will be centered on the lower-left corner of the parent label's texture instead, which is an unfortunate oddity in cocos2d's design. To correctly position such a node and center it on its parent node, you will have to use parent node's `contentSize.width / 2` and `contentSize.height / 2` as the child node's `position.x` and `position.y`.

I created the NodeHierarchy project to give you examples of relative positioning, and rotation, of nodes in a parent-child relationship and to allow you to familiarize yourself with the cocos2d node hierarchy. Consider it a test bed for your own experiments.

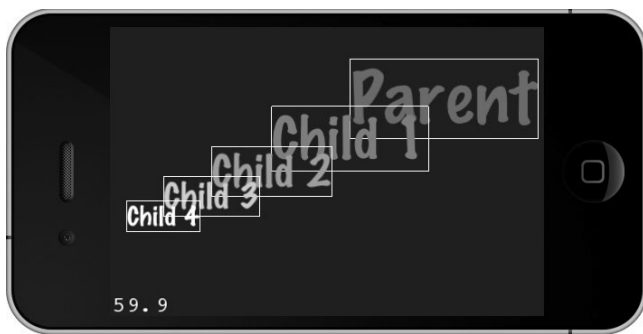


Figure 3-4. *Caution: the default position of a child node is unexpectedly offset from its parent's position.*

The CCNode Class Hierarchy

At this point, you may be wondering what classes derive from CCNode. Figure 3-5 shows the CCNode class hierarchy. The node classes that you'll be working with the most are highlighted, and you can make quite impressive games with just these classes. I will

explain the most important classes shortly, and over the course of the book, you'll get to know even the more obscure ones in depth.

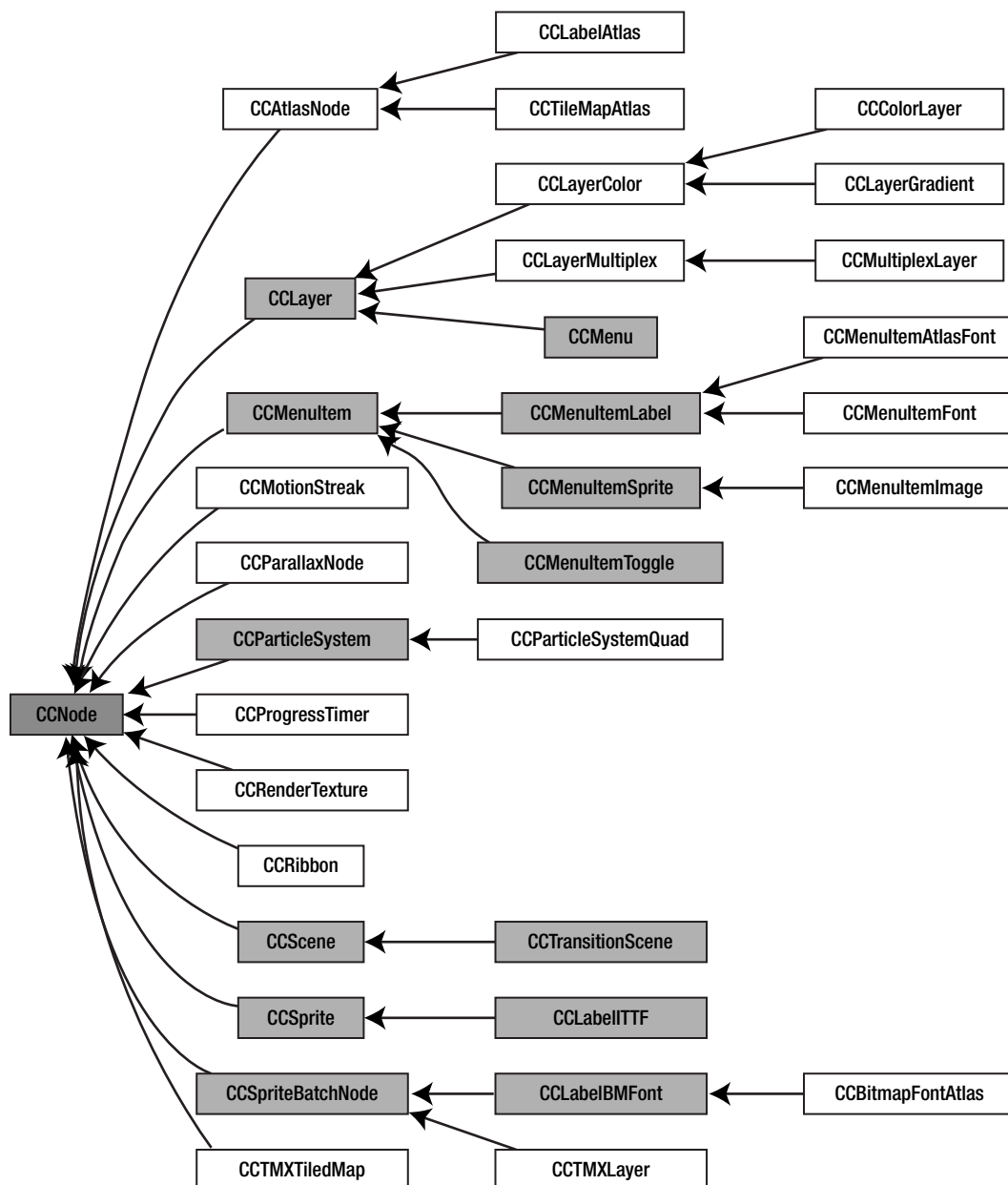


Figure 3–5. *The CCNode class hierarchy*

CCNode

CCNode is the base class for all nodes. It's an abstract class that has no visual representation and defines all properties and methods common to all nodes.

Working with Nodes

The CCNode class implements all the methods to add, get, and remove child nodes. Here are some of the ways you can work with child nodes:

- You can create a new node:

```
CCNode* childNode = [CCNode node];
```
- You can add the new node as a child:

```
[myNode addChild:childNode z:0 tag:123];
```
- You can retrieve the child node:

```
CCNode* retrievedNode = [myNode getChildByTag:123];
```
- You can remove the child node by tag; cleanup will also stop any running actions:

```
[myNode removeChildByTag:123 cleanup:YES];
```
- You can remove the node if you have a pointer to it:

```
[myNode removeChild:retrievedNode];
```
- You can remove every child of the node:

```
[myNode removeAllChildrenWithCleanup:YES];
```
- You can remove myNode from its parent:

```
[myNode removeFromParentAndCleanup:YES];
```

The `z` parameter in `addChild` determines the draw order of the node. The node with the lowest `z` value is drawn first; the one with the highest `z` value is drawn last. If multiple nodes have the same `z` value, they are simply drawn in the order they were added. Of course, this applies only to nodes that have a visual representation, such as sprites.

The `tag` parameter lets you identify and obtain specific nodes at a later time using the `getChildByTag` method.

NOTE: If several nodes end up with the same tag number, `getChildByTag` will return the first node with that tag number. The remaining nodes will be inaccessible. Make sure you use unique tag numbers for your nodes.

Note that actions can have tags, too. Node and action tags do not conflict, however, so an action and a node can have the same tag number without any problem.

Working with Actions

Nodes can also run actions. I'll cover actions more in a bit. For now, just know that actions can move, rotate, and scale nodes and do other things with nodes over time.

- Here's an action declaration:

```
CCAction* action = [CCBlink initWithDuration:10 blinks:20];  
action.tag = 234;
```

- Running the action makes the node blink:

```
[myNode runAction:action];
```

- If you need to access the action at a later time, you get it by its tag:

```
CCAction* retrievedAction = [myNode getActionByTag:234];
```

- You can stop the action by tag:

```
[myNode stopActionByTag:234];
```

- Or you can stop it by pointer:

```
[myNode stopAction:action];
```

- Or you can stop all actions running on this node:

```
[myNode stopAllActions];
```

Scheduled Messages

Nodes can schedule messages, which is Objective-C lingo for calling a method. In many cases, you'll want a particular update method to be running on a node in order to do some processing, such as checking for collisions. The simplest way to schedule the particular update method to be called every frame is like this:

```
-(void) scheduleUpdateMethod  
{  
    [self scheduleUpdate];  
}  
  
-(void) update:(ccTime)delta  
{  
    // this method is called every frame  
}
```

Dead simple, isn't it? Notice that the update method has a fixed signature, meaning it's always defined exactly this way. The delta parameter is the elapsed time since the method was last called. This is the preferred way to schedule updates that should take place every frame, but there are reasons to use other update methods that give you more flexibility.

If you want a different method to be run or if you don't want the method to be called every frame but every tenth of a second, you should use this method:

```

-(void) scheduleUpdateMethod
{
    [self schedule:@selector(updateTenTimesPerSecond:) interval:0.1f];
}

-(void) updateTenTimesPerSecond:(ccTime)delta
{
    // this method is called according to its interval, ten times per second
}

```

Note that if interval is 0, you should use the `scheduleUpdate` method instead. However, the previous code is the preferred choice if you ever need to unschedule a particular selector at a later time. The `scheduleUpdate` method won't let you do this.

The update method's signature is still the same; it receives a delta time as its only parameter. But this time it can be named any way you want, and it is called only every tenth of a second. This may be useful to check for win conditions if they are so complex you don't want to run them every frame. Or if you want something to happen after 10 minutes, you could schedule a selector with an interval of 600.

NOTE: The `@selector()` syntax may seem weird. It's the Objective-C way of referring to a specific method by name. The crucial thing here is not to overlook the colon at the end. It tells Objective-C to look for the method with the given name and exactly one parameter. If you forget to add the colon at the end, the program will still compile, but it will crash later. In the Debugger Console, the error log will read "unrecognized selector sent to instance ."

The number of colons in `@selector()` must always match the number and names of the parameters of the method. For the following method:

```
-(void) example:(ccTime)delta sender:(id)sender flag:(bool)aBool
```

the corresponding `@selector` should be as follows:

```
@selector(example:sender:flag:)
```

There's one major caveat with scheduling your own selectors, or with the `@selector()` keyword in general. By default, the compiler does not complain at all if the method's name doesn't exist. Instead, your app will simply crash when that selector is called. Since the call is done by cocos2d internally, you'll find it hard to figure out the cause of the problem. Luckily, there's a compiler warning you can enable. Figure 3–6 shows the `Undeclared Selector` warning enabled for the `NodeHierarchy` project, and the `Essentials Xcode` project for this chapter has it enabled as well.



Figure 3-6. Activating the build setting to warn about undeclared selectors

What's left is to show how to stop these scheduled methods from being called. You can do so by unscheduling them.

- You can stop all selectors of the node, even those scheduled with `scheduleUpdate`:
`[self unscheduleAllSelectors];`
- You can stop a particular selector, in this case the `updateTenTimesPerSecond` method:
`[self unschedule:@selector(updateTenTimesPerSecond:)];`

Note that this won't stop the update method scheduled by `scheduleUpdate`.

There's also a useful trick for scheduling and unscheduling selectors. Quite often you'll find that in the scheduled method you want a particular method to no longer be called, without having to replicate the exact name and number of parameters, because they can change. Here's how you'd run a scheduled selector and stop it on the first call:

```
-(void) scheduleUpdates
{
    [self schedule:@selector(tenMinutesElapsed:) interval:600];
}

-(void) tenMinutesElapsed:(ccTime)delta
{
    // unschedule the current method by using the _cmd keyword
    [self unschedule:_cmd];
}
```

The hidden variable `_cmd` is available in all Objective-C methods. It is the selector of the current method. In the previous example, `_cmd` is equivalent to writing `@selector(tenMinutesElapsed:)`. Unscheduling `_cmd` effectively stops the `tenMinutesElapsed` method from ever being called again. You can also use `_cmd` for scheduling the selector in the first place, if you want the current method to be scheduled. Let's assume you need a method called at varying intervals and this interval is changed each time the method is called. In this case, your code can make use of `_cmd` like this:

```

-(void) scheduleUpdates
{
    // schedule the first update as usual
    [self schedule:@selector(irregularUpdate:) interval:1];
}

-(void) irregularUpdate:(ccTime)delta
{
    // unschedule the method first
    [self unschedule:_cmd];

    // I assume you d have some kind of logic other than random to determine
    // the next time the method should be called
    float nextUpdate = CCRANDOM_0_1() * 10;

    // then re-schedule it with the new interval using _cmd as the selector
    [self schedule:_cmd interval:nextUpdate];
}

```

Using the `_cmd` keyword will save you a lot of pain in the long run because it avoids the dreaded issue of scheduling or unscheduling the wrong selector, and it decouples the code from having to know the name of the method it is used in.

There's one final scheduling issue to mention, and that's prioritizing updates. Take a look at the following code:

```

// in Node A
-(void) scheduleUpdates
{
    [self scheduleUpdate];
}

// in Node B
-(void) scheduleUpdates
{
    [self scheduleUpdateWithPriority:1];
}

// in Node C
-(void) scheduleUpdates
{
    [self scheduleUpdateWithPriority:-1];
}

```

This may take a moment to sink in. All nodes are still calling the same `...(void) update:(ccTime)delta` method for themselves. However, scheduling the update methods with a priority causes the one in Node C to be run first. Then the one in Node A is called because, by default, `scheduleUpdate` uses a priority of 0. Node B's update method is called last because it has the highest number. The update methods are called in the order from lowest-priority number to highest.

You might wonder where that's useful. To be honest, they're rarely needed, but in those cases it's quite useful to be able to prioritize updates, such as when applying forces to physics objects before or after the physics simulation itself is updated or ensuring that the game-over condition is checked only after all game objects have run their update

methods. And sometimes, usually late in the project, you may discover an odd bug that turns out to be a timing issue, and it forces you to run the player's update method after all other objects have updated themselves.

Until you need to solve a particular problem by using prioritized updates, you can safely ignore them. Also, keep in mind that each prioritized update method call adds a little overhead because the methods need to be called in a specific order.

Director, Scenes, and Layers

Like `CCNode`, the `CCScene` and `CCLayer` classes have no visual representation and are used internally as abstract concepts for the starting point of the scene graph, which is always a `CCScene`-derived object. The `CCScene` class is the container for all other nodes of the scene. And `CCLayer` is typically used to group nodes together, particularly to maintain the correct drawing order among multiple layers. But also to receive touch and accelerometer input on iOS, respectively, mouse and keyboard input on Mac OS X.

I'll start this section with the `CCDirector` class, however, because it is the class that, among other things, allows you to run and replace scenes.

The Director

The `CCDirector` class, or simply `Director` for short, is the heart of the cocos2d game engine. If you recall the `HelloWorld` application from Chapter 2, you'll remember that a lot of the cocos2d initialization procedure involved calls to `[CCDirector sharedDirector]`.

The `CCDirector` class is a singleton, which means there can be only one instance of the `CCDirector` class at any time, and it can be accessed globally by calling the class method `sharedDirector`. For now that's all you need to know about singletons. Later in this chapter, the [A Note on Singletons in cocos2d](#) section will explain what a singleton is and what other singleton classes cocos2d has to offer.

The `CCDirector` class stores global configuration settings for cocos2d and also manages the cocos2d scenes. The major responsibilities of the `CCDirector` class include the following:

- Providing access to the currently running scene
- Running, replacing, pushing, and popping scenes
- Providing access to cocos2d configuration details
- Providing access to cocos2d's OpenGL view and window
- Pausing, resuming, and ending the game
- Converting UIKit and OpenGL coordinates
- Determining how game state is updated

You can choose from four different types of Directors, which affects the way the Director updates the game states. This in turn can have a great impact on rendering performance and compatibility with UIKit views. You'll typically find these lines in the app delegate class, which sets the Director type:

```
// Try CADisplayLink director first.  
// If CADisplayLink is not available (iOS < 3.1) fall back to NSTimer director.  
if ( ! [CCDirector setDirectorType:kCCDirectorTypeDisplayLink] )  
    [CCDirector setDirectorType:kCCDirectorTypeDefault];
```

You can set four Director types:

- `kCCDirectorTypeDisplayLink` (fastest; requires iOS 3.1 or newer)
- `kCCDirectorTypeNSTimer` (slowest)
- `kCCDirectorTypeThreadMainLoop` (fast but has issues with UIKit views)
- `kCCDirectorTypeMainLoop` (fast but has issues with UIKit views)

The most commonly used and recommended Director type is the `kCCDirectorTypeDisplayLink`, which uses Apple's `CADisplayLink` class internally. It's the preferred and default choice and available only on iOS 3.1 or newer. The `CADisplayLink` class ensures that screen updates are synchronized with the screen refresh, which enables very smooth scrolling. It also cooperates well with UIKit views.

TIP: Since all devices can be upgraded to iOS 3.1.3 and iOS 3.1 was already released in September 2009, it is safe to assume that there's only a negligible fraction of devices left that do not run at least iOS 3.1. Various device statistics published by app developers and analytics companies support that notion and even indicate an adoption rate for iOS 4 of more than 90 percent for iPhone devices, as of April 2011. To ensure that the `CADisplayLink` class is available on any device running your app, you should set your project's iOS deployment target to iOS 3.1 and not iOS 3.0 (the lowest deployment target Xcode currently allows you to choose).

The default fallback option is `kCCDirectorTypeNSTimer`, which by default will be used only on devices running iOS 3.0.1 or older. Using the `kCCDirectorTypeNSTimer` is not recommended because it's the slowest Director type and uses an `NSTimer` object internally to drive frame updates. The resolution of `NSTimer` is somewhere around 50 to 100 milliseconds according to Apple's documentation. If you consider that you need one update every 16.6 milliseconds to render 60 frames per second, don't expect to be able to render 60 fps particularly on first- and second-generation devices when using `kCCDirectorTypeNSTimer`. Worse yet, the time between updates can fluctuate significantly because of reasons beyond your control that can result in a fluctuating framerate. At least it cooperates well with UIKit views.

The faster fallback options to the `kCCDirectorTypeNSTimer` would be the `kCCDirectorTypeMainLoop` or the `kCCDirectorTypeThreadMainLoop`, that is, unless you want to add UIKit views to your game. In that case, these two Director types can cause slow and unresponsive UIKit views and other issues. This is because they drive `cocos2d`

updates using an old-school, noncooperative update loop that will start the next update as soon as the last one ended. This leaves little to no CPU time for other things, including UIKit views, although the `kCCDirectorTypeThreadMainLoop` behaves better in this regard.

To sum up, use the `kCCDirectorTypeDisplayLink` (the default) and set your project's iOS deployment target to iOS 3.1 or newer (see Figure 3–7) to ensure the `CADisplayLink` is always available on all devices running your app. The other Director types can be considered relics of the ancient past (as in: last month's computer technology) but may still be useful in some obscure scenarios.

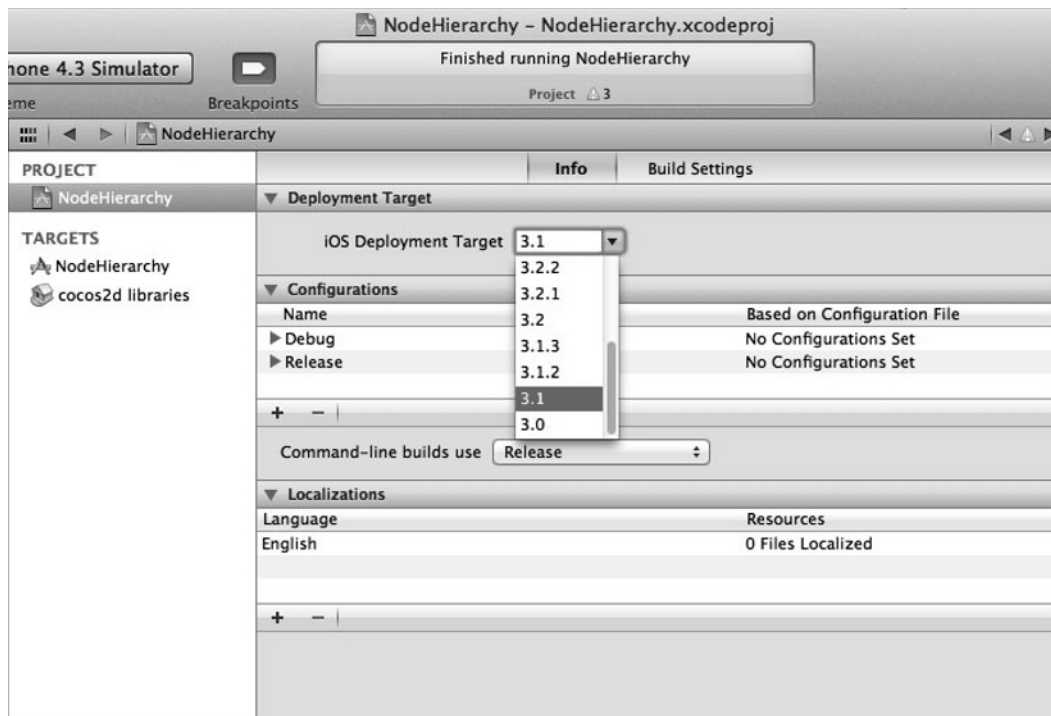


Figure 3–7. Setting the iOS deployment target to 3.1 ensures that the `CADisplayLink` Director type is available.

CCScene

A `CCScene` object is always the first node in the scene graph. In cocos2d, a scene is an abstract concept, and the `CCScene` class contains virtually no additional code compared to `CCNode`. But the `CCDirector` requires a `CCScene`-derived class to be able to change the currently active scene graph via the `CCDirector` `runWithScene`, `replaceScene`, and `pushScene` methods. You can also wrap a `CCScene` class into a class derived from `CCSceneTransition` in order to animate the transition between a currently running scene and the new scene.

Normally the only children of a `CCScene` are those derived from `CCLayer`, which in turn contain the individual game objects, although this convention is not enforced by

cocos2d. Since the scene object itself in most cases doesn't contain any game-specific code and is rarely subclassed, it's most often created in a static method `+(id) scene` inside the `CCLayer` object. I already mentioned this method in Chapter 2, but here it is again to refresh your memory:

```
+(id) scene
{
    CScene *scene = [CScene node];
    CCLayer* layer = [HelloWorld node];
    [scene addChild:layer];

    return scene;
}
```

The very first place you'll create a scene is at the end of the app delegate's `applicationDidFinishLaunching` method. You use the `Director` to start the first scene with the `runWithScene` method:

```
// only use this to run the very first scene
[[CCDirector sharedDirector] runWithScene:[HelloWorld scene]];
```

For all subsequent scene changes, you must replace the existing scene with the aptly named `replaceScene` method:

```
// use replaceScene to change all subsequent scenes
[[CCDirector sharedDirector] replaceScene:[HelloWorld scene]];
```

As you'll soon learn in an upcoming section, you can also use transitional effects when replacing scenes. The following is an example that uses the `CCTransitionShrinkGrow` class as an intermediary scene that manages the transition animation:

```
CScene* scene = [HelloWorld scene];
CSceneTransition* tran = [CCTransitionShrinkGrow transitionWithDuration:2 scene:scene];
[[CCDirector sharedDirector] replaceScene:tran];
```

NOTE: If you run this code in the `HelloWorld` scene, it will work just fine. It will create a new instance of `HelloWorld` and replace the old one, effectively reloading the scene. However, don't try to reload the current scene by passing `self` as a parameter to `replaceScene`. This will freeze your game!

Scenes and Memory

Keep in mind that when you replace one scene with another, the new scene is loaded into memory before the old scene's memory is freed. This creates a short spike in memory usage. Replacing scenes is always a crucial point where you can run into memory warnings or straight into a crash related to not enough free memory. You should test scene switching early and often when your app uses a lot of memory.

NOTE: Cocos2d does a good job of cleaning up its own memory when you replace one scene with another. It removes all nodes, stops all actions, and unschedules all selectors. I mention this because sometimes I come across code that makes explicit calls to the respective `removeAll` cocos2d methods. Remember, if in doubt, trust cocos2d's memory management.

This issue becomes even more pronounced when you start using transitions. What happens then is that the new scene is created, the transition runs, and only after the transition has done its job is the old scene removed from memory. It's good practice to add log statements to your scene or, respectively, to the layer that creates the scene:

```
-(id) init
{
    if ((self = [super init]))
    {
        CCLOG(@"%@: %@", NSStringFromSelector(_cmd), self);
    }
}

-(void) dealloc
{
    CCLOG(@"%@: %@", NSStringFromSelector(_cmd), self);

    // always call [super dealloc] in the dealloc method!
    [super dealloc];
}
```

Keep an eye on these log messages. If you ever notice that the `dealloc` log message is never sent when you switch from one scene to another, this is a huge warning sign. In that case, you're leaking the whole scene, not freeing its memory. Something like that is extremely unlikely to be caused by cocos2d. In almost all cases, it will boil down to retaining or not properly releasing nodes.

One thing you should never do is add a node as a child to the scene graph and then retain it yourself for some other purpose. Use cocos2d's methods to access node objects instead, or at the very least keep a weak reference to the pointer instead of retaining it. As long as you let cocos2d worry about managing the memory of nodes, you should be fine.

Pushing and Popping Scenes

While I'm still talking about changing scenes, I should mention the `pushScene` and `popScene` methods of the `Director`, which can be useful tools. What they do is run the new scene without removing the old scene from memory. If you think of your scenes as sheets of paper, then pushing a scene means that a new sheet of paper will be added on top of the currently visible sheet. The paper underneath stays in place and in memory, as opposed to replacing scenes. For each pushed scene, you then need to call `popScene` (this is equal to removing the topmost sheet of paper) until only the initial scene is left.

The idea is to make changing scenes faster. But there's the conundrum: if your scenes are lightweight enough that they can share memory with each other, they'll load fast anyway. And if they are complex and thus slow to load, chances are they take away each other's precious memory□ and memory usage quickly gets to a critical level.

The biggest issue with `pushScene` and `popScene` is that you need to keep track of how many scenes were pushed in order to pop the exact amount. If you're not very careful about managing your pushes and pops, you'll end up forgetting a pop or popping one scene too many. And this is besides the fact that all those scenes have to share the same memory.

There are cases where `pushScene` and `popScene` are very handy; one is if you have one common scene that's used in many places, such as the Settings screen where you can change music and sound volume. You can push the Settings scene to display it, and the Settings scene's Back button then simply calls `popScene`; the game will return to the previous state. Whether you opened the Settings menu from the main menu, from within the game, or someplace else, this technique works fine and lets you avoid having to keep track of where the Settings menu was opened.

Another case where `pushScene` and `popScene` are very useful is if you want to retain the state of the initial scene without having to revert to saving and loading that scene's state. For example, you could push the scene that shows the current leaderboard in a multiplayer game and then pop it again without having to save and load the game state, because the game scene remains in memory. And the game hasn't advanced while the player was inspecting the leaderboard because the game scene was automatically paused while the leaderboard scene was visible.

However, you need to ensure that there's always enough additional free memory available for the pushed scene at any time the scene could be pushed, which is hard to test for. It is recommended that any scene you may want to push should be very lightweight, consume little memory, and should only pop itself but never push other scenes or even call `replaceScene`.

For example, to display a Settings scene from anywhere on top of the current scene, use this code:

```
[[CCDirector sharedDirector] pushScene:[Settings scene]];
```

Now inside the Settings scene you need to call `popScene` when you want to go back to the previous scene that's still in memory:

```
[[CCDirector sharedDirector] popScene];
```

CAUTION: You can animate the `pushScene` call only with `CCSceneTransition` classes and not `popScene`. This is a drawback of pushing and popping scenes that you should be aware of.

CCTransitionScene

Transitions, meaning any class derived from CCTransitionScene, can give your game a really professional look. Figure 3–8 gives you an overview of the CCTransitionScene class hierarchy and shows the available transition classes. Figure 3–8 also complements the CCNode class hierarchy in Figure 3–5, where the CCTransitionScene subclasses were not included for the sake of clarity.

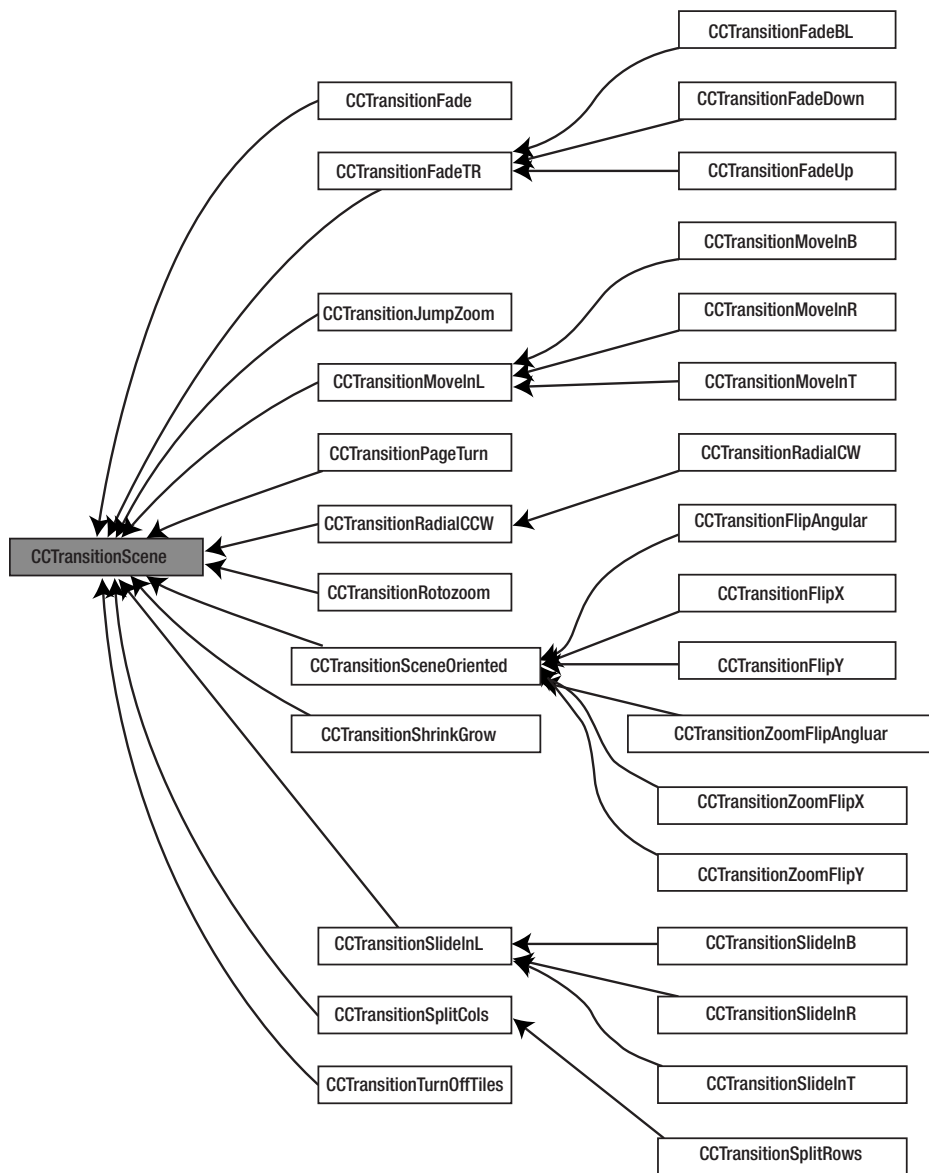


Figure 3–8. The CCTransitionScene class hierarchy

CAUTION: Not every transition is really useful in a game, even though they all look neat. What players care most about is the speed of a transition. Even just three seconds before they can interact with the next scene is quite a strain; I prefer to transition scenes within a second or avoid them altogether if it better fits the situation.

What you should certainly try to avoid is picking random transitions when replacing scenes. Players don't care, and as game developers, we know you just got a little over-excited by how cool the transitions are. If you don't know which transition is right for a particular change of scene, you shouldn't use any at all. In other words, just because you can doesn't mean you should.

Transitions add only one more line of code to replacing a scene, although admittedly that one line can be a long one given how long the names of transitions are and how they get even longer with the number of parameters they often require. Here's the very popular fade transition as an example; it fades to white in one second:

```
// initialize a transition scene with the scene we d like to display next
CCTransitionFade* tran = [CCTransitionFade transitionWithDuration:1
                           scene:[HelloWorld scene]
                           withColor:ccwHITE];

// use the transition scene object instead of HelloWorld
[[CCDirector sharedDirector] replaceScene:tran];
```

You can use a CCTransitionScene with replaceScene and pushScene, but as I said earlier, you can't use a transition with popScene.

A variety of transitions are available, although most are variations of directions, as in where the transition moves to or from which side it starts. Here's a list of the currently available transitions, along with a short description for each:

- CCTransitionFade: Fades to a specific color and back.
- CCTransitionFadeTR (three more variations): Tiles flip over to reveal a new scene.
- CCTransitionJumpZoom: Scene bounces and gets smaller; new scene does the reverse.
- CCTransitionMoveInL (three more variations): Scene moves out; new scene moves in at the same time, either from left, right, top, or bottom.
- CCTransitionSceneOriented (six more variations): A variety of transitions flipping the whole scene around.
- CCTransitionPageTurn: An effect like turning a page.
- CCTransitionRadialCCW (one variation): Like a radar screen that reveals the new scene with a radial wipe animation.
- CCTransitionRotoZoom: Scene rotates and gets smaller; new scene does reverse.

- `CCTransitionShrinkGrow`: Current scene shrinks; new scene grows over it.
- `CCTransitionSlideInL` (three more variations): New scene slides over the current scene, either from left, right, top, or bottom.
- `CCTransitionSplitCols` (one variation): Columns of scene move up or down to reveal new scene.
- `CCTransitionTurnOffTiles`: Tiles randomly replaced by tiles of new scene.

CCLayer

Sometimes you need more than one layer in your scene. In that case, you can add more `CCLayer` objects to your scene like this:

```
+(id) scene
{
    CScene* scene = [CScene node];

    CCLayer* backgroundLayer = [HelloWorldBackground node];
    [scene addChild: backgroundLayer];

    CCLayer* layer = [HelloWorld node];
    [scene addChild: layer];

    CCLayer* userInterfaceLayer = [HelloWorldUserInterface node];
    [scene addChild: userInterfaceLayer];

    return scene;
}
```

This scene now has three distinct layers: the `backgroundLayer`, the regular game object layer, and on top of that the `userInterfaceLayer`. Because the layers are added to the scene in the order they are created, any nodes added to the `backgroundLayer` will be drawn behind the other layers. Likewise, nodes added to the `userInterfaceLayer` will always be drawn on top of any nodes in the layer and `backgroundLayer`.

TIP: Again, strictly speaking, a layer does not have to be derived from the `CCLayer` class; it could also be a simple `CCNode`. This is often preferable when you need the layer only to group nodes together and don't require the layer to handle input.

One case where you might want to use multiple layers per scene is if you have a scrolling background and a static frame surrounding the background, possibly including user interface elements. Using two separate layers makes it easy to move the background layer by simply adjusting the layer's position while the foreground layer remains in place. In addition, all objects of the same layer will be either in front of or behind objects of another layer, depending on the z-order of the layers. Of course, you can achieve the same effect without layers, but it would require each individual object in the background to be moved separately. That's just ineffective, so avoid it if you can.

Like scenes, layers have the same dimension as the cocos2d OpenGL view. For iOS devices, this will almost always be the size of the screen; on Mac OS X, it will be the size of the window.

Layers are primarily a grouping concept. For example, you can use any action with a layer, and the action will affect all of the objects on the layer. That means you can move all layer objects around in unison or rotate and scale them all at once. In general, use a layer if you need a group of objects to perform the same actions and behaviors. Moving all objects to scroll them is one such case; sometimes you might want to rotate them or reorder them so they are drawn on top of other objects. If all these objects are children of a layer, you can simply change the layer's properties or run an action on the layer to affect all of its child nodes.

NOTE: There is a recommendation not to use too many CCLayer objects per scene. This is often misunderstood. You can use as many layers as you want without affecting performance any more than using any other node. However, things change if the layer also accepts input, because receiving touch or accelerometer events are costly tasks. So, you should not use several layers receiving touch or accelerometer input; one will do. Preferably one layer receives and handles input and, where necessary, informs other nodes or classes about the input events by forwarding them to registered objects. You usually do this via the `performSelector` method that calls a method with a defined method signature. See cocos2d's `CCScheduler` class for an example implementation.

Receiving Touch Events

The CCLayer class is designed to receive touch input but only if you explicitly enable it. To enable receiving touch events, set the property `isTouchEnabled` to YES:

```
self.isTouchEnabled = YES;
```

This is best done in the class's `init` method, but it can be changed at any time.

Once the `isTouchEnabled` property is set, a variety of methods for receiving touch input will start to get called. These are the events received when a new touch begins, when a finger is moved on the touchscreen, and when the user lifts his finger off the screen. Canceled touches are rare, and you can safely ignore this method for the most part or simply forward it to the `ccTouchesEnded` method.

- This is called when a finger just begins touching the screen:
`-(void) ccTouchesBegan:(NSSet *)touches withEvent:(UIEvent *)event`
- This is called whenever the finger moves on the screen:
`-(void) ccTouchesMoved:(NSSet *)touches withEvent:(UIEvent *)event`
- This is called when a finger is lifted off the screen:
`-(void) ccTouchesEnded:(NSSet *)touches withEvent:(UIEvent *)event`

- This is called to cancel a touch:

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

Cancel events are rare and should behave just like touches ended in most cases.

In many cases, you'll want to know where a touch occurred. Since the touch events are received by the Cocoa Touch API, the location must be converted to OpenGL coordinates. The following is a method that does this for you:

```
-(CGPoint) locationFromTouches:(NSSet *)touches
{
    UITouch *touch = [touches anyObject];
    CGPoint touchLocation = [touch locationInView: [touch view]];
    return [[CCDirector sharedDirector] convertToGL:touchLocation];
}
```

This method works only with a single touch since it uses `[touches anyObject]`. To keep track of multitouch locations, you have to keep track of each touch individually.

By default, the layer receives the same events as Apple's `UIResponder` class. Cocos2d also supports targeted touch handlers. The difference is that targeted touches receive only one touch at a time, in comparison to the `UIResponder` touch events that always receive a set of touches. The targeted touch handler simply splits those touches into separate events that, depending on your game's needs, may be easier to work with. More importantly, it allows you to remove certain touches from the event queue, specifying that you've handled this touch and don't want it to be forwarded to other layers. This makes it easy to sort out if touches are in a specific area of the screen; if they are, you mark the touch as claimed, and all the remaining layers don't need to do this area check again.

To enable the targeted touch handler, add the following method to your layer's class:

```
-(void) registerWithTouchDispatcher
{
    [[CCTouchDispatcher sharedDispatcher] addTargetedDelegate:self
                                                priority:INT_MIN+1
                                                swallowsTouches:YES];
}
```

CAUTION: If you leave the `registerWithTouchDispatcher` method empty, you won't receive any touches at all! If you want to keep the method but also want to use the default handler, you'll have to call `[super registerWithTouchDispatcher]` in this method.

Now, instead of using the default touch input methods, you'll be using a slightly different set of methods. They are almost equivalent with the exception of receiving a `(UITouch*)` touch instead of a `(NSSet*) touches` as the first parameter:

```
-(BOOL) ccTouchBegan:(UITouch *)touch withEvent:(UIEvent *)event {}
-(void) ccTouchMoved:(UITouch *)touch withEvent:(UIEvent *)event {}
-(void) ccTouchEnded:(UITouch *)touch withEvent:(UIEvent *)event {}
-(void) ccTouchCancelled:(UITouch *)touch withEvent:(UIEvent *)event {}
```

What's important to note here is that `ccTouchBegan` returns a `BOOL` value. If you return `YES` in that method, it means you don't want this particular touch to be propagated to other targeted touch handlers with a lower priority. You have effectively "swallowed" this touch.

NOTE: Cocos2d has no built-in support for recognizing gestures. You will have to roll out your own gesture recognition code, possibly with the help of Apple's Gesture Recognizers: <http://developer.apple.com/library/ios/#documentation/EventHandling/Conceptual/EventHandlingiPhoneOS/GestureRecognizers/GestureRecognizers.html>.

Receiving Accelerometer Events

Like touch input, the accelerometer must be specifically enabled to receive accelerometer events:

```
self.isAccelerometerEnabled = YES;
```

Once more, there's a specific method to be added to the layer that receives the accelerometer events:

```
-(void) accelerometer:(UIAccelerometer *)accelerometer
didAccelerate:(UIAcceleration *)acceleration
{
    CCLOG(@"acceleration: x:%f / y:%f / z:%f",
          acceleration.x, acceleration.y, acceleration.z);
}
```

You can use the `acceleration` parameter to determine the acceleration in any of the three directions.

Receiving Keyboard Events

If you're creating a Mac OS X app, you will want to be able to process keyboard presses. You first have to enable keyboard events:

```
self.isKeyboardEnabled = YES;
```

The callback methods to receive keyboard events are defined in the `CCKeyboardEventDelegate` protocol as follows:

```
-(BOOL) ccKeyDown:(NSEvent*)event
{
    CCLOG(@"key pressed: %@", [event characters]);
}

-(BOOL) ccKeyUp:(NSEvent*)event
{
    CCLOG(@"key released: %@", [event characters]);
}
```

```

-(BOOL) ccFlagsChanged:(NSEvent*)event
{
    CCLOG(@"flags changed: %@", [event characters]);
}

```

The flags changed event is received whenever the user presses or releases a modifier key, regardless of whether any other key is pressed at the same time. You can use it to implement controls that are assigned directly to a modifier key.

A very simplistic keyboard event check to react to presses or releases of the key D would go something like this:

```

NSString* key = [event charactersIgnoringModifiers];
if ([key isEqualToString:@"d"] || [key isEqualToString:@"D"])
{
    CCLOG(@"D key");
}

```

Checking for the uppercase variant is good style because the player may have the Caps Lock key activated by accident or otherwise. The cocos2d forum has a thread with more involved example code for implementing keyboard controls: www.cocos2d-iphone.org/forum/topic/11725.

To learn more about keyboard event handling in general, I recommend reading Apple's documentation on handling key events: <http://developer.apple.com/library/mac/#documentation/cocoa/conceptual/EventOverview/HandlingKeyEvents/HandlingKeyEvents.html>.

The NSEvent class is crucial not only for keyboard but also for mouse and other events. Because of this, it will be a good idea to take a look at the NSEvent class reference: http://developer.apple.com/library/mac/#documentation/Cocoa/Reference/ApplicationKit/Classes/NSEvent_Class/Reference/Reference.html.

Receiving Mouse Events

Similar to all other input methods, you first have to enable mouse input via the following:

```
self.isMouseEnabled = YES;
```

Then your layer will start receiving CCMouseEventDelegate protocol messages, of which there are quite a number:

```

// received when the mouse moves with no button pressed
-(BOOL) ccMouseMoved:(NSEvent*)event {}

// received when the mouse moves while the corresponding button is held down
-(BOOL) ccMouseDragged:(NSEvent*)event {}
-(BOOL) ccRightMouseDragged:(NSEvent*)event {}
-(BOOL) ccOtherMouseDragged:(NSEvent*)event {}

// received when the corresponding mouse button is pressed (left, right, other)
-(BOOL) ccMouseDown:(NSEvent*)event {}
-(BOOL) ccRightMouseDown:(NSEvent*)event {}
-(BOOL) ccOtherMouseDown:(NSEvent*)event {}

```

```
// received when the corresponding mouse button is released (left, right, other)
-(BOOL) ccMouseUp:(NSEvent*)event {}
-(BOOL) ccRightMouseUp:(NSEvent*)event {}
-(BOOL) ccOtherMouseUp:(NSEvent*)event {}

// received when the scroll wheel is turned
-(BOOL) ccScrollWheel:(NSEvent*)event {}
```

Since you get specific events for each mouse button, the `NSEvent` object is mainly used to get the current mouse cursor position. You will have to convert that position to `cocos2d` coordinates via the Director's `convertEventToGL` method:

```
CGPoint mousePos = [[CCDirector sharedDirector] convertEventToGL:event];
```

Apple has a great tutorial on handling mouse events if you need to learn more about mouse event handling:

<http://developer.apple.com/library/mac/#documentation/Cocoa/Conceptual/EventOverview/HandlingMouseEvents/HandlingMouseEvents.html>.

CCSprite

`CCSprite` is certainly the most commonly used class. It uses an image to display the sprite on-screen. The simplest way to create a sprite is from a file that will be loaded into a `CCTexture2D` texture and assigned to the sprite. You have to add the image file to the Resources group in Xcode; otherwise, the app will not be able to find the file:

```
CCSprite* sprite = [CCSprite spriteWithFile:@"Default.png"];
[self addChild:sprite];
```

Here's a question for you: where do you think this sprite will be positioned on the screen? Contrary to what you might be used to from other game engines, the texture is centered on the sprite's position. The sprite just initialized will be located at position 0,0, so it is positioned at the lower-left corner of the screen. Because the sprite's texture is centered on the sprite's position, the texture will be only partially visible. Assuming the image is 80 by 30 pixels in size, you'd have to move the sprite to position 40,15 to make the texture align perfectly with the lower-left corner of the screen and be fully visible.

While unusual at first glance, centering the texture on the sprite does have great advantages. Once you start using the rotation or scale properties of the sprite, the sprite will stay centered on its position.

WARNING: File names are case-sensitive on iOS devices. While you're on the simulator, the file name's case doesn't matter, but when you switch to testing on the device, it will most likely crash if the file name is actually something like `@ "default.PNG"`, as in the example.

This has caused many developers serious headaches, and it's another reason you should test on the device often. It's also a good idea to come up with a naming scheme for file names and stick to it. Personally, I keep them in lowercase, using dashes where needed to separate words.

Anchor Points Demystified

Every node has an anchor point, but it only starts to make a difference if the node has a texture, like `CCSprite` or `CCLabelTTF`. By default, the `anchorPoint` property is at 0.5,0.5, or, in other words, at the center of the texture.

The anchor point has nothing to do with the node's position, even though changing the `anchorPoint` will change where the texture is rendered on the screen. By modifying the `anchorPoint`, you change only where the texture of the node is drawn relative to the node's position. But that also raises the question, why would you want to modify the `anchorPoint`, and what effects can you achieve by doing so?

For example, setting the `anchorPoint` to 0,0 effectively moves the texture so that its lower-left corner aligns with the node's position. If you set the `anchorPoint` to 1,1 instead, the top-right corner of the texture will align with the node's position. Sometimes this can be useful to align textures with the screen borders or other elements; specifically, it is useful as a way for `CCLabel` classes to, for example, right align or top align the text.

Generally, you do not want to modify the `anchorPoint` unless you have a good reason to do so, because it can have wide-ranging side effects such as offsetting position-based collision checks. It will also have an effect on rotation and scaling because the texture will no longer be rotated or scaled around its center position.

In this example code, the sprite image will neatly align with the lower-left corner of the screen because its `anchorPoint` is set to 0,0, which causes the texture's lower-left corner to align with the sprite's default position, which is also 0,0:

```
CCSprite* sprite = [CCSprite spriteWithFile:@"Default.png "];
sprite.anchorPoint = CGPointMake(0, 0);
[self addChild:sprite];
```

Texture Dimensions

Texture dimensions deserve a special mention. iOS devices so far can work only with textures whose dimensions are a power of two, so the individual width and height of a texture can be 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, and, beginning with third-generation devices, even 2048 pixels. The textures don't need to be square, so a texture with a size of 8 by 1024 pixels is perfectly fine.

This comes into play whenever you create a texture, for example by creating a sprite from an image file. Let's check the worst-case scenario right away by assuming you have an image that is 260 by 260 pixels in size with 32-bit colors. In memory, the texture should use around 270KB, yet the texture is using a whopping 1MB !

The almost fourfold increase stems from the fact that textures need to be powers of two in width and height, so the iOS device simply creates a texture with the smallest dimension that's still a power of two but large enough to contain the image. In the case

of the 260 by 260 texture, the next best option is to create a 512 by 512 pixel texture in memory, which uses 1MB of memory.

There's nothing you can do about this, other than creating images that are already powers of two from the start. The 260 by 260 pixel texture should really be a 256 by 256 texture instead so it doesn't waste that much memory. If you work with an artist, make sure she is aware of this issue.

In Chapter 6, I'll show you how you can alleviate this problem to a great extent by creating and using a Texture Atlas.

CCLabelTTF

CCLabelTTF is the simplest choice when it comes to displaying text on the screen. Here's how to create a CCLabelTTF object to display some text:

```
CCLabelTTF* label = [CCLabelTTF labelWithString:@"text"
                                     fontName:@"AppleGothic"
                                     fontSize:32];

[self addChild:label];
```

In case you're wondering which TrueType fonts are available on iOS devices, you'll find a list of fonts in the Essentials code project for this chapter.

Internally, the given TrueType font is used to render the text on a CCTexture2D texture. Since this is done every time the text changes, it's not something you should do every frame. Re-creating the texture of a CCLabelTTF is really slow, and it is done every time the string of the label is modified:

```
[label setString:@"new text"];
```

You'll also notice that if you increase or decrease the length of the text of a label, the text behaves as if it is center-aligned on the label's position. The following sentences are center-aligned to illustrate this effect:

```

Hello World!
Hello World Once Again!
Hello Our World and all the other Worlds out there!
```

The center alignment is because of the anchor point and its default position of 0.5,0.5, which causes the center of the texture (in this case the label's text is the texture) to be center aligned with the label's position. In many cases you want to align labels left, right, up, or down, and you can use the `anchorPoint` property to easily achieve this. The following code shows how you can align a label by simply changing the `anchorPoint` property:

```
// align label to the right
label.anchorPoint = CGPointMake(1, 0.5f);
// align label to the left
label.anchorPoint = CGPointMake(0, 0.5f);
// align label to the top
label.anchorPoint = CGPointMake(0.5f, 1);
// align label to the bottom
label.anchorPoint = CGPointMake(0.5f, 0);
```

```
// use case: place label at top-right corner of the screen
// the label's text extends to the left and down and is always completely on screen
CGSize size = [[CCDirector sharedDirector] winSize];
label.position = CGPointMake(size.width, size.height);
label.anchorPoint = CGPointMake(1, 1);
```

Menus

You'll soon need some kind of button a user can click to perform an action, such as going to another scene or toggling music on and off. This is where the `CCMenu` class comes into play. `CCMenu` is a subclass of `CCLayer` and accepts only `CCMenuItem` nodes as children. The `CCMenuItem` class hierarchy can be reviewed in Figure 3–5 and is depicted again in Figure 3–9 for the sake of clarity.

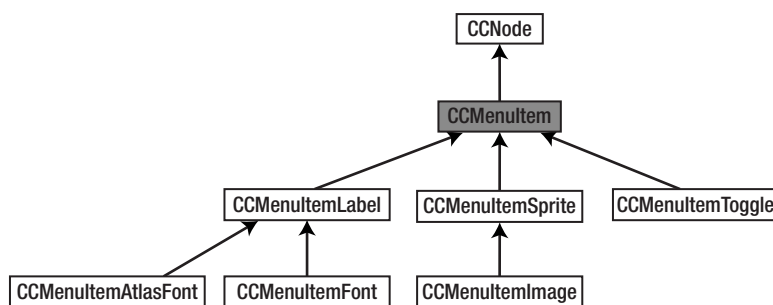


Figure 3–9. The `CCMenuItem` class hierarchy

Listing 3–1 shows the code for setting up a menu. You can find the menu code in the Essentials project in the `MenuScene` class.

Listing 3–1. Creating Menus in *cocos2d* with Text and Image Menu Items

```
CGSize size = [[CCDirector sharedDirector] winSize];

// set CCMenuItemFont default properties
[CCMenuItemFont setFontName:@"Helvetica-BoldOblique"];
[CCMenuItemFont setFontSize:26];

// create a few labels with text and selector
CCMenuItemFont* item1 = [CCMenuItemFont itemFromString:@"Go Back!" target:self
selector:@selector(menuItem1Touched:)];

// create a menu item using existing sprites
CCSprite* normal = [CCSprite spriteWithFile:@"Icon.png"];
normal.color = ccRED;
CCSprite* selected = [CCSprite spriteWithFile:@"Icon.png"];
selected.color = ccGREEN;
CCMenuItemSprite* item2 = [CCMenuItemSprite
                           itemFromNormalSprite:normal
                           selectedSprite:selected
                           target:self
                           selector:@selector(menuItem2Touched:)];
```



```
// create a toggle item using two other menu items (toggle works with images, too)
[CCMenuItemFont setFontName:@"STHeitiJ-Light"];
[CCMenuItemFont setFontSize:18];
CCMenuItemFont* toggleOn = [CCMenuItemFont itemFromString:@"I'm ON!"];
CCMenuItemFont* toggleOff = [CCMenuItemFont itemFromString:@"I'm OFF!"];
CCMenuItemToggle* item3 = [CCMenuItemToggle itemWithTarget:self
                                selector:@selector(menuItem3Touched:)
                                items:toggleOn, toggleOff, nil];

// create the menu using the items
CCMenu* menu = [CCMenu menuWithItems:item1, item2, item3, nil];
menu.position = CGPointMake(size.width / 2, size.height / 2);
[self addChild:menu];

// aligning is important, so the menu items don't occupy the same location
[menu alignItemsVerticallyWithPadding:40];
```

WARNING: The lists of menu items always end with `nil` as the last parameter. This is a technical requirement. If you forget to add `nil` as the last parameter, your app will crash at that particular line.

It takes a fair bit of code to set up a menu. The first menu item is based on `CCMenuItemFont` and simply displays a string. When the menu item is touched, it calls the method `menuItem1Touched`. Internally, `CCMenuItemFont` simply creates a `CCLabel`. If you already have a `CCLabel`, you can use that with the `CCMenuItemLabel` class instead.

Likewise, there are two menu item classes for images; one is `CCMenuItemImage`, which creates an image from a file and uses a `CCSprite` internally, and the other is one I've used here, `CCMenuItemSprite`. This class takes existing sprites as input, which I think is more convenient because you can use the same image and simply tint its color to achieve a highlighting effect when touched.

`CCMenuItemToggle` accepts exactly two `CCMenuItem`-derived objects and, when touched, will toggle between the two items. You can use either text labels or images with `CCMenuItemToggle`.

Finally, `CCMenu` itself is created and positioned. Since the menu items are all children of `CCMenu`, they will be positioned relative to the menu. To keep them from stacking up on each other, you have to call one of `CCMenu`'s align methods, like `alignItemsVerticallyWithPadding`, as I've done at the end of Listing 3-1.

Since `CCMenu` is a node containing all menu items, you can use actions on the menu to let it scroll in and out. This makes your menu screens appear less static, which is usually a good thing. See the Essentials project for an example. In the meantime, take a look at Figure 3-10 to see what our current menu looks like.



Figure 3-10. This is the menu produced by the code in Listing 3.1.

Actions

Actions are lightweight classes that are used on nodes to perform certain, well, actions. They allow you to move, rotate, scale, tint, fade, and do a lot of other things with a node. Because they work with every node, you can use them on sprites, labels, and even menus or whole scenes! That's what makes them so powerful.

In Figure 3-11 you can see the `CCAction` class hierarchy without the many subclasses of `CCActionInterval` and `CCActionInstant`. You will see their class hierarchies in Figure 3-12 and Figure 3-17, respectively.

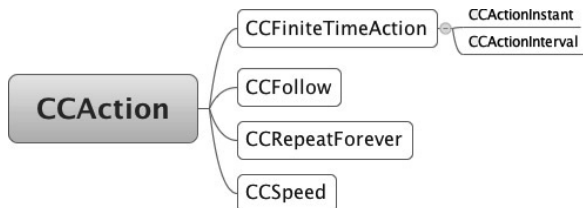


Figure 3-11. The `CCAction` class hierarchy with subclasses of `CCActionInstant` and `CCActionInterval` omitted

There are only three actions that directly derive from `CCAction`:

- `CCFollow` (allows a node to follow another node)
- `CCRepeatForever` (repeats an action indefinitely)
- `CCSpeed` (changes the update frequency of an action while it is running)

With the `CCFollow` action, you can instruct a node to follow another node. For example, to have a label follow the player character sprite, the code might look something like this:

```
[label runAction:[CCFollow actionWithTarget:playerSprite]];
```

You can also have actions or even a whole sequence of actions repeat (loop) forever with `CCRepeatForever`. You can create endlessly looping animations this way, for example. This code lets a node rotate forever like an endlessly spinning wheel:

```
CCRotateBy* rotateBy = [CCRotateBy actionWithDuration:2 angle:360];
CCRepeatForever* repeat = [CCRepeatForever actionWithAction:rotateBy];
[myNode runAction:repeat];
```

The `CCSpeed` action can be used to influence the speed of an action while it is running. Let's take the earlier rotation example and wrap it with a `CCSpeed` action:

```
CCRotateBy* rotateBy = [CCRotateBy actionWithDuration:2 angle:360];
CCRepeatForever* repeat = [CCRepeatForever actionWithAction:rotateBy];
CCSpeed* speedAction = [CCSpeed actionWithAction:repeat speed:0.5f];
speedAction.tag = 1;
[myNode runAction:speedAction];
```

Now the node will take twice as long to make a full revelation because the `CCSpeed` action's speed is set to 0.5f. You can later change the speed property of the `CCSpeed` action to influence the speed of the wrapped action while it is running. To make the node suddenly rotate faster, you only need to get the speed action and modify its speed property:

```
CCSpeed* speedAction = (CCSpeed*)[myNode getActionByTag:1];
speedAction.speed = 2;
```

NOTE: You cannot add a `CCSpeed` action to a `CCSequence` action, since only actions derived from `CCFiniteTimeAction` can be used in a sequence.

Interval Actions

Since most actions happen over time, like a rotation for three seconds, you'd normally have to write an update method and add variables to store the intermediate results. The `CCActionInterval` actions shown in Figure 3-12 wrap this kind of logic for you and turn it into simple, parameterized methods:

```
// have myNode move to 100, 200 and arrive there in 3 seconds
CCMoveTo* move = [CCMoveTo actionWithDuration:3 position:CGPointMake(100, 200)];
[myNode runAction:move];
```

Once you start using this particular code, you'll notice that depending on the distance `myNode` has to move, its speed will be different. This is a very common problem that has a simple solution. You need to calculate the distance from the current position to the target position and then divide it by the speed you want the node to move. The result is the correct duration to have the node move to the target position at the same speed, regardless of where the node and target positions are.

```
// have myNode move at a fixed speed to any position
CGPoint targetPos = CGPointMake(100, 200);
float speed = 10; // in pixels per second
float duration = ccpDistance(myNode.position, targetPos) / speed;
```

```
CCMoveTo* move = [CCMoveTo actionWithDuration:duration position:targetPos];
[myNode runAction:move];
```

By the way, you don't have to remove an action. Once an action has completed its task, it will remove itself from the node automatically and release the memory it uses.

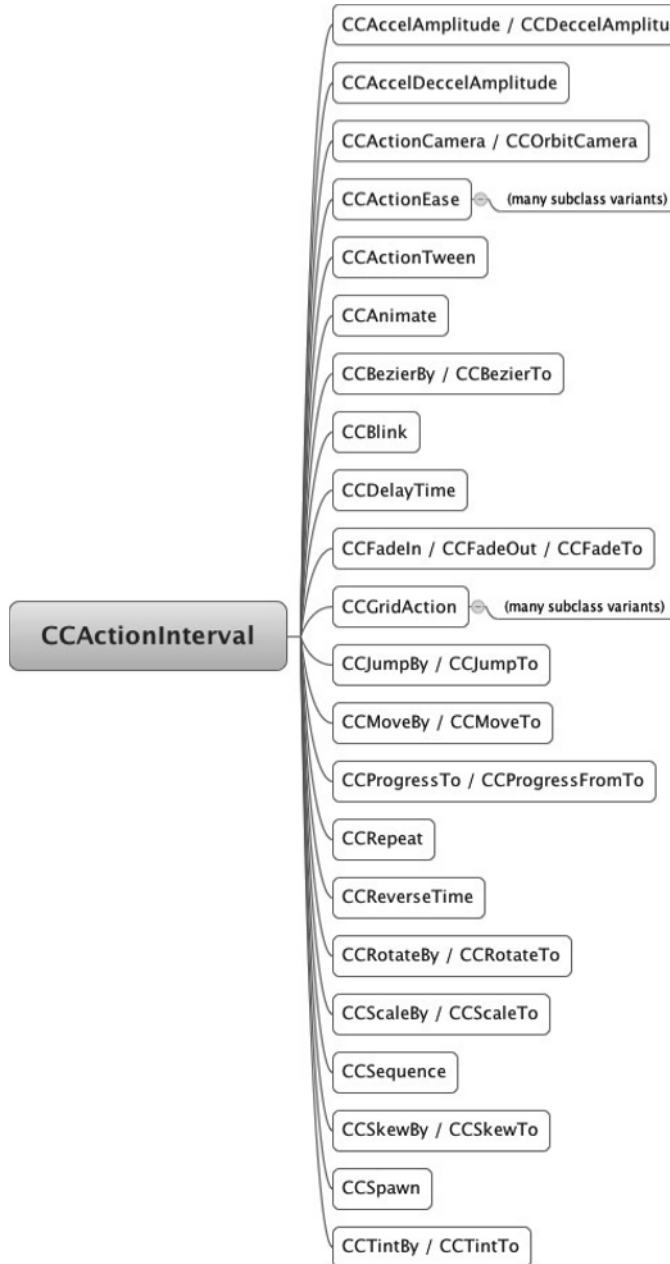


Figure 3–12. The `CCActionInterval` class hierarchy with subclasses of `CCActionEase` and `CCGridAction` omitted

Action Sequences

Normally, when you add several actions to a node, they all perform their duties at the same time. For example, you could have an object rotate and fade out at the same time by adding the corresponding actions. But what if you want to run the actions one after the other?

Sometimes it's more useful to sequence actions, and that's where `CCSequence` comes in. Since it is so frequently used, it deserves an extra mention. You can use any number and type of actions in a sequence, which makes it easy to have a node move to a target location and, at arrival, have it rotate around and then fade out, each action followed by the next one until the sequence is complete.

Here's how to cycle a label's colors from red to blue to green:

```
CCTintTo* tint1 = [CCTintTo actionWithDuration:4 red:255 green:0 blue:0];
CCTintTo* tint2 = [CCTintTo actionWithDuration:4 red:0 green:0 blue:255];
CCTintTo* tint3 = [CCTintTo actionWithDuration:4 red:0 green:255 blue:0];
CCSequence* sequence = [CCSequence actions:tint1, tint2, tint3, nil];
[label runAction:sequence];
```

You can also use a `CCRepeatForever` action with the sequence:

```
CCSequence* sequence = [CCSequence actions:tint1, tint2, tint3, nil];
CCRepeatForever* repeat = [CCRepeatForever actionWithAction:sequence];
[label runAction:repeat];
```

And being able to modify the speed of the entire repeating sequence can come in handy too:

```
CCSequence* sequence = [CCSequence actions:tint1, tint2, tint3, nil];
CCRepeatForever* repeat = [CCRepeatForever actionWithAction:sequence];
CCSpeed* speedAction = [CCSpeed actionWithAction:repeat speed:0.75f];
[label runAction:speedAction];
```

NOTE: As with menu items, a list of actions always ends with `nil`. If you forget to add `nil` as the last parameter, the line creating the `CCSequence` will crash!

Ease Actions

Actions become even more powerful by using actions based on the `CCActionEase` class. Ease actions allow you to modify the effect of an action over time. For example, if you use a `CCMoveTo` action on a node, the node will move the whole distance at the same speed until it has arrived. With `CCActionEase`, you can have the node start slow and speed up toward the target, or vice versa. Or you can let it move past the target location a little and then bounce back. Ease actions create very dynamic animations that are normally very time-consuming to implement. The following code shows how an ease action is used to modify the behavior of a regular action. The rate parameter determines how pronounced the effect of the ease action is and should be greater than 1 to see any effect.

```
// I want myNode to move to 100, 200 and arrive there in 3 seconds
CCMoveTo* move = [CCMoveTo actionWithDuration:3 position:CGPointMake(100, 200)];
// this time the node should slowly speed up and then slow down as it moves
CCEaseInOut* ease = [CCEaseInOut actionWithAction:move rate:4];
[myNode runAction:ease];
```

NOTE: In the example, the ease action is run on the node, not the move action. It's all too easy to forget to change the runAction line when you're working with actions. It's a common mistake that happens even to the most experienced cocos2d developers. If you notice your actions aren't working as expected or at all, double-check that you're actually running the correct action. And if the correct actions are used but you're still not seeing the desired result, verify that it's the correct node running the action. That is another common mistake.

Cocos2d implements the following CCActionEase classes:

- CCEaseBackIn, CCEaseBackInOut, CCEaseBackOut
- CCEaseBounceIn, CCEaseBounceInOut, CCEaseBounceOut
- CCEaseElasticIn, CCEaseElasticInOut, CCEaseElasticOut
- CCEaseExponentialIn, CCEaseExponentialInOut, CCEaseExponentialOut
- CCEaseIn, CCEaseInOut, CCEaseOut
- CCEaseSineIn, CCEaseSineInOut, CCEaseSineOut

In Chapter 4, I'll use a number of these ease actions in the DoodleDrop project so you can see what effect they have. I've added the CCActionEase class hierarchy in Figure 3-13 for reference.

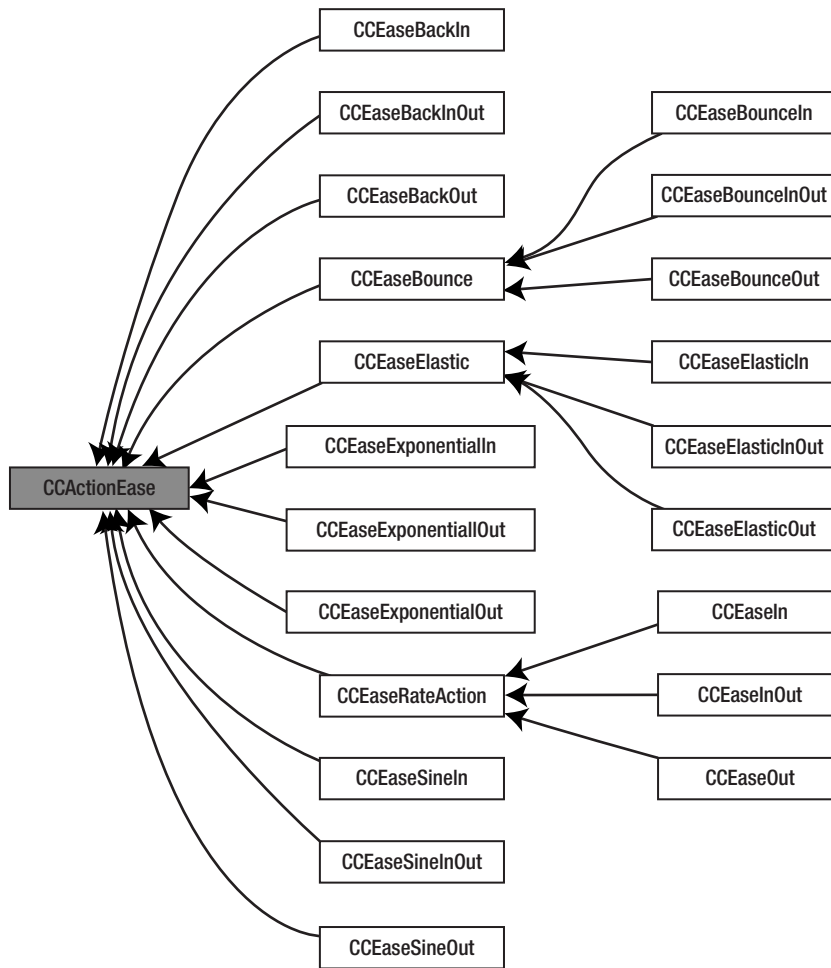


Figure 3-13. The *CCActionEase* class hierarchy

Grid Actions

Grid actions are purely visual actions derived from *CCGridAction* and one of its two subclasses, *CCGrid3DAction* and *CCTiledGrid3DAction*. Their class hierarchies are shown in Figure 3-14 and Figure 3-15.

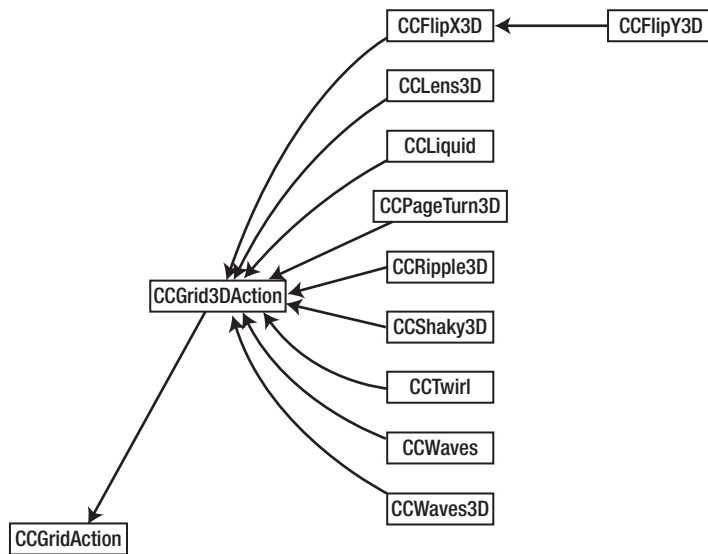


Figure 3-14. The *CCGrid3DAction* class hierarchy

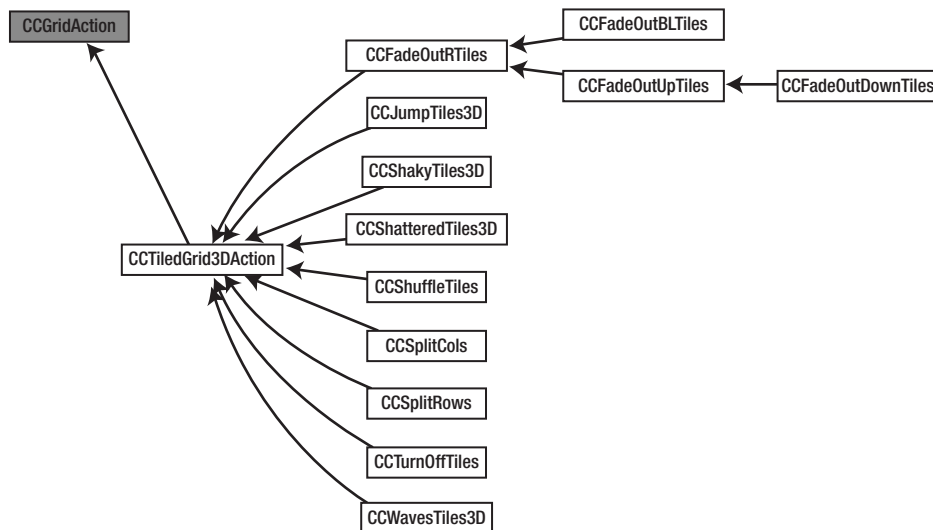


Figure 3-15. The *CCTiledGrid3DAction* class hierarchy

The specialty of the grid actions are three-dimensional effects such as the turning of a page (*CCPageTurn3D*; see Figure 3-16) or simulating waves and liquids (*CCWaves*, *CCLiquid*). The downside is that the 3D effects may show visual artifacts unless you enable depth buffering, which requires more memory and has a negative impact on rendering performance, particularly on first- and second-generation devices.

To enable depth buffering, you will have to change the line in your project's app delegate class that initializes the *EAGLView* class to add depth buffering support. You do

this by changing the `depthFormat` parameter from its default value 0 to either `GL_DEPTH_COMPONENT16_OES` for a 16-bit depth buffer or `GL_DEPTH_COMPONENT24_OES` for a 24-bit depth buffer:

```
EAGLView *glView = [EAGLView viewWithFrame:[window bounds]
                        pixelFormat:kEAGLColorFormatRGB565
                        depthFormat:GL_DEPTH_COMPONENT16_OES];
```

Ideally, you should try the 16-bit depth buffer first; it uses less memory, but in a few cases a 24-bit depth buffer may be necessary in case visual artifacts still occur when using 3D actions.

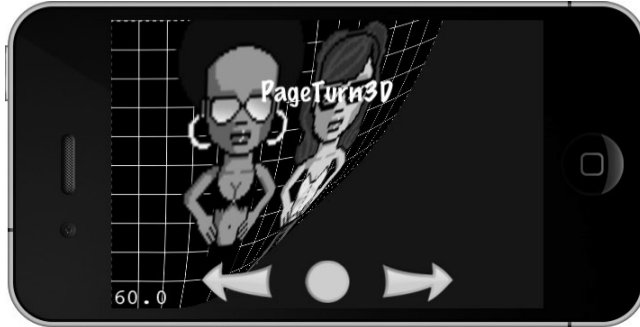


Figure 3–16. The *CCPageTurn3D* action in action

Instant Actions

You might wonder why there are instantaneous actions based on the *CCInstantAction* class (see Figure 3–17 for the class hierarchy), when you could just as well change the node's property to achieve the same effect. For example, there are instant actions to flip the node, to place it at a specific location, or to toggle its visible property.

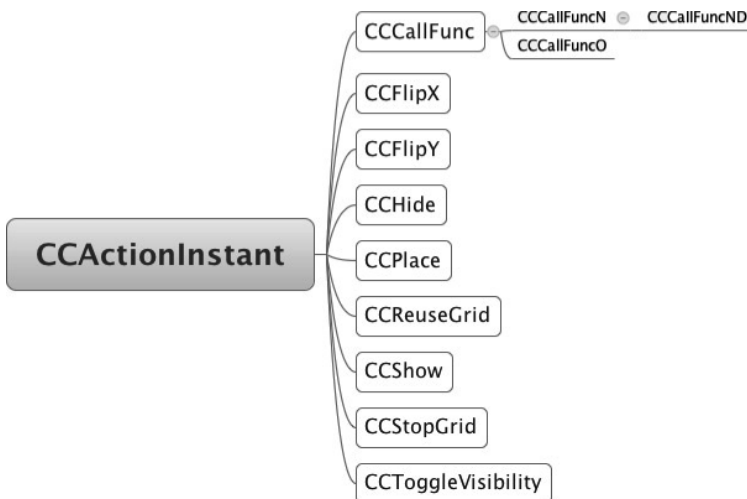


Figure 3–17. The *CCActionInstant* class hierarchy

The main reason that instant actions exist is because they are useful in action sequences. Sometimes in a sequence of actions you have to change a certain property of the node, such as visibility or position, and then continue with the sequence. Instant actions make this possible. True, they are rarely used — with one exception: `CCCallFunc` actions.

When using an action sequence, you may want to be notified at certain times, for example when the sequence has ended, and then perhaps start another sequence immediately thereafter. To do this, you can make use of several versions of `CCCallFunc` actions that will send a message whenever it is their turn in the sequence. Let's rewrite the color cycle sequence so that it calls a method each time one of the `CCTintTo` actions has done its job:

```
CCCallFunc* func = [CCCallFunc actionWithTarget:self
                      selector:@selector(onCallFunc)];

CCCallFuncN* funcN = [CCCallFuncN actionWithTarget:self
                      selector:@selector(onCallFuncN)];

CCCallFuncND* funcND = [CCCallFuncND actionWithTarget:self
                          selector:@selector(onCallFuncND:data:)
                          data:(void*)self];

CCCallFuncO* funcO = [CCCallFuncO actionWithTarget:self
                      selector:@selector(onCallFuncO:)
                      object:(id)self];

CCSequence* seq = [CCSequence actions:
    tint1, func, tint2, funcN, tint3, funcND, funcO, nil];
[label runAction:seq];
```

The difference between these variants of `CCCallFunc` is in which selector they will be calling and thus which context is available to the method they call. For example, when `CCCallFunc` calls the `onCallFunc` method, you have no way of knowing who called the method or why. There's no context, but in many cases that context is not needed.

The action sequence `seq` will call the methods in the following code one after another. The sender parameter will always be derived from `CCNode`; it's the node that's running the actions. The data parameter can be used in any way you want, including passing values, structs, or other pointers. You only have to properly cast the data pointer.

```
-(void) onCallFunc
{
    CCLOG(@"end of tint1!");
}

-(void) onCallFuncN:(id)sender
{
    CCLOG(@"end of tint2! sender: %@", sender);
}

-(void) onCallFuncND:(id)sender data:(void*)data
{
    // be careful when casting pointers like this!
    // you have to be 100% sure the object is of this type!
```

```

        CCSprite* sprite = (CCSprite*)data;
        CCLOG(@"end of sequence! sender: %@ - data: %@", sender, sprite);
    }
    -(void) onCallFunc0:(id)object
    {
        // object is the object you passed to CCCallFunc0
        CCLOG(@"call func with object %@", object);
    }
}

```

Of course, the `CCCallFunc` actions also work with `CCRepeatForever` sequences. Your methods will be called repeatedly at the appropriate time.

A Note on Singletons in cocos2d

Cocos2d makes good use of the Singleton design pattern, which I believe deserves mention because it's regularly and hotly debated. In principle, a *singleton* is a regular class that is instantiated only once during the lifetime of the application. To ensure that this is the case, a static method is used to both create and access the instance of the object. So, instead of using `alloc/init` or a static autorelease initializer, you gain access to a singleton object via methods that begin with `shared`. Here are some of cocos2d's most-used singleton classes and how you access them:

```

CCActionManager* sharedManager = [CCActionManager sharedManager];
CCDirector* sharedDirector = [CCDirector sharedDirector];
CCSpriteFrameCache* sharedCache = [CCSpriteFrameCache sharedSpriteFrameCache];
CCTextureCache* sharedTexCache = [CCTextureCache sharedTextureCache];
CCTouchDispatcher* sharedDispatcher = [CCTouchDispatcher sharedDispatcher];
CDAudioManager* sharedManager = [CDAudioManager sharedManager];
SimpleAudioEngine* sharedEngine = [SimpleAudioEngine sharedEngine];

```

The upside of a singleton is that it can be used anywhere by any class at any time. It acts almost like a global class, much like global variables. Singletons are very useful if you have a combination of data and methods you need to use in many different places. Audio is a good example of this, because any of your classes—whether the player, an enemy, a menu button, or a cutscene—might want to play a sound effect or change the background music. So, it makes a lot of sense to use a singleton for playing audio. Likewise, if you have global game stats, perhaps the size of the player's army and each platoon's number of troops, you might want to store that information in a singleton so you can carry it over from one level to another. Implementing a singleton is straightforward, as Listing 3–2 shows. This code implements the class `MyManager` as a singleton with minimal code. The `sharedManager` static method grants access to the single instance of `MyManager`. If the instance doesn't exist, a `MyManager` instance will be allocated and initialized; otherwise, the existing instance is returned.

Listing 3–2. Implementing the Exemplary Class `MyManager` as a Singleton

```

static MyManager *sharedManager = nil;

+(MyManager*) sharedManager
{
    if (sharedManager == nil)
    {

```

```
        sharedManager = [[MyManager alloc] init];
    }
    return sharedManager;
}
```

However, singletons also have ugly sides. Because they're simple to use and implement and can be accessed from any other class, there's a tendency to overuse them. They're like global variables that most programmers agree should be used scarcely and judiciously.

For example, you might think you have only one player object, so why not make the player class a singleton? Everything seems to be fine until you realize that whenever the player advances from one level to another, the singleton not only keeps the player's score but also his last animation frame, his health, and all the items he has picked up, and then he might even begin the new level in Berserk mode because that mode was active when he left the previous level.

To fix that, you add another method to reset certain variables when changing levels. So far, so good, but as you add more features to the game, you'll end up having to add and maintain more and more variables when switching a level. What's worse, suppose one day a friend suggests you give the iPad version a two-player mode. But, oh wait, your player is a singleton; you can have only one player at any time! This gives you a major headache: refactor a lot of code or miss out on the cool two-player mode?

Or why not make the second player a singleton, too? And whenever the second player needs to know something of the first player, it'll just use that singleton. So, they keep a reference to each other, which means you can't have a single-player game without initializing the other players as well. This is one side effect of classes strongly dependent on each other, also known as *tight coupling*. The more classes are tightly coupled with each other, the harder it is to make changes to any part of your code. It's like you're mixing cement that's slowly drying up until it's so hard that any change is easier done by hacking it, instead of improving the code. That's the point where bugs seem to occur everywhere, at random, with no connection to a recent change. In one word: frustrating.

The more you rely on singletons, the more likely such issues will arise. Before creating a singleton class, always consider whether you really need only one instance of this class and its data and whether this might change at a later time.

I understand it's hard for a beginner to judge when and where to use a singleton, particularly if you haven't had much experience with object-oriented programming. The Q&A web site [Stackoverflow.com](http://stackoverflow.com) hosts a discussion with additional links that illustrates the controversy around the Singleton design pattern and gives some food for thought: <http://stackoverflow.com/questions/137975/what-is-so-bad-about-singletons>.

My advice is to study popular uses of singletons. The use of singletons in the cocos2d game engine for resource management is acceptable, since it simplifies the overall design of the game engine. Most other game engines use singletons for similar purposes as well, and I've rarely heard anyone complain about that. And you'll even find singletons in the iOS SDK. Singletons are generally not as bad as some make them sound like, although they do tend to be very problematic by introducing strong dependencies, and the problems grow exponentially with the size of the code base.

Cocos2d Test Cases

Did you know that cocos2d comes with a lot of sample code? In your cocos2d-iphone folder, you'll find a project aptly named cocos2d-iphone that contains a lot of test targets you can build and run. You can see how things work and then check the code to see how it's implemented.

Summary

Wow! That was a lot to take in! I don't expect you to remember all of this chapter's content at once. Feel free to come back at any time to look again at cocos2d's scene graph and how to use the various CCNode classes. I wrote this chapter to be a good reference whenever you need it, as well as the accompanying NodeHierarchy and Essentials Xcode projects.

Armed with this chapter's knowledge and a fair bit of motivation on your side, you could be starting to write your own games now.

You know what, let's do that together. Read on to the next chapter, and I'll walk you through your first complete game project!

Chapter 4

Your First Game

In this chapter you'll build your first complete game. It won't win any awards, but you'll learn how to get the essential elements of cocos2d to work together. I'll guide you through the individual steps, so you'll also learn a bit about working with Xcode along the way.

The game is the inversion of the famous Doodle Jump game, aptly named DoodleDrop. The player's goal is to avoid falling obstacles for as long as possible by rotating the device to move the player sprite. Take a look at the final version in Figure 4–1 to get an idea of what you'll be creating in this chapter.



Figure 4–1. *The final version of the DoodleDrop game*

Step-by-Step Project Setup

Fire up Xcode now, and I'll walk you through the steps to create your first cocos2d game. In Xcode, select **File > New > New Project** and choose the cocos2d Application template, as shown in Figure 4-2. When asked to enter a name for the new project, enter **DoodleDrop** and, if necessary, find a suitable location to save the project. Xcode will automatically create a subfolder named DoodleDrop, so you don't have to create that folder.

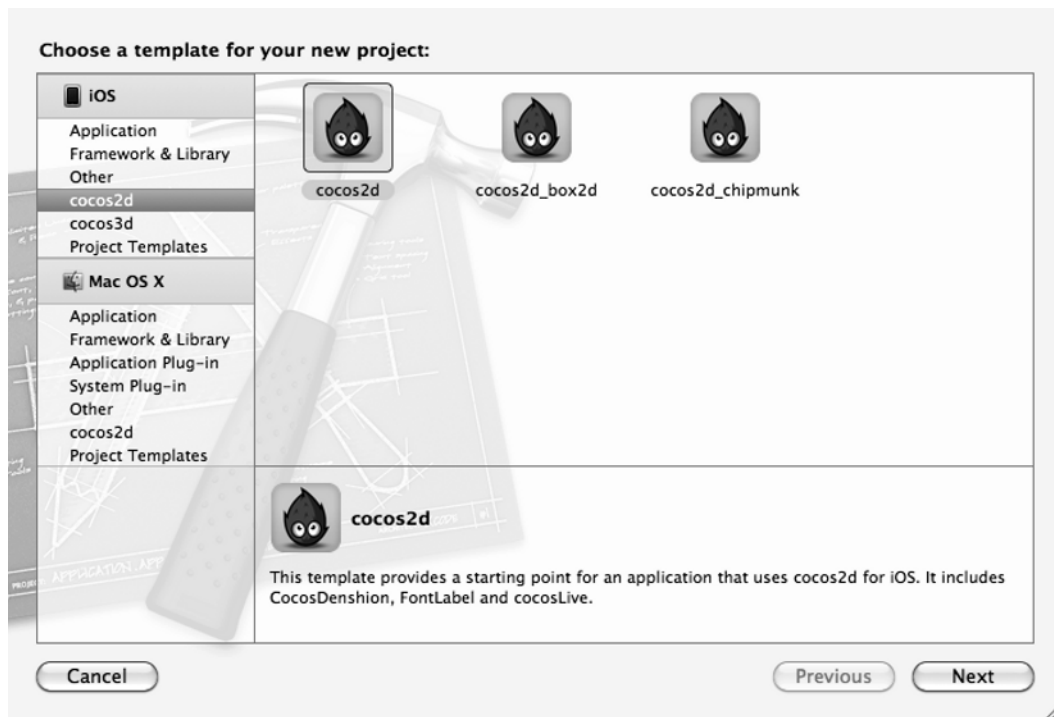


Figure 4-2. Create the project from the cocos2d Application template.

Xcode should present you with a project view similar to the one in Figure 4-3. Depending on the versions of cocos2d and Xcode you're using, there may be more files, or the names of the groups may be slightly different.

I've already unfolded the Classes and Resources groups because that's where you'll be adding the source code and game resource files, respectively. Anything that's not source code is considered to be a resource, be it an image, an audio file, a text file, or a plist. The grouping is not strictly necessary, but it does make it easier for you to navigate the project if you keep similar files grouped together.

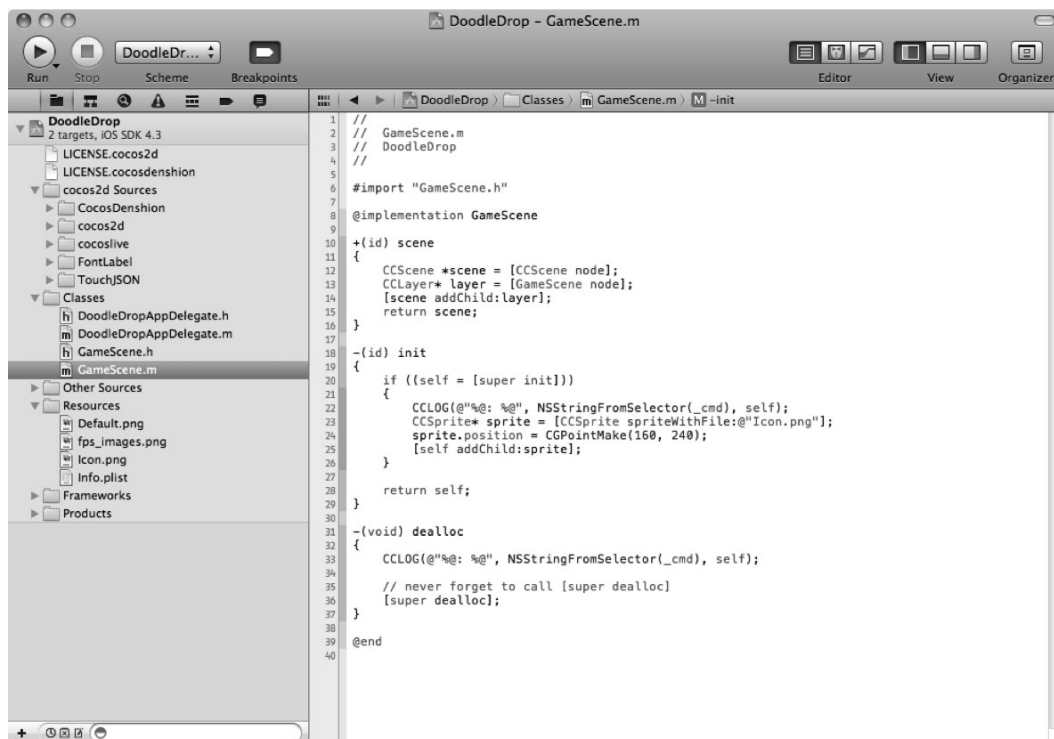


Figure 4-3. Let the games begin! The DoodleDrop project at this point is based on the HelloWorld cocos2d project template. Make sure you add subsequent files to the Classes and Resources groups accordingly to keep your game project organized.

The next step you're faced with is a decision: do you start working with the HelloWorldScene because it's there already, possibly renaming it later? Or do you go through the extra steps to create your own scene to replace the HelloWorldScene? I chose the latter because eventually you'll have to add new scenes anyway, so it's a good idea to learn the ropes here and now and start with a clean slate.

Make sure the Classes group is selected and then select **File > New > New File** or right-click the Classes folder and select **New File** to open the New File dialog shown in Figure 4-4. Since cocos2d comes with class templates for the most important nodes, it would be a shame not to use them. From the cocos2d User Templates section, select the CCNode class, click **Next**, and make sure it's set to **Subclass of CCLayer** before clicking **Next** again to bring up the save file dialog in Figure 4-5.

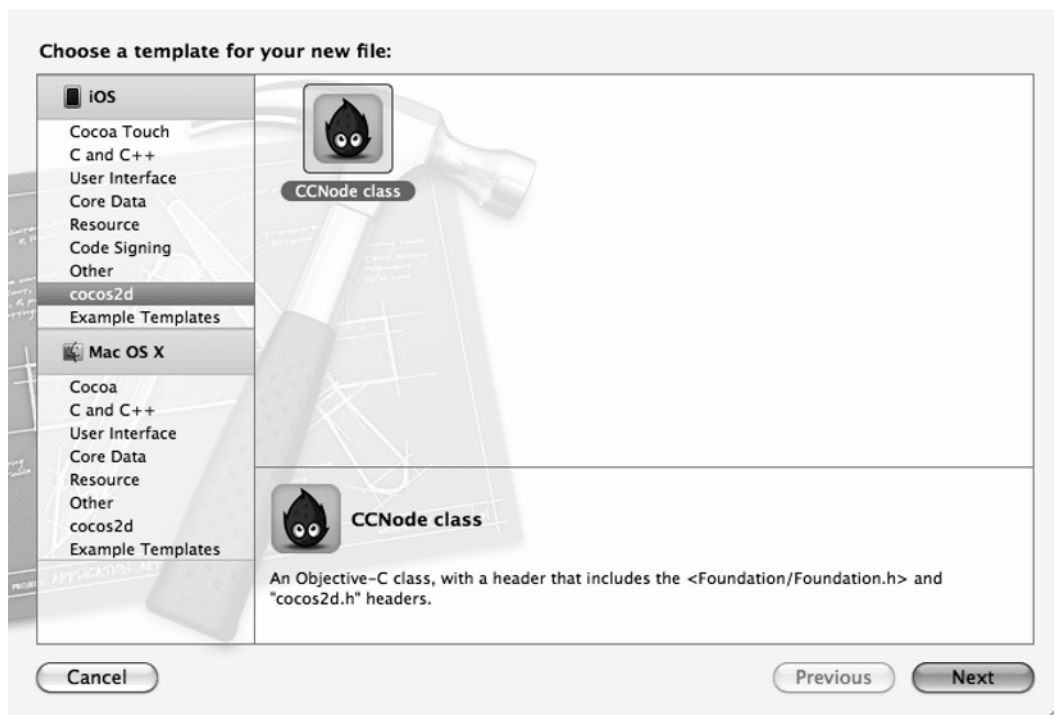


Figure 4–4. Adding new *CCNode*-derived classes is best done using the class templates provided by *cocos2d*. In this case, we want the *CCNode* class to be a subclass of *CCLayer* since we're setting up a new scene.

I prefer to name classes by function and in a generic way. I'm using `GameScene.m` in this case. It's going to be the scene where the DoodleDrop game play takes place, so that name seems appropriate. Be sure that the DoodleDrop target check box is checked. If you're using Xcode 3, the Also create `GameScene.h` check box must also be checked. Targets are Xcode's way of creating more or less different versions of the executable. For example, the iPad version of a game is usually created as a separate target. In this case, we have only one target, but once you create an iPad target, you want to make sure that, for example, the iPad's high-resolution images aren't accidentally added to the iPhone/iPod touch target.

NOTE: Not reviewing the target check boxes can lead to all kinds of issues, from compile errors to file not found errors or crashes during game play when files haven't been added to the targets that need them. Or you might simply waste space by adding files to targets that don't need them at all, for example by adding iPad and iPhone 4 high-res images to regular iPhone/iPod touch targets.

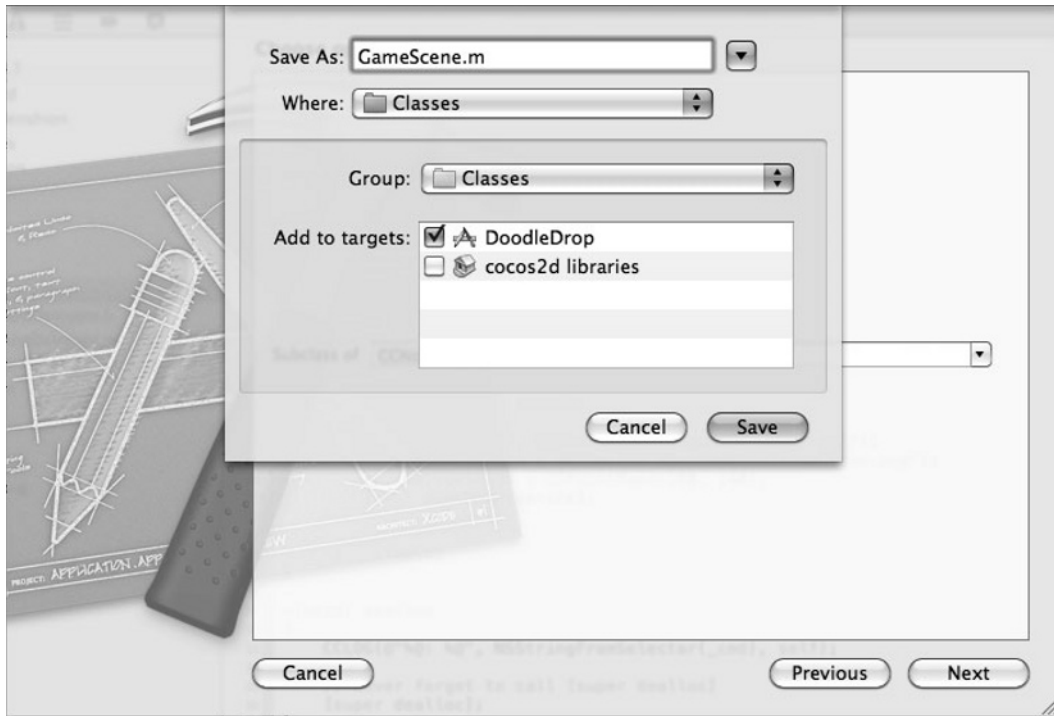


Figure 4-5. Naming the new scene and making sure it's added to the appropriate targets

At this point, our `GameScene` class is empty, and the first thing we need to do to set it up as a scene is to add the `+(id) scene` method to it. The code we'll plug in is essentially the same as in Chapter 3, with only the layer's class name changed. What you'll almost always need in any class are the `...(id) init` and `...(void) dealloc` methods, so it makes sense to add them right away. I'm also a very cautious programmer and decided to add the logging statements introduced in Chapter 3. The resulting `GameScene.h` is shown in Listing 4-1, and `GameScene.m` is in Listing 4-2.

Listing 4–1. *GameScene.h with the Scene Method*

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

@interface GameScene : CCLayer
{
}

+(id) scene;

@end
```

Listing 4–2. *GameScene.m with the Scene Method and Standard Methods Added, Including Logging*

```
#import "GameScene.h"

@implementation GameScene

+(id) scene
{
    CCScene *scene = [CCScene node];
    CCLayer* layer = [GameScene node];
    [scene addChild:layer];
    return scene;
}

-(id) init
{
    if ((self = [super init]))
    {
        CCLOG(@"%@: %@", NSStringFromSelector(_cmd), self);
    }

    return self;
}

-(void) dealloc
{
    CCLOG(@"%@: %@", NSStringFromSelector(_cmd), self);

    // never forget to call [super dealloc]
    [super dealloc];
}

@end
```

Now you can safely delete the HelloWorldScene class. When asked, select the **Also Move to Trash** option to remove the file from the hard drive as well, not just from the Xcode project. Select both files and choose **Edit ► Delete**, or right-click the files and choose **Delete** from the context menu. With the HelloWorldScene class gone, you have to modify DoodleDropAppDelegate.m to change any references to HelloWorldScene to GameScene. Listing 4–3 highlights the necessary changes to the `#import` and `runWithScene` statements. I also changed the device orientation to portrait mode since the game is designed to work best in that mode.

Listing 4–3. *Changing DoodleDropAppDelegate.m File to Use the GameScene Class Instead of HelloWorldScene*

```
// replace the line #import HelloWorldScene.h with this one:
#import "GameScene.h"

- (void) applicationDidFinishLaunching:(UIApplication*)application
{

    // Sets Portrait mode
    [director setDeviceOrientation:kCCDeviceOrientationPortrait];

    // replace HelloWorld with GameScene
    [[CCDirector sharedDirector] runWithScene: [GameScene scene]];
}
```

Compile and run, and you should end up with a blank scene. Success! If you run into any problems, compare your project with the DoodleDrop01 project that accompanies this book.

NOTE: Starting with cocos2d version 0.99.5, a file named GameConfig.h was added to the cocos2d Xcode project templates. If portrait mode isn't working for you and you started a new project based on one of the cocos2d Xcode project templates, it's probably because the game defaults to autorotation. If you see

```
#define GAME_AUTOROTATION kGameAutorotationUIViewController
```

in *GameConfig.h*, you have to change it to this line in order to disable autorotation:

```
#define GAME_AUTOROTATION kGameAutorotationNone.
```

Adding the Player Sprite

Next you'll add the player sprite and use the accelerometer to control its actions. To add the player image, select the Resources group in Xcode and select **File ►► Add Files to DoodleDrop** or, alternatively, right-click and from the context menu pick **Add Files to DoodleDrop** to open the file picker dialog. The player image *alien.png* is located in the Resources folder of the DoodleDrop project supplied with the book. You can also choose your own image, as long as it's 64 by 64 pixels in size.

Xcode will then ask you details about how and where to add the files, as in Figure 4–6. Make sure the **Add To Targets** check boxes are set for each target that will use the files, which in this case is only the DoodleDrop target. The defaults are good enough.

TIP: The preferred image format for iOS games is PNG (which stands for Portable Network Graphics). It's a compressed file format, but unlike JPG, the compression is lossless, retaining all pixels of the original image unchanged. Although you can also save JPEG files without compression, the same image in PNG format is typically smaller than an uncompressed JPEG file. This affects only the app size, not the memory (RAM) usage of the textures. In Chapter 16, you'll also learn about TexturePacker, a tool that manages images for you. It allows you to convert images into various compressed formats or reduce the color depth while retaining the best possible image quality through dithering and other techniques.

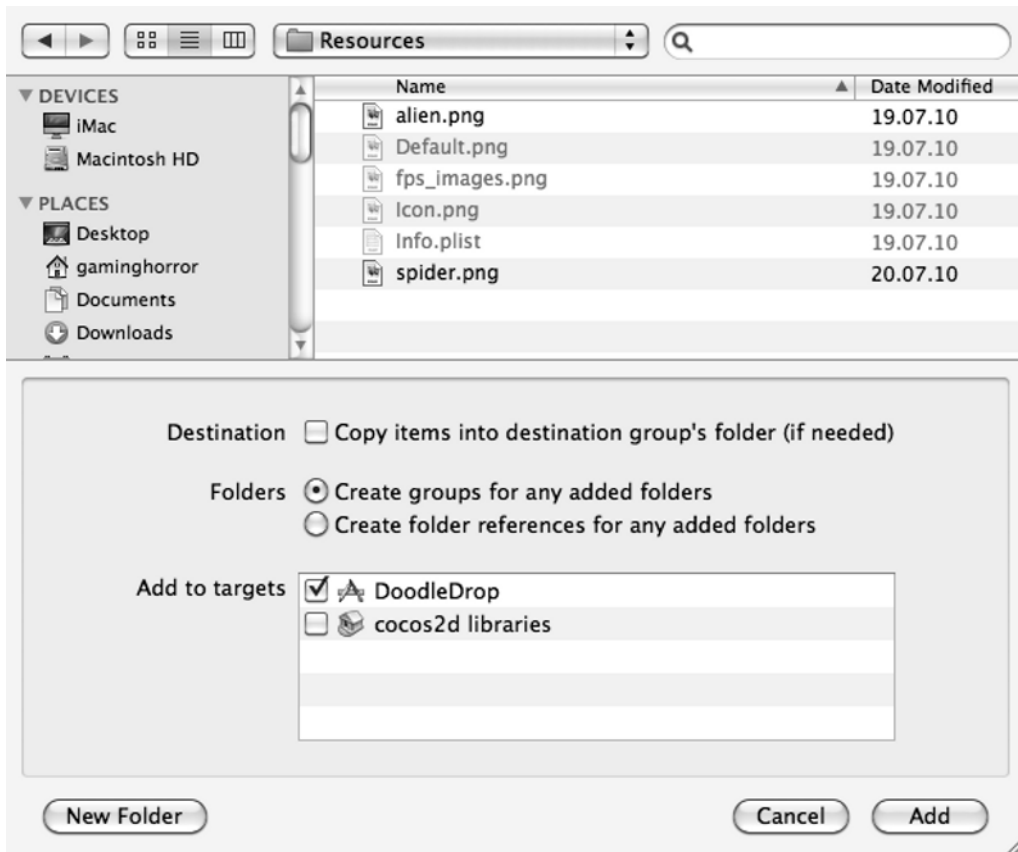


Figure 4–6. You'll see this dialog whenever you add resource files. In most cases you should use these default settings.

Now we'll add the player sprite to the game scene. I decided to add it as a `CCSprite*` member variable to the `GameScene` class. This is easier for now, and the game is simple enough for everything to go into the same class. Generally, that's not the recommended approach, so the projects following this one will create separate classes for individual game components as a matter of good code design.

Listing 4–4 shows the addition of the `CCSprite*` member to the `GameScene` header file.

Listing 4–4. *The CCSprite* Player Is Added as a Member Variable to the GameScene Class*

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

@interface GameScene : CCLayer
{
    CCSprite* player;
}

+(id) scene;

@end
```

Listing 4–5 contains the code I’ve added to the `init` method to initialize the sprite, assign it to the member variable, and position it at the bottom center of the screen. I’ve also enabled accelerometer input.

Listing 4–5. *Enabling Accelerometer Input and Creating and Positioning the Player Sprite*

```
-(id) init
{
    if ((self = [super init]))
    {
        CCLOG(@"%@: %@", NSStringFromSelector(_cmd), self);

        self.isAccelerometerEnabled = YES;

        player = [CCSprite spriteWithFile:@"alien.png"];
        [self addChild:player z:0 tag:1];

        CGSize screenSize = [[CCDirector sharedDirector] winSize];
        float imageHeight = [player texture].contentSize.height;
        player.position = CGPointMake(screenSize.width / 2, imageHeight / 2);
    }

    return self;
}
```

The player sprite is added as a child with a tag of 1, which will later be used to identify and separate the player sprite from all other sprites. Notice that I don’t retain the player sprite. Since we’ll add it as a child to the layer, `cocos2d` will retain it, and since the player sprite is never removed from the layer, that’s sufficient to keep the player sprite without specifically retaining it. Not retaining an object whose memory is managed by another class or object is called *keeping a weak reference*.

CAUTION: File names on iOS devices are case-sensitive. If you try to load `Alien.png` or `ALIEN.PNG`, it will work in the simulator but not on any iOS device because the real name is `alien.png` in all lowercase. That’s why it’s a good idea to stick to a naming convention like rigorously keeping all file names in all lowercase. Why lowercase? Because filename in all uppercase are typically harder to read.

You set the initial position of the player sprite by centering the x position at half the screen width, which puts the sprite in the center horizontally. Vertically we want the bottom of the player sprite's texture to align with the bottom of the screen. If you remember from the previous chapter, you know that the sprite texture is centered on the node's position. Positioning the sprite vertically at 0 would cause the bottom half of the sprite texture to be below the screen. That's not what we want; we want to move it up by half the texture height.

This is done by the call to `[player texture].contentSize.height`, which returns the sprite texture's content size. What exactly is the content size? In Chapter 3, I mentioned that the texture dimensions of iOS devices can only be powers of two. But the actual image size may be less than the texture size. For example, this is the case if the image is 100 by 100 pixels while the texture has to be 128 by 128 pixels. The `contentSize` property of the texture returns the original image's size of 100 by 100 pixels. In most cases, you'll want to work with the content size, not the texture size. Even if your image is a power of two, you should use `contentSize` because the texture might be a texture atlas containing multiple images. Texture atlases are described in Chapter 6.

By taking half of the image height and setting this as the position on the y-axis, the sprite image will align neatly with the bottom of the screen.

TIP: It's good practice to avoid using fixed positions wherever you can. If you simply set the player position to 160,32, you are making two assumptions you should avoid. First, you're assuming the screen width will be 320 pixels, but that will not hold true for every iOS device. Second, you're assuming that the image height is 64 pixels, but that might change too. Once you start to make assumptions like these, you're forming a habit to do so throughout the project.

The way I wrote the positioning code involves a bit more typing, but in the long run this pays off big time. You can deploy to different devices and it'll work, and you can use different image sizes and it'll work. There's no need to change this particular code anymore. One of the most time-consuming tasks a programmer faces is having to change code that was based on assumptions.

Imagine yourself three months down the road with lots of images and objects added to your game, having to change all those fixed numbers to create an iPad version that obviously also requires images of different sizes and then doing the same thing again to adjust the game to the iPhone 4's Retina Display. At that point, you have three different Xcode projects you need to maintain and add features to. Eventually it will lead to copy & paste hell, which is even more undesirable — don't go there!

Accelerometer Input

One last step, and then we should be done tilting the player sprite around. As I demonstrated in Chapter 3, you have to add the accelerometer method to the layer that receives accelerometer input. Here I use the `acceleration.x` parameter and add it to the player's position; multiplying by 10 is to speed up the player's movement.

```
-(void) accelerometer:(UIAccelerometer *)accelerometer
    didAccelerate:(UIAcceleration *)acceleration
{
    CGPoint pos = player.position;
    pos.x += acceleration.x * 10;
    player.position = pos;
}
```

Notice something odd? I wrote three lines where one might seem to suffice:

```
// ERROR: lvalue required as left operand of assignment
player.position.x += acceleration.x * 10;
```

Unlike other programming languages such as Java, C++, and C#, writing something like `player.position.x += value` won't work with Objective-C properties. The `position` property is a `CGPoint`, which is a regular C struct data type. Objective-C properties simply can't assign a value to a field in a struct directly. The problem lies in how properties work in Objective-C and also how assignment works in the C language, on which Objective-C is based.

The statement `player.position.x` is actually a call to the position getter method [`player position`], which means you're actually retrieving a temporary position and then trying to change the `x` member of the temporary `CGPoint`. But the temporary `CGPoint` would then get thrown away. The position setter [`player setPosition`] simply will not be called automatically. You can only assign to the `player.position` property directly, in this case a new `CGPoint`. In Objective-C you'll have to live with this unfortunate issue and possibly change programming habits if you come from a Java, C++, or C# background.

This is why the previous code has to create a temporary `CGPoint` object, change the point's `x` field, and then assign the temporary `CGPoint` to `player.position`. Unfortunately, this is how it needs to be done in Objective-C.

First Test Run

Your project should now be at the same level as the one in the `DoodleDrop02` folder of the code provided with this chapter. Give it a try now. Make sure you choose to run the app on the device, because you won't get accelerometer input from the simulator. Test how the accelerometer input behaves in this version.

If you haven't installed your development provisioning profiles in Xcode for this particular project yet, you'll get a CodeSign error. Code signing is required to run an app on an iOS device. Please refer to Apple's documentation to learn how to create and install the necessary development provisioning profiles (<http://developer.apple.com/ios/manage/provisioningprofiles/howto.action>).

Player Velocity

Notice how the accelerometer input isn't quite right? It's reacting slowly, and the motion isn't fluid. That's because the player sprite doesn't experience true acceleration and deceleration. Let's fix that now. The accompanying code changes are found in the DoodleDrop03 project.

The concept for implementing acceleration and deceleration is not to change the player's position directly but to use a separate `CGPoint` variable as a velocity vector. Every time an accelerometer event is received, the velocity variable accumulates input from the accelerometer. Of course, that means we also have to limit the velocity to an arbitrary maximum; otherwise, it'll take too long to decelerate. The velocity is then added to the player position every frame, regardless of whether accelerometer input was received.

NOTE: Why not use actions to move the player sprite? Well, move actions are a bad choice whenever you want to change an object's speed or direction very often, say multiple times per second. Actions are designed to be relatively long-lived objects, so creating new ones frequently creates additional overhead in terms of allocating and releasing memory. This can quickly drain a game's performance.

Worse yet, actions don't work at all if you don't give them any time to do their work. That's why adding a new action to replace the previous one every frame won't show any effect whatsoever. Many cocos2d developers have stumbled across this seemingly odd behavior.

For example, stopping all actions and then adding a new `MoveBy` action to an object every frame will not make it move at all! The `MoveBy` action will change the object's position only in the next frame. But that's when you're already stopping all actions again and adding another new `MoveBy` action. Repeat *ad infinitum*, but the object will simply not move at all. It's like the clichéd donkey: push it too hard, and it'll become a stubborn, immobile object.

Let's go through the code changes. The `playerVelocity` variable is added to the header:

```
@interface GameScene : CCLayer
{
    CCSprite* player;
    CGPoint playerVelocity;
}
```

If you wonder why I'm using a `CGPoint` instead of `float`, who's to say you'll never want to accelerate up or down a little? So, it doesn't hurt to be prepared for future expansions.

Listing 4–6 shows the accelerometer code, which I changed to use the velocity instead of updating the player position directly. It introduces three new design parameters for the amount of deceleration, the accelerometer sensitivity, and the maximum velocity. Those are values that don't have an optimum; you need to tweak them and find the right

settings that work best with your game's design (which is why they're called *design parameters*).

Deceleration works by reducing the current velocity before adding the new accelerometer value multiplied by the sensitivity. The lower the deceleration, the quicker the player can change the alien's direction. The higher the sensitivity, the more responsive the player will react to accelerometer input. These values interact with each other since they modify the same value, so be sure to tweak only one value at a time.

Listing 4-6. GameScene Implementation Gets playerVelocity

```
-(void) accelerometer:(UIAccelerometer *)accelerometer
    didAccelerate:(UIAcceleration *)acceleration
{
    // controls how quickly velocity decelerates (lower = quicker to change direction)
    float deceleration = 0.4f;
    // determines how sensitive the accelerometer reacts (higher = more sensitive)
    float sensitivity = 6.0f;
    // how fast the velocity can be at most
    float maxVelocity = 100;

    // adjust velocity based on current accelerometer acceleration
    playerVelocity.x = playerVelocity.x * deceleration + acceleration.x * sensitivity;

    // we must limit the maximum velocity of the player sprite, in both directions
    if (playerVelocity.x > maxVelocity)
    {
        playerVelocity.x = maxVelocity;
    }
    else if (playerVelocity.x < - maxVelocity)
    {
        playerVelocity.x = - maxVelocity;
    }
}
```

Now `playerVelocity` will be changed, but how do you add the velocity to the player's position? You do so by scheduling the update method in the `GameScene` init method, by adding this line:

```
// schedules the ...-(void) update:(ccTime)delta method to be called every frame
[self scheduleUpdate];
```

You also need to add the `...-(void) update:(ccTime)delta` method as shown in Listing 4-7. The scheduled update method is called every frame, and that's where we add the velocity to the player position. This way, we get a smooth constant movement in either direction regardless of the frequency of accelerometer input.

Listing 4-7. Updating the Player's Position with the Current Velocity

```
-(void) update:(ccTime)delta
{
    // Keep adding up the playerVelocity to the player's position
    CGPoint pos = player.position;
    pos.x += playerVelocity.x;

    // The Player should also be stopped from going outside the screen
    CGSize screenSize = [[CCDirector sharedDirector] winSize];
    float imageWidthHalved = [player.texture].contentSize.width * 0.5f;
```

```
float leftBorderLimit = imageWidthHalved;
float rightBorderLimit = screenSize.width - imageWidthHalved;

// preventing the player sprite from moving outside the screen
if (pos.x < leftBorderLimit)
{
    pos.x = leftBorderLimit;
    playerVelocity = CGPointZero;
}
else if (pos.x > rightBorderLimit)
{
    pos.x = rightBorderLimit;
    playerVelocity = CGPointZero;
}

// assigning the modified position back
player.position = pos;
}
```

TIP: The `contentSize` width is not divided by two but rather multiplied by 0.5 in order to calculate `imageWidthHalved`. This is a conscious choice and leads to the same results because any division can be rewritten as a multiplication.

The update method is called in every frame, and every piece of code that runs in every frame should be running at top speed. And since the ARM CPUs of the iOS devices don't support division operations in hardware, multiplications are generally a bit faster. It's not going to be noticeable in this case, but it's a good habit to get into since, unlike other (premature) optimizations, it doesn't make the code harder to read or maintain.

Another option for making this code just a little bit faster would be to precalculate values like `imageWidthHalved`, `leftBorderLimit`, and `rightBorderLimit` and then store them in instance variables. This is going to work as long as the player's texture remains the same while the game is running. The trade-off here is higher memory usage for fewer CPU cycles, and you'll generally find that this trade-off is very common for most code optimizations. But, as long you don't need to squeeze out speed, it's advisable to write readable and flexible code first and optimize only when you really must.

A boundary check prevents the player sprite from leaving the screen. Once again, we have to take the player texture's `contentSize` into account, since the player position is at the center of the sprite image but we don't want either side of the image to be off the screen. For this, we calculate `imageWidthHalved` and then use it to check whether the newly updated player position is within the left and right border limits. The code may be a bit verbose at this point, but that makes it easier to understand. And that's it with the player accelerometer input logic.

TIP: You'll notice that this straightforward implementation of accelerometer control doesn't give you the same dynamic feeling that you may be used to from games like *Tilt to Live*. The reason is that smooth, dynamic accelerometer controls require additional computations on the input values that go beyond the scope of this book.

The technique to smooth the accelerometer input is called *filtering*. To be specific, you might want to search the cocos2d forum primarily for high-pass and low-pass filtering. Filtering the accelerometer input values reduces the effects of sudden peaks in acceleration (high-pass filter) as well as cancels out the effects of gravity or simply one's natural hand tremor caused by our pulse (low-pass filter).

In addition, there are several ways to have an accelerometer-controlled object appear to have momentum. To achieve this effect, you would apply an easing function to the input values. Easing functions can simulate the effect of an object wanting to continue to move in the same direction with the same velocity unless an external force is exerted on the object. In other words, easing can emulate Newton's first and second laws of motion.

Adding Obstacles

This game isn't any good until we add something for the player to avoid. The DoodleDrop04 project introduces an abomination of nature: a six-legged man-spider. Who wouldn't want to avoid that?

As with the player sprite, you should add the spider.png to the Resources group. Then the `GameScene.h` file gets three new member variables added to its interface: a `spiders` `CCArray` whose class reference is shown in Listing 4–9 and the `spiderMoveDuration` and `numSpidersMoved`, which are used in Listing 4–12:

```
@interface GameScene : CCLayer
{
    CCSprite* player;
    CGPoint playerVelocity;

    CCArray* spiders;
    float spiderMoveDuration;
    int numSpidersMoved;
}
```

And in the `GameScene` `init` method add the call to the `initSpiders` method discussed next, right after `scheduleUpdate`:

```
-(id) init
{
    if ((self = [super init]))
    {
```

```

        [self scheduleUpdate];
        [self initSpiders];
    }
    return self;
}

```

After that a fair bit of code is added to the `GameScene` class, beginning with the `initSpiders` method in Listing 4–8, which is creating the spider sprites.

Listing 4–8. *For Easier Access, Spider Sprites Are Initialized and Added to a `CCArray`*

```

-(void) initSpiders
{
    CGSize screenSize = [[CCDirector sharedDirector] winSize];

    // using a temporary spider sprite is the easiest way to get the image's size
    CCSprite* tempSpider = [CCSprite spriteWithFile:@"spider.png"];

    float imageWidth = [tempSpider texture].contentSize.width;

    // Use as many spiders as can fit next to each other over the whole screen width.
    int numSpiders = screenSize.width / imageWidth;

    // Initialize the spiders array using alloc.
    spiders = [[CCArray alloc] initWithCapacity:numSpiders];

    for (int i = 0; i < numSpiders; i++)
    {
        CCSprite* spider = [CCSprite spriteWithFile:@"spider.png"];
        [self addChild:spider z:0 tag:2];

        // Also add the spider to the spiders array.
        [spiders addObject:spider];
    }

    // call the method to reposition all spiders
    [self resetSpiders];
}

```

There are a few things to note. I create a `tempSpider` `CCSprite` only to find out the sprite's image width, which is then used to decide how many spider sprites can fit next to each other. The easiest way to get an image's dimensions is by simply creating a temporary `CCSprite`. Note that I did not add the `tempSpider` as child to any other node. This means its memory will be released automatically.

This is in contrast to the `spiders` array I'm using to hold references to the spider sprites. This array must be created using `alloc`; otherwise, its memory would be released, and subsequent access to the sprites array would crash the app with an `EXC_BAD_ACCESS` error. And since I took control over managing the sprites' array memory, I must not forget to release the `spiders` array in the `dealloc` method, as shown here:

```

-(void) dealloc
{
    CCLOG(@"%@: %@", NSStringFromSelector(_cmd), self);

    // The spiders array must be released, it was created using [CCArray alloc]
    [spiders release];
}

```

```

    spiders = nil;

    // Never forget to call [super dealloc]!
    [super dealloc];
}

```

The CCArry class is, at this time of writing, an undocumented but fully supported class of cocos2d. You can find the CCArry class files in the cocos2d/Support group in the Xcode project. It's used internally by cocos2d and is similar to Apple's NSMutableArray class except that it performs better. The CCArry class implements a subset of the NSArray and NSMutableArray classes and also adds new methods to initialize a CCArry from an NSArray. It also implements fastRemoveObject and fastRemoveObjectAtIndex methods by simply assigning the last object in the array to the deleted position in order to avoid copying parts of the array's memory. This is faster, but it also means objects in CCArry will change positions, so if you rely on a specific ordering of objects, you shouldn't use the fastRemoveObject methods. In Listing 4–9 you can see the full CCArry class reference because it doesn't implement all of the methods of NSArray and NSMutableArray while adding its own.

Listing 4–9. CCArry Class Reference

```

+ (id) array;
+ (id) arrayWithCapacity:(NSUInteger)capacity;
+ (id) arrayWithArray:(CCArray*)otherArray;
+ (id) arrayWithNSArray:(NSArray*)otherArray;

- (id) initWithCapacity:(NSUInteger)capacity;
- (id) initWithArray:(CCArray*)otherArray;
- (id) initWithNSArray:(NSArray*)otherArray;

- (NSUInteger) count;
- (NSUInteger) capacity;
- (NSUInteger) indexOfObject:(id)object;
- (id) objectAtIndex:(NSUInteger)index;
- (id) lastObject;
- (BOOL) containsObject:(id)object;

#pragma mark Adding Objects

- (void) addObject:(id)object;
- (void) addObjectsFromArray:(CCArray*)otherArray;
- (void) addObjectsFromNSArray:(NSArray*)otherArray;
- (void) insertObject:(id)object atIndex:(NSUInteger)index;

#pragma mark Removing Objects

- (void) removeLastObject;
- (void) removeObject:(id)object;
- (void) removeObjectAtIndex:(NSUInteger)index;
- (void) removeObjectsInArray:(CCArray*)otherArray;
- (void) removeAllObjects;
- (void) fastRemoveObject:(id)object;
- (void) fastRemoveObjectAtIndex:(NSUInteger)index;

- (void) makeObjectsPerformSelector:(SEL)aSelector;
- (void) makeObjectsPerformSelector:(SEL)aSelector withObject:(id)object;

- (NSArray*) getNSArray;

```

At the end of Listing 4–8, the method `[self resetSpiders]` is called; this method is shown in Listing 4–10. The reason for separating the initialization of the sprites and positioning them is that eventually there will be a game over, after which the game will need to be reset. The most efficient way to do so is to simply move all game objects to their initial positions. However, that may stop being feasible once your game scene gets to a certain complexity. Eventually, it'll be easier to simply reload the whole scene, at the cost of having the player wait for the scene to reload.

CAUTION: Speaking of reloading a scene, you may be tempted to write something like `[[CCDirector sharedDirector] replaceScene:[GameScene node]];` within the `GameScene` class, or even `[[CCDirector sharedDirector] replaceScene:self];`, to reload the same scene. Both attempts will cause a crash! Instead, you should load another (intermediary) scene first before loading the same scene again. You'll learn how to write a loading scene in Chapter 5.

Listing 4–10. Resetting Spider Sprite Positions

```
-(void) resetSpiders
{
    CGSize screenSize = [[CCDirector sharedDirector] winSize];

    // Get any spider to get its image width
    CCSprite* tempSpider = [spiders lastObject];
    CGSize size = [tempSpider texture].contentSize;

    int numSpiders = [spiders count];
    for (int i = 0; i < numSpiders; i++)
    {
        // Put each spider at its designated position outside the screen
        CCSprite* spider = [spiders objectAtIndex:i];
        spider.position = CGPointMake(size.width * i + size.width * 0.5f,
                                       screenSize.height + size.height);

        [spider stopAllActions];
    }

    // Unschedule the selector just in case. If it isn't scheduled it won't do anything.
    [self unschedule:@selector(spidersUpdate)];

    // Schedule the spider update logic to run at the given interval.
    [self schedule:@selector(spidersUpdate) interval:0.7f];

    // reset the moved spiders counter and spider move duration (affects speed)
    numSpidersMoved = 0;
    spiderMoveDuration = 4.0f;
}
```

Once again I obtain a reference to one of the existing spiders temporarily to get its image size via the texture's `contentSize` property. I don't create a new sprite here since there are already existing sprites of the same kind, and since all spiders use the same

image with the same size, I don't even care which sprite I'm getting. So, I simply choose the get the last spider from the array.

Each spider's position is then modified so that together they span the entire width of the screen. Half of the image size's width is added, and once again this is because of the sprite's texture being centered on the node's position. As for the height, each sprite is also set to be one image size above the upper screen border. This is an arbitrary distance, as long as the image isn't visible, which is all I want to achieve. Because the spider might still be moving when the reset occurs, I'll also stop all of its actions at this point.

TIP: To save a few CPU cycles, it's good practice not to use method calls in the conditional block of for or other loops if it's not strictly necessary. In this case, I created a variable `numSpiders` to hold the result of `[spiders count]`, and I use that in the conditional check of the for loop. The count of the array remains the same during the for loop's iterations because the array itself isn't modified in the loop. That's why I can cache this value and save the repeated calls to `[spiders count]` during each iteration of the for loop.

I'm also scheduling the `spidersUpdate: selector` to run every 0.7 seconds, which is how often another spider will drop down from the top of the screen. But before doing so, I make sure the same selector is unscheduled, just to be safe. The `resetSpiders` method may be called while `spidersUpdate:` is still scheduled, and I don't want the method to be called twice, effectively doubling the spider drop-down rate. The `spidersUpdate:` method, shown in Listing 4–11, randomly picks one of the existing spiders, checks whether it is idle, and lets it fall down the screen by using a sequence of actions.

Listing 4–11. *The spidersUpdate: Method Frequently Lets a Spider Fall*

```
-(void) spidersUpdate:(ccTime)delta
{
    // Try to find a spider which isn't currently moving.
    for (int i = 0; i < 10; i++)
    {
        int randomSpiderIndex = CCRANDOM_0_1() * [spiders count];
        CCSprite* spider = [spiders objectAtIndex:index:randomSpiderIndex];

        // If the spider isn't moving it won't have any running actions.
        if ([spider numberOfRunningActions] == 0)
        {
            // This is the sequence which controls the spiders' movement
            [self runSpiderMoveSequence:spider];

            // Only one spider should start moving at a time.
            break;
        }
    }
}
```

I don't let any listing pass without some curiosity, do I? In this case, you might wonder why I'm iterating exactly ten times to get a random spider. The reason is that I don't

know if the randomly generated index will get me a spider that isn't moving already, so I want to be reasonably sure that eventually a spider is randomly picked that is currently idle. If after ten tries[□] and this number is arbitrary[□] I did not have the luck to get an idle spider chosen randomly, I'll simply skip this update and wait for the next.

I could brute-force my way and just keep trying to find an idle spider using a do/while loop. However, it's possible that all spiders could be moving at the same time, since this depends on design parameters such as the frequency with which new spiders are being dropped. In that case, the game would simply lock up, looping endlessly trying to find an idle spider. Moreover, I'm not so keen on trying too hard; it really doesn't matter much for this game if I'm unable to send another spider falling down for a couple of seconds. That said, if you check out the DoodleDrop04 project, you'll see I added a logging statement that will print out how many retries it took to find an idle spider.

Since the movement sequence is the only action the spiders perform, I simply check whether the spider is running any actions at all, and if not, I assume it is idle. And that brings us to the `runSpiderMoveSequence` in Listing 4–12.

Listing 4–12. Spider Movement Is Handled by an Action Sequence

```
-(void) runSpiderMoveSequence:(CCSprite*)spider
{
    // Slowly increase the spider speed over time.
    numSpidersMoved++;
    if (numSpidersMoved % 8 == 0 && spiderMoveDuration > 2.0f)
    {
        spiderMoveDuration -= 0.1f;
    }

    // This is the sequence which controls the spiders' movement.
    CGPoint belowScreenPosition = CGPointMake(spider.position.x, ←
        -[spider texture].contentSize.height);
    CCMoveTo* move = [CCMoveTo actionWithDuration:spiderMoveDuration
        position:belowScreenPosition];
    CCCallFuncN* callDidDrop = [CCCallFuncN actionWithTarget:self
        selector:@selector(spiderDidDrop:)];
    CCSequence* sequence = [CCSequence actions:move, callDidDrop, nil];
    [spider runAction:sequence];
}
```

The `runSpiderMoveSequence` method keeps track of the number of dropped spiders. Every eighth spider, the `spiderMoveDuration` is decreased, and thus any spider's speed is increased. In case you are wondering about the `%` operator, it's called the *modulo* operator. The result is the remainder of the division operation, meaning if `numSpidersMoved` is divisible by 8, the result of the modulo operation will be 0.

The action sequence consists only of a `CCMoveTo` action and a `CCCallFuncN` action. It could be improved to let spiders drop down a bit, wait, and then drop all the way, as evil six-legged man-spiders would normally do. I leave this improvement up to you. For now it's only important to know that I chose the `CCCallFuncN` variant because I want the `spiderDidDrop` method to be called with the spider sprite as a sender parameter. This way, I get a reference to the spider that has reached its destination (it dropped past the player character), and I don't have to jump through hoops to find the

right spider. Listing 4–13 reveals how it’s done by resetting the spider position to just above the top of the screen whenever a spider has reached its destination just below the screen.

Listing 4–13. *Resetting a Spider Position So It Can Fall Back Down Again*

```
-(void) spiderDidDrop:(id)sender
{
    // Make sure sender is actually of the right class.
    NSAssert([sender isKindOfClass:[CCSprite class]], @"sender is not a CCSprite!");
    CCSprite* spider = (CCSprite*)sender;

    // move the spider back up outside the top of the screen
    CGPoint pos = spider.position;
    CGSize screenSize = [[CCDirector sharedDirector] winSize];
    pos.y = screenSize.height + [spider texture].contentSize.height;
    spider.position = pos;
}
```

NOTE: Being a defensive programmer, I’ve added the `NSAssert` line to make sure that sender is of the right class, since I’m assuming that sender will be a `CCSprite`, but it might not be one.

Indeed, when I first ran this code, I forgot to use `CCCallFuncN` and actually used a `CCCallFunc`, which led to sender being `nil`, since `CCCallFunc` doesn’t pass the sender parameter. `NSAssert` caught this case, too. With sender being `nil`, the method `isKindOfClass` was never called and the return value became `nil`, causing the `NSAssert` to trigger. It wasn’t the error I expected, but `NSAssert` caught it anyway. With that information, it was easy to figure out what I was doing wrong and fix it.

Once I’m sure that sender is of the class `CCSprite`, I can cast it to `CCSprite*` and use it to adjust the sprite’s position. The process should be familiar by now.

So far, so good. You might want to try the game and play it a little. I think you’ll quickly notice what’s still missing. Hint: read the next headline.

Collision Detection

You may be surprised to see that collision detection can be as simple as in Listing 4–14. Admittedly, this only checks the distance between the player and all spiders, which makes this type of collision detection a radial check. For this type of game, it’s sufficient. The call to `[self checkForCollision]` is added to the end of the `...(void) update:(ccTime)delta` method.

Listing 4–14. *A Simple Range-Check or Radial Collision-Check Suffices*

```
-(void) checkForCollision
{
    // Assumption: both player and spider images are squares.
    float playerImageSize = [player texture].contentSize.width;
    float spiderImageSize = [[spiders lastObject] texture].contentSize.width;
```

```

float playerCollisionRadius = playerImageSize * 0.4f;
float spiderCollisionRadius = spiderImageSize * 0.4f;

// This collision distance will roughly equal the image shapes.
float maxCollisionDistance = playerCollisionRadius + spiderCollisionRadius;

int numSpiders = [spiders count];
for (int i = 0; i < numSpiders; i++)
{
    CCSprite* spider = [spiders objectAtIndex:i];

    if ([spider numberOfRunningActions] == 0)
    {
        // This spider isn't even moving so we can skip checking it.
        continue;
    }

    // Get the distance between player and spider.
    float actualDistance = ccpDistance(player.position, spider.position);

    // Are the two objects closer than allowed?
    if (actualDistance < maxCollisionDistance)
    {
        // Game Over (just restart the game for now)
        [self resetSpiders];
        break;
    }
}
}

```

The image sizes of the player and spider are used as hints for the collision radii. The approximation is good enough for this game. If you check the DoodleDrop05 project, you'll also notice that I've added a debug drawing method that renders the collision radii for each sprite.

NOTE: The correct plural of radius is radii. You can also say radiuses without being expelled from the country, though I wouldn't dare say it in the vicinity of programmers. Less critically but also hotly debated is the plural of vertex, which is correctly spelled *vertices*, but *vertexes* is also deemed acceptable.

I'm iterating over all the spiders but ignoring those that aren't moving at the moment because they'll definitely be out of range. The distance between the current spider and the player is calculated by the `ccpDistance` method. This is another undocumented but fully supported cocos2d method. You can find these and other useful math functions in the `CGPointExtension` files in the `cocos2d/Support` group in the Xcode project.

The resulting distance is then compared to the sum of the player's and spider's collision radius. If the actual distance is smaller than that, a collision has occurred. Since no game-over has been implemented, I chose to simply reset the spiders to restart the game.

Labels and Bitmap Fonts

Labels are the second most important graphical element of cocos2d games, right after sprites. The easiest and most flexible solution seems to be the `CCLabelTTF` class, but its performance is terrible if you need to change the displayed text frequently. The alternative is the `CCLabelBMFont` class, which renders bitmap fonts instead of TrueType fonts. Along with that I'll introduce you to Glyph Designer, an elegant tool for converting TrueType fonts into bitmap fonts and enhancing them along the way with effects like shadows, color gradients, and so on.

Adding the Score Label

The game needs some kind of scoring mechanism. I decided to add a simple time-lapse counter as the score. I start by adding the score's Label in the `init` method of the `GameScene` class:

```
scoreLabel = [CCLabelTTF labelWithString:@"0" fontName:@"Arial" fontSize:48];
scoreLabel.position = CGPointMake(screenSize.width / 2, screenSize.height);
```

```
// Adjust the label's anchorPoint's y position to make it align with the top.
scoreLabel.anchorPoint = CGPointMake(0.5f, 1.0f);
```

```
// Add the score label with z value of -1 so it's drawn below everything else
[self addChild:scoreLabel z:-1];
```

I consciously chose a `CCLabelTTF` object, which would likely be the first choice for most beginning cocos2d programmers. I'll show you how quickly these label objects rear their ugly heads, though. Add the following code anywhere in the `update:` method so that the score label will be updated like a stopwatch counter every second:

```
// Update the Score (Timer) once per second.
totalTime += delta;
int currentTime = (int)totalTime;
if (score < currentTime)
{
    score = currentTime;
    [scoreLabel setString:[NSString stringWithFormat:@"%i", score]];
}
```

The `delta` parameter of the `update:` method is continuously added to the `totalTime` member variable to keep track of the time that has passed. Because `totalTime` is a floating-point variable, I simply assign it to an integer variable, effectively removing the fractional part. That makes it possible to compare the score with the `currentTime`; if the score is lower, `currentTime` becomes the new score, and the `CCLabelTTF`'s string is updated.

And that's where things get ugly. In the previous chapter, I mentioned that updating a `CCLabelTTF`'s text is slow. The whole texture is re-created using iOS font-rendering methods, and they take their time, besides allocating a new texture and releasing the old one. If you play this version of the game on the device, you may notice how the spiders

seem to jump a little every time the score label changes. The game is no longer running smoothly.

If you want to see this effect more pronounced, comment out the line with the `if` statement so that `[scoreLabel setString:]` is run every frame. On my iPhone 3GS, the framerate suddenly drops from a previous constant 60 frames per second to below 30—all this because of one measly `CCLabelTTF` updated every frame!

But keep in mind that `CCLabelTTF` is slow only when changing its string frequently. If you create the `CCLabelTTF` once and never change it, it's just as fast as any other `CCSprite` of the same dimensions.

Introducing CCLabelBMFont

Labels that update fast at the expense of more memory usage, like any other `CCSprite`, are the specialty of the `CCLabelBMFont` class. I've replaced the `CCLabelTTF` with a `CCLabelBMFont` in `DoodleDrop07`. It's relatively straightforward; besides changing the declaration of the `scoreLabel` variable from `CCLabelTTF` to `CCLabelBMFont` in the header file, you only have to change the line in the `init` method, as shown here:

```
scoreLabel = [CCLabelBMFont labelWithString:@"0" fntFile:@"bitmapfont.fnt"];
```

NOTE: Bitmap fonts are a great choice for games because they are fast, but they do have one disadvantage. The size of any bitmap font is fixed. If you need the same font but larger or smaller in size, you can scale the `CCLabelBMFont`—but you lose image quality scaling up, and you're potentially wasting memory scaling down. The other option is to create a separate font file with the new size, but this uses up more memory since each bitmap font comes with its own texture, even if only the font size changed.

But there's a catch, obviously. You need to add the `bitmapfont.fnt` file as well as the accompanying `bitmapfont.png`, which are both in the project's `Resources` folder. More importantly, you do want to create your own bitmap fonts sooner or later. The tool to use for that used to be `Hiero`; that was before `Glyph Designer` came along. Now `Hiero` is only the tool of choice if you really don't want to spend any money. `Hiero` was written by Kevin James Glass. It's a free Java Web application and is available from <http://slick.cokeandcode.com/demos/hiero.jnlp>.

The downside is, it's a free Java web application. It will ask you to trust the application because of a missing security certificate. On the other hand, many developers use the tool, and so far there has been no evidence that the application is untrustworthy. `Hiero` also features several odd and downright annoying bugs, including an obnoxious one that has the resulting image file flipped upside down. If you see only garbage instead of a bitmap font text in your app, you may have to flip the bitmap font PNG image upside down with an image-editing program. I have documented these issues and how to fix them in my `Hiero` tutorial: www.learn-cocos2d.com/knowledge-base/tutorial-bitmap-fonts-hiero/.

Some developers also swear by BMFont. But as a Windows program, it requires a Windows computer or Windows installed in a virtual machine on your Mac. That's why it's not more widely used in the Mac developer community. You can download BMFont from www.angelcode.com/products/bmfont/.

Finally, along came a tool that satisfied everyone who wouldn't hesitate to trade a few dollars for convenience and reliability: Glyph Designer.

Creating Bitmap Fonts with Glyph Designer

After the first edition of this book was printed, the guys at www.71squared.com released their Hiero replacement tool called Glyph Designer. Although it's not free, it's definitely worth every cent.

You can download a trial version of Glyph Designer on <http://glyphdesigner.71squared.com>, and if you're already familiar with Hiero, you'll notice a striking similarity in features, although the user interface is a lot easier to use and encourages experimentation. Mike Daley also mentioned in an episode of the Cocos2D Podcast available at <http://cocos2dpodcast.wordpress.com> that Glyph Designer will get a new feature that allows you to share font designs with other users of the tool.

Figure 4–7 shows Glypgh Designer in action. The process of creating a bitmap font is relatively playful, and it doesn't hurt to change the various knobs, buttons, and colors as you please. I'll outline just the basic editing areas for you.



Figure 4–7. *Glyph Designer allows you to create bitmap fonts from any TrueType font. It can export FNT and PNG files compatible with the CCLabelBMFont class of cocos2d.*

On the left side you have the list of TrueType fonts, and if those aren't enough, you can use the **Load Font** icon to load any TTF file. Below the list you can change the size of the font with the slider and also switch to bold, italic, and other font styles.

TIP: Creating Retina-enabled bitmap fonts is easy. Create your font as usual and export it. This will be your non-Retina or SD font. Then simply change the size of the font in Glyph Designer to twice its normal size; for example, move the slider from a font size of 30 to a font size of 60. Then reexport the font using the same name but adding the **-hd** suffix. Now you have the same font in regular/SD and Retina/HD sizes.

Cocos2d will automatically recognize and use the font with the **-hd** suffix if Retina support is enabled and the game is running on a Retina device.

In the center of the screen, you see the resulting texture atlas used for your current font settings. You'll notice that the texture atlas size and the order of the glyphs frequently change as you modify the font settings. You can also select a glyph and see its info on the right pane under Glyph Info.

Further down on the right pane you can change the texture atlas settings, although in most cases you don't have to. Glyph Designer makes sure that the texture atlas size is always large enough to contain all the glyphs in a single texture.

With the Glyph Fill settings, you can change the color and the way glyphs are being filled, including a gradient setting. Alongside that, you have the option to change the Glyph Outline, which is a black thin line around each glyph, and the Glyph Shadow, which allows you to create a 3D-ish look of the font.

At the very bottom on the right pane you find the Included Glyphs section. With that you can choose from a predefined selection of glyphs to include in the atlas. If you absolutely know for sure that you won't be needing certain characters, you can also enter your own list of characters to reduce the size of the texture. This is especially helpful for score strings where you may only need digits plus very few extra characters.

Once you are satisfied with your bitmap font, you can save the entire project so that you can restore previous settings. To save the font in a format usable by cocos2d, you have to save it via **File ► Export** in the .fnt (Cocos2d Text) format. You can then add the FNT and PNG files created by Glyph Designer to your Xcode project and use the FNT file with the `CCLabelBMFont` class.

CAUTION: If you try to display characters using a `CCLabelBMFont`, which are not available in the .fnt file, they will simply be skipped and not displayed. For example, if you do `[label setString:@" Hello, World!]` but your bitmap font contains only lowercase letters and no punctuation characters, you'll see the string `ello orld` displayed instead.

Simply Playing Audio

I've added some audio files to complete this game. In the Resources folder of the DoodleDrop07 project, you'll find the audio files named `blues.mp3` and `alien-sfx.caf` that you can add to your project. The first choice and the easiest way to play audio files in cocos2d is by using the `SimpleAudioEngine`. Audio support is not an integral part of cocos2d; this is the domain of `CocosDenshion`, a third-party addition to cocos2d and fortunately distributed with cocos2d.

TIP: If you're looking for an alternative sound engine, I recommend `ObjectAL` available from <http://kstenerud.github.com/ObjectAL-for-iPhone>, which is the preferred audio engine in `Kobold2D` (see Chapter 16). `ObjectAL` has a cleanly written API and excellent documentation.

Because CocosDenshion is treated as separate code from cocos2d, you have to add the corresponding header files whenever you use the CocosDenshion audio functionality, like so:

```
#import "GameScene.h"
#import "SimpleAudioEngine.h"
```

You'll find playing music and audio using the SimpleAudioEngine is straightforward, as shown here:

```
[[SimpleAudioEngine sharedEngine] playBackgroundMusic:@"blues.mp3" loop:YES];
[[SimpleAudioEngine sharedEngine] playEffect:@"alien-sfx.caf"];
```

For music and longer speech files, playing MP3 files is the preferred choice. Note that you can play only one MP3 file in the background at a time. Technically, it's possible to play two or more MP3 files, but only one can be decoded in hardware. The extra strain on the CPU is undesirable for games, so playing multiple MP3 files at the same time is out of the question for most. This also means that short-lived sound effects should not be in MP3 format. For those audio effects, I've had only good experiences with 16-bit PCM (uncompressed) audio in either the WAV or CAF file format. The sampling rate can be 22.5 kHz for most game sound effects, unless you need or want crystal-clear audio quality, in which case you should use 44.1 kHz.

A good and complete audio-editing tool for Mac OS X is Audacity, which you can download for free from <http://audacity.sourceforge.net>. If you need only to quickly convert audio files from one format to another, possibly changing some basic settings such as sampling rate, I recommend SoundConverter, which was developed by Steve Dekorte. The tool is free to use for files up to 500KB in size, and the license to use SoundConverter without restrictions is just \$15. You can download SoundConverter from <http://dekorte.com/projects/shareware/SoundConverter/>.

A free alternative to SoundConverter is the command-line tool `afconvert`. Familiarity with Terminal is recommended. You can do a lot with `afconvert`, but being a command-line tool, you'll also have to type all settings. To get help for `afconvert`, open the Terminal app and type the following:

```
afconvert -h
```

The preferred audio format for iOS devices is 16-bit, little endian, linear PCM packaged as CAF file (Apple CAF audio format code: LEI16), according to Apple's Audio Coding How To, which contains generally helpful advice for audio programming (http://developer.apple.com/library/ios/#codinghowtos/AudioAndVideo/_index.html)

To convert any audio file that `afconvert` supports to the preferred iOS audio format, you would run the `afconvert` command like this:

```
afconvert -f caff -d LEI16 myInputFile.mp3 myOutputFile.caf
```

The `-f` (or `-file`) switch denotes the file format, which is `caff` for CAF files. With the `-d` switch, you specify the audio data format, here `LEI16`. You can get a list of the audio data formats supported by `afconvert` by running `afconvert` with the `-hf` switch.

NOTE: If you ever find yourself in the situation where an audio file just won't play or results in a garbled mess of noise, don't worry. There are countless audio applications and numerous audio codecs that all create their own variations of the respective formats. Some are unable to play on iOS devices but play fine otherwise. Particularly, WAV files seem to be affected, which is why I prefer to use Apple's more native audio container format CAF. Typically, the way you can fix broken audio files is to open the audio file in an audio-editing program that you know is capable of saving iOS-compatible audio files and then save it again. You can do this with the aptly named SoundConverter or the audio application of your choice. Usually, after this resave, the file will play just fine on the iOS device.

Porting to iPad

With all coordinates taking the screen's size into account, the game should simply scale up without any problems when running it on the iPad's bigger screen. And it does. Just like that. In contrast, if you had been using fixed coordinates, you'd be facing a serious rework of your game.

One Universal App or Two Separate Apps?

When porting your app to iPad, you generally have to decide whether your app will be treated as a single (Universal) app on the App Store or whether it should be treated as two separate apps. Both options have their pros and cons, and generally you could say that Universal apps are better and fairer for customers, whereas separate apps are usually better for developers.

Universal apps include code and assets for both iPhone/iPod touch and iPad devices. This has the drawback that all assets are added to the same Xcode target, increasing the app's size. That's the technical drawback; there are no performance penalties.

With a Universal app, you will not be able to set different prices for iPhone/iPod touch and iPad versions, and user reviews and comments for all devices will be listed under the same app. Moreover, you won't know which percentage of your download's respective purchases were made by iPad users.

Regardless of that, Universal apps will still be ranked separately by device in the App Store charts. If the user downloads or purchases the app on an iPhone or iPod touch device, it is accounted for in the iPhone charts. The same goes for downloads/purchases on the iPad, which add to your app's ranking in the iPad charts. That leaves the question of how iTunes downloads/purchases are accounted for. They are simply accounted for in the iPhone rankings. That makes it impossible to even estimate how many of your users are iPad users, unless you add analytics tracking code to your app.

Splitting your app into two separate apps for iPhone/iPod touch and iPad allows you to keep the game assets separate. Most importantly, if an iOS user wants both the iPhone and iPad versions, he'll have to buy both. That is good for you but bad for the customer. And some won't hesitate to give your app a worse rating just because of that.

But since your app will be treated as two entirely separate apps in the App Store, at least the customer reviews and comments will be specific to the particular app version. You'll also be able to optimize each app's description and screenshots for the target platform and update each version separately. Splitting your app is also a good choice if your app has been on the App Store for a while, since adding support for new devices in a Universal app will not have your app appear in the What's New section of the App Store.

Porting to iPad with Xcode 3

Porting an iPhone project is a simple process. If you are using Xcode 3, then select the target in the Groups & Files pane that you want to convert and select **Project ► Upgrade Current Target for iPad**, which will bring up the dialog in Figure 4-8.

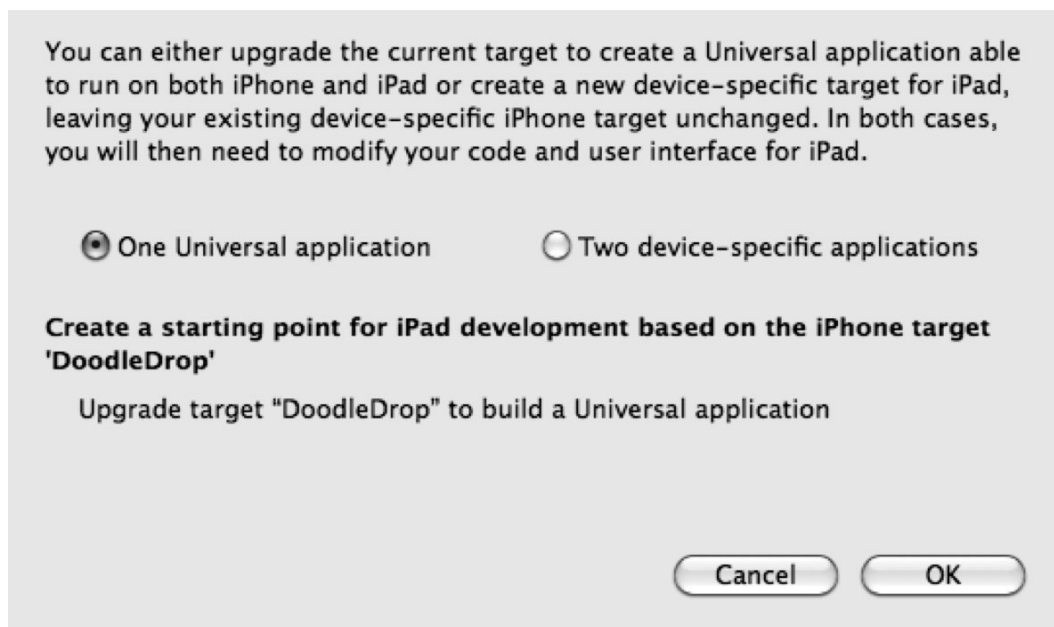


Figure 4-8. Upgrading a target for iPad in Xcode 3 gives you two choices. Universal application requires you to add both iPhone and iPad assets to the same target, increasing the app's download size.

For a simple game, the default **One Universal application** is fine. It does have the drawback that all assets are added to the same target, increasing the app's size. That's the only technical drawback; there are no performance penalties. The resulting app can be run on both the iPhone and iPad. Choosing the **Two device-specific applications** option lets you

keep game assets separate, but you end up with two apps, which you would have to submit to the App Store individually.

The easier option for now is to choose **One Universal application** and run the code. On the device, the app will automatically detect which device is connected and run the appropriate version. If you want to try it in the iPad Simulator, just select the iPad Simulator as the active executable, as in Figure 4–9.



Figure 4–9. To run the new Universal app in the iPad Simulator, make it the active executable.

Porting to iPad with Xcode 4

Xcode 4 makes porting apps incredibly straightforward, so much so that you can change the targeted device with a simple drop-down.

First, select the project DoodleDrop in the Project Navigator. This brings up the list of targets where you select the DoodleDrop target and then choose the **Summary** tab. Under the iOS Application Target section, there's a pop-up control labeled **Devices** that gives you three choices: iPhone, iPad, and Universal.

Depending on the device setting, your app is either built for iPhone or iPad specifically or supports both iPhone and iPad as a Universal app. In Figure 4–10 the target is set to build a Universal app.



Figure 4–10. Changing the application target to build a Universal app

Now you may be wondering, what if I want two separate targets for iPhone and iPad? This can be useful to charge different prices for iPhone and iPad versions or simply to reduce the download size of either version.

In that case, all you have to do is to select the target, DoodleDrop in this case, and choose **Edit ► Duplicate** from the menu. Or just right-click the target and select **Duplicate**. This creates a duplicate of the target, which allows you to set one target's Devices setting to iPhone and the other target's Devices setting to iPad. Now you have a separate target for each device type, and you might want to label the targets accordingly to avoid confusion.

Summary

I hope you had fun building this first game. It was surely a lot to take in, but I'd rather err on the side of too much information than too little.

At this point, you've learned how to create your own game-layer class and how to work with sprites. You've used the accelerometer to control the player and added velocity to allow the player sprite to accelerate and decelerate, giving it a more dynamic feel.

I also introduced you to the undocumented `CCArray` class, cocos2d's replacement for `NSMutableArray`. This should be your preferred choice when you need to store a list of data. Simple radial collision detection using the distance check method from the likewise undocumented `CGPointExtensions` was also on the menu.

And for dessert you had a potpourri of labels, bitmap fonts, and the Glyph Designer tool, garnished with some audio programming.

What's left? Maybe going through the source code for this chapter. I've added a finalized version of this game that includes some game-play improvements, a startup menu, and a game-over message.

There's just one thing about the DoodleDrop project I haven't mentioned yet: it's all in one class. For a small project, this may suffice, but it'll quickly get messy as you move on to implement more features. You need to add some structure to your code design. The next chapter will arm you with cocos2d programming best practices and show you how to lay out your code and the various ways information can be passed between objects if they are no longer in the same class.

Game Building Blocks

The game DoodleDrop in the previous chapter was written to be easy to understand if you're new to cocos2d. If you're a more experienced developer, though, you probably noticed that there is no separation of code; everything is in just one file. Clearly, this doesn't scale, and if you're going to make bigger, more exciting games than DoodleDrop, you'll have to find a suitable way to structure your code. Otherwise, you might end up with one class driving your game's logic. The code size can quickly grow to thousands of lines, making it hard to navigate and tempting to change anything from anywhere, very likely introducing subtle and hard-to-find bugs.

Each new project demands its own code design. In this chapter, I'll introduce you to some of the building blocks for writing more complex cocos2d games. The code foundation laid out in this chapter will then be used to create the side-scrolling shooter game we'll be building in the next few chapters.

Working with Multiple Scenes

The DoodleDrop game had only one scene and one layer. More complex games will surely need several scenes and multiple layers. How and when to use them will become second nature for you. Let's see what's involved.

Adding More Scenes

The basics still apply. In Listings 4-1 and 4-2 in the previous chapter, I outlined the basic code needed to create a scene. Adding more scenes is a matter of adding more classes built on that same basic code. It's when you're transitioning between scenes that things get a little more interesting. There's a set of three methods in `CCNode` that are called on each node object in the current scene hierarchy when you're replacing a scene via the `CCDirector replaceScene` method.

The `onEnter` and `onExit` methods get called at certain times during a scene change, depending on whether a `CCTransitionScene` is used. You must always call the super

implementation of these methods to avoid input problems and memory leaks. Take a look at Listing 5–1, and note that all of these methods call the super implementation.

Listing 5–1. *The onEnter and onExit Methods*

```
-(void) onEnter
{
    // Called right after a node's init method is called.
    // If using a CCTransitionScene: called when the transition begins.

    [super onEnter];
}

-(void) onEnterTransitionDidFinish
{
    // Called right after onEnter.
    // If using a CCTransitionScene: called when the transition has ended.

    [super onEnterTransitionDidFinish];
}

-(void) onExit
{
    // Called right before node's dealloc method is called.
    // If using a CCTransitionScene: called when the transition has ended.

    [super onExit];
}
```

NOTE: If you don't make the call to the super implementation in the onEnter methods, your new scene may not react to touch or accelerometer input. If you don't call super in onExit, the current scene may not be released from memory. Since it's easy to forget this and the resulting behavior doesn't lead you to realize that it may be related to these methods, it's important to stress this point. You can see this behavior in the ScenesAndLayer01_WithBugs project.

These methods are useful whenever you need to do something in any node (CCNode, CCLayer, CCScene, CCSprite, CCLabelTTF, and so on) right before a scene is changed or right after. The difference from simply writing the same code in a node's init or dealloc method is that the scene is already fully set up during onEnter, and it still contains all nodes during onExit.

This can be important. For example, if you perform a transition to change scenes, you may want to pause certain animations or hide user interface elements until the transition finishes. Here's the sequence in which these methods get called, based on the logging information from the ScenesAndLayers02 project:

1. scene: OtherScene
2. init: <OtherScene = 066B2130 | Tag = -1>
3. onEnter: <OtherScene = 066B2130 | Tag = -1>

4. `// Transition is running here`
5. `onExit: <FirstScene = 0668DF40 | Tag = -1>`
6. `onEnterTransitionDidFinish: <OtherScene = 066B2130 | Tag = -1>`
7. `dealloc: <FirstScene = 0668DF40 | Tag = -1>`

At first, `OtherScene's +(id) scene` method is called to initialize the `CCScene` and the `CCLayer` it contains. The `OtherScene CCLayer's init` method is then called, directly followed by the `onEnter` method in line 3. In line 4, the transition is sliding the new scene in, and when it's done, the `FirstScene onExit` method gets called, followed by `onEnterTransitionDidFinish` in `OtherScene`.

TIP: The numbers following the class name, like `FirstScene = 0668DF40`, are the memory address where this object is stored in memory. If you have multiple objects of the same class, they will log the same class name, but their memory addresses will be different. This and the possibly unique tag values can help you identify objects of the same class and assist in debugging. If you have little experience in debugging code, I strongly suggest you read the section on debugging applications in Apple's *iOS Development Guide*: developer.apple.com/library/ios/#documentation/Xcode/Conceptual/iphone_development/130-Debugging_Applications/debugging_applications.html.

Debugging an application is not as hard as it may sound. You don't need to know assembler or machine code, and you don't need to be a hacker or expert coder. The debugger is an invaluable tool for every developer to pinpoint issues, crashes in particular. A great number of users seeking for help on the Internet because their application is crashing could have easily solved their problem if they knew just the debugging basics: setting breakpoints, stepping through code, and inspecting the values of variables. That covers about 80 percent of the time spent with the debugger.

Note that the `FirstScene dealloc` method is called last. This means that during `onEnterTransitionDidFinish`, the previous scene is still in memory. If you want to allocate memory-intensive nodes at this point, you'll have to schedule a selector to wait at least one frame before doing the memory allocations, to be certain that the previous scene's memory is released. You can schedule a method that is guaranteed to be called the next frame by omitting the interval parameter when scheduling the method:

```
[self schedule:@selector(waitOneFrame:)];
```

The method that will be called then needs to unschedule itself by calling `unschedule` with the `hidden_cmd` parameter:

```
-(void) waitOneFrame:(ccTime)delta
{
    [self unschedule:_cmd];

    // delayed code goes here ...
}
```

Another strategy would be to release as much memory as possible in the previous scene's `onExit` method. However, this works only if the memory was allocated by you. For example, if your `init` method allocates an `NSMutableArray` instance variable like so:

```
myArray = [[NSMutableArray alloc] init];
```

you will eventually have to call `release` on `myArray` to free its memory. Normally you would do so in the `dealloc` method, but you can also release the memory early in the `onExit` method:

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

    [myArray release];
    myArray = nil;
}
```

Notice that I set `myArray` to `nil`. It's still possible that `myArray` might be used after `onExit` and before the node is deallocated. Setting instance variables to `nil` when they're being released is good practice to prevent a potential crash. Since the memory address `myArray` points to has been freed, any code that tries to send a message to the now-released `myArray` would cause an `EXC_BAD_ACCESS` error.

Loading Next Paragraph, Please Stand By

Sooner or later you'll face noticeable loading times during scene transitions. As you add more content, loading times will correspondingly increase. Creating a new scene actually happens before the scene transition starts. If you have very complex code or load a lot of assets in the new scene's `init` or `onEnter` methods, there will be an obvious delay before the transition begins. This is especially problematic if the new scene takes more than one second to load and the user initiated the scene change by clicking a button. The user may get the impression that the game has locked up or frozen. The way to alleviate this problem is to add another scene in between: a loading scene. You'll find an example implementation in the `ScenesAndLayers03` project.

In effect, the `LoadingScene` acts as an intermediate scene. It is derived from the `cocos2d` `CCScene` class. You don't have to create a new `LoadingScene` for each transition; you can use one scene for which you simply specify the target scene you'd like to be loaded. An enum works best for this; it's defined in the `LoadingScene` header file shown in Listing 5-2.

Listing 5-2. *LoadingScene.h*

```
typedef enum
{
    TargetSceneINVALID = 0,
    TargetSceneFirstScene,
    TargetSceneOtherScene,
    TargetSceneMAX,
} TargetScenes;
```

```
// LoadingScene is derived directly from Scene. We don't need a CCLayer for this scene.
@interface LoadingScene : CCScene
```

```

{
    TargetScenes targetScene_;
}

+(id) sceneWithTargetScene:(TargetScenes)targetScene;
-(id) initWithTargetScene:(TargetScenes)targetScene;

```

TIP: It is good practice to set the first enum value to be an INVALID value, unless you intend to make the first the default. Variables in Objective-C are initialized to 0 automatically unless you specify a different value.

In addition, you can also add a MAX or NUM entry at the end of the enum if you intend to iterate over every enum value, as in:

```
for (int i = TargetSceneINVALID + 1; i < TargetScenesMAX; i++) { .. }
```

In the case of the LoadingScene, it's not necessary, but I tend to add these entries merely out of habit, even if I don't need them.

This brings me to the LoadingScene class implementation of the ScenesAndLayers03 project in Listing 5-3. You'll notice that the scene is initialized differently and that it uses `scheduleUpdate` to delay replacing the LoadingScene with the actual target scene.

Listing 5-3. *The LoadingScene Class Uses a Switch to Decide Whether to Load the Target Scene*

```

+(id) sceneWithTargetScene:(TargetScenes)targetScene;
{
    // This creates an autorelease object of the current class
    return [[[self alloc] initWithTargetScene:targetScene] autorelease];
}

-(id) initWithTargetScene:(TargetScenes)targetScene
{
    if ((self = [super init]))
    {
        targetScene_ = targetScene;

        CCLabelTTF* label = [CCLabelTTF labelWithString:@"Loading ..."
                                                         fontName:@"Marker Felt"
                                                         fontSize:64];

        CGSize size = [[CCDirector sharedDirector] winSize];
        label.position = CGPointMake(size.width / 2, size.height / 2);
        [self addChild:label];

        // Must wait one frame before loading the target scene!
        [self scheduleUpdate];
    }

    return self;
}

-(void) update:(ccTime)delta
{

```

```

[self unscheduleAllSelectors];

// Decide which scene to load based on the TargetScenes enum.
switch (targetScene_)
{
    case TargetSceneFirstScene:
        [[CCDirector sharedDirector] replaceScene:[FirstScene scene]];
        break;
    case TargetSceneOtherScene:
        [[CCDirector sharedDirector] replaceScene:[OtherScene scene]];
        break;

    default:
        // Always warn if an unspecified enum value was used
        NSAssert2(nil, @"%@: unsupported TargetScene %i", ⚡,
            NSStringFromSelector(_cmd), targetScene_);
        break;
}
}

```

Because the `LoadingScene` is derived from `CCScene` and requires a new parameter passed to it, it's no longer sufficient to call `[CCScene node]`. The `sceneWithTargetScene` method first allocates `self`, calls the `initWithTargetScene` method, and returns the new object as autoreleased. This is the same way `cocos2d` initializes its own classes, and it's the reason you can rely on `cocos2d` objects being autoreleased. If you're deriving your own classes, you should always add the appropriate static autorelease initializers, like in this case `sceneWithTargetScene`.

The `init` method simply stores the target scene in a member variable, creates the `Loading` `label`, and runs `scheduleUpdate`.

CAUTION: Why not just call `replaceScene` right inside the `init` method? There are two rules about that. Rule number 1 is never call `CCDirector`'s `replaceScene` in a node's `init` method. Rule number 2 is follow rule number 1. The reason: it crashes. The `Director` can't cope with replacing a scene from a node that is currently being initialized.

The `update` method then uses a simple `switch` statement based on the provided `TargetScenes` enum to determine which scene is to be replaced. The default `switch` contains an `NSAssert`, which always triggers when the default case is hit. This is good practice because you'll be editing and expanding this list several times, and if you forgot to update the `switch` statement with a new case, you'll be notified of that.

This is a very simple `LoadingScene` implementation that you can use in your own games. Simply extend the enum and `switch` statement with more target scenes, or use the same target scene multiple times but with different transitions. But as I mentioned, don't overdo the transitions just because they're cool-looking.

Using the `LoadingScene` has an important effect regarding memory. Since you're replacing the existing scene with the lightweight `LoadingScene` and then replacing the `LoadingScene` with the actual target scene, you're giving `cocos2d` enough time to free up

the previous scene's memory. Effectively there's no longer any overlap with two complex scenes in memory at the same time, reducing spikes in memory usage during scene changes.

Working with Multiple Layers

The project `ScenesAndLayers04` illustrates how multiple layers can be used to scroll the contents of a game's objects layer while the content of the user interface layer, where it says "Here be your Game Scores etc" (see Figure 5-1), remains static. You'll learn how multiple layers can cooperate and react only to their own touch input, as well as how to access the various layers from any node.



Figure 5-1. *The ScenesAndLayers04 project. So far, so normal.*

I'll start by putting the `MultiLayerScene` together in the `init` method. If you skim over the code in Listing 5-4, you'll barely notice anything different from what we've done so far.

Listing 5-4. *Initializing the MultiLayerScene*

```
-(id) init
{
    if ((self = [super init]))
    {
        multiLayerSceneInstance = self;

        // The GameLayer will be moved, rotated and scaled independently
        GameLayer* gameLayer = [GameLayer node];
        [self addChild:gameLayer z:1 tag:LayerTagGameLayer];
        gameLayerPosition = gameLayer.position;

        // The UserInterfacelayer remains static and relative to the screen area.
        UserInterfacelayer* uilayer = [UserInterfacelayer node];
        [self addChild:uilayer z:2 tag:LayerTagUserInterfacelayer];
    }

    return self;
}
```

TIP: It's worth mentioning that I've started using enum values for tags, like `LayerTagGameLayer`. This has the advantage over using numbers in that you can actually read whose tag it is, instead of having to remember which layer had tag 7 assigned to it. It also shows that the actual tag values are not important; what's important is that you use the same value consistently for the same node. Using a human-readable tag makes that task easier and less error prone. The same goes for action tags, of course.

You may have noticed the variable `multiLayerSceneInstance` and that it gets `self` assigned. A bit strange, isn't it? What would that be good for? If you recall from Chapter 3, I explained how to create a singleton class. In this case, I'll turn the `MultiLayerScene` class into a singleton. See Listing 5-5, and if you want, compare it with Listing 3-1 to spot the differences.

Listing 5-5. *Turning the MultiLayerScene into a Semi-Singleton Object*

```
static MultiLayerScene* multiLayerSceneInstance;

+(MultiLayerScene*) sharedLayer
{
    NSAssert(multiLayerSceneInstance != nil, @"MultiLayerScene not available!");
    return multiLayerSceneInstance;
}

-(void) dealloc
{
    // MultiLayerScene will be deallocated now, you must set it to nil
    multiLayerSceneInstance = nil;

    // don't forget to call "super dealloc"
    [super dealloc];
}
```

Simply put, the `multiLayerSceneInstance` is a static global variable that will hold the current `MultiLayerScene` object during its lifetime. The static keyword denotes that the `multiLayerSceneInstance` variable is accessible only within the implementation file it is defined in. At the same time, it is not an instance variable; it lives outside the scope of any class. That's why it is defined outside any method, and it can be accessed in class methods like `sharedLayer`.

When the layer is deallocated, the `multiLayerSceneInstance` variable is set back to `nil` to avoid crashes, because the `multiLayerSceneInstance` variable would be pointing to an already released object after the `dealloc` method. Remember, variables declared static live on past the deallocation of a class. Using static variables to store pointers to dynamically allocated classes requires great care to ensure that the static variable either points to a valid object or is `nil`.

The reason for this semi-singleton is that you'll be using several layers, each with its own child nodes, but you still need to somehow access the main layer. It's a very comfortable way to give access to the main layer to other layers and nodes of the current scene.

CAUTION: This semi-singleton works only if there is only ever one instance of `MultiLayerScene` allocated at any one time. It also can't be used to initialize `MultiLayerScene`, unlike a regular singleton class.

Access to the `GameLayer` and `UserInterfaceLayer` is granted through property getter methods, for ease of use. The properties are defined in Listing 5–6, which shows the relevant part from `MultiLayerScene.h`.

Listing 5–6. *Property Definitions for Accessing the `GameLayer` and `UserInterfaceLayer`*

```
@property (readonly) GameLayer* gameLayer;
@property (readonly) UserInterfaceLayer* uiLayer;
```

The properties are defined as `readonly`, since we only ever want to retrieve the layers, never set them through the property. Their implementation in Listing 5–7 is a straightforward wrapper to the `getChildByTag` method, but they also perform a safety check just in case, verifying that the retrieved object is of the correct class.

Listing 5–7. *Implementation of the Property Getters*

```
-(GameLayer*) gameLayer
{
    CCNode* layer = [self getChildByTag:LayerTagGameLayer];
    NSAssert([layer isKindOfClass:[GameLayer class]], @"%@: not a GameLayer!", ⚡
             NSStringFromSelector(_cmd));
    return (GameLayer*)layer;
}

-(UserInterfaceLayer*) uiLayer
{
    CCNode* layer = [[MultiLayerScene sharedLayer] getChildByTag:LayerTagUILayer];
    NSAssert([layer isKindOfClass:[UserInterfaceLayer class]], @"%@: not a UILayer!", ⚡
             NSStringFromSelector(_cmd));
    return (UserInterfaceLayer*)layer;
}
```

This makes it easy to access the various layers from any node of the `MultiLayerScene`.

- You can access the `sceneLayer` of `MultiLayerScene`:

```
MultiLayerScene* sceneLayer = [MultiLayerScene sharedLayer];
```
- You can access the other layers through the scene layer:

```
GameLayer* gameLayer = [sceneLayer gameLayer];
UserInterfaceLayer* uiLayer = [sceneLayer uiLayer];
```
- As an alternative, because of the `@property` definition, you can also use the dot accessor:

```
GameLayer* gameLayer = sceneLayer.gameLayer;
UserInterfaceLayer* uiLayer = sceneLayer.uiLayer;
```

The `UserInterfaceLayer` and `GameLayer` classes both handle touch input, but independently. To achieve the correct results, we need to use `TargetedTouchHandlers`,

and by using the priority parameter, we can make sure that the `UserInterfaceLayer` gets to look at a touch event before the `GameLayer`. The `UserInterfaceLayer` uses the `isTouchForMe` method to determine whether it should handle the touch, and it will return YES from the `ccTouchBegan` method if it did handle the touch. This will keep other targeted touch handlers from receiving this touch. Listing 5–8 illustrates the important bits of the touch event code for the `UserInterfaceLayer`.

Listing 5–8. Touch Input Processing Using TargetedTouchDelegate

```
// Register TargetedTouch handler with higher priority than GameLayer
-(void) registerWithTouchDispatcher
{
    [[CCTouchDispatcher sharedDispatcher] addTargetedDelegate:self
                                                priority:-1
                                                swallowsTouches:YES];
}

// Checks if the touch location was in an area that this layer wants to handle as input.
-(bool) isTouchForMe:(CGPoint)touchLocation
{
    CCNode* node = [self getChildByTag:UILayerTagFrameSprite];
    return CGRectContainsPoint([node boundingBox], touchLocation);
}

-(BOOL) ccTouchBegan:(UITouch*)touch withEvent:(UIEvent *)event
{
    CGPoint location = [MultiLayerScene locationFromTouch:touch];
    bool isTouchHandled = [self isTouchForMe:location];
    if (isTouchHandled)
    {
        CCNode* node = [self getChildByTag:UILayerTagFrameSprite];
        NSAssert([node isKindOfClass:[CCSprite class]], @"node is not a CCSprite");

        // Highlight the UI layer's sprite for the duration of the touch
        ((CCSprite*)node).color = ccRED;

        // Access the GameLayer via MultiLayerScene.
        GameLayer* gameLayer = [MultiLayerScene sharedLayer].gameLayer;

        // Run Actions on GameLayer (code removed for clarity)
    }

    return isTouchHandled;
}

-(void) ccTouchEnded:(UITouch*)touch withEvent:(UIEvent *)event
{
    CCNode* node = [self getChildByTag:UILayerTagFrameSprite];
    NSAssert([node isKindOfClass:[CCSprite class]], @"node is not a CCSprite");
    ((CCSprite*)node).color = ccWHITE;
}
```

In `registerWithTouchDispatcher`, the `UserInterfaceLayer` registers itself as a targeted touch handler with a priority of -1. Because `GameLayer` uses the same code but with a priority of 0, the `UserInterfaceLayer` will be the first layer to receive touch input.

In `ccTouchBegan`, the first thing to do is to check whether this touch is of relevance to the `UIInterfaceLayer`. The `isTouchForMe` method implements a simple `CGRect` in `boundingBox` check via `CGRectContainsPoint` to see whether the touch began on the `uiFrame` sprite. There are more useful methods available in `CGGeometry` to test intersection, containing points, or equality. Please refer to Apple's documentation to learn more about the `CGGeometry` methods (<http://developer.apple.com/mac/library/documentation/GraphicsImaging/Reference/CGGeometry/Reference/reference.html>).

If the touch location check determines that the touch is on the sprite, `ccTouchBegan` will return `YES`, signaling that this touch event was used and should not be processed by other layers with a targeted touch delegate of lower priority.

Only if the `isTouchForMe` check fails will the `GameLayer` receive the touch input and use it to scroll itself when the user moves a finger over the screen. You can compare `GameLayer`'s input handling code in Listing 5–9.

Listing 5–9. *GameLayer Receives the Remaining Touch Events and Uses Them to Scroll Itself*

```
-(void) registerWithTouchDispatcher
{
    [[CCTouchDispatcher sharedDispatcher] addTargetedDelegate:self
                                                priority:0
                                                swallowsTouches:YES];
}

-(BOOL) ccTouchBegan:(UITouch*)touch withEvent:(UIEvent *)event
{
    lastTouchLocation = [MultiLayerScene locationFromTouch:touch];

    // Stop the move action so it doesn't interfere with the user's scrolling.
    [self stopActionByTag:ActionTagGameLayerMovesBack];

    // Always swallow touches, GameLayer is the last layer to receive touches.
    return YES;
}

-(void) ccTouchMoved:(UITouch*)touch withEvent:(UIEvent *)event
{
    CGPoint currentTouchLocation = [MultiLayerScene locationFromTouch:touch];

    // Take the difference of the current to the last touch location.
    CGPoint moveTo = ccpSub(lastTouchLocation, currentTouchLocation);

    // Then reverse to give the impression of moving the background
    moveTo = ccpMult(moveTo, -1);

    lastTouchLocation = currentTouchLocation;

    // Adjust the layer's position accordingly, and with it all child nodes.
    self.position = ccpAdd(self.position, moveTo);
}

-(void) ccTouchEnded:(UITouch*)touch withEvent:(UIEvent *)event
{
    // Move the game layer back to its designated position.
    CCMoveTo* move = [CCMoveTo actionWithDuration:1 position:gameLayerPosition];
```

```

    CCEaseIn* ease = [CCEaseIn actionWithAction:move rate:0.5f];
    ease.tag = ActionTagGameLayerMovesBack;
    [self runAction:ease];
}

```

Since `GameLayer` is the last layer to receive input, it doesn't need to do any `isTouchForMe` checks and simply swallows all touches.

Using the `ccTouchMoved` event, the difference between the previous and current touch location is calculated. It is then reversed by multiplying it by `-1` to change the effect from moving the camera over the background to moving the background under the camera. If you have a hard time imagining what I mean by this, try the `ScenesAndLayers04` project, and then try it a second time, commenting out the `moveTo = ccpMult(moveTo, -1);` line. You'll notice the second time that every finger movement moves the layer in the opposite direction.

In `ccTouchEnded`, the layer is then simply moved back to its center position automatically when the user lifts the finger off the screen. Figure 5–2 shows this project in action with the whole `GameLayer` rotated and zoomed out. Every game object on the `GameLayer` abides by every movement, rotation, and scaling of the `GameLayer` automatically, whereas the `UserInterfaceLayer` always stays put.



Figure 5–2. *The utility of multiple layers becomes clear once the `GameLayer` is zoomed out and rotated, with all its nodes adhering to the layer's behavior, while the `UserInterfaceLayer` on top remains in place unaffected.*

How to Best Implement Levels

So far we've examined multiple scenes and multiple layers. Now you want to cover what, levels?

Well, the concept of levels is very common in many games, so I don't think I need to explain that. What's much harder is deciding which approach best serves a level-based game. In `cocos2d` you can go either way, choosing a new scene for each level or using separate layers to manage multiple levels. Both have their uses, and which one to choose depends mostly on what purpose levels serve in your game.

Scenes as Levels

The most straightforward approach is to run each level as a separate scene. You can either create a new Scene class for each level or choose to initialize one common LevelScene class and pass as a parameter the level number or other information necessary to load the correct level data.

This approach works best if you have clearly separated levels and most everything that happens within a level is no longer relevant or needed after the player has progressed through that level. Maybe you'll keep the player's score and the number of lives left, but that's about it. The user interface is probably minimal and noninteractive and likely is purely informative without any interactive elements other than a pause button.

I imagine this approach works best for twitch-based action game levels.

Layers as Levels

Using separate layers in the same scene to load and display levels is an approach that's recommended if you have a complex user interface that should not be reset when a level changes. You might even want to keep the player and other game objects in the exact same positions and states when changing levels.

You'll probably have a number of variables that keep the current game state and user interface settings, such as an inventory. It would be more work to save and restore these game settings and reset all visual elements than to switch out one layer with another within the same scene.

This may be the ideal solution for a hidden object or adventure game, where you move from room to room, especially if you want to replace the level contents using an animation that moves in or out beneath the user interface.

The CCMultiplexLayer class may be the ideal solution for such an approach. It can contain multiple nodes, but only one will be active at any given time. Listing 5–10 shows an example of using the CCMultiplexLayer class. The only drawback is that you can't transition between the layers. There's only one layer visible at a time, which makes any transition effects impossible.

Listing 5–10. *Using the CCMultiplexLayer Class to Switch Between Multiple Layers*

```
CCLayer* layer1 = [CCLayer node];
CCLayer* layer2 = [CCLayer node];
CCMultiplexLayer* mplayer = [CCMultiplexLayer layerWithLayers:layer1, layer2, nil];

// Switches to layer2 but keeps layer1 as child of mplayer.
[mplayer switchTo:1];

// Switches to layer1, removes layer2 from mplayer and releases its memory.
// After this call you must not switch back to layer2 (index: 1) anymore!
[mplayer switchToAndReleaseMe:0];
```

CCLayerColor

In the ScenesAndLayers project, so far the background is simply a black screen. You can see it when you scroll to the edge of the grassy background image or tap the `user` interface to have the GameLayer zoom out. To change the background color, cocos2d provides a `CCLayerColor`, which is added to the ScenesAndLayers05 project and works like this:

```
// Set background color to magenta. The most unobtrusive color imaginable.
CCLayerColor* colorLayer = [CCLayerColor layerWithColor:ccc4(255, 0, 255, 255)];
[self addChild:colorLayer z:0];
```

If you're somewhat familiar with OpenGL, it may seem like overkill to add a separate layer just to change the background color. You can achieve the same effect using OpenGL, like this:

```
glClearColor(1, 0, 1, 1);
```

But please test this with your game's scene transitions first, since changing the `glClearColor` can have an adverse effect on scene transitions. For example, when using `CCTransitionFade`, the clear color will shine through regardless of the color you use to initialize `CCTransitionFade`.

Subclassing Game Objects from CCSprite

Very often your game objects will implement logic of their own. It makes sense to create a separate class for each type of game object. This could be your player character, various enemy types, bullets, missiles, platforms, and about everything else that can be individually placed in a game's scene and needs to run logic of its own.

The question then is, where to subclass from?

A lot of developers choose the seemingly obvious route of subclassing `CCSprite`. I don't think that's a good idea. The relationship of subclassing is a `is a` relationship. Think closely, is your player character a `CCSprite`? Are all of your enemy characters `CCSprites`?

At first the answer seems logical: of course they are sprites! That's what they use to display themselves. But wait a second. Could they be something else other than `CCSprite`? For all I know, game characters can also be characters in the literal sense. In Rogue-like games, your player character is an @. So, would that character be a `CCLabelTTF` then?

I think the confusion comes from `CCSprite` being the most widely used class to display anything on-screen. But the true relationship of your game characters to `CCNode` classes is a `has a` relationship. Your player class `has a` `CCSprite` that it uses to display itself. In a Rogue-like game, the player character class `has a` `CCLabelTTF` to display itself. And if you want to get fancy, as in OpenGL and lots of particle effects, your player class `has a` system of particle effects to represent it visually on-screen.

The distinction becomes even clearer when you think of why you'd normally subclass the `CCSprite` class: in general, to add new features to the `CCSprite` class—for example, to have a `CCSprite` class that uses a `CCRenderTexture` to modify how it is displayed based on what is beneath it on the screen.

But what kind of code do you add to the `CCSprite` class when you would subclass it to be your player object? Input handling, animating the player, collision detection, physics; in general: game logic. None of these things belong to a `CCSprite` class because this code is about how an object behaves and how you interact with it, not how a sprite is drawn on the screen. Another way to look at this is to consider which of the code you add to a subclassed `CCSprite` is generally useful and usable by all sprites. For example, the code that handles user input to control the player character is useful and usable only by the player object and not all sprites in general.

You may still be wondering why this has any relevance, or why you should care? Consider the case where you want your player to have several visual representations that it should be able to switch to seamlessly. If the player needs to morph from one sprite to another using `FadeIn/FadeOut` actions, you're going to have to use two sprites. Or if you want your game objects to appear on different parts of the screen at the same time, like in a game like *Asteroids* where the asteroid leaving the top of the screen should also show up partially at the bottom of the screen. You need two sprites to do this, and that's just one reason why composition (or aggregation) is preferable to subclassing (or inheritance). Inheritance causes tight coupling between classes, with changes to parent classes potentially introducing bugs and anomalies in subclasses. The deeper the class hierarchy, the more code will reside in base classes, which amplifies the problem.

Another good reason is that a game object encapsulates its visual representation. If the logic is self-contained, only the game object itself should ever change any of the `CCNode` properties, such as position, scale, rotation, or even running and stopping actions. One of the core problems many game developers face sooner or later is that their game objects are directly manipulated by outside influences. For example, you may inadvertently create situations where the scene's layer, the user interface, and the player object itself all change the player sprite's position. This is undesirable. You want all of the other systems to notify the player object to change its position, giving the player object itself the final say about how to interpret these commands, whether to apply them, modify them, or ignore them.

Composing Game Objects Using `CCSprite`

Let's try this. Instead of creating a new class derived from a `CCSprite` class, create it from cocos2d's base class `CCNode`. Although you can also base your class on the Objective-C base class `NSObject`, it's easier to stick to `CCNode` because it is generally easier to work with `CCNode` as your base class.

I decided to turn the spiders in the `ScenesAndLayers05` project into a class of their own. The class is simply called `Spider`. Listing 5-11 reveals the `Spider` class's header file.

Listing 5–11. The Spider Class Interface

```
#import "cocos2d.h"

@interface Spider : CCNode
{
    CCSprite* spiderSprite;
    int numUpdates;
}

+(id) spiderWithParentNode:(CCNode*)parentNode;
-(id) initWithParentNode:(CCNode*)parentNode;

@end
```

You can see that the `CCSprite` is added as a member variable to the class and is named `spiderSprite`. This is called *composition* since you compose the `Spider` class of a `CCSprite` used to display it and later possibly other objects (including additional `CCSprite` classes) and variables.

The `Spider` class in Listing 5–12 has a static autorelease initializer like any `CCNode` class. This mimics `cocos2d`'s memory management scheme, and it is good practice to follow that. Another convenient feature I added is to pass a `parentNode` as a parameter to the `initWithParentNode` method. By doing that, the `Spider` class can add itself to the node hierarchy by calling `[parentNode addChild:self]`, which makes creating an instance of a `Spider` class a one-liner in `GameLayer.m` because you only need to write `[Spider spiderWithParentNode:self]`.

On the other hand, the `spiderSprite` should be self-contained because it is created and managed by the `Spider` class. The `spiderSprite` is added via `[self addChild:spiderSprite]` to the `Spider` class and not the `parentNode`. Although you could do that, it is not recommended because it breaks encapsulation. For one, the `parentNode` code could possibly remove the `spiderSprite` from its hierarchy, and moving the `Spider` object would no longer move the `spiderSprite`.

Listing 5–12. The Spider Class Implementation

```
#import "Spider.h"

@implementation Spider

// Static autorelease initializer, mimics cocos2d's memory allocation scheme.
+(id) spiderWithParentNode:(CCNode*)parentNode
{
    return [[[self alloc] initWithParentNode:parentNode] autorelease];
}

-(id) initWithParentNode:(CCNode*)parentNode
{
    if ((self = [super init]))
    {
        [parentNode addChild:self];

        CGSize screenSize = [[CCDirector sharedDirector] winSize];

        spiderSprite = [CCSprite spriteWithFile:@"spider.png"];
    }
}
```

```

        spiderSprite.position = CGPointMake(CCRANDOM_0_1() * screenSize.width,
        CCRANDOM_0_1() * screenSize.height);
        [self addChild:spiderSprite];

        [self scheduleUpdate];
    }

    return self;
}

-(void) update:(ccTime)delta
{
    numUpdates++;
    if (numUpdates > 50)
    {
        numUpdates = 0;
        [spiderSprite stopAllActions];

        // Let the Spider move randomly.
        CGPoint moveTo = CGPointMake(CCRANDOM_0_1() * 200 - 100,
        CCRANDOM_0_1() * 100 - 50);
        CCMoveBy* move = [CCMoveBy actionWithDuration:1 position:moveTo];
        [spiderSprite runAction:move];
    }
}

@end

```

The Spider class now uses its own game logic to move the spiders around on the screen. Granted, it won't win any prizes at this year's Artificial Intelligence Symposium; it's just meant as an example.

Up to this point you know that you can receive touch input using CCLayer nodes, but in fact any class can receive touch input by using the CCTouchDispatcher class directly. Your class only needs to implement either the CCStandardTouchDelegate or CCTargetedTouchDelegate protocol. You'll find these changes in the ScenesAndLayers06 project, and the appropriate protocol definition added to the header file is shown in Listing 5–13.

Listing 5–13. The CCTargetedTouchDelegate Protocol

```

@interface Spider : CCNode <CCTargetedTouchDelegate>
{
}

```

The implementation in Listing 5–14 highlights the changes made to the Spider class. The Spider class now reacts to targeted touch input. Whenever you tap a spider, it will quickly move away from your finger. Contrary to common perceptions, spiders are usually more afraid of humans than the other way around, exceptions notwithstanding.

The Spider class is registered with CCTouchDispatcher to receive input as a touch delegate, but this delegate association must also be removed on dealloc. Otherwise, the scheduler or touch dispatcher would still keep a reference to the Spider class even

though it was released from memory, and that would likely cause a crash shortly thereafter.

Listing 5–14. The Changed Spider Class

```

-(id) initWithParentNode:(CCNode*)parentNode
{
    if ((self = [super init]))
    {

        // Manually add this class as receiver of targeted touch events.
        [[CCTouchDispatcher sharedDispatcher] addTargetedDelegate:self
                                                    priority:-1
                                                    swallowsTouches:YES];

    }

    return self;
}

-(void) dealloc
{
    // Must manually remove this class as touch input receiver!
    [[CCTouchDispatcher sharedDispatcher] removeDelegate:self];

    [super dealloc];
}

// Extract common logic into a separate method accepting parameters.
-(void) moveAway:(float)duration position:(CGPoint)moveTo
{
    [spiderSprite stopAllActions];
    CCMoveBy* move = [CCMoveBy actionWithDuration:duration position:moveTo];
    [spiderSprite runAction:move];
}

-(void) update:(ccTime)delta
{
    numUpdates++;
    if (numUpdates > 50)
    {
        numUpdates = 0;

        // Move at regular speed.
        CGPoint moveTo = CGPointMake(CCRANDOM_0_1() * 200 - 100, ⤵
                                     CCRANDOM_0_1() * 100 - 50);
        [self moveAway:2 position:moveTo];
    }
}

-(BOOL) ccTouchBegan:(UITouch *)touch withEvent:(UIEvent *)event
{
    // Check if this touch is on the Spider's sprite.
    CGPoint touchLocation = [MultiLayerScene locationFromTouch:touch];
    BOOL isTouchHandled = CGRectContainsPoint([spiderSprite boundingBox], ⤵
        touchLocation);
    if (isTouchHandled)
    {

```



```

    // Reset move counter.
    numUpdates = 0;

    // Move away from touch location rapidly.
    CGPoint moveTo;
    float moveDistance = 60;
    float rand = CCRANDOM_0_1();

    // Randomly pick one of four corners to move away to.
    if (rand < 0.25f)
        moveTo = CGPointMake(moveDistance, moveDistance);
    else if (rand < 0.5f)
        moveTo = CGPointMake(-moveDistance, moveDistance);
    else if (rand < 0.75f)
        moveTo = CGPointMake(moveDistance, -moveDistance);
    else
        moveTo = CGPointMake(-moveDistance, -moveDistance);

    // Move quickly:
    [self moveAway:0.1f position:moveTo];
}

return isTouchHandled;
}

```

I decided to improve the move logic by extracting the functionality into a separate method. The straightforward way would have been to just copy the existing code from the update method to the `ccTouchBegan` method. However, copy-and-paste is evil. If you join the Cult of Programmology, be aware that it is considered a deadly sin to duplicate existing code.

CAUTION: Using copy-and-paste is very easy, and everyone knows how to do it, which makes it so tempting. But whenever you duplicate code, you duplicate the effort needed to change that code later. Consider the case where you duplicate the same sequence of actions ten times and you need to change the duration of the action sequence. You can't change it just once; you have to change it ten times now, and more importantly you have to test the change ten times because you might have forgotten to apply the change to one of the ten places. More code also means more chances of introducing a bug, and since this bug will possibly happen in only one of ten cases, it'll be harder to find as well.

Using methods to extract common functionality, and exposing what needs to be flexible as parameters, is indeed a very simple task. I hope that the `moveAway` method in Listing 5–14 illustrates my point well. It does not contain much code, but even the smallest amount of duplicated code increases your time spent on maintaining your code.

The `ccTouchBegan` method takes the touch location and checks via the `CGRectContainsPoint` if the touch location is inside the spider sprite's `boundingBox`. If so, it handles the touch and runs the code that lets the spider move away quickly in one of four directions.

In summary, using a `CCNode` as a base class for your game objects makes a few things more inconvenient at first glance. The benefits do become visible when you start creating larger projects, however, like when you are dealing with more than a dozen of classes for game objects. It is OK if you prefer to subclass `CCSprite` for now. But when you get more proficient and more ambitious, please come back to this chapter again and try this approach. It leads to a better code structure and more clearly separated boundaries and responsibilities of the individual game elements.

Curiously Cool `CCNode` Classes

Please remain seated as I walk you through a few more `CCNode`-derived classes that fulfill very specific purposes. They are `CCProgressTimer`, `CCParallaxNode`, `CCRibbon`, and `CCMotionStreak`.

`CCProgressTimer`

In the `ScenesAndLayers07` project, I've added a `CCProgressTimer` node to the `UserInterfaceLayer` class. You can see how it cuts off chunks from a sprite in a radial fashion in Figure 5-3.



Figure 5-3. *The `CCProgressTimer` in action. I would say it's about 10 past 12.*

The progress timer class is useful for any kind of progress display, like a loading bar or the time it takes an icon to become available again. Think of the action buttons in *World of Warcraft* and their recast timer. The progress timer takes a sprite and, based on a percentage, displays only part of it to visualize some kind of progress in your game. See Listing 5-15 for how to initialize the `CCProgressTimer` node.

Listing 5-15. *Initializing a `CCProgressTimer` Node*

```
// Progress timer is a sprite that is only partially displayed
// to visualize some kind of progress.
CCProgressTimer* timer = [CCProgressTimer progressWithFile:@"firething.png"];
timer.type = kCCProgressTimerTypeRadialCCW;
timer.percentage = 0;
[self addChild:timer z:1 tag:UILayerTagProgressTimer];
```

```
// The update is needed for the progress timer.
[self scheduleUpdate];
```

The timer type is from the `CCProgressTimerType` enum defined in `CCProgressTimer.h`. You can choose between radial, vertical, and horizontal progress timers. But there's one caveat: the timer doesn't update itself. You have to change the timer's percentage value frequently to update the progress. That's why I included the `scheduleUpdate` in Listing 5-15. The implementation of the update method that does the actual progressing is shown in Listing 5-16. The `CCProgressTimer` node's percentage property must be frequently updated as needed—it won't progress by itself automatically. The progress here is simply the passing of time. Isn't that what games are all about?

Listing 5-16. The Implementation of the update Method

```
-(void) update:(ccTime)delta
{
    CCNode* node = [self getChildByTag:UILayerTagProgressTimer];
    NSAssert([node isKindOfClass:[CCProgressTimer class]], @"node is not a ~
        CCProgressTimer");

    // Updates the progress timer
    CCProgressTimer* timer = (CCProgressTimer*)node;
    timer.percentage += delta * 10;
    if (timer.percentage >= 100)
    {
        timer.percentage = 0;
    }
}
```

CCParallaxNode

Parallaxing is an effect used in 2D games to give the impression of depth, created by using layered images that move at different rates. The images in the foreground move faster relative to the images in the background. Although you can't see the parallax effect in a static image, Figure 5-4 at least gives you a first impression of a background made up of individual layers. The clouds are at the very back; the mountains are in front of the clouds but behind the (ugly) trees. At the very bottom is the road where you might typically add a player character moving left and right along this scenery.

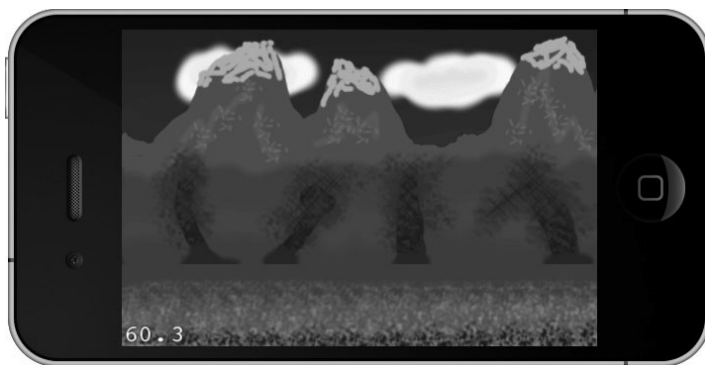


Figure 5–4. *The CCParallaxNode allows you to create an illusion of depth.*

NOTE: Why does parallaxing create the illusion of depth? Because our minds are trained to this effect. Think of traveling in a car at high speed, and you're looking out of a side window. You'll notice the trees next to the road (closest to you) zipping by so fast you can hardly focus on a single one. Look a little further, and you'll see the barnyard passing you by at a seemingly much slower rate. Then look to the mountains at the horizon, and you'll hardly notice that you are moving past them at all. This is the parallax effect in a three-dimensional world, where there are an infinite number of parallax layers. In a 2D game we have to (very roughly) simulate the same effect with around two to eight parallax layers. Each layer tries to fool your mind into thinking that it is a certain distance away from your viewpoint, simply because it is moving at a certain speeds relative to other layers. It works surprisingly well.

Cocos2d has a specialized node you can use to create this effect. The code to create a CCParallaxNode in Listing 5–17 is also in the ScenesAndLayers08 project.

Listing 5–17. *The CCParallaxNode Requires a Lot of Setup Work, but the Results Are Worth It*

```
// Load the sprites for each parallax layer, from background to foreground.
CCSprite* para1 = [CCSprite spriteWithFile:@"parallax1.png"];
CCSprite* para2 = [CCSprite spriteWithFile:@"parallax2.png"];
CCSprite* para3 = [CCSprite spriteWithFile:@"parallax3.png"];
CCSprite* para4 = [CCSprite spriteWithFile:@"parallax4.png"];

// Set the correct offsets depending on the screen and image sizes.
para1.anchorPoint = CGPointMake(0, 1);
para2.anchorPoint = CGPointMake(0, 1);
para3.anchorPoint = CGPointMake(0, 0.6f);
para4.anchorPoint = CGPointMake(0, 0);
CGPoint topOffset = CGPointMake(0, screenSize.height);
CGPoint midOffset = CGPointMake(0, screenSize.height / 2);
CGPoint downOffset = CGPointZero;
```

```
// Create a parallax node and add the sprites to it.
CCParallaxNode* paraNode = [CCParallaxNode node];
[paraNode addChild:para1
                 z:1
                 parallaxRatio:CGPointMake(0.5f, 0)
                 positionOffset:topOffset];

[paraNode addChild:para2 z:2 parallaxRatio:CGPointMake(1, 0) positionOffset:topOffset];
[paraNode addChild:para3 z:4 parallaxRatio:CGPointMake(2, 0) positionOffset:midOffset];
[paraNode addChild:para4 z:3 parallaxRatio:CGPointMake(3, 0) positionOffset:downOffset];
[self addChild:paraNode z:0 tag:ParallaxSceneTagParallaxNode];

// Move the parallax node to show the parallaxing effect.
CCMoveBy* move1 = [CCMoveBy actionWithDuration:5 position:CGPointMake(-160, 0)];
CCMoveBy* move2 = [CCMoveBy actionWithDuration:15 position:CGPointMake(160, 0)];
CCSequence* sequence = [CCSequence actions:move1, move2, nil];
CCRepeatForever* repeat = [CCRepeatForever actionWithAction:sequence];
[paraNode runAction:repeat];
```

To create a `CCParallaxNode`, you first create the desired `CCSprite` nodes that make up the individual parallaxing images, and then you have to properly position them on the screen. In this case, I chose to modify their anchor points instead because it was easier to align the sprites with the screen borders. The `CCParallaxNode` is created like any other node, but its children are added using a special initializer. With it you specify the `parallaxRatio`, which is a `CGPoint` used as a multiplier for any movement of the `CCParallaxNode`. In this case, the `CCSprite` `para1` would move at half the speed, `para2` at normal speed, `para3` at double the speed of the `CCParallaxNode`, and so on.

Using a sequence of `CCMoveBy` actions, the `CCParallaxNode` is moved from left to right and back. You will notice how the clouds in the background move slowest while the trees and gravel in the foreground scroll by the fastest. This gives the illusion of depth.

NOTE: You can't modify the positions of individual child nodes once they are added to the `CCParallaxNode`. You can only scroll as far as the largest and fastest-moving image before the background shows through. You can see this effect if you modify the `CCMoveBy` actions to scroll a lot further. You can increase the scrolling distance by adding more of the same sprites with the appropriate offsets. But if you require endless scrolling in one or both directions, you will have to implement your own parallax system. In fact, this is what we're going to do in Chapter 7.

CCRibbon

The CCRibbon node creates a band of images, like a chain or, as in Figure 5–5, like a millipede crawling over the parallaxing scene in the ScenesAndLayers09 project.



Figure 5–5. *Nasty. A CCRibbon of spiders. Looks like a millipede crawling over the screen.*

The CCRibbon class, together with touch input, can be used to create the line-drawing effects of popular games. Listing 5–18 shows how the CCRibbon is implemented with the touch events. What’s notable is that you can’t remove individual points from a CCRibbon. You can only remove the whole CCRibbon by removing it as child from its parent. The width and length parameters of the CCRibbon initializer determine how big individual ribbon elements are drawn. In this case, I chose to make it as big as the spider.png image, which is 32 pixels wide and high. If you choose other values, the image will be scaled up or down accordingly.

Listing 5–18. *The CCRibbon Class*

```
-(void) resetRibbon
{
    // Removes the ribbon and creates a new one.
    [self removeChildByTag:ParallaxSceneTagRibbon cleanup:YES];
    CCRibbon* ribbon = [CCRibbon ribbonWithWidth:32
                                     image:@"spider.png"
                                     length:32
                                     color:ccc4(255, 255, 255, 255)
                                     fade:0.5f];
    [self addChild:ribbon z:5 tag:ParallaxSceneTagRibbon];
}

-(CCRibbon*) getRibbon
{
    CCNode* node = [self getChildByTag:ParallaxSceneTagRibbon];
    NSAssert([node isKindOfClass:[CCRibbon class]], @"node is not a CCRibbon");
    return (CCRibbon*)node;
}

-(void) addRibbonPoint:(CGPoint)point
```

```

{
    CCRibbon* ribbon = [self getRibbon];
    [ribbon addPointAt:point width:32];
}

-(BOOL) ccTouchBegan:(UITouch*)touch withEvent:(UIEvent *)event
{
    [self addRibbonPoint:[MultiLayerScene locationFromTouch:touch]];
    return YES;
}

-(void) ccTouchMoved:(UITouch*)touch withEvent:(UIEvent *)event
{
    [self addRibbonPoint:[MultiLayerScene locationFromTouch:touch]];
}

-(void) ccTouchEnded:(UITouch*)touch withEvent:(UIEvent *)event
{
    [self resetRibbon];
}

```

CCMotionStreak

CCMotionStreak is essentially a wrapper around CCRibbon. It causes the CCRibbon elements to more or less slowly fade out and disappear after you've drawn them. Try it in the ScenesAndLayers10 project and take a look at Figure 5–6 to get an impression how the fade-out effect might look like.

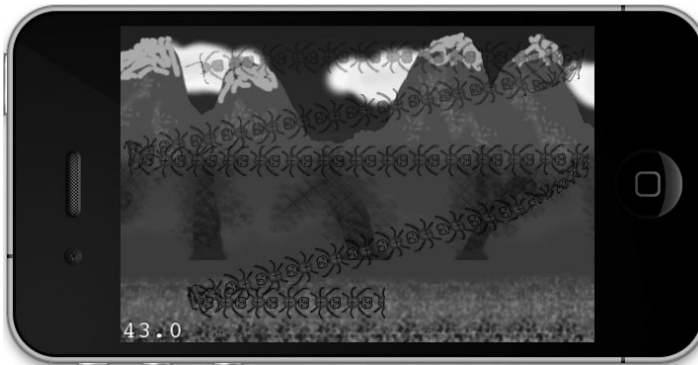


Figure 5–6. The CCMotionStreak class allows you to slowly fade out the CCRibbon elements.

As you can see in Listing 5–19, you use it almost identically to CCRibbon, except that the CCRibbon is now a property of CCMotionStreak. The fade parameter determines how fast ribbon elements fade out; the smaller the number, the quicker they disappear. The minSeg parameter seems to have no discernable effect, although an interesting graphical glitch occurs if you set it to negative values.

Listing 5–19. *The CCMotionStreak Lets the Ribbon's Elements Fade Out, Creating a Streak Effect*

```
-(void) resetMotionStreak
{
    // Removes the CCMotionStreak and creates a new one.
    [self removeChildByTag:ParallaxSceneTagRibbon cleanup:YES];
    CCMotionStreak* streak = [CCMotionStreak streakWithFade:0.7f
                                                minSeg:10
                                                image:@"spider.png"
                                                width:32
                                                length:32
                                                color:ccc4(255, 0, 255, 255)];
    [self addChild:streak z:5 tag:ParallaxSceneTagRibbon];
}

-(void) addMotionStreakPoint:(CGPoint)point
{
    CCMotionStreak* streak = [self getMotionStreak];
    [streak.ribbon addPointAt:point width:32];
}
```

Summary

In this chapter, you learned more about scenes and layers—how and when to use them and for what. I explained why it's usually not a good idea to subclass game objects directly from CCSprite, and I showed you how to create a fully self-contained game object class that derives from CCNode instead.

Finally, you learned how to use specialized CCNode classes like CCProgressTimer, CCParallaxNode, CCRibbon, and CCMotionStreak.

You now have enough knowledge about cocos2d to start creating more complex games, like the side-scrolling shooter I'm preparing you for. And with complex games come complex graphics, including animations. How to handle all of these sprites efficiently both in terms of memory and performance is the topic of the next chapter.

Sprites In-Depth

In this chapter, I'll focus on working with sprites. There are numerous ways to create sprites from individual image files and texture atlases. I will also explain how to create and play sprite animations.

A *texture atlas* is a regular texture that contains more than one image. Often it is used to store all animation frames of a single character in one texture, but it is not limited to that. In fact, you can place any image into a texture atlas. The goal is to get as many images as possible into each texture atlas. To help create a texture atlas, a great tool to rely on is TexturePacker, which I'll also introduce in this chapter.

Sprite batching is a technique for speeding up the drawing of sprites. As the name implies, batching sprites allows the GPU to render all the sprites in one go, or in technical terms, in one draw call. It speeds up drawing identical sprites but is most effective when using a texture atlas. If you use a texture atlas along with sprite batching, you can draw all of the images in that texture atlas in one draw call.

A *draw call* is the process of transmitting the necessary information to the graphics hardware in order to render a texture or parts of it. When you are using CCSprites, each CCSprite will cause one draw call. The CPU overhead for issuing each draw call can add up so much that it can decrease the framerate, particularly with the more sprites you want to display.

CCSpriteBatchNode functions like an extra layer to which you can add sprite nodes, as long as they all use the same texture. From then on, all the children of the CCSpriteBatchNode will be drawn with a single draw call. Effectively, the CPU tells the GPU what texture it should draw from but also passes a long list of frames and positions to the GPU so that it can render a large number of sprites from that texture all by itself.

To summarize, sprite batching speeds up drawing identical sprites using the same texture and is most effective when using a texture atlas.

The lessons you'll learn in this chapter will become the foundation for the parallax-scrolling shoot-'em-up game that I'll be discussing in Chapters 7 and 8.

Retina Display

The newer iPhone models starting with iPhone 4 use the new high-resolution display called the Retina display. It has a resolution of 960×640 pixels, which duplicates the number of pixels in each direction; previous-generation devices had a resolution of 480×320 pixels. To make this distinction, the Retina display graphics are referred to as high-definition (HD) graphics, whereas non-Retina devices are referred to as having standard-definition (SD) graphics.

In Table 6–1 you’ll get a brief overview over the technical specifications of the iOS devices up to and including the fourth generation.

Table 6–1. *The Technical Specifications of iOS Devices*

Device	Processor	Max. Texture Size	Display Resolution	Memory
iPhone classic	PowerVR MBX Lite	1024×1024	480×320	128MB
iPhone 3G	PowerVR MBX Lite	1024×1024	480×320	128MB
iPhone 3GS	PowerVR SGX	2048×2048	480×320	256MB
iPhone 4	Apple A4	2048×2048	960×640	512MB
iPad	Apple A4	2048×2048	1024×768	256MB
iPad 2	Apple A5	2048×2048	1024×768	512MB
iPod touch first generation	PowerVR MBX Lite	1024×1024	480×320	128MB
iPod touch second generation	PowerVR MBX Lite	1024×1024	480×320	128MB

For the devices using PowerVR MBX Lite, there is another important limitation. They can use only 24MB of RAM for graphics. This limit does not exist on newer devices. This limit is specifically important to graphic-intensive applications where you have to balance the usage of texture memory with regular memory required by the app.

Although first- and second-generation devices are no longer produced or sold, there are still millions of people who own and use these devices. Whether you want to support these devices or decide to ignore them is your decision. But it’s one that you should make before you start writing your app because it’s a lot harder to optimize a work in progress or even finished app and make it perform well on less capable hardware. Fortunately, cocos2d helps you accomplish the task of targeting newer and older devices at the same time.

Cocos2d uses a resolution-independent coordinate system using points instead of pixels. One point is exactly one pixel on the SD devices, but one point is two pixels on

Retina display devices. By writing all positions in points, the coordinates will be the same on both devices!

TIP: To set an object to an exact pixel location on Retina devices, the points can be expressed in fractions. For example, the point 100.5,99.5 will set an object to the pixel coordinate 201,200 on a Retina device. However, this also sets the pixel position to 100.5,99.5 on a non-Retina device, which is referred to as *subpixel rendering*. Since the image is not on an exact pixel location, some blending can occur, which can cause objects to not align properly or to leave gaps. It is generally recommended to avoid this kind of situation.

Cocos2d is smart when it comes to loading images. If your code has Retina support enabled and the app is running on a device that has a Retina display, cocos2d will try to load a sprite with the `-hd` suffix first. So if you load a file named `ship.png` on a Retina display device, it will first try to load `ship-hd.png`, and if that file does not exist or the device is not a Retina display device, the SD resolution image `ship.png` will be loaded.

Of course, this makes sense only if all `-hd` images have exactly twice the resolution of the non `-hd` image. Otherwise, you'll notice that `-hd` images that are not exactly twice the resolution will be more or less offset when you display them in your app. In general, you should avoid using `-hd` images whose pixel resolution is not divisible by two without a remainder. If you do support Retina displays, you should create all images in HD resolution and simply scale them down by 50 percent to save them as the SD images. Upscaling SD images will not give you Retina display quality.

The nice thing about cocos2d's HD image support is that your game does not even have to know if it is running on a Retina device. The code will be the same. The only thing you do need to care about is that you need two images instead of one.

In theory, it is also possible to use only HD graphics and then scale them down on the fly using the `scale` property of sprites to match the display resolution. But this comes with a major drawback: memory usage quadruples! The HD image in Figure 6-1 uses 128KB of texture memory, assuming 32-bit color quality. The SD version consumes only 32KB of memory. The non-Retina devices have generally less memory than Retina devices and would quickly run out of memory. Then consider that displaying a scaled-down version of an image incurs a performance penalty because four times more pixels need to be processed each frame to display a scaled-down version of the image.

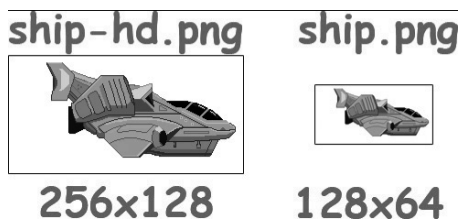


Figure 6-1. Sprites in two resolutions for HD (Retina) and SD (non-Retina) displays

To enable support for Retina display resolutions in cocos2d, you must call the CCDirector method `enableRetinaDisplay`:

```
if (![director enableRetinaDisplay:YES])  
{  
    CCLOG(@"Retina Display Not supported");  
}
```

CAUTION: If you enable Retina display support, you should supply HD images for all your sprites, bitmap fonts, particle effects, and so on. If you don't, the resulting effect will be that your app looks normal on SD devices, but all the visuals that don't have an HD version will be drawn in half the size on Retina displays.

CCSpriteBatchNode

Every time a texture is drawn on the screen, the graphics hardware has to prepare the rendering, render the graphics, and clean up after rendering. There is an inherent overhead caused by starting and ending the rendering of a single texture. This can be alleviated by letting the graphics hardware know that you have a group of sprites that should be rendered using the same texture. In that case, the graphics hardware will perform the preparation and cleanup steps only once for a group of sprites.

Figure 6-2 shows an example of this kind of batch rendering. As you can see, there are hundreds of identical bullets on the screen. If you rendered them each one at a time, your framerate would drop by at least 15 percent in this case. With a `CCSpriteBatchNode`, you can avoid the repeated effort.

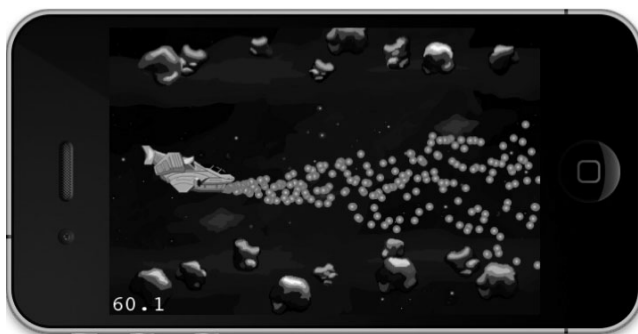


Figure 6-2. Drawing many `CCSprite` nodes using the same texture is more efficient when they are added to a `CCSpriteBatchNode`.

As a refresher, here's how you'd create a `CCSprite` the normal way:

```
CCSprite* sprite = [CCSprite spriteWithFile:@"bullet.png"];  
[self addChild:sprite];
```

Listing 6–1 changes the creation of the same `CCSprite` to use a `CCSpriteBatchNode` instead. Of course, just adding one `CCSprite` to it won't give you any benefit, so I'll be adding a number of sprites using the same texture to the `CCSpriteBatchNode`.

Listing 6–1. *Creating Multiple CCSprites and Adding Them to a CCSpriteBatchNode to Render Them Faster*

```
CCSpriteBatchNode* batch = [CCSpriteBatchNode batchNodeWithFile:@"bullet.png"];
[self addChild:batch];

for (int i = 0; i < 100; i++)
{
    CCSprite* bullet = [CCSprite spriteWithFile:@"bullet.png"];
    [batch addChild:bullet];
}
```

You will notice that in Listing 6–1 the `CCSpriteBatchNode` takes a file as an argument, even though the `CCSpriteBatchNode` itself isn't displayed. It's more like a `CCLayer` in that regard, except that you can add only `CCSprite` nodes to it. The reason it takes an image file as an argument is that all `CCSprite` nodes added to the `CCSpriteBatchNode` must use the same texture. If you make that mistake, you'll see the following error message in the Debugger Console window:

```
SpriteBatches[13879:207] *** Terminating app due to uncaught exception
'NSInternalInconsistencyException', reason: 'CCSprite is not using the same texture id'
```

When to Use CCSpriteBatchNode

`CCSpriteBatchNode` can be used whenever you display two or more `CCSprites` of the same kind. The more `CCSprites` you can group together, the greater the benefit of using `CCSpriteBatchNode` will be.

There are limitations, however. Since all the `CCSprite` nodes are added to the `CCSpriteBatchNode`, all `CCSprite` nodes added to it will be drawn at the same z-order (depth). If your game is supposed to have bullets flying behind and in front of enemies, you would have to use two `CCSpriteBatchNodes` to group the bullet sprites of the lower and the higher z-order independently.

Another drawback is that all `CCSprites` added to the `CCSpriteBatchNode` need to use the same texture. But that also means the `CCSpriteBatchNode` becomes most important when you are using a texture atlas. With a texture atlas, you are not limited to drawing only one image; instead, you can add a number of different images to the same texture atlas and draw all of these images using the same `CCSpriteBatchNode`, speeding up the rendering of all the images of the same texture atlas.

If all your game's images can fit into the same texture atlas, you could compose almost your entire game using just a single `CCSpriteBatchNode` (although this will be the rare exception).

Think of the `CCSpriteBatchNode` as similar to a `CCLayer`, except that it only accepts `CCSprite` nodes using the same texture. With that mind-set, I'm sure you'll find the right places to use the `CCSpriteBatchNode`.

Demo Projects

There are four projects—named Sprites01 through Sprites04—that are intended as a demonstration of using a `CCSpriteBatchNode`. They are the first steps toward the scrolling shoot-'em-up game that I'll discuss in Chapters 7 and 8. Since developers often start by using just the `CCSprite` node and then they improve the code to support `CCSpriteBatchNode`, I found it interesting to show exactly this process and how the code changes along the way.

The projects use two classes, named `Ship` and `Bullet`, both derived from `CCSprite`, to illustrate how a project may be changed from using regular `CCSprite` objects to a `CCSpriteBatchNode`.

A Common and Fatal Mistake

The Sprites01 project shows a common trap developers new to Objective-C can quickly find themselves in. It's easy to make this mistake but hard to figure out the cause and how to fix it. I'll spare you the headache. Take a look at Listing 6–2; do you see what's wrong with this code?

Listing 6–2. *A Commonly Made Fatal Mistake When Subclassing CCSprite (or Other Classes for That Matter)*

```
-(id) init
{
    if ((self = [super initWithFile:@"ship.png"]))
    {
        [self scheduleUpdate];
    }
    return self;
}
```

No, it's not about `scheduleUpdate`; that's just to throw you off guard. The problem lies in the fact that the `-(id) init` method is the default initializer, which is eventually called by any other specialized initializer like `initWithFile`. Can you imagine now what's wrong with the code?

Well, `initWithFile` will eventually call the default initializer, `-(id) init`. Then, since this class's implementation overrides it, it will call `[super initWithFile: ..]` again. Repeat *ad infinitum*.

The solution is very simple. As shown in Listing 6–3, it is sufficient to give the initializer method a different name—something other than `-(id) init`.

Listing 6–3. *Fixing the Infinite Loop Caused by the Code in Listing 6.2*

```
-(id) initWithShipImage
{
    if ((self = [super initWithFile:@"ship.png"]))
    {
        [self scheduleUpdate];
    }
    return self;
}
```

CAUTION: In general, to avoid this problem, you should never call anything but `[super init]` in the default initializer `-(id) init`. If you have to call `[super initWith...]` in your class's initializer method, then you should name the initializer method accordingly, as `-(id) initWith...` (for example, like `-(id) initWithShipImage`, as in Listing 6.3).

Bullets Without a SpriteBatch

The Sprites02 project creates a new `CCSprite` for each `Bullet`. Notice how in Listing 6–4 the ship sprite is adding the bullets to its parent node—it does not add them to itself; otherwise, all the flying bullets would be positioned relative to the ship and mimic the ship's movement.

Listing 6–4. *The Ship Shooting the Bullets*

```
-(void) update:(ccTime)delta
{
    // Keep creating new bullets
    Bullet* bullet = [Bullet bulletWithShip:self];

    // Add the bullets to the ship's parent
    CCNode* gameScene = [self parent];
    [gameScene addChild:bullet z:0 tag:GameSceneNodeTagBullet];
}
```

The `Bullet` sprites are added to the ship's parent for the simple reason that adding them to the ship would make all flying bullets positioned at an offset to the ship. This means that if the ship were to move—which I suppose you will want eventually; otherwise, this would be a really boring game—all the flying bullets would change their positions relative to the ship, as if they were somehow attached to it.

TIP: The bullets are all added with the same z-order, 0. All nodes using the same z-order are drawn in the order they are added to the scene hierarchy. Meaning, the node added last will be drawn in front of all other previously added nodes with the same z-order.

In addition, all bullets use the same tag. Tags do not need to be unique, and sometimes it can be helpful to use a tag to denote group membership of nodes. You could then loop through all children of a node and perform different code based on the node tag.

The bullets also use an update method to update their position and to remove themselves at some point. Although sprites aren't drawn if they are outside the screen, they still consume memory and CPU power, so it's a necessity to remove any stray objects that leave the screen area at some point in time. In this case, you simply check the bullet's position against the right side of the screen, as shown in Listing 6–5.

Listing 6–5. Moving and Removing the Bullets

```
-(void) update:(ccTime)delta
{
    // update position of the bullet
    // multiply the velocity by the time since the last update was called
    // this ensures same bullet velocity even if framerate drops
    self.position = ccpAdd(self.position, ccpMult(velocity, delta));

    // delete the bullet if it leaves the screen
    if (self.position.x > outsideScreen)
    {
        [self removeFromParentAndCleanup:YES];
    }
}
```

The bullet position is updated by multiplying its velocity (speed and direction given by a `CGPoint`) and then adding the result to the bullet's position. The velocity simply determines how many pixels to move in each direction every second. Multiplying the velocity by the time results in the distance the bullet traveled. The reason for using the delta time when updating the position is that this makes the movement of the bullets independent from the framerate. If you don't do that for all moving objects, your game would slow down proportionally with a decreasing framerate, for example when there's a boss fight with lots of sprites on the screen.

Calculating the movement in the aptly named update method `update` is much more effective than using `CCMoveTo` or `CCMoveBy` actions in this case. It avoids some overhead and the problem that actions run for a given duration. If the ship were to move closer to the right side of the screen, the move actions would cause the bullets to move more slowly since they need to travel a shorter distance in the same time.

TIP: You can have the `CCMoveTo` and `CCMoveBy` actions move a node at a fixed speed to any position. To do so, you would first have to calculate the distance between the node's current position to its desired destination by using the `ccpDistance` method. Then divide the distance by the desired speed (in pixels per frame). This works well enough, but the drawback is that `ccpDistance` calls the `sqrtof` (square root) method, which is computationally expensive. You would want to avoid doing this regularly, so for something as simple as updating node positions every frame, it's generally advisable to avoid using move actions for continuously moving nodes.

Introducing the `CCSpriteBatchNode`

In the `Sprites03` project, the `CCSpriteBatchNode` for bullets has been added. I decided to add it to the `GameScene` itself, since bullets are not supposed to be added to the `Ship` class. Since the `Ship` class has no access to the `GameScene`, I also needed to add the pseudosingleton accessor `sharedGameScene` to allow the ship to get to the `CCSpriteBatchNode`, as shown in Listing 6–6.

Listing 6–6. *The GameScene Gets a CCSpriteBatchNode for Bullets and Accessors for the Ship Class*

```

static GameScene* instanceOfGameScene;
+(GameScene*) sharedGameScene
{
    NSAssert(instanceOfGameScene != nil, @" instance not yet initialized!");
    return instanceOfGameScene;
}

-(id) init
{
    if ((self = [super init]))
    {
        instanceOfGameScene = self;

        CCSpriteBatchNode* batch = [CCSpriteBatchNode batchNodeWithFile:@"bullet.png"];
        [self addChild:batch z:1 tag:GameSceneNodeTagBulletSpriteBatch];
    }
    return self;
}

-(void) dealloc
{
    instanceOfGameScene = nil;
    [super dealloc];
}

-(CCSpriteBatchNode*) bulletSpriteBatch
{
    CCNode* node = [self getChildByTag:GameSceneNodeTagBulletSpriteBatch];
    NSAssert([node isKindOfClass:[CCSpriteBatchNode class]], @"not a SpriteBatch");
    return (CCSpriteBatchNode*)node;
}

```

CAUTION: I know this singleton and the additional accessor method `bulletSpriteBatch` may not be to everyone's liking. Why didn't I simply pass the `CCSpriteBatchNode` as a pointer to the `Ship` class, either in the initializer or via a property?

One reason is that `Ship` does not own the bullet sprite batch, and therefore it should not keep a reference to it. Moreover, if the `Ship` class also retains the sprite batch, it could cause your whole scene to not be deallocated if you're not careful. Nodes should never retain other nodes that do not belong to them.

In particular, parent nodes should not be retained by one of their child nodes. For one, a child node never needs to retain its parent node that is already taken care of by cocos2d's node hierarchy. A child node that does retain its parent or any of its grandparents would cause the parent not to get deallocated because it is retained by one of its children. And the children won't be deallocated unless its parent gets deallocated. This vicious circle causes a memory leak and potentially weird side effects as nodes that should long be gone keep executing their code.

Now the Ship class can add the bullets to the sprite batch directly by using the `sharedGameScene` and `bulletSpriteBatch` accessors. This is shown in Listing 6–7.

Listing 6–7. *GameScene Gets a CCSpriteBatchNode for Bullets and Accessors for the Ship Class*

```
-(void) update:(ccTime)delta
{
    Bullet* bullet = [Bullet bulletWithShip:self];
    [[[GameScene sharedGameScene] bulletSpriteBatch] addChild:bullet
                                                    z:0
                                                    tag:GameSceneNodeTagBullet];
}
```

Optimizations

While I’m optimizing this code, why not get rid of the unnecessary memory allocations and releases caused by the Bullet class? Allocating and releasing memory is an expensive operation you should aim to minimize during game play. A common solution is to instantiate a fixed number of objects when the game starts and then simply enable or disable/hide the objects as needed. This is called *object pooling*.

Because you can safely define an upper limit for the number of bullets that can be on the screen at the same time, bullets are an excellent candidate for pooling to avoid allocating and releasing bullets during game play. Since the bullets share the same texture, the additional memory used by having a greater number of bullets that resides in memory at all times is negligible. Listing 6–8 shows the changes to GameScene’s `init` method implemented in the Sprites04 project.

Listing 6–8. *Creating a Reasonable Number of Bullet Sprites Up Front Avoids Unnecessary Memory Allocations During Game Play*

```
CCSpriteBatchNode* batch = [CCSpriteBatchNode batchNodeWithFile:@"bullet.png"];
[self addChild:batch z:0 tag:GameSceneNodeTagBulletSpriteBatch];

// Create a number of bullets up front and reuse them whenever necessary.
for (int i = 0; i < 400; i++)
{
    Bullet* bullet = [Bullet bullet];
    bullet.visible = NO;
    [batch addChild:bullet];
}
```

All the bullets are made invisible since we don’t use them just yet. The GameScene class gets a new method in Listing 6–9 that allows it to shoot bullets from the ship by reactivating inactive bullets in sequence. This process is often referred to as *object pooling*. Shooting is now rerouted through the GameScene since it contains the `CCSpriteBatchNode` used for the bullets. Once an inactive bullet has been selected, it is instructed to shoot itself.

Listing 6–9. Shooting Is Now Rerouted

```

-(void) shootBulletFromShip:(Ship*)ship
{
    NSArray* bullets = [self.bulletSpriteBatch children];

    CCNode* node = [bullets objectAtIndex:nextInactiveBullet];
    NSAssert([node isKindOfClass:[Bullet class]], @"not a bullet!");

    Bullet* bullet = (Bullet*)node;
    [bullet shootBulletFromShip:ship];

    nextInactiveBullet++;
    if (nextInactiveBullet >= [bullets count])
    {
        nextInactiveBullet = 0;
    }
}

```

By keeping the reference counter `nextInactiveBullet`, each shot uses the sprite-batched bullet from that index. Once all bullets have been shot once, the index is reset. This works fine as long as the number of bullets in the pool is always greater than the maximum number of bullets on the screen.

The `Bullet` class's `shoot` method in Listing 6–10 only performs the necessary steps to reinitialize a bullet, including rescheduling its update selector by first unscheduling the update selector in case it is already running. Most importantly, the `Bullet` is set to be visible again. Its position and velocity are also reset. The `Bullet` class's `shoot` method simply resets the relevant variables such as position and velocity and then sets the bullet to be visible. Once the bullet has reached the end of its lifetime, it's simply set to not be visible again.

Listing 6–10. The Bullet Class's Shoot Method Reinitializing a Bullet

```

-(void) shootBulletFromShip:(Ship*)ship
{
    float spread = (CCRANDOM_0_1() - 0.5f) * 0.5f;
    velocity = CGPointMake(1, spread);

    outsideScreen = [[CCDirector sharedDirector] winSize].width;

    self.position = CGPointMake(ship.position.x + ship.contentSize.width * 0.5f, ←
        ship.position.y;
    self.visible = YES;

    [self unscheduleUpdate];
    [self scheduleUpdate];
}

-(void) update:(ccTime)delta
{
    self.position = ccpAdd(self.position, velocity);

    if (self.position.x > outsideScreen)
    {
        self.visible = NO;
        [self unscheduleUpdate];
    }
}

```

```
}
}
```

TIP: Note that in `shootBulletFromShip` the update selector is first unscheduled to prevent the update selector from being scheduled twice, which is an error. To unschedule a selector, you do not need to check whether it is already scheduled, however. Cocos2d may print out a warning message to the Debugger Console window in that case, but otherwise cocos2d will ignore the command if the selector currently isn't scheduled. What cocos2d doesn't like is when you try to schedule a selector that is already scheduled; thus, the preemptive unscheduling of a selector is a relatively common task when working with scheduled selectors in cocos2d.

Sprite Animations the Hard Way

Now brace yourself. I'd like to show you how sprite animations work. Figure 6–3 shows the ship's animation frames. Remember that we have the entire animation in both HD and SD resolution.



Figure 6–3. *The ship's animation five frames with different flames*

Sprite animations are another good reason to use `CCSpriteBatchNode`, because you can put all animation frames into the same texture to conserve memory. It's quite a bit of code actually, as you'll see in Listing 6–11 and in the `Sprites05` project. After that, I'll show you how to create the same animation using a texture atlas, which cuts down the amount of code you'll have to write.

Listing 6–11. *Adding an Animation to the Ship Without Using a Texture Atlas Requires Quite a Bit of Code*

```
// Load the ship's animation frames as textures and create a sprite frame
NSMutableArray* frames = [NSMutableArray arrayWithCapacity:5];
for (int i = 0; i < 5; i++)
{
    // Create a texture for the animation frame
    NSString* file = [NSString stringWithFormat:@"ship-anim%i.png", i];
    CCTexture2D* texture = [[CCTextureCache sharedTextureCache] addImage:file];

    // The whole image should be used as the animation frame
    CGSize texSize = [texture contentSize];
    CGRect texRect = CGRectMake(0, 0, texSize.width, texSize.height);

    // Create a sprite frame from the texture
    CCSpriteFrame* frame = [CCSpriteFrame frameWithTexture:texture rect:texRect];
    [frames addObject:frame];
}
```

```
// Create an animation object from all the sprite animation frames
CCAnimation* anim = [CCAnimation animationWithFrames:frames delay:0.08f];

// Run the animation by using the CCAanimate action and loop it with CCRepeatForever
CCAnimate* animate = [CCAnimate actionWithAnimation:anim];
CCRepeatForever* repeat = [CCRepeatForever actionWithAction:animate];
[self runAction:repeat];
```

All that just to create a sprite animation with five frames? I'm afraid so. I'll walk you through the code backward this time, which may explain the setup better. At the very end we're using a CCAanimate action to play an animation. In this case, we're also using a CCRepeatForever action to loop the animation.

The CCAanimate action uses a CCAnimation object (which is a container for animation frames) that defines the delay between each individual frame. In many cases, you'll later want to refer to a previously created animation. For that purpose, cocos2d has a CCAnimationCache class that stores CCAnimation instances by name, as shown in Listing 6–12. Using the CCAnimationCache, you can later access a particular animation by name.

Listing 6–12. The CCAnimationCache Class Can Store Animations for You, Which Can Be Retrieved by Name

```
CCAnimation* anim = [CCAnimation animationWithFrames:frames delay:1];

// Store the animation in the CCAnimationCache
[[CCAnimationCache sharedAnimationCache] addAnimation:anim name:@"move "];

// Sometime later: retrieve the move animation from the CCAnimationCache
CCAnimation* move = [[CCAnimationCache sharedAnimationCache] animationByName:@"move "];
```

Going back to Listing 6–11, notice the for loop. This is where it gets complicated. The CCAnimation class must be initialized with an NSArray containing CCSpriteFrame objects. A *sprite frame* consists only of a reference to a texture and a rectangle that defines the area of the texture that should be drawn. The texture rectangle equals the texture's `contentSize` property—in other words, the size of the actual image contained in the texture. Keep in mind that the texture can be bigger than the image it contains because textures can only have dimensions that are powers of two.

Now, the CCSpriteFrame unfortunately doesn't take an image file name as input; it only accepts existing CCTexture2D objects. The texture is created using the CCTextureCache singleton's `addImage` method, normally used to preload images as textures into memory without having to create a CCSprite or other object. The file name is constructed using NSString's `stringWithFormat` method, which allowed me to use the loop variable `i` to be appended to the file name, instead of having to write out all five file names.

To recap, from top to bottom, here's how you can create and run a sprite animation:

1. Create NSMutableArray.
2. For each animation frame:
 - a. Create a CCTexture2D for each image.
 - b. Create a CCSpriteFrame using the CCTexture2D.
 - c. Add each CCSpriteFrame to the NSMutableArray.

3. Create a CCAAnimation using the frames in the NSMutableArray.
4. Optionally, add the CCAAnimation to the CCAAnimationCache with a name.
5. Use a CCAAnimate action to play the animation.

Shh, calm down—no need to take that Valium. If you pack your animation frames into a texture atlas, things will get a bit easier and more efficient at the same time. More helpful is to encapsulate all this code into a helper method and stick to a naming convention for your animation files.

Animation Helper Category

Since the code to create the animation frames and the animation is common to all animations, you should consider encapsulating this into a helper method. I have done so in the project `Sprite05_WithAnimHelper`. Instead of using static methods, I decided to extend the CCAAnimation class using an Objective-C feature called a *category*. It offers a way to add methods to an existing class without having to modify the original class. The only downside is that with a category you cannot add member variables to the class; you can only add methods. The following code is the @interface for the CCAAnimation category, which I simply named Helper:

```
@interface CCAAnimation (Helper)
+ (CCAAnimation*) animationWithFile:(NSString*)name
                           frameCount:(int)frameCount
                           delay:(float)delay;
@end
```

The @interface for a Objective-C category uses the same name as the class it extends and adds a category name within parentheses. The category name is like a variable name and thus cannot contain spaces or other characters you can't use in variables, like punctuation characters, for example. The @interface also must not contain curly brackets, since adding member variables to a category is not possible and not allowed.

The actual @implementation for the CCAAnimation category uses the same schema as the @interface by appending the category name in parentheses after the class name. Everything else is just like writing regular class methods; in this case, my extension method is named `animationWithFile` and takes the file name, the number of frames, and the animation delay as input:

```
@implementation CCAAnimation (Helper)

// Creates an animation from single files
+ (CCAAnimation*) animationWithFile:(NSString*)name
                           frameCount:(int)frameCount
                           delay:(float)delay
{
    // Load the animation frames as textures and create the sprite frames
    NSMutableArray* frames = [NSMutableArray arrayWithCapacity:frameCount];
    for (int i = 0; i < frameCount; i++)
    {
        // Assuming all animation files are named "nameX.png"
```

```

NSString* file = [NSString stringWithFormat:@"%0%i.png", name, i];
CCTexture2D* texture = [[CCTextureCache sharedTextureCache] addImage:file];

// Assuming that image file animations always use the whole image
CGSize texSize = texture.contentSize;
CGRect texRect = CGRectMake(0, 0, texSize.width, texSize.height);
CCSpriteFrame* frame = [CCSpriteFrame frameWithTexture:texture rect:texRect];

[frames addObject:frame];
}

// Return an animation object from all the sprite animation frames
return [CCAnimation animationWithFrames:frames delay:delay];
}

@end

```

Here's how the naming convention comes into play. The Ship's animations have the base name `ship-anim` followed by a consecutive number starting with 0 and ending in the `.png` file extension. For example, the file names for the Ship's animation are named `ship-anim0.png` through `ship-anim4.png`. If you create all your animations using that naming scheme, you can use the preceding `CCAnimation` extension method for all of your animations.

TIP: I can't help but notice that a lot of developers and artists have a habit of consecutively naming files with a fixed number of digits, by adding leading zeros where necessary. For example, you might be tempted to name your files `my-anim0001` through `my-anim0024`. I think this habit goes back to the good ol' computer operating systems that were incapable of natural sorting and thus incorrectly sorted file names with consecutive numbers. Those days are long gone, and you'll actually make it harder for the programmer to load files named like that in a `for` loop, since you'll have to take into account how many leading zeros should be prepended. There is a nice formatting shortcut, `%03i`, to prepend zeros so that the number is always at least three digits long. However, I think it's better in our modern world to just name file names consecutively without prepending any leading zeros. You gain a little bit of simplicity and peace of mind.

This simplifies the code used to create an animation from individual files a lot:

```

// The whole shebang is now encapsulated into a Category extension method
CCAnimation* anim = [CCAnimation animationWithFile:@"ship-anim"
                                frameCount:5
                                delay:0.08f];

```

Essentially this cuts down the number of lines from nine to just this one. As the file name, you only need to pass the base name of your animation—in this case `ship-anim`. The helper method adds the consecutive numbers based on the `frameCount` parameter and also appends the `.png` file extension. You can also use the base name for the animation as the name for the animation when you add it to the `CCAnimationCache` so that you don't have to remember alternate names for the same animation. Previously I named the ship's animation `move`. Now it's called `ship-anim`, in line with the file names.

You could store and access the animation from the `CCAnimationCache` by using its base name like so:

```
NSString* shipAnimName = @"ship-anim";

CCAnimation* anim = [CCAnimation animationWithFile:shipAnimName
                                           frameCount:5
                                           delay:0.08f];
[[CCAnimationCache sharedAnimationCache] addAnimation:anim name:shipAnimName];

// sometime later:
CCAnimation* shipAnim = [shipSprite animationByName:shipAnimName];
```

The `animationWithFile` helper method makes two assumptions: animation image file names are consecutively numbered beginning with 0, and the files must be .png files. It's up to you whether you want to stick to this exact naming convention or change it to accommodate your own needs. For example, you might find it more convenient to start numbering your animations starting with 1 instead of 0. In that case, you'll have to change the for loop so that the name string is formatted with `i + 1`. The important part is to stick to whatever naming convention you choose to make your life (and your code) easier.

You should take away three things from this:

- Encapsulate commonly used code by defining your own methods.
- Use Objective-C categories to add methods to existing classes.
- Define resource file naming conventions to support your code.

Working with Texture Atlases

Texture atlases help conserve precious memory, and they also help speed up the rendering of sprites. Since a texture atlas is nothing but a big texture, you can render all the images it contains using a `CCSpriteBatchNode`, thus reducing the draw call overhead. Using texture atlases is a win-win for both memory usage and performance.

What Is a Texture Atlas?

So far, for all the sprites used, I simply loaded the image file they need to display. Internally, this image becomes the sprite's texture, which contains the image, but the texture width and height always have to be a power of two—for example, 1024×128 or 256×512. The texture size is increased automatically to conform to this rule, possibly taking up more memory than the image size would suggest. For example, an image with dimensions of 140×600 becomes a texture with dimensions of 256×1024 in memory. This texture is wasting a lot of precious memory, and the amount of wasted memory becomes significant if you have several such images and you load them each into individual textures.

That's where the texture atlas comes in. It is simply an image that is already aligned to a power-of-two dimension and contains multiple images. Each image contained in the texture atlas has a sprite frame that defines the rectangle area the image is at within the texture atlas. In other words, a sprite frame is a CGRect structure that defines which part of the texture atlas should be used as the sprite's image. These sprite frames are saved in a separate .plist file so that cocos2d can render very specific images from a large texture atlas texture.

Introducing TexturePacker

Packing images into a texture atlas and noting the rectangular sprite frames they occupy would be a monumental task if it weren't for TexturePacker, a 2D sprite-packing tool (shown in Figure 6-4). The TexturePacker app is available as both free and paid versions and can be downloaded from www.texturepacker.com.

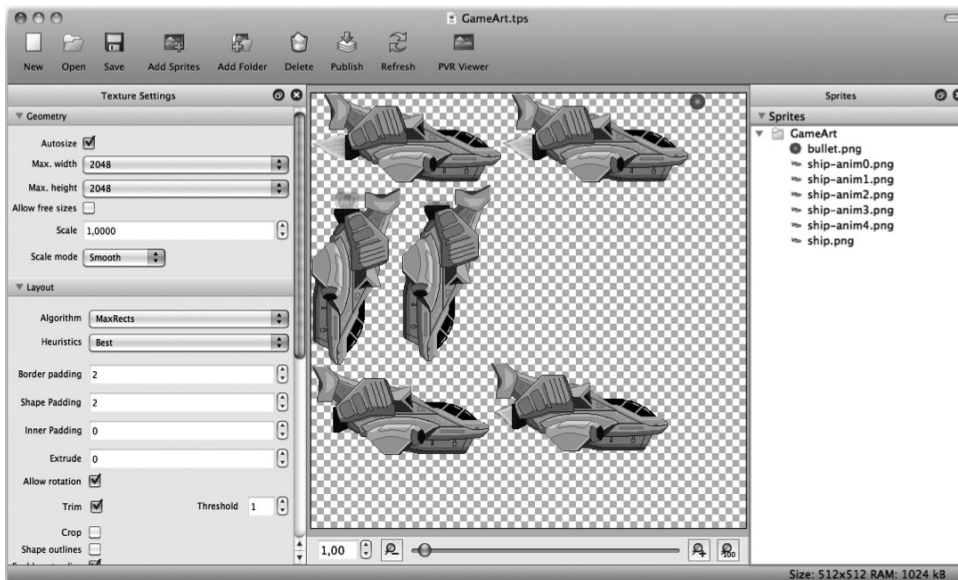


Figure 6-4. TexturePacker with the ship's animation frames already packed into a texture atlas

The free version TexturePacker Essential is sufficient for basic needs and can be used to create commercial apps. It doesn't have the Pro version's more advanced features like saving high-resolution data for Retina displays, scaling down images for non-Retina displays on the fly, or optimizing the graphics to save memory. The Pro version requires a paid license that comes at a reasonably low price.

TexturePacker can also be run as a command-line tool via the Terminal app. This allows it to be integrated in your Xcode build process. You can find more information on the TexturePacker command-line tool and how to use it on the TexturePacker web page.

In this chapter, we will use TexturePacker Pro since it can also export to the PVR image format, which is the native image format for iPhone's PowerVR graphics chip. The Pro

version also conveniently creates the SD and HD textures we need to be able to run our project on all variants of the iPhone.

Preparing the Project for TexturePacker

To use TexturePacker, the project Sprites06 needs to be reorganized a little. Currently all images in the HD and SD variants are in the Resources folder. Since you are going to use a texture atlas that contains all the images in one texture, the individual images don't need to be copied onto the device anymore.

First create a new folder called Assets that keeps all the source images and the save file for the texture atlases. You do not need to have both HD and SD versions of each image anymore since TexturePacker will scale down the images for you. So, with TexturePacker you'll be working only with the HD variants, and the image files don't need to have the -hd suffix anymore. Figure 6-5 shows the Assets folder of the Sprites06 project.

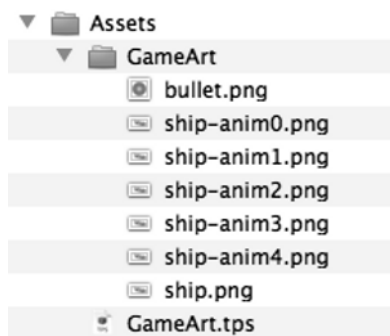


Figure 6-5. *The folder contents of the Assets folder*

Another tip that makes handling image files easier is to put all image files that should be packed into the same texture atlas in the same subfolder. In the case of the Sprites06 project, the Assets folder has a subfolder called GameArt that contains all the image files that are packed into the game-art texture atlas.

TexturePacker will simply add new image files in that folder when you are refreshing the texture atlas in TexturePacker, so you don't have to add individual images manually anymore. TexturePacker will create one .tps (TexturePacker Save) file for each folder.

CAUTION: If you have an image called `bullet.png`, the HD version must be named `bullet-hd.png`, and in `cocos2d` you will have to load the file with the string `@ "bullet.png"`. On Retina devices `cocos2d` will automatically load the `.hd` version instead.

A common mistake that happens when you move from individual images to a texture atlas is to add images to the texture atlas that have the `.hd` prefix in their filename. In that case, the resulting texture atlas for SD and HD versions might be named `bulletatlas.pvr.czz` and `bulletatlas-hd.pvr.czz`. But the images contained in the texture atlas, now called *sprite frames*, will all have the name `bullet-hd.png` even in the SD texture atlas. Thus, you would have to change all references in `cocos2d` from `@ "bullet.png"` to `@ "bullet-hd.png"`. It is recommended to use HD version images without the `.hd` suffix in the file name for creating a texture atlas to avoid this issue.

Even worse would be to manually create two texture atlases, one for SD images containing only image files without the `.hd` suffix and one for HD images containing only image files with the `.hd` suffix. If you would then load `@ "bullet.png"` in `cocos2d`, it would not be able to find the `.hd` image on Retina devices. If you were to load `@ "bullet-hd.png"`, the SD images wouldn't load. This is because the distinction between SD and HD is done at the texture atlas level; you will have a texture atlas with the `.hd` suffix and one without. The names of the sprite frames inside a texture atlas must be identical for both SD and HD texture atlases.

Creating a Texture Atlas with TexturePacker

Working with TexturePacker is very straightforward and involves only a few steps, as illustrated in Figure 6–6. In most cases you will be fine using the default settings.

First you'll have to add the images you want to add to the texture atlas. You can always add more at a later time or remove existing ones. Click the **Add Sprites** or **Add Folder** button, or simply drag and drop sprites or folders on the right pane. TexturePacker can load images from the most common graphics formats. In this case, we will add all ship images and animation frames, as well as the bullet image. You can find them in the `Assets` folder in the `Sprites06` example.

Simply drop the `GameArt` folder on the right pane. After doing so, the sprites immediately appear in the center pane, which is the real-time preview of our texture atlas. It updates immediately whenever you change any setting, including image optimizations and image layout.

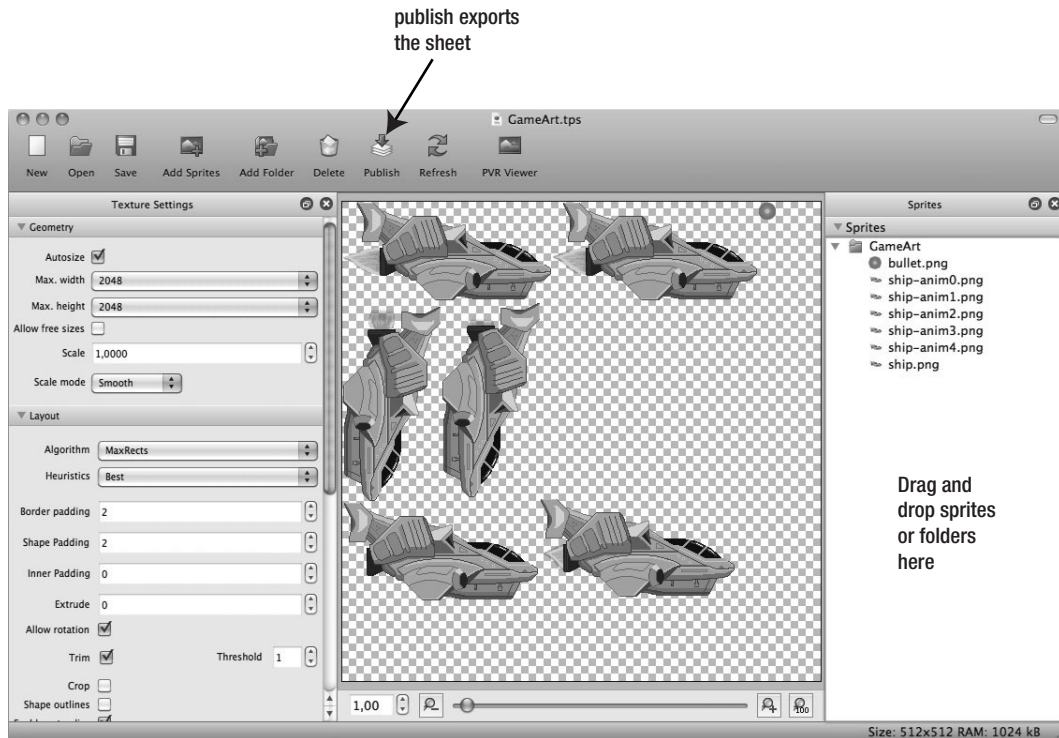


Figure 6-6. The process of working with TexturePacker is straightforward.

In the status bar at the bottom-right corner, you also see how big the resulting texture will be and how much memory it will use on the device. In this case, the texture atlas is 512×512 pixels and will use up 1MB of memory. This information is for the HD texture. The SD texture will be about 256×256 pixels in size and consume a quarter of the memory, which is 256KB. Note that the SD texture atlas may not always be exactly half the width and height of the HD atlas. This is because some layout features such as padding do not scale when the SD texture is created.

CAUTION: Unless you develop your game exclusively for third-generation and newer devices, you should not use a width or height of 2048. Older devices support texture dimensions only up to a maximum of 1024×1024 pixels. You can limit the maximum texture size that TexturePacker should generate in the left pane's Geometry section.

TexturePacker uses several tricks to optimize the texture space to create an optimal packing rate. First it trims the transparent border pixels of each image. Cocos2d compensates for this by adding the trimmed area as an offset to the sprite's position when drawing it. This has two advantages: it reduces the texture size, and it speeds up the rendering of the sprites.

You may also notice that some images are rotated. TexturePacker does this to optimize how the texture atlas space is used. Again, cocos2d compensates by restoring the original orientation when loading the image into memory. If you add two or more exactly identical images, TexturePacker will add only one image to further save texture atlas space. If the images are referenced by different file names, then you can load the image in cocos2d by using any of the image's source file names. Whenever an image in a texture atlas has multiple source files containing the same image, TexturePacker will draw a small overlay (a stack of paper) over the image in the center pane.

In case you are wondering about the **Border Padding** and **Shape Padding** settings, they determine how many pixels of space is left between all images and the border of the atlas. The default of 2 pixels ensures that all the images in the texture atlas can be drawn without any artifacts. With less padding, images can show stray pixels around their borders when they are displayed in your game. The amount and color of these stray pixels depend on the surrounding pixels from other images in the texture atlas. This is a technical issue that has to do with how the graphics hardware filters textures, and the only solution is to leave a certain amount of padding between all images in a texture atlas.

Before you can save the texture atlas, you need to make adjustments to the TexturePacker **Output** settings depicted in Figure 6-7.

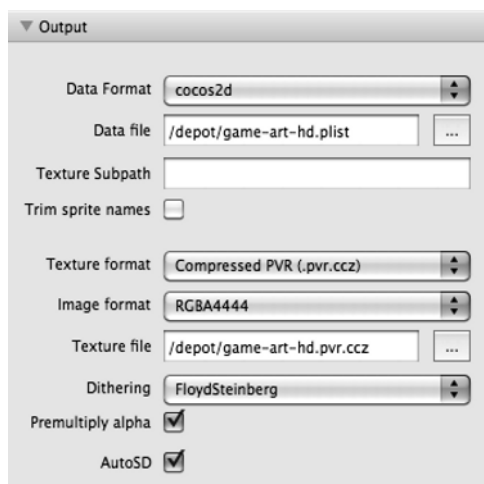


Figure 6-7. *TexturePacker Output settings*

First make sure that the data format is set to **cocos2d** since TexturePacker can be used with other game engines as well.

For the data file, you should specify a file that is located in the project's Resources folder. Make sure the file has the **-hd** suffix. In the Sprites06 project, the file is named **game-art-hd.plist**. TexturePacker will save the HD texture to the given file name but will omit the **-hd** suffix for the SD version, which is exactly how cocos2d wants the data. To have TexturePacker automatically create the SD version of the texture atlas, you also need to check the **AutoSD** check box at the bottom.

Finally, you choose **Texture Format** and **Image Format** for the exported texture atlas. The recommended **Texture Format** setting is **Compressed PVR (.pvr.ccz)**, which is a compressed version of the iPhone's native PVR format. This format typically loads faster than PNG, and if you have images with a 16-bit color depth, the compressed PVR format will create smaller files than PNG. The PNG file format will always store 32-bit color values regardless of the actual color bit depth of the image.

NOTE: Color depth, bit depth, or bits per pixel is the number of bits that are stored to represent a pixel's color information. A color depth of 4 bits allows each pixel in the image to have one of 16 possible colors. With a color depth of 16 bits, an image's pixel can display a range of 65,536 colors. And with a 24-bit color depth, an image can have millions of colors—so many in fact that an image with 24-bit color depth is sometimes referred to as a *true-color* image. Why is this format commonly referred to as 32-bit? Because the additional 8 bits in the most common file formats are used to store the opacity of each pixel.

I encourage you to read the Wikipedia articles on color depth (http://en.wikipedia.org/wiki/Color_depth) and the RGB color model (<http://en.wikipedia.org/wiki/RGB>) to learn more about how computing devices display color.

When using the PVR format, you should also enable the **Premultiply alpha** check box to avoid dark borders around sprites in some cases. In your project's app delegate, after cocos2d has been initialized and shortly before running the first scene, you should also let cocos2d know that your PVR images use premultiplied alpha:

```
// Enable pre multiplied alpha for PVR textures to avoid artifacts
[[CCTexture2D PVRImagesHavePremultipliedAlpha:YES];
```

```
// run the main scene
[[CCDirector sharedDirector] runWithScene: [GameScene scene]];
```

For the **Image Format** setting, you have a variety of options. The default format is **RGBA8888**, which gives the best-looking results. It provides 24-bit color depth and an 8-bit alpha channel. The downside is that this format is also the slowest to render, and particularly on first- and second-generation devices, it is recommended to fall back to a lower-quality image format and favor rendering speed. However, providing device-specific variants of texture atlases is cumbersome to work with. You may want to simply render fewer sprites on older devices or possibly even drop support for those devices altogether.

The best compromise between quality, memory usage, and rendering speed is provided by the **RGBA4444** format. It uses 4 bits per color and 4 bits for the alpha channel. This is the most commonly used image format for sprites.

If transparency is not important to you and you'd like to have more color variations, you should use the **RGB5551** format. It provides 5 bits per color and only 1 bit for the alpha

channel. The 1 bit for the alpha channel can be either set or not, which means your image can have only fully transparent or fully opaque pixels. In other words, RGB5551 sprites won't be able to blend with pixels in the background.

If you do not need any transparency at all, like for background images, you can use the **RGB565** format, which provides 5 bits for red, 6 bits for green, and 5 bits for the blue color channel. It does not use an alpha channel. The fact that there are 6 bits for green colors and only 5 bits for the other two color channels has something to do with our hunter-gatherer background. Our retina is simply trained to differentiate better between green color tones, so that extra bit is provided to the green channel where we would notice a "missing color bit" more easily.

The **PVRTC2** and **PVRTC4** image formats provide 2 and 4 bits per pixel, respectively, and no alpha channel. This format should be used only for monotonous or dark background images and if you really need to squeeze out some memory and rendering speed since those formats come with a severe impact on image quality. Think of the artifacts seen in JPEG images with a relatively low-quality setting.

The Dithering option allows you to optimize the image quality whenever the image format requires the image's color depth to be reduced. Dithering emulates gradients by randomly distributing pixels with similar color tones across a larger area. This effectively reduces the "banding" effect when the color depth of an image is reduced. Since all of the dithering options are applied in real time in the TexturePacker preview without affecting your source images, you can just try the various dithering algorithms to find out which provides the best quality.

TIP: While evaluating dithering algorithms, keep in mind that the ultimate quality test is of course your game running on a device. Some artifacts that are clearly visible on your computer screen may not be noticeable on the device. In particular, this is because the color profile of your computer screen will be different from the device's color profile, either through manual adjustments (for example, brightness, contrast, color tone), limitations imposed by the display technology, or a change in color vibrancy as the display ages. That also means a single device is not representative for the final look of the game. At the very least, you should test the game with the device brightness set to minimum and maximum levels.

When you are done with the **Output** settings, simply click **Publish**, and TexturePacker will write the HD and SD textures and the accompanying plist files to your Resources folder.

Using the Texture Atlas with cocos2d

The next thing you should do is add the new texture atlas to the Xcode project's Resource group. Cocos2d only needs the `game-art-hd.pvr.ccz`, `game-art.pvr.ccz`, `game-art-hd.plist`, and `game-art.plist` files for the texture atlas. The TexturePacker `.tsp` files and individual source image files should not be added to your project. The code in Listing 6-13 now replaces the code in Listing 6-11.

Listing 6–13. *The Ship Class Now Uses the Texture Atlas for Its Initial Frame and the Animation*

```
// Load the texture atlas sprite frames; this also loads the Texture with the same name
CCSpriteFrameCache* frameCache = [CCSpriteFrameCache sharedSpriteFrameCache];
[frameCache addSpriteFramesWithFile:@"game-art.plist"];

// Loading the ship's sprite using a sprite frame name (e.g., the file name)
if ((self = [super initWithSpriteFrameName:@"ship.png"]))
{
    // Load the ship's animation frames
    NSMutableArray* frames = [NSMutableArray arrayWithCapacity:5];
    for (int i = 0; i < 5; i++)
    {
        NSString* file = [NSString stringWithFormat:@"ship-anim%i.png", i];

        CCSpriteFrame* frame = [frameCache spriteFrameByName:file];
        [frames addObject:frame];
    }

    // Create an animation object from all the sprite animation frames
    CCAAnimation* anim = [CCAAnimation animationWithFrames:frames delay:0.08f];

    // Run the animation by using the CCAanimate action
    CCAanimate* animate = [CCAanimate actionWithAnimation:anim];
    CCRRepeatForever* repeat = [CCRRepeatForever actionWithAction:animate];
    [self runAction:repeat];
}
```

At the very beginning of the code, I assigned the `sharedSpriteFrameCache` to a local variable. The only reason to do so is that the `[CCSpriteFrameCache sharedSpriteFrameCache]` singleton accessor is pretty lengthy to write.

To load a texture atlas, you use the `CCSpriteFrameCache`'s method `addSpriteFramesWithFile` and pass it the name of the `.plist` file for this texture atlas. The `CCSpriteFrameCache` will load the sprite frames and will also try to load the texture. Cocos2d automatically tries to load the `-hd` suffixed files on Retina display devices.

NOTE: If you're using a large texture atlas texture with dimensions of 1024×1024 or higher you should load this texture before game play begins. It will take a moment to load such a large texture (in the worst case it will freeze the game for a few seconds).

Because the `Ship` class derives from `CCSprite` and because I wanted it to use the `ship.png` image from the texture atlas, I changed its initialization to use the `initWithSpriteFrameName` method. This is identical to the code that initializes a regular `CCSprite` from a texture atlas using a sprite frame name.

```
CCSprite* sprite = [CCSprite spriteWithSpriteFrameName:@"ship.png"];
```

If you load several texture atlases and only one contains the sprite frame with the name `ship.png`, cocos2d will still find that frame and use the correct texture for the sprite. In essence, you work with the sprite frames by name as if they were the image's file names, but you do not need to know which texture contains the actual image (unless

you use a `CCSpriteBatchNode`, of course, which requires that all of its children use the same texture).

In Listing 6–13 I could get rid of most of the extra code required to initialize a `CCSpriteFrame` object. There's no need any more to load a `Texture2D` and define the texture's dimensions. Instead, I simply call `[CCSpriteFrame spriteFrameByName:file]` to create the sprite frame with the corresponding name.

Updating the CCAAnimation Helper Category

While the code to create a `CCAnimation` could be reduced significantly by using a texture atlas, it's still worthwhile to encapsulate this code into the `CCAAnimationHelper` class. After all, one line of code is still less than five lines, especially if you would otherwise use the same five lines of code everywhere. Without further ado, Listing 6–14 shows the extended `CCAAnimation` Helper interface declaration, which adds the `animationWithFrame` method.

Listing 6–14. *The @interface for the CCAAnimation Helper Category*

```
@interface CCAAnimation (Helper)
+ (CCAAnimation*) animationWithFile:(NSString*)name
                        frameCount:(int)frameCount
                        delay:(float)delay;

+ (CCAAnimation*) animationWithFrame:(NSString*)frame
                        frameCount:(int)frameCount
                        delay:(float)delay;

@end
```

This code is essentially the same method using the same parameters, except that this method uses sprite frames instead of file names. The implementation is nothing spectacular and is very similar to the `animationWithFile` method shown in Listing 6–15.

Listing 6–15. *The animationWithFrame Helper Method Makes It Easier to Create an Animation*

```
// Creates an animation from sprite frames
+ (CCAAnimation*) animationWithFrame:(NSString*)frame
                        frameCount:(int)frameCount
                        delay:(float)delay
{
    // load the ship's animation frames as textures and create a sprite frame
    NSMutableArray* frames = [NSMutableArray arrayWithCapacity:frameCount];
    for (int i = 0; i < frameCount; i++)
    {
        NSString* file = [NSString stringWithFormat:@"%02i.png", frame, i];
        CCSpriteFrameCache* frameCache = [CCSpriteFrameCache sharedSpriteFrameCache];
        CCSpriteFrame* frame = [frameCache spriteFrameByName:file];
        [frames addObject:frame];
    }

    // Return an animation object from all the sprite animation frames
    return [CCAAnimation animationWithFrames:frames delay:delay];
}
```

The big plus is now, once again, that you can create an animation from a texture atlas using sprite frame names with just one line of code:

```
// Create an animation object from all the sprite animation frames
CCAnimation* anim = [CCAnimation animationWithFrame:@"ship-anim"
                                     frameCount:5
                                     delay:0.08f];
```

The much, much bigger plus is, however, that you can now work with your animations as single files and only later create a texture atlas. All you have to do is to change one line of code from using `animationWithFile` to the `animationWithFrame` method. This allows you to quickly prototype animations using individual files, and only when you're satisfied will you pack the animation frames into a texture atlas and load the animation images from it.

You'll find this code in the project `Sprites06_WithAnimHelper`.

All into One and One for All

Whenever possible, you should add all your game's images into one texture atlas or as few as possible. It is more effective both from a workflow perspective and for performance to use three texture atlases with dimensions of 1024×1024 than 20 smaller ones.

Unlike code, which you should separate into distinct logical components, with a texture atlas your goal should be to put as many images as possible into the same texture atlas while trying to reduce the wasted space of each texture atlas as much as possible.

It may seem logical to use one texture atlas for your player's images; another for monster A, B, and C and their animations; and so on. But that would mean more draw calls. However, that is helpful only if you have a huge number of images for each game object and you want to be selective about which images to load into memory at any one time. One such scenario might be a shoot-'em-up game with different worlds where you know that each world has separate types of enemies. In that case, it makes sense not to mix and match enemies of different worlds into the same texture atlas. Otherwise, just for organization's sake, you should not split up your texture atlases by game objects but rather fill each texture atlas as much as possible.

As long as your game's images can fit into three or four texture atlases of 1024×1024 size, you should just put all the images into those texture atlases and load them up front. This will use 12MB to 16MB of memory for your textures. Your actual program code and other assets such as audio files don't take up that much space, so you should be able to keep these texture atlases in memory even on first- and second-generation iOS devices with just 24MB of texture memory. Newer devices have no texture memory limit.

Once you pass that point, however, you need a better strategy to handle your texture memory. One such strategy, as mentioned earlier, could be to divide your game's images into worlds and load only the texture atlases needed for the current world. This will introduce a short delay when a new world is loaded and would be a good use for the `LoadingScene` described in Chapter 5.

Since cocos2d automatically caches all images, you'll need a way to specifically unload textures that you know you don't need. You can rely on cocos2d to do that for you:

```
[[CCSpriteFrameCache sharedSpriteFrameCache] removeUnusedSpriteFrames];  
[[CCTextureCache sharedTextureCache] removeUnusedTextures];
```

Obviously you should call these methods only when there are unused textures that you want to be removed. This is typically done after changing scenes and should not be done during game play. Keep in mind that changing scenes causes the previous scene to be deallocated only after the new scene has been initialized. This means you can't use the `removeUnused` methods in the `init` method of a scene—that is, unless you use the `LoadingScene` from Chapter 5 in between two scenes, in which case you should extend it so that it removes unused textures before replacing itself with the new scene.

If you absolutely want to remove all textures from memory before loading new ones, you should use the `purge` methods instead:

```
[CCSpriteFrameCache purgeSharedSpriteFrameCache];  
[CCTextureCache purgeSharedTextureCache];
```

Summary

In this chapter, you learned how to use a `CCSpriteBatchNode` to render multiple sprites using the same texture faster, whether that texture is a single image or a sprite frame of a texture atlas.

Subclassing your game objects from `CCSprite` also introduces a few subtle differences and stumbling blocks, which I demonstrated earlier in this chapter, before I moved on to show you how to create sprite animations. Since the code to create animations is very complex, I gave you the solution in the form of a `CCAnimationHelper` category.

I also showed you how to work with texture atlases and why and how you should use them. Of course, one can't say "Texture Atlas" without mentioning `TexturePacker` in the same sentence. It's hands down the best tool to create and modify texture atlases, and if you don't want to spend money on it, you can still use the free version but without the advanced features.

In the next chapter, you'll be working on your next game, as we work on making the shooter game playable.

Scrolling with Joy

Continuing with the beginnings of the game from the previous chapter, I now want to turn it into something resembling an actual shoot-'em-up game. The very first thing will be to make the player's ship controllable. Accelerometer controls don't make sense in this case; a virtual joystick would be much more appropriate. But instead of reinventing the wheel, we'll be using a cool source code package called *SneakyInput* to add a virtual joystick to this *cocos2d* game.

Moving the player's ship around is just one thing. We also want the background to scroll, to give the impression of moving into a certain direction. For this to happen, we'll implement our own solution for parallax scrolling, since *CCParallaxNode* is too limited in that it does not allow an infinitely scrolling parallax background.

In addition, I'll illustrate what you've learned about texture atlases and sprite batching in the previous chapter. There will be one texture atlas containing all of the game's graphics since there's no need to group the images separately when using a texture atlas.

Advanced Parallax Scrolling

I mentioned before that *CCParallaxNode* is limited in that it doesn't allow infinite scrolling. For this shooting game, we'll add a *ParallaxBackground* node, which will do just that. Moreover, it will use a *CCSpriteBatchNode* to speed up the rendering of the background images.

Creating the Background As Stripes

First, I would like to illustrate how I created the background stripes that will create the parallaxing effect. This is crucial to understanding how the texture atlas created by *TexturePacker* can help you save memory and performance, but also save time positioning the individual stripes. Take a look at Figure 7-1; it shows the background layer as a *Seashore* image composed of individual parallax layers. This image is also in the *Assets* folder of the *ScrollingWithJoy01* project, as *background-parallax.xcf*.

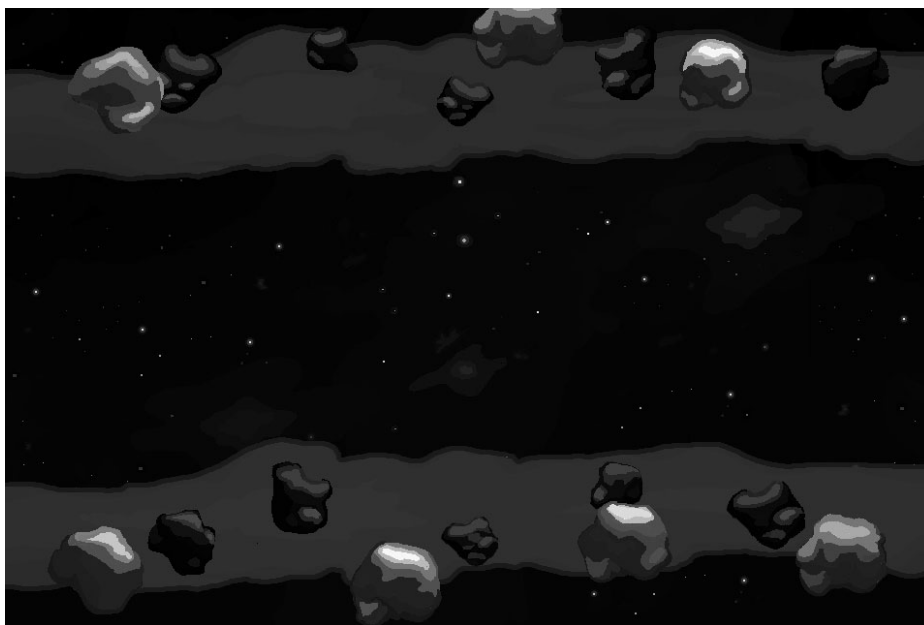


Figure 7-1. *The source image for the parallax scrolling background*

Each of these stripes is on its own layer in the image-editing program Seashore. In Figure 7-2 you can see the various layers, and if you look in the Assets folder, you'll notice that there are images named bg0.png through bg6.png, which correspond to the seven individual stripes that make up the background.

There are several reasons for creating the parallax background image in this way. You can create the image as a whole, but you're able to save each layer to an individual file. All of these files will be 960×640 pixels in size, which may seem wasteful at first. But you're not adding the individual images to the game; instead, you'll be adding them to a texture atlas. Since TexturePacker removes the surrounding transparent space of each image, it will shrink the individual stripes down to a bare minimum. You can see this in Figure 7-3.

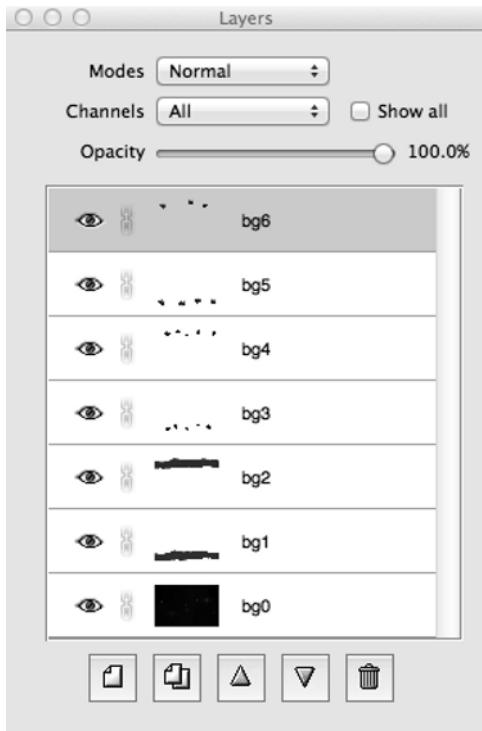


Figure 7–2. Each stripe of the background is on its own layer. This helps in creating the individual images and positioning them in the game.

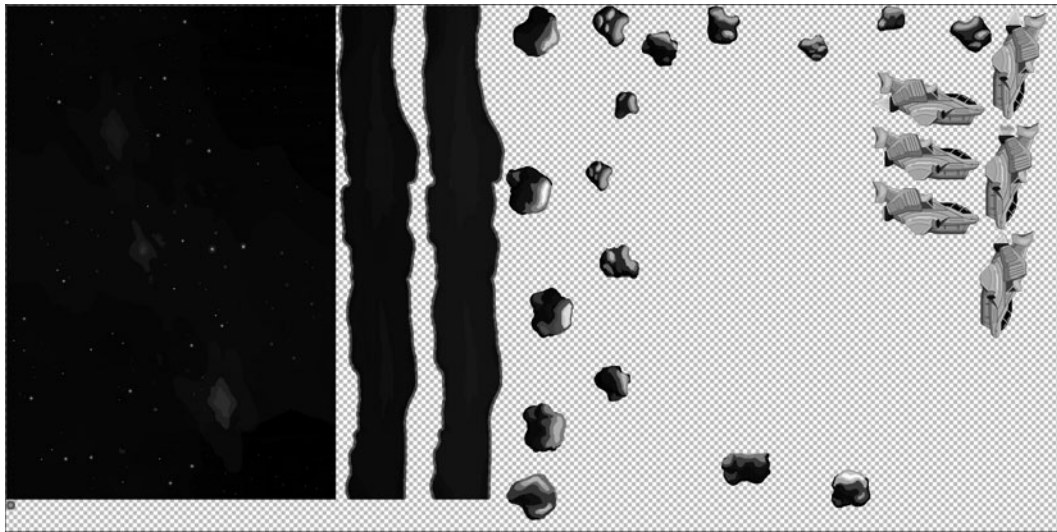


Figure 7–3. The background stripes as they would appear in a texture atlas

Splitting the stripes into individual images is not only going to be helpful for drawing the images at the correct z-order. Strictly speaking, the images `bg5.png` and `bg6.png` can be at the same z-order since they do not overlap, yet I chose to save them into separate files. In Figure 7–3 you can see these two files as the two topmost stripes. You'll notice how little space they actually use up in the texture atlas, and that's because TexturePacker was able to remove most of the surrounding transparent parts of these images.

Now suppose I left both these stripes in the same 960×640 image—one would be at the top and the other at the bottom of the image, with a big gaping hole of transparency in between them. TexturePacker is not able to remove the transparent part between these two sprites, so they would have remained as a 960×640 image in the texture atlas, which is a lot more space than they take up as individual images.

Splitting the stripes into individual images also helps maintain a high framerate. The iOS devices are very limited in *fill rate* (that is, the number of pixels they can draw every frame). Since images can overlap each other, and they frequently do, the iOS device will often have to draw the same pixel several times in every frame. The extreme scenario would be a full-screen image that is on top of another full-screen image. You can see only one of the two images, but the device actually has to draw both. The technical term for that is *overdraw*. Separating the background into individual stripes with as little overlap as possible reduces the number of pixels drawn.

TIP: By using 16-bit textures (RGB565, RGBA5551, and RGBA4444) or even PVR compressed textures, you can improve the rendering performance of your game. Of course, you lose some image quality, but in many cases it is hardly noticeable on the iPhone screen, even if you do see some artifacts on your computer's screen. TexturePacker enables you to reduce the color depth of your images while retaining most of the image quality through dithering techniques. And if you experience long load times, saving images in the compressed pvr.czz format is worth a try because this particular image file format loads noticeably faster than other image file formats.

Re-creating the Background in Code

You may be wondering by now how you can put these images back together in the source code without spending a lot of time properly positioning these stripped-down images. The answer is you don't have to do this. Since all these images were saved as full-screen images, TexturePacker will remember the image offsets. All you really have to do is center each of these images on the screen, and they will be at the correct place.

Let's take a look at the code for the `ParallaxBackground` node newly added to the `ScrollingWithJoy01` project. The header file is pretty straightforward:

```
@interface ParallaxBackground : CCNode
{
    CCSpriteBatchNode* spriteBatch;
    int numSprites;
}
@end
```

I only chose to keep a reference to the `CCSpriteBatchNode` around because I will be accessing it in the code frequently. Storing a node as a member variable is faster than asking cocos2d for the node via the `getNodeByTag` method. If you do that every frame, it saves you a few CPU cycles. It is nothing too dramatic, and it is certainly not worth keeping several hundreds of member variables around. It's a minor optimization that is just very convenient in this case.

In the `init` method of the `ParallaxBackground` class, the `CCSpriteBatchNode` is created, and all seven background images are added from the texture atlas, as shown in Listing 7–1.

Listing 7–1. Loading the Background Images

```
CGSize screenSize = [[CCDirector sharedDirector] winSize];

// Get the game's texture atlas texture by adding it
CCTexture2D* gameArtTexture = [[CCTextureCache sharedTextureCache]
    addImage:@"game-art.pvr.czz"];

spriteBatch = [CCSpriteBatchNode batchNodeWithTexture:gameArtTexture];
[self addChild:spriteBatch];

numSprites = 7;

// Add the six different layer objects and position them on the screen
for (int i = 0; i < numSprites; i++)
{
    NSString* frameName = [NSString stringWithFormat:@"bg%i.png", i];
    CCSprite* sprite = [CCSprite spriteWithSpriteFrameName:frameName];
    sprite.position = CGPointMake(screenSize.width / 2, screenSize.height / 2);
    [spriteBatch addChild:sprite z:i];
}
```

You'll see that first I'm adding the `game-art.pvr.czz` texture atlas image to `CCTextureCache`. Actually, the texture atlas has already been added in the `GameScene` class, so why would I add it a second time here? The reason is simply that I need the `CCTexture2D` object to create the `CCSpriteBatchNode`, and adding the same image another time is the only way to retrieve an already cached image from the `CCTextureCache`. This doesn't load the texture a second time; the `CCTextureCache` singleton knows that the texture is already loaded and returns the cached version, which is a fast operation. It's just a bit unusual that there is no specific `getTextureByName` method, but that's the way it is.

With the `CCSpriteBatchNode` created and set up, the next step is to load the seven individual background images. I deliberately chose to number them from 0 to 6 so that I can use `stringWithFormat` to create the file names as strings in a very effective way:

```
NSString* frameName = [NSString stringWithFormat:@"bg%i.png", i];
```

With that `sprite frameName`, I create a `CCSprite` as usual and then position it at the center of the screen:

```
sprite.position = CGPointMake(screenSize.width / 2, screenSize.height / 2);
```


Of course, once you create an iPad version of this project, the images won't fit perfectly anymore since they were designed for a screen of 960×640 resolution. If you want to create an iPad version, you can follow the exact same steps, except make your original image 1024×768 in size. You can then downscale the 960×640 versions from that easily, since there's only a few extra pixels of overlap.

TIP: It takes surprisingly little effort to re-create the source image in cocos2d, and it's all thanks to TexturePacker saving the image offsets for you. It's also a great way to create your game screen layouts. You can have an artist design each screen as separate layers, as many as are needed. Then each layer is exported as an individual full-screen file with transparency. Next you create a texture atlas from these files and end up having the artist-envisioned screen design in cocos2d with no hassle of positioning individual files and no wasted memory either.

Because the `ParallaxBackground` class is derived from `CCNode`, I only need to add it to the `GameScene` layer to add the `ParallaxBackground` to the game, like this:

```
ParallaxBackground* background = [ParallaxBackground node];  
[self addChild:background z:-1];
```

This replaces the `CCLayerColor` and the background `CCSprite`, which were placeholders from the previous chapter.

Moving the ParallaxBackground

In the `ScrollingWithJoy01` project, I also added a quick-and-dirty scrolling of the background stripes. It does show a parallax effect, although the images quickly leave the screen, revealing the blank background behind them. Figure 7–4 isn't exactly what I had in mind, but I'm getting there.

NOTE: You'll also notice that Figure 7.4 doesn't look at all like Figure 7.1 or the game project. Figure 7.4 uses dummy graphics to better illustrate the parallax images leaving the screen. You'll notice this effect much better in motion.

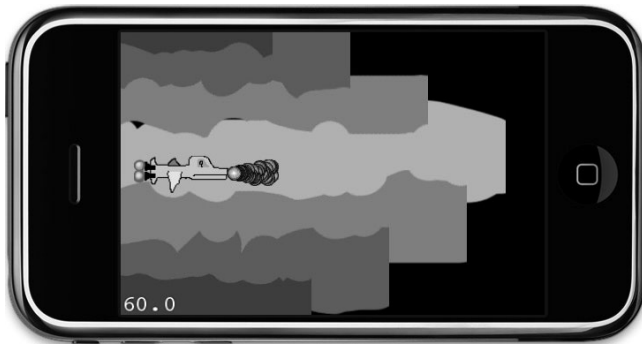


Figure 7-4. *The background stripes are moving, but they're also leaving the screen forever.*

Still, Listing 7-2 shows that the code to make the background scroll as a first test is surprisingly simple.

Listing 7-2. *Moving the Background Stripes*

```
-(void) update:(ccTime)delta
{
    CCSprite* sprite;
    CCARRAY_FOREACH([spriteBatch children], sprite)
    {
        CGPoint pos = sprite.position;
        pos.x -= (scrollSpeed + sprite.zOrder) * (delta * 20);
        sprite.position = pos;
    }
}
```

Every background image's x position gets subtracted a bit every frame to scroll it from right to left. How much an image moves depends on the predefined `scrollSpeed` plus the sprite's `zOrder`. The `delta` multiplier is used to make the scrolling speed independent of the framerate, which is then multiplied by 20 to make the scrolling reasonably fast. Images that are closer to the screen scroll faster. However, using the `zOrder` property causes the stripes that should be at the same visual depth to scroll at different speeds.

The position is also multiplied by the *delta time* to make the scrolling speed independent of the framerate. The *delta time* itself is just a tiny fraction—it's the time between two calls to the update method. At exactly 60 frames per second (fps), it's 1/60 of a second, which is a *delta time* of 0.167 seconds. For that reason, I multiply *delta* by 20 just to get a reasonably fast scroll; otherwise, the images would be moving too slowly.

Parallax Speed Factors

Somehow the stripes of the same color need to scroll at the same speed, and the stripes should repeat so that the background doesn't show up. My solutions to these issues are in the `ScrollingWithJoy02` project.

The first change is regarding the scrolling speed. I decided to use a `CCArray` to store the speed factor with which individual stripes move. There are other solutions, but this

allows me to illustrate a key issue of CCArray and in fact all iOS SDK collection classes: they can store only objects, never values such as integers and floating-point numbers.

The way around this is to box numbers into an NSNumber object. The following code is the newly added CCArray* speedFactors, which stores floating-point values. The array is defined in the ParallaxBackground class header:

```
@interface ParallaxBackground : CCNode
{
    CCSpriteBatchNode* spriteBatch;

    int numStripes;
    CCArray* speedFactors;
    float scrollSpeed;
}
@end
```

Then it is filled with factors in the init method of the ParallaxBackground class. Notice how NSNumber numberWithFloat is used to store a float value inside the array:

```
// Initialize the array that contains the scroll factors for individual stripes
speedFactors = [[CCArray alloc] initWithCapacity:numStripes];
[speedFactors addObject:[NSNumber numberWithFloat:0.3f]];
[speedFactors addObject:[NSNumber numberWithFloat:0.5f]];
[speedFactors addObject:[NSNumber numberWithFloat:0.5f]];
[speedFactors addObject:[NSNumber numberWithFloat:0.8f]];
[speedFactors addObject:[NSNumber numberWithFloat:0.8f]];
[speedFactors addObject:[NSNumber numberWithFloat:1.2f]];
[speedFactors addObject:[NSNumber numberWithFloat:1.2f]];
NSAssert([speedFactors count] == numStripes, @"speedFactors count mismatch!");
```

The final assert is simply a safety check for human error. Consider that you may be adding or removing stripes from the background for whatever reason but might forget to adjust the number of values you're adding to the speedFactors array. If you forget to modify the speedFactors initialization, the assert will remind you of it, instead of potentially crashing the game seemingly at random.

In Figure 7–5 you can see which speed factor is applied to which stripe. Stripes with higher speed factors move faster than those with slower speed factors, which creates the parallax effect. Once again, dummy graphics are used to make each individual stripe clearly visible.

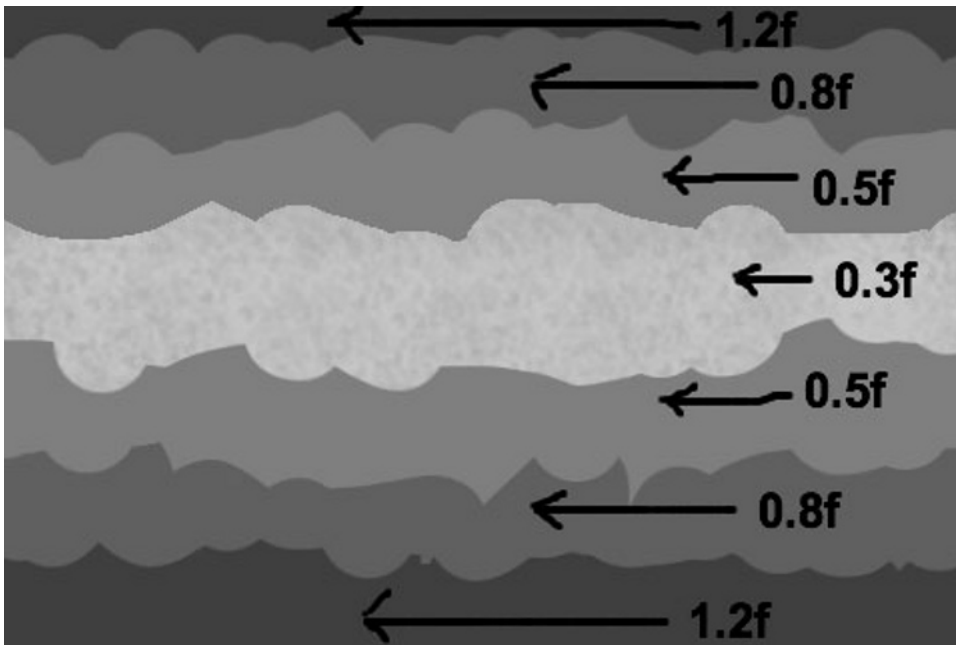


Figure 7-5. *The speed factors applied to each background stripe*

The `speedFactors` array is created with `alloc`, so its memory must also be released, which is done in the newly added `dealloc` method:

```
-(void) dealloc
{
    [speedFactors release];
    [super dealloc];
}
```

To use the newly introduced speed factors, the `update` method has been changed to this:

```
-(void) update:(ccTime)delta
{
    CCSprite* sprite;
    CCARRAY_FOREACH([spriteBatch children], sprite)
    {
        NSNumber* factor = [speedFactors objectAtIndex:sprite.zOrder];

        CGPoint pos = sprite.position;
        pos.x -= scrollSpeed * [factor floatValue] * (delta * 50);
        sprite.position = pos;
    }
}
```

Based on the `sprite`'s `zOrder` property, a speed factor `NSNumber` is obtained from the `speedFactors` array. This will be multiplied by the `scrollSpeed` to speed up or slow down the movement of individual stripes. You can't multiply the `NSNumber` object directly because it is a class instance storing a primitive data type such as `float`, `int`, `char`, or

others. `NSNumber` has a `floatValue` method that returns the floating-point value, but it supports a number of different methods. You could also use `intValue`, even though this `NSNumber` stores a floating-point value. It is essentially the same as casting a float to an int. Once again, the delta time is also factored in to make the scrolling speed independent of the framerate.

By using the `speedFactors` array and giving the same-colored stripes the same factor, the background stripes will now move as expected. But there's still the issue of making endless scrolling.

Scrolling to Infinity and Beyond

Also in `ScrollingWithJoy02` is the first step toward endless scrolling. As you can see in Listing 7–3, I simply added seven more background stripes to the `CCSpriteBatchNode`, although with a slightly different setup.

Listing 7–3. *Adding Off-Screen Background Images*

```
// Add seven more stripes, flip them, and position them next to their neighbor stripe
for (int i = 0; i < numStripes; i++)
{
    NSString* frameName = [NSString stringWithFormat:@"bg%i.png", i];
    CCSprite* sprite = [CCSprite spriteWithSpriteFrameName:frameName];

    // Position the new sprite one screen width to the right
    sprite.position = CGPointMake(screenSize.width + screenSize.width / 2,
                                  screenSize.height / 2);

    // Flip the sprite so that it aligns perfectly with its neighbor
    sprite.flipX = YES;

    // Add the sprite using the same tag offset by numStripes
    [spriteBatch addChild:sprite z:i tag:i + numStripes];
}
```

The idea is to add one more stripe of the same type each. They are positioned so that they align with the right end of the first stripe's position. In effect, this doubles the total width of the background stripes, and that will be enough for endless scrolling. But I'll get to that shortly.

First I need to point out that the new neighboring images' X coordinates are flipped (mirrored along the y-axis). This is done so that the images match visually where they are aligned, to avoid any sharp edges. The new images also get a different tag number, one that is offset by the number of stripes in use. This way, it will be easy to get to the neighboring stripe by either adding or deducting `numStripes` from the tag number.

Right now, the background image scrolls just a bit longer before showing the blank canvas behind the images. The project `ScrollingWithJoy03` completes the effort and adds totally infinite scrolling. But first the `anchorPoint` property of the stripes needs to change to make things a little easier. The x position is now simply at 0:

```
sprite.anchorPoint = CGPointMake(0, 0.5f);
sprite.position = CGPointMake(0, screenSize.height / 2);
```

The same goes for the secondary, flipped stripes, which now only need to be offset by the screen width:

```
sprite.anchorPoint = CGPointMake(0, 0.5f);
sprite.position = CGPointMake(screenSize.width, screenSize.height / 2);
```

The stripes' `anchorPoint` is changed from its default (0.5f, 0.5f) to (0, 0.5f). This makes it easier to work with the parallax sprites, since in this particular case you don't want to have to take into account that that texture's origin and the sprite's x position aren't at the same location. Figure 7-6 shows how this makes it easier to calculate the x position.

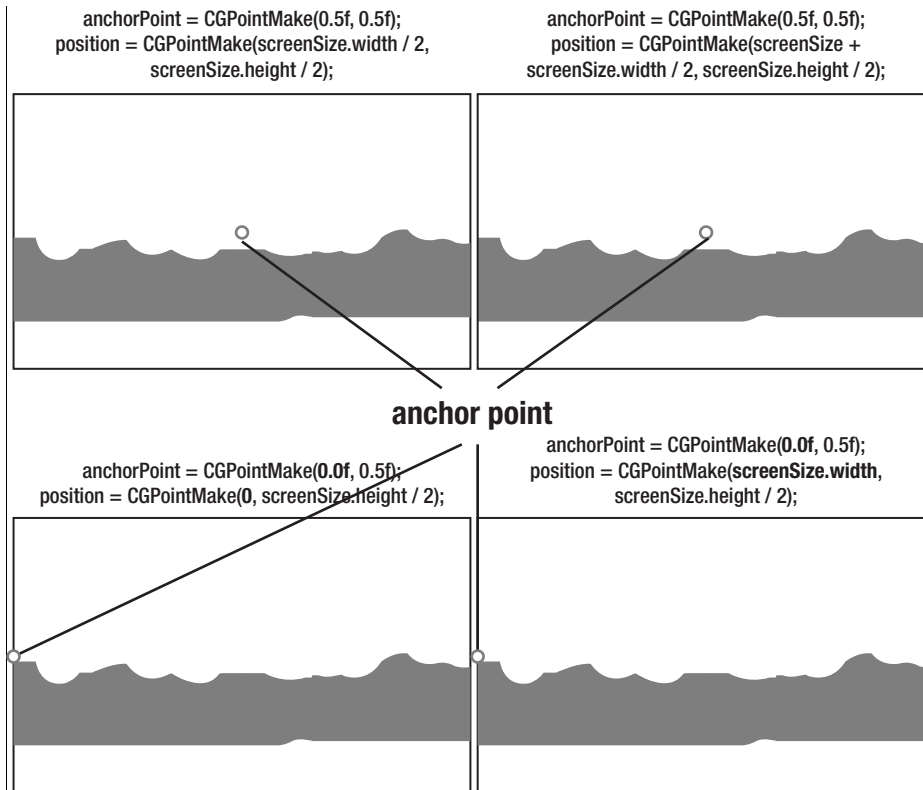


Figure 7-6. Anchor point moved to the left for simplicity

You can see in Listing 7-4 how this is helpful in the changed update method, which now gives us endless scrolling.

Listing 7-4. Moving the Image Pairs Seamlessly

```
-(void) update:(ccTime)delta
{
    CCSprite* sprite;
    CCARRAY_FOREACH([spriteBatch children], sprite)
    {
        NSInteger* factor = [speedFactors objectAtIndex:sprite.zOrder];
```

```
CGPoint pos = sprite.position;
pos.x -= scrollSpeed * [factor floatValue] * (delta * 50);

// Reposition stripes when they're out of bounds
if (pos.x < -screenSize.width)
{
    pos.x += screenSize.width * 2;
}

sprite.position = pos;
}
```

The stripes' x position now only needs to be checked if it is less than negative screen width and, if it is, multiplied by twice the screen width. Essentially, this moves the sprite that has just left the screen on the left side to the right side, just outside the screen. This repeats forever with the same two sprites, giving the effect of endless scrolling.

TIP: Notice that the background of the screen scrolls, but the ship stays in place. Inexperienced game developers often have the misconception that everything on the screen needs to be scrolling to achieve the effect of game objects passing by the player character as he progresses through the game world. Instead, you can much more easily create the illusion of objects moving on the screen by moving the background layers but keeping the player character fixed in place. Popular examples of games that make use of this visual illusion are Super Turbo Action Pig, Canabalt, Super Blast, Doodle Jump, and Zombieville USA. Typically, the game objects scrolling into view are randomly generated shortly before they appear, and when they leave the screen, they are removed from the game. In Chapter 11, I'll make use of the same effect where the player character remains in the center of the screen while only the game world underneath the player is actually moving, giving the impression that the player moves around in the world.

Fixing the Flicker

So far, so good. There's only one issue remaining. If you look closely, you will notice a vertical, flickering line appearing where the two background stripes meet. That's where they align with each other. This line appears because of rounding errors in their positions in combination with subpixel rendering. From time to time, a 1-pixel-wide gap can appear for just a fraction of a second. It's still noticeable and is something you should get rid of for a commercial-quality game.

The simplest solution is to overlap the stripes by 1 pixel. In the ScrollingWithJoy04 project I changed the initial positions for the flipped background stripes by subtracting 1 pixel from the x position:

```
sprite.position = CGPointMake(screenSize.width - 1, screenSize.height / 2);
```

This also requires updating the stripe-repositioning code in the update method so that the stripe is positioned 2 pixels further to the left than before:

```
// Reposition stripes when they're out of bounds
if (pos.x < -screenSize.width)
{
    pos.x += (screenSize.width * 2) - 2;
}
```

Why 2 pixels? Well, since the initial position of the flipped stripes is already moved to the left by 1 pixel, we have to move all of them 2 pixels to the left each time they flip around to maintain the same distance and to keep the overlap of 1 pixel.

An alternative solution would be to update only the position of the currently leftmost sprite and then find the sprite that is aligned to the right of it and offset it by exactly the screen width. This way, you also avoid the rounding errors. Figure 7–7 shows the finished result with the final graphics.

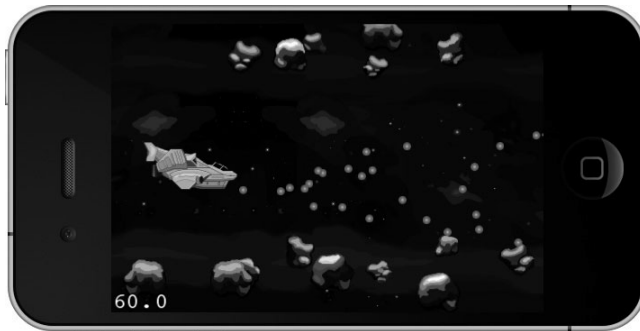


Figure 7–7. *The result: an infinitely scrolling parallax background*

Repeat, Repeat, Repeat

Another neat trick deserves mention in this chapter. You can set any texture to repeat over a certain rectangular area. If you make this area big enough, you can have this texture repeat nearly endlessly. At least several thousand pixels or dozens of screen areas can be covered with a repeating texture, with no penalty to performance or memory usage.

The trick is to use the `GL_REPEAT` texture parameter supported by OpenGL. But it only works with square images that are exactly a power of 2, like 32×32 or 128×128 pixels. Listing 7–5 shows the code.

Listing 7–5. Repeating Background with GL_REPEAT

```

CGRect repeatRect = CGRectMake(-5000, -5000, 5000, 5000);
CCSprite* sprite = [CCSprite spriteWithFile:@"square.png" rect:repeatRect];
ccTexParams params =
{
    GL_LINEAR, // texture minifying function
    GL_LINEAR, // texture magnification function
    GL_REPEAT, // how texture should wrap along X coordinates
    GL_REPEAT, // how texture should wrap along Y coordinates
};
[sprite.texture setTexParameters:&params];

```

In this case, the sprite must be initialized with a rect that determines the area the sprite will occupy. The ccTexParams struct is initialized with the wrap parameters for the texture coordinates set to GL_REPEAT. Don't worry if that doesn't mean anything to you. These OpenGL parameters are then set on the sprite's texture using the CCTexture2D method setTexParameters.

The result is a tiled area repeating the same square.png image over and over again. If you move the sprite, the whole area covered by the repeatRect is moved. You could use this trick to remove the bottommost background stripe and replace it with a smaller image that simply repeats. I'll leave that up to you as an exercise.

A Virtual Joypad

Since the iOS devices all use a touchscreen for input and have no buttons, D-pads, or analog joypads like conventional mobile gaming devices, we need something called a *virtual joypad*. This emulates the behavior of digital or analog thumbsticks by allowing you to touch the screen where the digipad or thumbstick is displayed and move your finger over it to control the action on the screen. Buttons are also designated areas of the touchscreen that you can tap or hold to cause actions on the screen. Figure 7–8 shows a virtual joypad in action.



Figure 7–8. A skinned analog thumbstick and fire button created with SneakyInput

CAUTION: Virtual joypads resemble joypad controllers in looks but never in feel. The user is still touching a flat surface with no feedback to the fingers as to how far the virtual analog stick was moved or whether the virtual button has been pressed correctly. A number of players feel uneasy controlling a game with virtual joypads. It has become more or less common wisdom that a virtual joypad should have no more than two or three controller elements, usually a stick/pad (sometimes limited to two directions) and one or two buttons. Add more, and you're going to make your game exponentially harder to control.

Introducing SneakyInput

Over time, many a developer has faced the problem of implementing a virtual joypad. There are many ways to go about this, and even more ways to fail at it. But why spend time on that if there's a ready-to-use solution?

This is generally sound advice. Before you program anything that seems reasonably common that others might have worked on before, always check to see whether there is a general-purpose solution available that you can just use instead of having to spend a lot of time creating it yourself. In this case, SneakyInput is just too good to be ignored.

SneakyInput was created by Nick Pannuto, with skinning examples by CJ Hanson. SneakyInput is open source software and free to download, but if you like this product, please consider making a donation to Nick Pannuto at this link: <http://pledgie.com/campaigns/9124>.

The SneakyInput source code is hosted on GitHub, a social coding web site, at this link: <http://github.com/sneakyness/SneakyInput>.

It may not be immediately obvious what you have to do to download the source code from GitHub. When you browse to the web site and click any of the files, you'll see the actual source code displayed in your browser. But you want the full source code project, not individual files. What you do is locate the **Download Source** button in the upper-right corner of the web site. You then choose one of the archive types—it doesn't really matter which—and save the file to your computer and extract it.

At this point, you might want to open the SneakyInput Xcode project and compile it. Since SneakyInput comes together with cocos2d integrated into the project, it's possible that the cocos2d version used by SneakyInput may not be the most current version. If the project is up and running, you'll see an analog pad and a button as simple circles. The analog virtual thumbstick moves the "Hello World" label on the screen, while the button changes the label's colors on touch.

Figure 7-9 shows the SneakyInput sample project running in the iPhone Simulator. The sample project includes a virtual analog thumbstick that controls the "Hello World" label's position, as well as a button in the lower-right corner that changes the label's color when touched. The buttons aren't skinned; they use the `ColoredCircleSprite` class.



Figure 7–9. *The SneakyInput sample project*

But of course the sticks and buttons can also be skinned. Skinning, sometimes referred to as *texturing*, means using images from textures instead of flat colors to display the sticks and buttons, as you saw in Figure 7–8.

Integrating SneakyInput

I already have a project fully set up and functioning□ in this case `ScrollingWithJoy05`. I don't want to use the project provided by SneakyInput. How do I get it to work with my project?

This is an issue that isn't limited to SneakyInput but possibly any source code project you can download that comes already bundled with its own version of cocos2d. In most cases, and as long as the programming language of that source code is Objective-C, you only have to figure out which of the project's files are necessary and add them to your own project. There's no clear guideline, however, since every project is different.

I can tell you which files you need to add to your project regarding SneakyInput, however. It consists at its core of five classes:

- `SneakyButton` and `SneakyButtonSkinnedBase`
- `SneakyJoystick` and `SneakyJoystickSkinnedBase`
- `ColoredCircleSprite` (optional)

The remaining files are not needed but serve as references. Figure 7–10 shows the selection I've made in the `Add Files To ...` dialog. My rationale for not including certain classes was by first making an educated guess and then seeing whether I was right by compiling the project after having added the files I believed to be necessary.

The `HelloWorldScene` class is created by a cocos2d project template and most likely doesn't contain anything but example code. Of course, I already have an `AppDelegate` in my project, so I don't need to be adding SneakyInput's `AppDelegate` class□ it might be in conflict with the existing `AppDelegate`. Then there are two classes explicitly suffixed with `Example`, which indicates that these files are not core classes for SneakyInput but further example code.

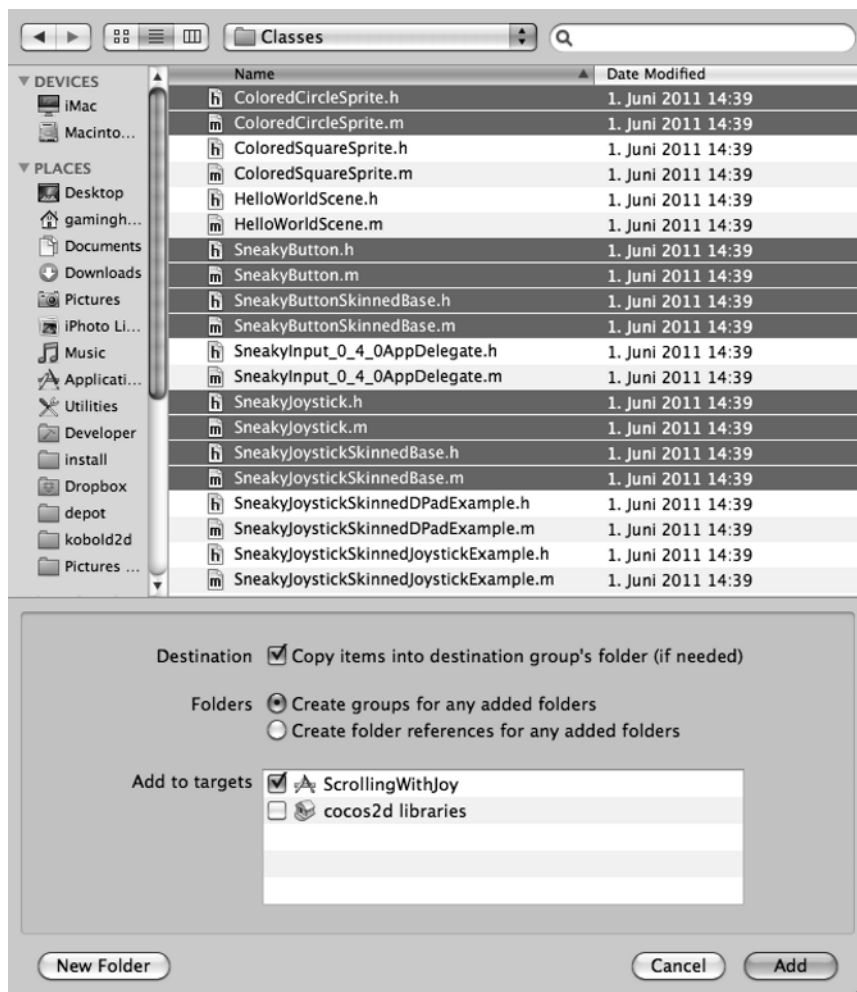


Figure 7-10. These files are needed to get *SneakyInput* to work in your own project; the other files are used only for example code.

Touch Button to Shoot

Let's try this. With the *SneakyInput* source code added to the project *ScrollingWithJoy05*, the first goal is to add a button that allows the player to shoot bullets from the player's ship. I'm going to add a separate *InputLayer* class to the project, which is derived from *CCLayer* and added to the *GameScene* class. In Listing 7-6, I updated the scene method to add the new *InputLayer* to it, and I'm giving both layers a tag just in case I need to identify them later.

Listing 7–6. Adding the InputLayer to the GameScene

```

+(id) scene
{
    CScene* scene = [CScene node];
    GameScene* layer = [GameScene node];
    [scene addChild:layer z:0 tag:GameSceneLayerTagGame];
    InputLayer* inputLayer = [InputLayer node];
    [scene addChild:inputLayer z:1 tag:GameSceneLayerTagInput];
    return scene;
}

```

The new tags are defined in the GameScene header file as follows:

```

typedef enum
{
    GameSceneLayerTagGame = 1,
    GameSceneLayerTagInput,
} GameSceneLayerTags;

```

With the InputLayer in place, the next step is to add the header files for the SneakyInput files we want to use to the InputLayer.h header file. I'm not picky, and we're probably going to use most of the classes, so I simply added all of the SneakyInput header files:

```

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

// SneakyInput headers
#import "ColoredCircleSprite.h"
#import "SneakyButton.h"
#import "SneakyButtonSkinnedBase.h"
#import "SneakyJoystick.h"
#import "SneakyJoystickSkinnedBase.h"

@interface InputLayer : CCLayer
{
    SneakyButton* fireButton;
}
@end

```

In addition, I added a SneakyButton member variable for easier access to the button I'm going to create now. This is done in the addFireButton method in Listing 7–7.

Listing 7–7. Creating a SneakyButton

```

-(void) addFireButton
{
    float buttonRadius = 80;
    CGSize screenSize = [[CCDirector sharedDirector] winSize];

    fireButton = [[[SneakyButton alloc] initWithRect:CGRectZero] autorelease];
    fireButton.radius = buttonRadius;
    fireButton.position = CGPointMake(screenSize.width - buttonRadius, buttonRadius);
    [self addChild:fireButton];
}

```

The CGRect parameter of the button's initWithRect method isn't used by SneakyButton, which is why I'm simply passing CGRectZero. The actual touch code uses the radius property to determine whether the button should react to the touch. The button in this

case should be neatly tucked into the lower-right corner. Subtracting the `buttonRadius` from the screen width and setting its height to `buttonRadius` places it exactly at the desired location.

TIP: The use of the `buttonRadius` variable allows you to change the radius of the button in one place. Otherwise, you would have to update several values in several places. This is not only extra work for a value you might want to tweak several times before you get it exactly the way you want, but it can also introduce subtle bugs, because you're a human and you tend to forget things, such as changing that one value over there. Suddenly the button is offset or worse, the input doesn't match the button's location.

The `InputLayer` class schedules the update method:

```
[self scheduleUpdate];
```

The update method is used to check whether the button is touched:

```
-(void) update:(ccTime)delta
{
    if (fireButton.active)
    {
        CCLOG(@"FIRE!!!");
    }
}
```

Instead of shooting a bullet, I wanted to keep things simple for now and simply log a successful button press. If you try the `ScrollingWithJoy05` project now, you'll notice that there isn't any button drawn. Yet when you touch the screen at the lower-right corner, you'll see the `▣FIRE!!!▣` message appear in the Debugger Console window. So, all is well and right, except that the button can't be seen▣ which we'll need to fix.

Skinning the Button

Eeww! No, it's not what you think. *Skinning* in computer graphics refers to giving an otherwise featureless object a texture or simply a different look. In this case, we want to actually see our button, so we need an image for that.

I also prefer to add `cocos2d`-style static initializers to external classes using a category, as done in the previous chapter. This helps avoid potentially forgetting to send `alloc` or `autorelease` messages to a `SneakyButton` object. I'll start with that by adding a `SneakyExtensions` class to the `ScrollingWithJoy06` project and then stripping the header file down to this:

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

// SneakyInput headers
#import "ColoredCircleSprite.h"
#import "SneakyButton.h"
#import "SneakyButtonSkinnedBase.h"
#import "SneakyJoystick.h"
#import "SneakyJoystickSkinnedBase.h"
```

```
@interface SneakyButton (Extension)
+(id) button;
+(id) buttonWithRect:(CGRect)rect;
@end
```

Once more, all the SneakyInput headers are added because I plan to make more categories for every SneakyInput class that doesn't conform to regular cocos2d initializers. In this case, a SneakyButton category named Extension is added, which adds two methods, named button and buttonWithRect. They are implemented as shown in Listing 7–8.

Listing 7–8. A Category with SneakyButton Autorelease Initializers

```
#import "SneakyExtensions.h"

@implementation SneakyButton (Extension)
+(id) button
{
    return [[[SneakyButton alloc] initWithRect:CGRectZero] autorelease];
}

+(id) buttonWithRect:(CGRect)rect
{
    return [[[SneakyButton alloc] initWithRect:rect] autorelease];
}
@end
```

They simply wrap the alloc and autorelease calls. Also, I decided to add a simple, parameterless button initializer because the CGRect parameter isn't really used anyway. This allows me to initialize the fireButton in a straightforward manner:

```
fireButton = [SneakyButton button];
```

It's just a little extra effort for more convenience and cleaner code. I'll be adding more convenience methods to SneakyExtensions without discussing them in the book, because the principle is the same.

Now my mind is at peace, and I can start skinning the button. I created four button images that are 100×100 pixels in size—twice the final button radius of 50 that I'm going with. The button images come in four variations: Default, Pressed, Activated, and Disabled. The default state is what the button looks like when it isn't pressed, which should make it obvious what the Pressed state is. The Activated state comes into play only for toggle buttons, meaning the toggle button is *active*, or *on*. The Disabled image is used if the button currently has no function. For example, when the ship's weapons are overheated and you can't shoot for a few seconds, you could disable the button, and it would show the Disabled image. For the shoot button, I only needed to use the Default and Pressed images. Listing 7–9 shows the updated addFireButton method.

Listing 7–9. Replacing Listing 7.7 with a Skinned Button

```
float buttonRadius = 50;
CGSize screenSize = [[CCDirector sharedDirector] winSize];

fireButton = [SneakyButton button];
fireButton.isHoldable = YES;
```

```

SneakyButtonSkinnedBase* skinFireButton = [SneakyButtonSkinnedBase skinnedButton];
skinFireButton.position = CGPointMake(screenSize.width - buttonRadius, buttonRadius);
skinFireButton.defaultSprite = [CCSprite spriteWithSpriteFrameName: ◀
    @"fire-button-idle.png"];
skinFireButton.pressSprite = [CCSprite spriteWithSpriteFrameName: ◀
    @"fire-button-pressed.png"];
skinFireButton.button = fireButton;
[self addChild:skinFireButton];

```

I initialized the `fireButton` as usual except that I made it holdable, which means you can keep it pressed down for a continuous stream of bullets. It also doesn't set the `radius` property anymore, since the images of the `SneakyButtonSkinnedBase` class determine the radius now. Keep in mind that I added a category to `SneakyButtonSkinnedBase` in the `SneakyExtension` source files created earlier. The `skinnedButton` initializer is in there and wraps the `alloc` and `autorelease` messages.

Instead of positioning the `fireButton`, the `skinned button` is now used to position the button on the screen; the actual button is updated accordingly by `SneakyButtonSkinnedBase`.

At this point, it makes sense to also write the firing code; Listing 7–10 shows the update method now sending the fire message to the `GameScene` class.

Listing 7–10. Shooting Bullets Whenever the Fire Button Is Active

```

-(void) update:(ccTime)delta
{
    totalTime += delta;

    if (fireButton.active && totalTime > nextShotTime)
    {
        nextShotTime = totalTime + 0.5f;

        GameScene* game = [GameScene sharedGameScene];
        [game shootBulletFromShip:[game defaultShip]];
    }

    // Allow faster shooting by quickly tapping the fire button
    if (fireButton.active == NO)
    {
        nextShotTime = 0;
    }
}

```

The two `ccTime` variables, `totalTime` and `nextShotTime`, are used to limit the number of bullets the ship will emit to two per second. If the fire button is not active (meaning that it isn't pressed), the `nextShotTime` is set to 0 so that the next time you press the button a shot is guaranteed to be fired. Tap the button quickly, and you should be able to shoot more bullets than with continuous fire.

Controlling the Action

You can't fly a ship without some form of input. This is where SneakyJoystick will give us a helping hand (I mean a helping virtual thumbstick). Behold, ScrollingWithJoy07 is here!

As usual, the first thing I did was add another Extension category to the new classes so I could initialize them like any other CCNode. For the joystick, I'll go right ahead and create a skinned one in the addJoystick method in Listing 7–11.

Listing 7–11. Adding a Skinned Joystick

```
-(void) addJoystick
{
    float stickRadius = 50;

    joystick = [SneakyJoystick joystickWithRect:CGRectMake(0, 0, stickRadius, ◀
        stickRadius)];
    joystick.autoCenter = YES;
    joystick.hasDeadzone = YES;
    joystick.deadRadius = 10;

    SneakyJoystickSkinnedBase* skinStick = [SneakyJoystickSkinnedBase ◀
        skinnedJoystick];
    skinStick.position = CGPointMake(stickRadius * 1.5f, stickRadius * 1.5f);
    skinStick.backgroundSprite = [CCSprite spriteWithSpriteFrameName: ◀
        @"joystick-back.png"];
    skinStick.backgroundSprite.color = ccYELLOW;
    skinStick.thumbSprite = [CCSprite spriteWithSpriteFrameName: ◀
        @"joystick-stick.png"];
    skinStick.joystick = joystick;
    [self addChild:skinStick];
}
```

The SneakyJoystick is initialized with a CGRect, and contrary to the SneakyButton, the CGRect is actually used to determine the joystick's radius. I set the joystick to autoCenter so that the thumb controller jumps back to the neutral position, like most real-world game controllers. The dead zone is also enabled; this is a small area defined by the deadRadius, in which you can move the thumb controller without any effect. This gives users a certain radius where they can keep the thumb controller centered. Without the dead zone, it would be almost impossible to center the thumb controller manually.

The SneakyJoystickSkinnedBase is positioned a small distance away from the edge of the screen. The button's position and size may not be ideal for the game, but it allows me to better demonstrate the controls. If you align the thumb controller with the screen edges, it's too easy to inadvertently move your finger off the touchscreen and thus lose control of the ship.

TIP: Gray areas are useful! I mean gray images like joystick-back.png. By using just grayscale colors, you can colorize the image using the `color` property of the sprite. You can create red, green, yellow, blue, magenta, and other colored versions of the same image, saving both download size and in-game memory. The only drawback is that it will be a flat color, so instead of shades of gray, your image will be using shades of red. This trick works best with images that are supposed to be shades of a single color.

You want the thumbstick on the screen to be used to control the ship, of course. As usual, the update method processes the input, as shown in Listing 7–12.

Listing 7–12. Moving the Ship Based on Joystick Input

```
-(void) update:(ccTime)delta
{

    // Moving the ship with the thumbstick
    GameScene* game = [GameScene sharedGameScene];
    Ship* ship = [game defaultShip];

    // Velocity must be scaled up by a factor that feels right
    CGPoint velocity = ccpMult(joystick.velocity, 7000 * delta);
    if (velocity.x != 0 && velocity.y != 0)
    {
        ship.position = CGPointMake(ship.position.x + velocity.x * delta,
                                     ship.position.y + velocity.y * delta);
    }
}
```

I have added a `defaultShip` accessor method to the `GameScene` class in the meantime so that the `InputLayer` has access to the ship. The `velocity` property of the joystick is used to change the ship's position but not without scaling it up. The velocity values are a tiny fraction of a pixel, so multiplying the velocity using cocos2d's `ccpMult` method, which takes a `CGPoint` and a float factor, is necessary for the joypad velocity to have a noticeable effect. The scale factor is arbitrary; it's just a value that feels good for this game.

To ensure smooth movement even if the update method is called at uneven intervals, the update method's `delta` parameter is factored in as well. The `delta` parameter is passed by cocos2d and contains the time elapsed since the update method was last called. This isn't strictly necessary, but it's good practice. If you don't do it, you run the risk that the ship will move more slowly whenever the framerate drops below 60 fps. Tiny things like these can be pretty annoying to players, and as game developers, our goal is the exact opposite of annoying players.

At this point, it is still possible to move the ship outside the screen area. I bet you'd like for the ship to stay on the screen as much as I do. And you may be tempted to add this code directly to the `InputLayer` where the ship's position is updated. That brings up a question: do you want to prevent the joystick input from moving the ship outside the screen, or do you want to prevent the ship from ever being able to move outside the screen? The latter

is the more general solution and preferable in this case. To do so, you only need to override the `setPosition` method in the `Ship` class, as shown in Listing 7–13.

Listing 7–13. Overriding the Ship's `setPosition` Method

```
// Override setPosition to keep the ship within bounds
-(void) setPosition:(CGPoint)pos
{
    CGSize screenSize = [[CCDirector sharedDirector] winSize];
    float halfWidth =.contentSize_.width * 0.5f;
    float halfHeight =.contentSize_.height * 0.5f;

    // Cap the position so the ship's sprite stays on the screen
    if (pos.x < halfWidth)
    {
        pos.x = halfWidth;
    }
    else if (pos.x > (screenSize.width - halfWidth))
    {
        pos.x = screenSize.width - halfWidth;
    }

    if (pos.y < halfHeight)
    {
        pos.y = halfHeight;
    }
    else if (pos.y > (screenSize.height - halfHeight))
    {
        pos.y = screenSize.height - halfHeight;
    }

    // Must call super with the new position
    [super setPosition:pos];
}
```

Every time the ship's position is updated, the preceding code performs the check to see whether the ship's sprite is still inside the screen boundaries. If not, then the X or Y coordinate is set to a distance of half the `contentSize` away from the respective screen border.

Because `position` is a property, the `setPosition` method gets called by this code:

```
ship.position = CGPointMake(200, 100);
```

The dot notation is shorthand for getter and setter messages to Objective-C properties, and it can be rewritten:

```
[ship setPosition:CGPointMake(200, 100)];
```

You can override other base class methods in this way to change the behavior of your game objects. For example, if an object is allowed to be rotated only between 0 and 180 degrees, you would override the `setRotation:(float)rotation` method and add the code to limit the rotation to it.

Digital Controls

You can turn the `SneakyJoystick` class into a digital controller as well, often referred to as a *D-pad*. The necessary code changes are minimal. You can find them in the `ScrollingWithJoy08` project or right here:

```
joystick = [SneakyJoystick joystickWithRect:CGRectMake(0, 0, stickRadius, stickRadius)];
joystick.autoCenter = YES;

// Now with fewer directions
joystick.isDPad = YES;
joystick.numberOfDirections = 8;
```

The dead zone properties have been removed—they are not needed for a digital controller. The joystick is set to digital controls by setting the `isDPad` property to `YES`. You can also define the number of directions. While D-pads regularly have four directions, in many games you can keep two directions pressed at the same time to have the character move in a diagonal direction. To achieve the same effect, the `numberOfDirections` property is set to 8. `SneakyJoystick` automatically ensures that these directions are evenly divided onto the thumbpad controller. Of course, you will get strange results if you set the number of directions to 6, but then again, maybe that's exactly what you need to travel across a hexagonal tile map.

Summary

In this chapter you learned several tricks to make an effective parallax scrolling background. Not only did you learn to scroll the background infinitely and without flickering edges, but you also learned how to properly separate the parallax layers so that `TexturePacker` can cut down on any unnecessary transparent areas while keeping the image's offset so you don't have to fumble with positioning the parallax layers.

The downside to this approach is that it is specific to a certain screen resolution. If you wanted to create an iPad version, for example, you could use the same process, except you'd have to create the image at a 1024×768 resolution. I'll leave that up to you as an exercise.

The second half of this chapter introduced you to `SneakyInput`, an open source project to add virtual thumbsticks and buttons to any `cocos2d` game. It may not be perfect, but it's good enough for most games, and it definitely beats writing your own virtual thumbstick code.

The ship is now controllable and stays within the screen's boundaries, and it's able to shoot with the press of a button—but the game is still lacking something. A shoot-'em-up isn't a shoot-'em-up without something to shoot down, is it? The next chapter addresses that lack.

Shoot em Up

What does a game of this kind need above all else? Something to shoot up and bullets to evade. In this chapter, you'll be adding enemies to the game and even a boss monster.

Both enemies and player will use the new `BulletCache` class to shoot a variety of bullets from the same pool. The cache class reuses inactive bullets to avoid constantly allocating and releasing bullets from memory. Likewise, enemies will use their own `EnemyCache` class because they, too, will appear in greater numbers on the screen.

Obviously the player will be able to shoot these enemies. I will also introduce the concept of component-based programming, which allows you to extend the game's actors in a modular way. Besides shooting and moving components, we also create a healthbar component for the boss monster. After all, a boss monster should not be an instant kill but require several hits before it is destroyed.

Adding the BulletCache Class

The `BulletCache` class is the one-stop shop for creating new bullets in the `ShootEmUp01` project. Previously all of this code was in the `GameScene` class, but it shouldn't be the responsibility of the `GameScene` to manage and create new bullets. Listing 8-1 shows the new `BulletCache` header file, and it now contains the `CCSpriteBatchNode` and the inactive bullets counter.

Listing 8-1. *The @interface of the BulletCache Class*

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

@interface BulletCache : CCNode
{
    CCSpriteBatchNode* batch;
    int nextInactiveBullet;
}

-(void) shootBulletAt:(CGPoint)startPosition velocity:(CGPoint)velocity
                    frameName:(NSString*)frameName;

@end
```

To refactor the bullet-shooting code out of the `GameScene` class, I needed to move both the initialization and the method to shoot bullets to the new `BulletCache` class (Listing 8–2). Along the way, I decided to keep the `CCSpriteBatchNode` in a member variable instead of using the `CCNode getChildByTag` method every time I need the sprite batch object. It's a minor performance optimization. Since I'll be adding the `BulletCache` class as child to the `GameScene`, I can simply add the sprite batch node to the `BulletCache` class.

NOTE: There's little harm in increasing the depth of the scene hierarchy by adding an in-between `CCNode` like `BulletCache`. If you're concerned about scene hierarchy depth, the alternative would be to add the sprite batch node to the `GameScene` class as usual and use an accessor method to get to the sprite batch node in the `BulletCache` class. But the additional function call overhead could possibly void any performance gain. If in doubt, always prefer to make your code more readable and then refactor later to improve performance where necessary, and only where necessary.

Listing 8–2. *The `BulletCache` Maintains a Pool of Bullets for Reuse*

```
#import "BulletCache.h"
#import "Bullet.h"

@implementation BulletCache

-(id) init
{
    if ((self = [super init]))
    {
        // get any bullet image from the texture atlas we're using
        CCSpriteFrame* bulletFrame = [[CCSpriteFrameCache sharedSpriteFrameCache]
            spriteFrameByName:@"bullet.png"];

        // use the bullet's texture
        batch = [CCSpriteBatchNode batchNodeWithTexture:bulletFrame.texture];
        [self addChild:batch];

        // Create a number of bullets up front and re-use them
        for (int i = 0; i < 200; i++)
        {
            Bullet* bullet = [Bullet bullet];
            bullet.visible = NO;
            [batch addChild:bullet];
        }
    }

    return self;
}

-(void) shootBulletAt:(CGPoint)startPosition velocity:(CGPoint)velocity
    frameName:(NSString*)frameName
{
    CCArrary* bullets = [batch children];
    CCNode* node = [bullets objectAtIndex:nextInactiveBullet];
    NSAssert([node isKindOfClass:[Bullet class]], @"not a Bullet!");
```

```

Bullet* bullet = (Bullet*)node;
[bullet shootBulletAt:startPosition velocity:velocity frameName:frameName];

nextInactiveBullet++;
if (nextInactiveBullet >= [bullets count])
{
    nextInactiveBullet = 0;
}
}
@end

```

The shootBulletAt method has changed the most as you can see. It now takes three parameters, startPosition, velocity, and frameName, instead of a pointer to the Ship class. It then passes on these parameters to the Bullet class's shootBulletAt method, which I had to refactor as well:

```

-(void) shootBulletAt:(CGPoint)startPosition velocity:(CGPoint)vel
                    frameName:(NSString*)frameName
{
    self.velocity = vel;
    self.position = startPosition;
    self.visible = YES;

    // change the bullet's texture by setting a different SpriteFrame to be displayed
    CCSpriteFrame* frame = [[CCSpriteFrameCache sharedSpriteFrameCache]
        spriteFrameByName:frameName];
    [self setDisplayFrame:frame];

    [self unscheduleUpdate];
    [self scheduleUpdate];
}

```

Both velocity and position are now directly assigned to the bullet. This means that the code calling the shootBulletAt method has to determine the position, direction, and speed of the bullet. This is exactly what I wanted: full flexibility for shooting bullets, including changing the bullet's sprite frame by using the setDisplayFrame method. Since the bullets are all in the same texture atlas and thus use the same texture, all it needs to do to change which bullet is displayed is to set the desired sprite frame. In effect, this is simply going to render a different part of the texture and comes at no extra cost.

While I was in the Bullet class, I also fixed the boundary issues the bullets would have had that only bullets moving outside the right side of the screen would have been set invisible and put back on the waiting list. By using the CGRectIntersectsRect check with the bullet's boundingBox and the screenRect in the update method, any bullet having moved completely outside the screen area will be marked for reuse:

```

// When the bullet leaves the screen, make it invisible
if (CGRectIntersectsRect([self boundingBox], screenRect) == NO)
{
}

```

The screenRect variable itself is now stored for convenience and performance reasons as a static variable, so it can be accessed by other classes and doesn't need to be re-created

for each use. Static variables like `screenRect` are available in the class implementation file where they are declared. They are like global variables to the class; any class instance can read and modify the variable, as opposed to class member variables, which are local to every class instance. Since the screen size never changes during game play and all `Bullet` instances need to use this variable, it makes sense to store it in a static variable for all class instances. The first bullet to be initialized sets the `screenRect` variable. The `CGRectIsEmpty` method checks whether the `screenRect` variable is still uninitialized; since the variable is static, I want to initialize it only once.

```
static CGRect screenRect;
```

```
// make sure to initialize the screen rect only once
if (CGRectIsEmpty(screenRect))
{
    CGSize screenSize = [[CCDirector sharedDirector] winSize];
    screenRect = CGRectMake(0, 0, screenSize.width, screenSize.height);
}
```

With these changes implemented, what's left is to clean up the `GameScene` by removing any of the methods and member variables previously used for shooting bullets. Specifically, I need to replace the `CCSpriteBatchNode` initialization with the initialization of the `BulletCache` class:

```
BulletCache* bulletCache = [BulletCache node];
[self addChild:bulletCache z:1 tag:GameSceneNodeTagBulletCache];
```

I also need to add a `bulletCache` accessor so other classes can access the `BulletCache` instance through the `GameScene`:

```
-(BulletCache*) bulletCache
{
    CCNode* node = [self getChildByTag:GameSceneNodeTagBulletCache];
    NSAssert([node isKindOfClass:[BulletCache class]], @"not a BulletCache");
    return (BulletCache*)node;
}
```

The `InputLayer` can now use the new `BulletCache` class and uses it to shoot the player's bullets. The bullet properties, such as starting position, velocity, and the sprite frame to use, are now passed by the shooting code in the update method of the `InputLayer`:

```
if (fireButton.active && totalTime > nextShotTime)
{
    nextShotTime = totalTime + 0.4f;

    GameScene* game = [GameScene sharedGameScene];
    Ship* ship = [game defaultShip];
    BulletCache* bulletCache = [game bulletCache];

    // Set the position, velocity and spriteframe before shooting
    CGPoint shotPos = CGPointMake(ship.position.x + 45, ship.position.y ... 19);

    float spread = (CCRANDOM_0_1() - 0.5f) * 0.5f;
    CGPoint velocity = CGPointMake(200, spread * 50);
    [bulletCache shootBulletAt:shotPos velocity:velocity frameName:@"bullet.png"];
}
```


This short refactoring session adds much-needed flexibility to shooting bullets. I'm sure you can imagine how enemies can now use the very same code to shoot their own bullets.

What About Enemies?

At this point, there's only a fuzzy idea about what the enemies are, what they do, and what their behavior will be. That's the thing with enemies—you never quite know what they're up to.

In the case of games, that means going back to the drawing board, planning out what you want the enemies to do, and then deducing from that plan what you need to program. Contrary to real life, you have full control over the enemies. Doesn't that make you feel powerful? But before you or anyone can have fun, you need to come up with a plan for world domination.

I already created the graphics for three different types of enemies. At this point, I know only that at least one of them is supposed to be a boss monster. Take a look at Figure 8-1 and try to imagine what these enemies could be up to.

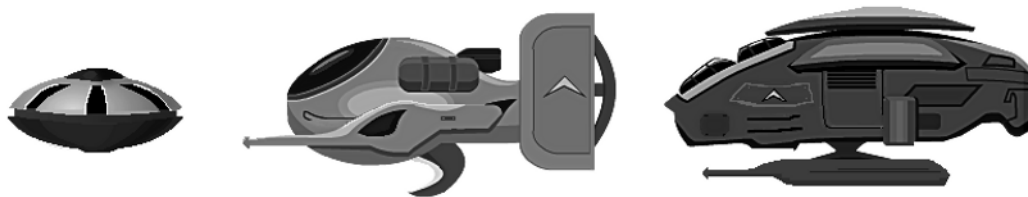


Figure 8-1. *The graphics used as the game's enemy characters*

Before you start programming, you should have a good understanding of which behaviors the enemies will have in common so that you program those parts only once. Eliminating code duplication is the single most important goal of clean code design.

Let's see what we know for sure is common to all enemies:

- Shoots bullets
- Has logic that determines when and where to shoot what bullet
- Can be hit by player's bullets
- Cannot be hit by other enemy's bullets
- Can take one or more hits (has health)
- Has a specific behavior and movement pattern
- Has a specific behavior or animation when destroyed
- Will appear outside the screen area and move inside
- Will not leave the screen area once inside

When you look at that list, you may notice that some of these attributes also apply to the player's ship. It certainly can shoot bullets, we may want it to sustain multiple hits, and it should have a specific behavior or animation when destroyed. It makes sense to consider the player's ship as just a special type of enemy and take it into consideration as well.

Looking at this feature set, I see three possible approaches. I could create one class that contains all the code for the ship, the enemies, and the boss monster. Certain parts of the code would run conditionally, depending on the type of enemy. For example, the shooting code may have different code paths for each type of game object. With a limited number of different objects, this approach works reasonably well□ but it doesn't scale. As you add more and more types of game objects, you end up with a bigger and fatter class containing all of the game logic code. Any change to any part of that class has the potential to cause undesirable side effects in enemies' or even the player ship's behavior. Determining which code path to execute depending on a type variable is quite reminiscent of pure C programming and doesn't make use of the object-oriented nature of the Objective-C programming language. But if used judiciously, it's a very powerful concept even today.

The second approach is to create a class hierarchy with the Entity class as the base class and then derive a ship, two monsters, and a boss monster class from it. This is what a lot of programmers actually do, and it also works reasonably well for a small number of game objects. But in essence, it's little different from the first approach, in that common code often ends up piling up in the base Entity class when it is needed by some of the subclasses, but not all of them. It takes a turn for the worse as soon as that code in the Entity class starts to add switches based on the type of the enemy, to skip parts of the code or execute code paths specific to that type of enemy. That's the same problem of the first C-style programming approach. With a little care, you can make sure the code specific to an enemy is part of that enemy's class, but it's all too easy to end up making most changes in the Entity class itself.

The third option is to use a component system. This means that individual code paths are separated from the Entity class hierarchy and only added to the subclasses that need the components, such as a healthbar component. Since component-based development would justify a book on its own and is likely to be overkill for a small project like this shoot-'em-up game, I'll use a mixture of the class hierarchy approach and component design to at least give you an idea how compositing game objects out of individual parts works in principle and what the benefits are.

I do want to point out that there is no one best approach to code design. Certain choices are entirely subjective and depend on personal preferences and experience. Working code is often preferable to clean code if you're willing to refactor your codebase often as you learn more about the game you're making. Experience allows you to make more of these decisions up front in the planning stage, enabling you to create more complex games faster. So if that's your goal, start by making and completing smaller games, and slowly push yourself to new limits and new challenges. It's a learning process, and unfortunately the easiest way to kill your motivation is to be over-ambitious. There's a reason why every seasoned game programmer will tell a beginner to start simple and to re-create classic arcade games like Tetris, Pac-Man, or Asteroids first.

The Entity Class Hierarchy

For the base class, I created the Entity class in the ShootEmUp02 project. Entity is a generic class derived from CCSprite, which contains only the setPosition code of the Ship class to keep all instances of Entity inside the screen. I made just a small improvement to this code so that objects outside the screen area are allowed to move inside, but once inside, they can no longer leave the screen area. In this shoot-'em-up example game, the enemies won't pass by you; they'll stop mid-screen in order to illustrate the EnemyCache, introduced shortly. The screen area check simply checks whether the screen rectangle fully contains the sprite's boundingBox, and only if that's the case will the code to keep the sprite inside the screen rectangle run:

```
-(void) setPosition:(CGPoint)pos
{
    // If the current position is outside the screen no adjustments should be made!
    // This allows entities to move into the screen from outside.
    if ([self isOutsideScreenArea])
    {

    }

    [super setPosition:pos];
}
```

The function isOutsideScreenArea is implemented as follows:

```
-(BOOL) isOutsideScreenArea
{
    return (CGRectContainsRect([GameScene screenRect], [self boundingBox]));
}
```

The Ship class has been replaced with ShipEntity. Since the Entity base class now contains the setPosition code, the only thing left in ShipEntity is the initWithShipImage method. It's the same as before, so I won't re-create it here.

The EnemyEntity Class

I do need to go more in depth with the EnemyEntity class and what it does, starting with the header file in Listing 8-3.

Listing 8-3. *The @interface of the EnemyEntity Class*

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

typedef enum
{
    EnemyTypeUFO = 0,
    EnemyTypeCruiser,
    EnemyTypeBoss,

    EnemyType_MAX,
} EnemyTypes;
```

```

@interface EnemyEntity : Entity
{
    EnemyTypes type;
}

+(id) enemyWithType:(EnemyTypes)enemyType;
+(int) getSpawnFrequencyForEnemyType:(EnemyTypes)enemyType;
-(void) spawn;
@end

```

There is nothing too exciting here. The `EnemyTypes` enum is used to differentiate between the three different types of enemies currently supported, with `EnemyType_MAX` used as the upper limit for loops, as you'll see soon. The `EnemyEntity` class has a member variable that stores the type, so that I can use switch statements to branch the code depending on the type of enemy as needed.

The implementation of `EnemyEntity` contains a lot of code I'd like to discuss, so I'll split the discussion into several topics and present only the relevant code, beginning with the `initWithType` method in Listing 8-4.

Listing 8-4. Initializing an Enemy with a Type

```

-(id) initWithType:(EnemyTypes)enemyType
{
    type = enemyType;

    NSString* enemyFrameName;
    NSString* bulletFrameName;
    float shootFrequency = 6.0f;
    switch (type)
    {
        case EnemyTypeUFO:
            enemyFrameName = @"monster-a.png";
            bulletFrameName = @"shot-a.png";
            break;
        case EnemyTypeCruiser:
            enemyFrameName = @"monster-b.png";
            bulletFrameName = @"shot-b.png";
            shootFrequency = 1.0f;
            break;
        case EnemyTypeBoss:
            enemyFrameName = @"monster-c.png";
            bulletFrameName = @"shot-c.png";
            shootFrequency = 2.0f;
            break;

        default:
            [NSEException exceptionWithName:@"EnemyEntity Exception"
             reason:@"unhandled enemy type"
             userInfo:nil];
    }

    if ((self = [super initWithSpriteFrameName:enemyFrameName]))
    {
        // Create the game logic components
        [self addChild:[StandardMoveComponent node]];
    }
}

```



```

        // spawn one enemy immediately
        [self spawn];
    }
}

+(int) getSpawnFrequencyForEnemyType:(EnemyTypes)enemyType
{
    NSAssert(enemyType < EnemyType_MAX, @"invalid enemy type");
    NSNumber* number = [spawnFrequency objectAtIndex:enemyType];
    return [number intValue];
}

-(void) dealloc
{
    [spawnFrequency release];
    spawnFrequency = nil;

    [super dealloc];
}

```

We store the spawn frequency values for each type of enemy in a static `spawnFrequency` `CCArray`. It's a static variable because the spawn frequency isn't needed for each enemy but only for each enemy type. The first `EnemyEntity` instance that executes the `initSpawnFrequency` method will find that the `spawnFrequency` `CCArray` is `nil` and so initializes it.

Since a `CCArray` can store only objects and not primitive data types like integers, the values have to be wrapped into an `NSNumber` class using the `numberWithInt` initializer. I chose to use `insertObject` here instead of `addObject` because it not only ensures that the values will have the same index as the enemy type defined in the enum, but it also tells any other programmer looking at this code that the index used has a meaning. In this case, the index is synonymous with the enemy type. Although it's technically unnecessary to specify the index here, it helps to show which value is used for which enemy type.

Of course, the `dealloc` method releases the `spawnFrequency` `CCArray`, and it also sets it to `nil`, which is very important. Being a static variable, the first `EnemyEntity` object to run its `dealloc` method will release the `spawnFrequency`'s memory. If it wasn't set to `nil` immediately thereafter, the next `EnemyEntity` running its `dealloc` method would try the same and thus over-release the `spawnFrequency` `CCArray`, leading to a crash. On the other hand, if the `spawnFrequency` variable is `nil`, any message sent to it, like the `release` message, will simply be ignored. I said it before, but it can't be repeated often enough: sending messages to `nil` objects is perfectly fine in Objective-C; the message will simply be ignored.

Spawning an entity is done by the `spawn` method:

```

-(void) spawn
{
    // Select a spawn location just outside the right side of the screen
    CGRect screenRect = [GameScene screenRect];
    CGSize spriteSize = [self contentSize];
    float xPos = screenRect.size.width + spriteSize.width * 0.5f;

```

```

float yPos = CCRANDOM_0_1() * (screenRect.size.height - spriteSize.height) +
    spriteSize.height * 0.5f;
self.position = CGPointMake(xPos, yPos);

// Finally set yourself to be visible, this also flag the enemy as "in use"
self.visible = YES;
}

```

Because an `EnemyCache` is used to create all instances of enemies up front, the whole spawning process is limited to choosing a random y position just outside the right side of the screen and then setting the `EnemyEntity` sprite to be visible. The visible status is used elsewhere in the project, specifically by component classes, to determine whether the `EnemyEntity` is currently in use. If it's not visible, it can be spawned to make it visible, but it should run its game logic code only while it's visible.

The EnemyCache Class

I just mentioned the `EnemyCache` class. By its name, it should remind you of the `BulletCache` class, which also holds a number of pre-initialized objects for fast and easy reuse. This avoids creating and releasing objects during game play that can be a source of minor performance hiccups. Especially for action games, those small glitches can have a devastating effect on the player's experience. With that said, let's look at the unspectacular header file of the `EnemyCache` in Listing 8–6.

Listing 8–6. *The @interface of the EnemyCache Class*

```

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

@interface EnemyCache : CCNode
{
    CCSpriteBatchNode* batch;
    CCArrary* enemies;

    int updateCount;
}
@end

```

After the `spriteBatch`, which contains all the enemy sprites, there's an `enemies CCArrary` that stores a list of enemies of each type. The `updateCount` variable is increased every frame and used to spawn enemies at regular intervals. The `init` method of the `EnemyCache` is quite similar to the `BulletCache` `init` with its initialization of the `CCSpriteBatchNode`:

```

-(id) init
{
    if ((self = [super init]))
    {
        // get any image from the texture atlas we're using
        CCSpriteFrame* frame = [[CCSpriteFrameCache sharedSpriteFrameCache]
            spriteFrameByName:@"monster-a.png"];
        batch = [CCSpriteBatchNode batchNodeWithTexture:frame.texture];
        [self addChild:batch];
    }
}

```

```

        [self initEnemies];
        [self scheduleUpdate];
    }

    return self;
}

```

But since the code for initializing the enemies is a bit more complex, I extracted it into its own method, as shown in Listing 8–7.

Listing 8–7. Initializing the Pool of Enemies for Later Reuse

```

-(void) initEnemies
{
    // create the enemies array containing further arrays for each type
    enemies = [[CCArray alloc] initWithCapacity:EnemyType_MAX];

    // create the arrays for each type
    for (int i = 0; i < EnemyType_MAX; i++)
    {
        // depending on enemy type the array capacity is set
        // to hold the desired number of enemies
        int capacity;
        switch (i)
        {
            case EnemyTypeUFO:
                capacity = 6;
                break;
            case EnemyTypeCruiser:
                capacity = 3;
                break;
            case EnemyTypeBoss:
                capacity = 1;
                break;

            default:
                [NSException exceptionWithName:@"EnemyCache Exception"
                                     reason:@"unhandled enemy type"
                                   userInfo:nil];

                break;
        }

        // no alloc needed since the enemies array will retain anything added to it
        CCArray* enemiesOfType = [CCArray arrayWithCapacity:capacity];
        [enemies addObject:enemiesOfType];
    }

    for (int i = 0; i < EnemyType_MAX; i++)
    {
        CCArray* enemiesOfType = [enemies objectAtIndex:i];
        int numEnemiesOfType = [enemiesOfType capacity];

        for (int j = 0; j < numEnemiesOfType; j++)
        {
            EnemyEntity* enemy = [EnemyEntity enemyWithType:i];
            [batch addChild:enemy z:0 tag:i];
            [enemiesOfType addObject:enemy];
        }
    }
}

```



```

    }
}

-(void) dealloc
{
    [enemies release];
    [super dealloc];
}

```

The interesting part here is that the `CCArray* enemies` itself contains more `CCArray*` objects, one per enemy type. It's what's called a two-dimensional array. The member variable `enemies` requires the use of `alloc`, because otherwise its memory would be freed after leaving the `initEnemies` method. In contrast, the `CCArray*` objects added to the `enemies` `CCArray` do not need to be created using `alloc`, because the `enemies` `CCArray` retains the objects added to it.

The initial capacity of each `enemiesOfType` `CCArray` also determines how many enemies of that type can be on the screen at once. In this way, you can keep the maximum number of enemies on the screen under control. Each `enemiesOfType` `CCArray` is then added to `enemies` `CCArray` using `addObject`, just like any other object. If you want, you can create deep hierarchies in this way. As a matter of fact, the `cocos2d` node hierarchy is built on `CCNode` classes containing `CCArray*` `children` member variables that can contain more `CCNode` classes, and so on.

I split the array initialization and the creation of enemies into separate loops even though both could be done in the same loop. They are simply different tasks and should be kept separate. The additional overhead of going through all enemy types once again is negligible.

Based on the initial capacity set in the `CCArray` initialization loop, the desired number of enemies is created, added to the `CCSpriteBatchNode`, and then added to the corresponding `enemiesOfType` `CCArray`. While the enemies could also be accessed through the `CCSpriteBatchNode`, keeping references to the enemy entities in separate arrays makes it easier to process them during later activities such as spawning, as shown in Listing 8–8.

Listing 8–8. Spawning Enemies

```

-(void) spawnEnemyOfType:(EnemyTypes)enemyType
{
    CCArray* enemiesOfType = [enemies objectAtIndex:enemyType];

    EnemyEntity* enemy;
    CCARRAY_FOREACH(enemiesOfType, enemy)
    {
        // find the first free enemy and respawn it
        if (enemy.visible == NO)
        {
            CLOG(@"spawn enemy type %i", enemyType);
            [enemy spawn];
            break;
        }
    }
}

```

```

}

-(void) update:(ccTime)delta
{
    updateCount++;

    for (int i = EnemyType_MAX - 1; i >= 0; i--)
    {
        int spawnFrequency = [EnemyEntity getSpawnFrequencyForEnemyType:i];

        if (updateCount % spawnFrequency == 0)
        {
            [self spawnEnemyOfType:i];
            break;
        }
    }
}

```

The update method increases a simple update counter. It does not take into effect the actual time passed, but since the variances are typically minimal, it's a fair trade-off to make life a bit easier. This for loop oddly starts at `EnemyType_MAX - 1` and runs until `i` is negative. The only purpose for this is that higher-numbered `EnemyTypes` have spawn precedence over lower-numbered `EnemyTypes`. If a boss monster is scheduled to appear at the same time as a cruiser, the boss will be spawned. Otherwise, it could happen that the cruiser takes the boss's spawn slot by trying to spawn at the same time, blocking the boss from ever spawning. It's a side effect of the spawning logic, and I leave it up to you to extend and improve this code, because you'll probably have to do anyway if you decide to write your own version of a classic shoot-'em-up game.

The `spawnFrequency` is obtained from `EnemyEntity`'s `getSpawnFrequencyForEnemyType` method:

```

+(int) getSpawnFrequencyForEnemyType:(EnemyTypes)enemyType
{
    NSAssert(enemyType < EnemyType_MAX, @"invalid enemy type");
    NSNumber* number = [spawnFrequency objectAtIndex:index:enemyType];
    return [number intValue];
}

```

First the method asserts that the `enemyType` number is actually within the defined range. Then the `NSNumber` object for that `enemyType` is obtained and returned as `intValue`.

The modulo operator `%` returns the remainder left after the division of the two operands `updateCount` and `spawnFrequency`. This means an enemy is spawned only when `updateCount` can be evenly divided by `spawnFrequency`, resulting in a remainder of 0.

The `spawnEnemyOfType` method then gets the `CCArray` from the `enemies CCArray`, which contains the list of `enemiesOfType`, another `CCArray`. You can now iterate over only the desired enemy types, rather than having to go through all sprites added to the `CCSpriteBatchNode`. As soon as one enemy is found that is not visible, its spawn method is called. If all enemies of that type are visible, the maximum number of enemies is currently on-screen and no further enemies are spawned, effectively limiting the number of enemies of a type on the screen at any time.

The Component Classes

Component classes are intended as plug-ins for game logic. If you add a component to an entity, the entity will execute the behavior of the component: moving, shooting, animating, showing a healthbar, and so on. The big benefit is that you program these components to work generically almost automatically, because they interact with the parent CCNode class and should make as few assumptions about the parent class as possible. Of course, in some cases a component requires the parent to be an EnemyEntity class, but then you can still use it with any EnemyEntity.

Component classes can also be configured depending on the class that's using the component. As an example for component classes, let's take a look at the StandardShootComponent initialization in the EnemyEntity class:

```
StandardShootComponent* shootComponent = [StandardShootComponent node];
shootComponent.shootFrequency = shootFrequency;
shootComponent.bulletFrameName = bulletFrameName;
[self addChild:shootComponent];
```

The variables shootFrequency and bulletFrameName have been set previously based on the EnemyType. By adding the StandardShootComponent to the EnemyEntity class, the enemy will shoot specific bullets at certain intervals. Since this component makes no assumptions about the parent class, you could even add an instance of it to the ShipEntity class and have your player's ship shoot automatically at specific intervals! Or by simply activating and deactivating specialized shooting components, you can create the effect of changing weapons for the player with very little code. You just program the shooting code in isolation and then plug it into a game entity and add some parameters to it. The only logic left for programming the switching of weapons is simply when to deactivate which components. What's more, you can reuse these components in other games if they make sense. Components are great for writing reusable code and are a standard mechanism in many, many game engines. If you'd like to learn more about game components, please refer to this blog post on my web site: www.learn-cocos2d.com/2010/06/prefer-composition-inheritance/.

Let's look at the StandardShootComponent's source code, starting with the header file:

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

@interface StandardShootComponent : CCSprite
{
    float updateCount;
    float shootFrequency;
    NSString* bulletFrameName;
}

@property (nonatomic) float shootFrequency;
@property (nonatomic, copy) NSString* bulletFrameName;

@end
```

There are two things of note. The first is that `StandardShootComponent` is derived from `CCSprite`, even though it doesn't use any texture. That is a workaround because all `EnemyEntity` objects are added to a `CCSpriteBatchNode`, which can contain only `CCSprite`-based objects. This also extends to any child node of the `EnemyEntity` class; thus, the `StandardShootComponent` needs to inherit from `CCSprite` to satisfy the requirement of the `CCSpriteBatchNode`.

Next there's an `NSString*` pointer, `bulletFrameName`, with a corresponding `@property`. If you look closely, you'll notice the keyword `copy` in the `@property` definition. That means assigning an `NSString` to this property will create a copy of it. This is important for strings since they are generally autoreleased objects, and we want to make sure this string is ours. We could also retain it, but the problem is that the source string could be an `NSMutableString` that can be changed. And that would also change the `bulletFrameName`, which would be undesirable in this case. Of course, with the `copy` keyword comes the responsibility to release the `bulletFrameName` memory on `dealloc`, as shown in the implementation in Listing 8–9.

Listing 8–9. *The StandardShootComponent Implementation in Its Entirety*

```
#import "StandardShootComponent.h"
#import "BulletCache.h"
#import "GameScene.h"

@implementation StandardShootComponent

@synthesize shootFrequency;
@synthesize bulletFrameName;

-(id) init
{
    if ((self = [super init]))
    {
        [self scheduleUpdate];
    }

    return self;
}

-(void) dealloc
{
    [bulletFrameName release];
    [super dealloc];
}

-(void) update:(ccTime)delta
{
    if (self.parent.visible)
    {
        updateCount += delta;

        if (updateCount >= shootFrequency)
        {
            updateCount = 0;

            GameState* game = [GameState sharedInstance];
```

```

CGPoint startPos = ccpSub(self.parent.position,
    CGPointMake(self.parent.contentSize.width * 0.5f, 0));
[game.bulletCache shootBulletFrom:startPos
    velocity:CGPointMake(-200, 0)
    rameName:bulletFrameName];
    }
}
@end

```

The actual shooting code first checks whether the parent is visible, because if the parent isn't visible, the code obviously shouldn't shoot. The `BulletCache` is what shoots the bullet, using the `bulletFrameName` provided to the component and a fixed velocity. For the start position, the component's position itself is irrelevant. Instead, the parent position and `contentSize` properties are used to calculate the correct starting position: in this case, at the left side of the enemy's sprite.

This bullet `startPos` works reasonably well for regular enemies but may need tweaking for the boss enemy. I'll leave it up to you to add another property to this component with which to set the bullet `startPosition`. Alternatively, you could also create a separate `BossShootComponent` and add this only to boss enemies to create more complex shooting patterns. The same goes for `StandardMoveComponents`, which for the boss might require hovering at a certain position at the right side of the screen.

Shooting Things

I almost forgot—you actually want to shoot the enemies, right? Well, in the `ShootEmUp03` project, you can!

The ideal starting point to check if a bullet has hit something is in the `BulletCache` class. I've added just the method to do that. Actually, I've added three methods, two of them public; the third is private to the class and combines the common code (see Listing 8–10). The idea behind using the two wrapper methods, `isPlayerBulletCollidingWithRect` and `isEnemyBulletCollidingWithRect`, is to hide the internal detail of determining which kinds of bullets are used for collision checks. You could also expose the `usePlayerBullets` parameter to other classes, but doing so would only make it harder to eventually change the parameter from a `bool` to an `enum`, in case you want to introduce a third type of bullet.

Listing 8–10. Checking for Collisions with Bullets

```

-(bool) isPlayerBulletCollidingWithRect:(CGRect)rect
{
    return [self isBulletCollidingWithRect:rect usePlayerBullets:YES];
}

-(bool) isEnemyBulletCollidingWithRect:(CGRect)rect
{
    return [self isBulletCollidingWithRect:rect usePlayerBullets:NO];
}

-(bool) isBulletCollidingWithRect:(CGRect)rect usePlayerBullets:(bool)usePlayerBullets

```

```

{
    bool isColliding = NO;

    Bullet* bullet;
    CCARRAY_FOREACH([batch children], bullet)
    {
        if (bullet.visible && usePlayerBullets == bullet.isPlayerBullet)
        {
            if (CGRectIntersectsRect([bullet boundingBox], rect))
            {
                isColliding = YES;

                // remove the bullet
                bullet.visible = NO;
                break;
            }
        }
    }

    return isColliding;
}

```

Only visible bullets can collide, of course, and by checking the bullet's `isPlayerBullet` property, we ensure that enemies can't shoot themselves. The actual collision test is a simple `CGRectIntersectsRect` test, and if the bullet has actually hit something, the bullet itself is also set to be invisible to make it disappear.

The `EnemyCache` class holding all `EnemyEntity` objects is the perfect place to call this method to check whether any enemy was hit by a player bullet. The class has a new `checkForBulletCollisions` method, which is called from its update method:

```

-(void) checkForBulletCollisions
{
    EnemyEntity* enemy;
    CCARRAY_FOREACH([batch children], enemy)
    {
        if (enemy.visible)
        {
            BulletCache* bulletCache = [[GameScene sharedGameScene] bulletCache];
            CGRect bbox = [enemy boundingBox];
            if ([bulletCache isPlayerBulletCollidingWithRect:(bbox)])
            {
                // This enemy got hit ...
                [enemy gotHit];
            }
        }
    }
}

```

Here again it's convenient to be able to iterate over all the enemy entities in the game, skipping those that are currently not visible. Using each enemy's `boundingBox` to check with the `BulletCache isPlayerBulletCollidingWithRect` method, we can quickly find if an enemy got hit by a player bullet. If so, the `EnemyEntity` method `gotHit` is called, which simply sets the enemy to be invisible.

I'll leave it as an exercise for you to implement the player's ship being hit by enemy bullets. You'll have to schedule an update method in `ShipEntity` and then implement the `checkForBulletCollisions` method and call it from the update method. You'll have to change the call to `isPlayerBulletCollidingWithRect` to `isEnemyBulletCollidingWithRect` and decide how you want to react to being hit by a bullet, for example by playing a sound effect.

A Healthbar for the Boss

The boss monster shouldn't be an easy, one-hit kill. It also should give the player feedback about its health by displaying a healthbar that decreases with each hit. The first step toward a healthbar is adding the `hitPoints` member variable to the `EnemyEntity` class, which tells us how many hits the enemy can take before being destroyed. The `initialHitPoints` variable stores the maximum hit points, since after an enemy gets killed we need to be able to restore the original hit points. This is the changed header file in the `EnemyEntity` class:

```
@interface EnemyEntity : Entity
{
    EnemyTypes type;
    int initialHitPoints;
    int hitPoints;
}

@property (readonly, nonatomic) int hitPoints;
```

To display the healthbar, a component class is ideal because it provides a plug-and-play solution. The header file for the `HealthbarComponent` proves to be highly unspectacular:

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

@interface HealthbarComponent : CCSprite
{
}

-(void) reset;

@end
```

The implementation of the `HealthbarComponent` class is more interesting, as Listing 8–11 shows.

Listing 8–11. *The HealthBarComponent Updates Its scaleX Property Based on the Enemy's Remaining Hit Points*

```
#import "HealthbarComponent.h"
#import "EnemyEntity.h"

@implementation HealthbarComponent

-(id) init
{
    if ((self = [super init]))
    {
```

```

        self.visible = NO;
        [self scheduleUpdate];
    }

    return self;
}

-(void) reset
{
    float parentHeight = self.parent.contentSize.height;
    float selfHeight = self.contentSize.height;
    self.position = CGPointMake(self.parent.anchorPointInPixels.x,
        parentHeight + selfHeight);
    self.scaleX = 1;
    self.visible = YES;
}

-(void) update:(ccTime)delta
{
    if (self.parent.visible)
    {
        NSAssert([self.parent isKindOfClass:[EnemyEntity class]], @"not a EnemyEntity");
        EnemyEntity* parentEntity = (EnemyEntity*)self.parent;
        self.scaleX = parentEntity.hitPoints / (float)parentEntity.initialHitPoints;
    }
    else if (self.visible)
    {
        self.visible = NO;
    }
}
@end

```

The healthbar turns its visible state on and off to be in line with its parent `EnemyEntity` object. The reset method places the healthbar sprite at the proper location just above the head of the `EnemyEntity`'s sprite. Since a decrease in health is displayed by modifying the `scaleX` property, it, too, needs to be reset to its default state.

In the update method and when its parent is visible, the `HealthbarComponent` first asserts that the parent is of class `EnemyEntity`. Since this component relies on certain properties only available in the `EnemyEntity` class and its subclasses, we need to make sure that it's the right parent class. We modify the `scaleX` property as a percentage of the current hit points divided by the initial hit points. Since there's currently no way of telling when the hit points change, the calculation is done every frame, regardless of whether it's needed. The overhead here is minimal, but for more complex calculations, a call to the `HealthbarComponent` from the `onHit` method of the `EnemyEntity` class would be preferable.

NOTE: The `parentEntity.initialHitPoints` is cast to float. If I hadn't done this, the division would be an integer division that would always result in 0 since integers can't represent fractional numbers and are always rounded down. Casting the divisor to a float data type ensures that floating-point division is done and the desired result is returned.

In the init method of the EnemyEntity, the HealthbarComponent is added if the enemy type is EnemyTypeBoss:

```
if (type == EnemyTypeBoss)
{
    HealthbarComponent* healthbar = [HealthbarComponent
        spriteWithSpriteFrameName:@"healthbar.png"];
    [self addChild:healthbar];
}
```

The spawn method has been extended to include resetting the hit points to their initial values and calling the reset method of any possible HealthbarComponent added to the entity. I omitted the explicit check if the enemy is a boss type here simply because the HealthbarComponent is universal and could be used by any type of enemy.

```
-(void) spawn
{
    // Select a spawn location just outside the right side of the screen
    CGRect screenRect = [GameScene screenRect];
    CGSize spriteSize = [self contentSize];
    float xPos = screenRect.size.width + spriteSize.width * 0.5f;
    float yPos = CCRANDOM_0_1() * (screenRect.size.height - spriteSize.height) +
        spriteSize.height * 0.5f;
    self.position = CGPointMake(xPos, yPos);

    // Finally set yourself to be visible, this also flags the enemy as "in use"
    self.visible = YES;

    // reset health
    hitPoints = initialHitPoints;

    // reset certain components
    CCNode* node;
    CCARRAY_FOREACH([self children], node)
    {
        if ([node isKindOfClass:[HealthbarComponent class]])
        {
            HealthbarComponent* healthbar = (HealthbarComponent*)node;
            [healthbar reset];
        }
    }
}
```

Summary

Creating a complete and polished game is quite an effort, one that involves a lot of refactoring, changing working code to improve its design and to allow for more features existing in harmony with each other.

In this chapter, you learned the value of classes like BulletCache and EnemyCache, which manage all instances of certain classes so that you have easy access to them from one central point. And pooling objects together helps improve performance.

The Entity class hierarchy serves as just one example of how you can divide your classes without requiring each game object to be its own class. By using component classes and cocos2d's node hierarchy to your advantage, you can create plug-and-play classes with very specific functionality. This helps you construct your game objects using composition rather than inheritance. It's a much more flexible way to write game logic code and leads to better code reuse.

Finally, you learned how to shoot enemies and how the `BulletCache` and `EnemyCache` classes help perform such tasks in a straightforward manner. And the `HealthbarComponent` provided the perfect example of the component system at work.

The game to this point leaves a couple things open for you to build on. First and foremost, the player doesn't get hit yet. And you might want to add a healthbar to the cruiser monster and write specialized move and shoot components for the boss monster's behavior. Overall, it's an excellent starting point for your own side-scrolling game, just waiting for you to improve on it.

In the next chapter, I'll show you how to add visual eye candy to the shoot-'em-up game by using particle effects.

Particle Effects

To create special visual effects, game programmers often make use of *particle systems*. Particle systems work by emitting vast numbers of tiny particles and rendering them efficiently, much more efficiently than if they were sprites. This allows you to simulate effects such as rain, fire, snow, explosions, vapor trails, and many more.

Particle systems are driven by a great number of properties. By great I mean about 30 properties, which all influence not only the appearance and behavior of individual particles but the whole particle effect. The particle effect is the totality of all particles working together to create a particular visual outcome. One particle alone does not make a fire effect; ten still don't get close enough. You would want several dozens, if not hundreds, of particles to work together in just the right way to create the fire effect.

Creating convincing particle effects is a trial-and-error process. Trying all the various properties in source code and tweaking a particle system by compiling the game, seeing what it looks like, and then making changes and repeating this process is cumbersome to say the least. That's where a particle design tool comes in handy, and I know just the right one: it's called Particle Designer, and I'll explain how it works in this chapter.

Example Particle Effects

Cocos2d comes with a number of built-in particle effects that give you a good idea of the kinds of effects they'll produce. You can use them as placeholders in your game or subclass and modify the examples defined in the `CCParticleExamples.m` file if you just want to apply some minor tweaks. The good thing about them is that you don't need any outside help; you create the example particle effects as if they were simple `CCNode` objects. As a matter of fact, they are actually derived from `CCNode`.

I created a project called `ParticleEffects01` that shows all the cocos2d example particle effects. You can cycle through the examples by quickly tapping the screen, and you can also drag and move them with your finger. Many particle effects look totally different as soon as they start moving, as you can see in Figure 9–1, so what seems like just a huge blob of particles may well work as an engine exhaust effect if the particle effect is moving.

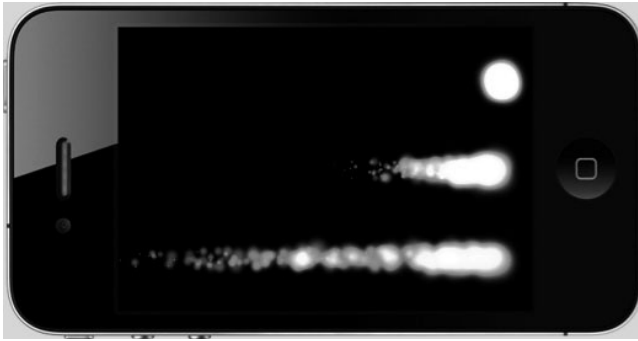


Figure 9-1. *The same particle effect from top to bottom: motionless, moving slowly, moving fast*

There's only one type of effect that can't be moved once started, and these are one-time effects like the `CCParticleExplosion` shown in Figure 9-2. What's special about this effect is that it emits all its particles at once and stops emitting new particles instantly. All other particle effects run continuously, always creating new particles while those that have exceeded their lifetime are removed. The challenge in that situation is to balance the total number of particles that are on the screen.

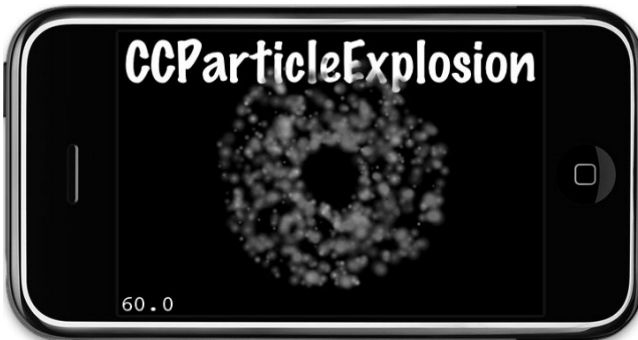


Figure 9-2. *The `CCParticleExplosion` is an example effect provided by `cocos2d`.*

Listing 9-1 shows the relevant methods used in the `ParticleEffects01` example project. By using the current `particleType` variable in the switch statement, the corresponding built-in particle effect is created. Note that a `CCParticleSystem` pointer is used to store the particles, so I need to use the `addChild` code only once at the end of the `runEffect` method. Every example particle effect is derived from `CCParticleSystem`.

Listing 9-1. *Using the Built-in Effects*

```
-(void) runEffect
{
    // remove any previous particle FX
    [self removeChildByTag:1 cleanup:YES];

    CCParticleSystem* system;

    switch (particleType)
    {
```

```

        case ParticleTypeExplosion:
            system = [CCParticleExplosion node];
            break;
        case ParticleTypeFire:
            system = [CCParticleFire node];
            break;
        case ParticleTypeFireworks:
            system = [CCParticleFireworks node];
            break;
        case ParticleTypeFlower:
            system = [CCParticleFlower node];
            break;
        case ParticleTypeGalaxy:
            system = [CCParticleGalaxy node];
            break;
        case ParticleTypeMeteor:
            system = [CCParticleMeteor node];
            break;
        case ParticleTypeRain:
            system = [CCParticleRain node];
            break;
        case ParticleTypeSmoke:
            system = [CCParticleSmoke node];
            break;
        case ParticleTypeSnow:
            system = [CCParticleSnow node];
            break;
        case ParticleTypeSpiral:
            system = [CCParticleSpiral node];
            break;
        case ParticleTypeSun:
            system = [CCParticleSun node];
            break;

        default:
            // do nothing
            break;
    }

    [self addChild: system z:1 tag:1];

    [label setString:NSStringFromClass([system class])];
}

-(void) setNextParticleType
{
    particleType++;
    if (particleType == ParticleTypes_MAX)
    {
        particleType = 0;
    }
}

```

NOTE: The `NSStringFromClass` method is very helpful in this example for printing out the name of the class without having to enter dozens of matching strings. It's one of the cool runtime features of the Objective-C language that you're able to get a class's name as a string. Try to do that in C++, and you'll be biting your toenails. The Objective-C Runtime Programming Guide is a good starting point if you like to dive into this advanced topic or if you just want to learn how Objective-C works on a lower level:

<http://developer.apple.com/library/mac/#documentation/Cocoa/Conceptual/ObjCRuntimeGuide/Introduction/Introduction.html>.

For game-play code, the `NSStringFromClass` and related methods hardly play any role, but they're very helpful debugging and logging tools. You can find a complete list and description of these methods in Apple's Foundation Function Reference: http://developer.apple.com/library/documentation/Cocoa/Reference/Foundation/Miscellaneous/Foundation_Functions/Reference/reference.html.

If you use one of these example effects in your own project, you might be shocked to see ugly, square pixels. Figure 9-3 shows this effect very clearly. This occurs because all of the built-in particle effects try to load a specific texture named `fire.png`, which is distributed with `cocos2d-iphone` in the `Resources/Images` folder. You can still create very good particle effects even without a texture, provided that the particle sizes remain fairly small. But to see the built-in particle effects as they were intended, you need to add the `fire.png` image to your Xcode project.



Figure 9-3. If your example particle effects, like this `CCParticleFireworks`, display huge, square particles, you forgot to add the `fire.png` image to your Xcode project.

Creating a Particle Effect the Hard Way

You can easily create your own subclass of the `CCParticleSystem` class. What's not so easy is creating a convincing particle effect with it, let alone one that comes close to what you originally envisioned. The following is the list of properties grouped by function that determine the particle system's look and behavior:

- `emitterMode = gravity`
 - `sourcePosition`
 - `gravity`
 - `radialAccel, radialAccelVar`
 - `speed, speedVar`
 - `tangentialAccel, tangentialAccelVar`
- `emitterMode = radius`
 - `startRadius, startRadiusVar, endRadius, endRadiusVar`
 - `rotatePerSecond, rotatePerSecondVar`
- `duration`
- `posVar`
- `positionType`
- `startSize, startSizeVar, endSize, endSizeVar`
- `angle, angleVar`
- `life, lifeVar`
- `emissionRate`
- `startColor, startColorVar, endColor, endColorVar`
- `blendFunc, blendAdditive`
- `texture`

As you can imagine, there's a lot to tweak here, and that is the main problem of achieving what you want: you don't see the effect of your changes until you rebuild and run your project. You'll learn that when we get to Particle Designer later in this chapter how much it streamlines the creation of new particle effects.

For now let's do it the hard way first and start at the beginning to gain an understanding about how the cocos2d particle system works. To program a particle effect from scratch, you'll first learn how to subclass the `CCParticleSystem` class and how to initialize it. A detailed description of the particle system's properties follows.

Subclassing CCParticleSystem: Point or Quad?

To create your own particle effect without Particle Designer, you should subclass either from CCParticleSystemPoint or from CCParticleSystemQuad. Point particles are a bit faster on first- and second-generation iOS devices but don't perform well on third- and fourth-generation devices like the iPhone 3GS, iPad, and iPhone 4. This is because of the change in CPU architecture. The ARMv7 CPU architecture used in third- and fourth-generation devices introduced optimizations and new features, such as a vector floating-point processor and the SIMD instruction set (NEON). The CCParticleSystemQuad class benefits from both.

If in doubt, I prefer to use the CCParticleSystemQuad since it performs reasonably well across all devices and creates exactly the same visual effects. Or let cocos2d make that decision for you based on the build target. You'll see how you can do so in the ParticleEffectSelfMade class I've added to the ParticleEffects02 project, which you can see in Listing 9-2.

Listing 9-2. Subclassing from the Optimal Particle System Class

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

// Depending on the targeted device the ParticleEffectSelfMade class will either derive
// from CCParticleSystemPoint or CCParticleSystemQuad
@interface ParticleEffectSelfMade : ARCH_OPTIMAL_PARTICLE_SYSTEM
{
}
@end
```

The preprocessor definition ARCH_OPTIMAL_PARTICLE_SYSTEM, instead of an actual class name, is used to determine during compilation which of the two particle systems this class should subclass from. The definition in cocos2d is based on the processor architecture and results either in a CCParticleSystemQuad or in a CCParticleSystemPoint:

```
// build each architecture with the optimal particle system
#ifdef __ARM_NEON__
    // armv7
    #define ARCH_OPTIMAL_PARTICLE_SYSTEM CCParticleSystemQuad
#elif __arm__ || TARGET_IPHONE_SIMULATOR
    // armv6 or simulator
    #define ARCH_OPTIMAL_PARTICLE_SYSTEM CCParticleSystemPoint
#else
    #error(unknown architecture)
#endif
```

Now let's look at the implementation of the self-made particle effect, which uses all available properties. I'll attempt to explain each of them, but it's much better to see it for yourself and experiment with the parameters, so I encourage you to tweak the properties in this project. In the ParticleEffects02 project (Listing 9-3), you'll also find comments describing each parameter in brief.

Listing 9-3. Manually Setting a Particle System's Properties

```

#import "ParticleEffectSelfMade.h"

@implementation ParticleEffectSelfMade
-(id) init
{
    return [self initWithTotalParticles:250];
}

-(id) initWithTotalParticles:(int)numParticles
{
    if ((self = [super initWithTotalParticles:numParticles]))
    {
        self.duration = kCCParticleDurationInfinity;
        self.emitterMode = kCCParticleModeGravity;

        // some properties must only be used with a specific emitterMode!
        if (self.emitterMode == kCCParticleModeGravity)
        {
            self.sourcePosition = CGPointMake(-15, 0);
            self.gravity = CGPointMake(-50, -90);
            self.radialAccel = -90;
            self.radialAccelVar = 20;
            self.tangentialAccel = 120;
            self.tangentialAccelVar = 10;
            self.speed = 15;
            self.speedVar = 4;
        }
        else if (self.emitterMode == kCCParticleModeRadius)
        {
            self.startRadius = 100;
            self.startRadiusVar = 0;
            self.endRadius = 10;
            self.endRadiusVar = 0;
            self.rotatePerSecond = -180;
            self.rotatePerSecondVar = 0;
        }

        self.position = CGPointZero;
        self.posVar = CGPointZero;
        self.positionType = kCCPositionTypeFree;

        self.startSize = 40.0f;
        self.startSizeVar = 0.0f;
        self.endSize = kCCParticleStartSizeEqualToEndSize;
        self.endSizeVar = 0;

        self.angle = 0;
        self.angleVar = 0;

        self.life = 5.0f;
        self.lifeVar = 1.0f;

        self.emissionRate = 30;
        self.totalParticles = 250;

        startColor.r = 1.0f;

```

```

        startColor.g = 0.25f;
        startColor.b = 0.12f;
        startColor.a = 1.0f;
        startColorVar.r = 0.0f;
        startColorVar.g = 0.0f;
        startColorVar.b = 0.0f;
        startColorVar.a = 0.0f;
        endColor.r = 0.0f;
        endColor.g = 0.0f;
        endColor.b = 0.0f;
        endColor.a = 1.0f;
        endColorVar.r = 0.0f;
        endColorVar.g = 0.0f;
        endColorVar.b = 1.0f;
        endColorVar.a = 0.0f;

        self.blendFunc = (ccBlendFunc){GL_SRC_ALPHA, GL_DST_ALPHA};
        // or use this shortcut to set the blend func to: GL_SRC_ALPHA, GL_ONE
        //self.blendAdditive = YES;

        self.texture = [[CCTextureCache sharedTextureCache] addImage:@"fire.png"];
    }
    return self;
}
@end

```

CCParticleSystem Properties

In Listing 9–3 you will have noticed how verbose the code is simply because so many particle system properties can be initialized. And most of them need to be set to acceptable values in order to get a reasonably meaningful particle effect displayed on the screen. Some properties are even mutually exclusive and can't be used together. It's time to take a close look at what these particle system properties actually do.

Variance Properties

You'll notice that many properties have companion properties suffixed with `Var`. These are variance properties, and they determine the range of fuzzyness that is allowed for the corresponding property. Take, for example, the properties `life = 5` and `lifeVar = 1`. These values mean that on average each particle will live for five seconds. The variance allows a range of $5-1$ to $5+1$. So, each particle will get a random lifetime between four to six seconds.

If you don't want any variation, set the `Var` variable to 0. Variation is what gives particle effects their organic, fuzzy behavior and appearance. But variation can also be confusing when you design a new effect, so unless you have some experience, I recommend starting with a particle effect that has little or no variance.

Number of Particles

Let's get acquainted with particles by starting with the total number of particles in the particle effect, controlled by the `totalParticles` property. The `totalParticles` variable is usually set by the `initWithTotalParticles` method but can be changed later. The number of particles has a direct impact both on the look of the effect and on performance.

```
-(id) init
{
    return [self initWithTotalParticles:250];
}
```

Use too few particles and you won't get a nice glow, but it may be sufficient to sprinkle a few stars around the player's head when he runs into a wall. Use too many particles and it might not be what you want either, because many particles are rendered on top of one another and possibly blended so you basically end up with a white blob. Furthermore, using too many particles easily kills your framerate. There's a reason why the Particle Designer tool won't let you create effects with more than 2,000 particles.

TIP: In general, you should aim to achieve the desired effect with the smallest number of particles. Particle size also plays an important role—the smaller the size of individual particles, the better the performance will be. Especially with Particle Effects, it is important to test them on first- and second-generation devices because they can have a severe and negative impact on performance on the older devices.

Emitter Duration

The `duration` property determines how long particles will be emitted. If set to 2, it will create new particles for two seconds and then stop. It's that simple:

```
self.duration = 2.0f;
```

If you'd like the particle effect node to be automatically removed from its parent node once the particle system has stopped emitting particles and the last particles have vanished, set the `autoRemoveOnFinish` property to YES:

```
self.autoRemoveOnFinish = YES;
```

The `autoRemoveOnFinish` property is a convenience feature that is meaningful only if used in conjunction with particle systems that don't run infinitely, like one-time explosion effects. Cocos2d defines a constant `kCCParticleDurationInfinity` (equals: -1) for infinitely running particle effects.

```
self.duration = kCCParticleDurationInfinity;
```

The majority of particle effects are infinitely running and can be stopped only by removing them from the node hierarchy. Infinitely running particle effects ignore the `autoRemoveOnFinish` property.

Emitter Modes

There are two emitter modes: gravity and radius, controlled by the `emitterMode` property. These two modes create fundamentally different effects even if most of the parameters are the same, as you can see when you compare Figure 9–4 with Figure 9–5. Both modes use several exclusive properties (see Listing 9–3) that must not be set if they are not supported by the current mode; otherwise, you’ll receive a runtime exception from cocos2d like this:

```
ParticleEffects[6332:207] *** Terminating app due to uncaught exception
      'NSInternalInconsistencyException', reason: 'Particle Mode should be Radius'
```

Emitter Mode: Gravity

Gravity mode lets particles fly toward or away from a center point. Its strength is that it allows very dynamic, organic effects. You can set gravity mode with this line:

```
self.emitterMode = kCCParticleModeGravity;
```

Gravity mode uses the following exclusive properties, which can be used only when `emitterMode` is set to `kCCParticleModeGravity`:

```
self.sourcePosition = CGPointMake(-15, 0);
self.gravity = CGPointMake(-50, -90);
self.radialAccel = -90;
self.radialAccelVar = 20;
self.tangentialAccel = 120;
self.tangentialAccelVar = 10;
self.speed = 15;
self.speedVar = 4;
```

The `sourcePosition` determines the offset as a `CGPoint` from the node’s position where new particles appear. The name is a bit misleading in that the actual center of gravity is the node’s position, and `sourcePosition` is an offset to that center of gravity. The `gravity` property then determines the speed with which particles accelerate in the x and y directions. In this case, the negative values indicate that the gravitational force will accelerate particles toward the left (-50) and downward (-90). But this acceleration is relative to the particle node’s position. Since particles start out at a `sourcePosition` offset of -15 (slightly offset to the left), they will perform a counterclockwise movement around the particle node’s position. You can see this effect in Figure 9–4, and tweaking the values in the `ParticleEffects02` project helps to understand how `sourcePosition` and `gravity` affect the movement of particles.

For the center of gravity to have any impact, the `gravity` of the particles shouldn’t be too high, and the `sourcePosition` should not be offset too far. The previous values give you a good working example that you can tweak.

The `radialAccel` property defines how fast particles accelerate the further they move away from the emitter. This parameter can also be negative, which makes particles slow down as they move away. The `tangentialAccel` property is similar in that it lets particles rotate around the emitter and speed up as they move away. Negative values let the particles spin clockwise, and positive values spin them counterclockwise.

The `speed` property should be fairly obvious—it's simply the speed of the particles. It has no particular unit of measurement. Figure 9–4 shows an example particle effect using gravity mode. Particles are attracted by the particle node's position and start out slightly to the left of the particle node's position, so they perform a counterclockwise radial movement.

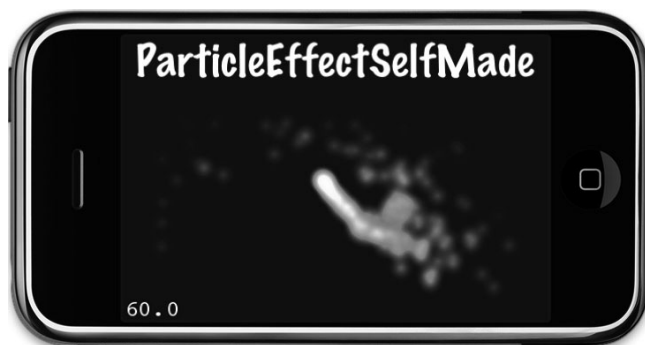


Figure 9–4. *The ParticleEffectSelfMade from the ParticleEffects02 project in gravity mode*

Emitter Mode: Radius

Radius mode causes particles to rotate in a circle. It also allows you to create spiral effects with particles either rushing inward or rotating outward. You set radius mode with this line:

```
self.emitterMode = kCCParticleModeRadius;
```

Like gravity mode, radius mode has exclusive properties. The following properties can be used only when `emitterMode` is set to `kCCParticleModeRadius`:

```
self.startRadius = 100;
self.startRadiusVar = 0;
self.endRadius = 10;
self.endRadiusVar = 0;
self.rotatePerSecond = -180;
self.rotatePerSecondVar = 0;
```

The `startRadius` property determines how far away from the particle effect node's position the particles will be emitted. Likewise, the `endRadius` determines the distance from the node's position the particles will rotate toward. If you want to achieve a perfect circle effect, you can set `endRadius` to the same as the `startRadius` using this constant:

```
self.endRadius = kCCParticleStartRadiusEqualToEndRadius;
```

Using the `rotatePerSecond` property, you can influence the direction the particles move and the speed with which they move, and thus the number of times they rotate around if `startRadius` and `endRadius` are different.

The same particle effect that was shown in Figure 9–4 using gravity mode is shown in Figure 9–5 using radius mode, and you’ll notice how different it looks, despite all other properties—except for the exclusive ones—being the same. To test this, uncomment the following line in the `ParticleEffects02` project:

```
//self.emitterMode = kCCParticleModeRadius;
```



Figure 9–5. *The very same effect using radius mode looks completely different.*

Particle Position

By moving the node, you also move the effect. But the effect also has a `posVar` property that determines the variance in the position where new particles will be created. By default, both are at the center of the node:

```
self.position = CGPointZero;
self.posVar = CGPointZero;
```

A very important aspect of particle positions is whether existing particles should move relative to the node’s movement or whether they should not be influenced at all by the node’s position. For example, if you have a particle effect that creates stars around your player-character’s head, you would want the stars to follow the player as he moves around. You can achieve this effect by setting this property:

```
self.positionType = kCCPositionTypeGrouped;
```

On the other hand, if you want to set your player on fire and you want the particles to create a trail-like effect as the player moves around, you should set the `positionType` property like this:

```
self.positionType = kCCPositionTypeFree;
```

The free movement is best used with effects like steam, fire, engine exhaust smoke, and similar effects that move around with the object they are attached to and should give the impression of not being connected to the object that emits these particles.

Particle Size

The size of particles is given in pixels using the `startSize` and `endSize` properties, which determine the size of the particles when they are emitted and how big they are when they are removed. The size of the particle gradually scales from `startSize` to `endSize`.

```
self.startSize = 40.0f;  
self.startSizeVar = 0.0f;  
self.endSize = kCCParticleStartSizeEqualToEndSize;  
self.endSizeVar = 0;
```

The constant `kCCParticleStartSizeEqualToEndSize` can be used to ensure that the particle size does not change during a particle's lifetime.

Particle Direction

The direction in which particles are initially emitted is set with the `angle` property and measured in degrees from 0 to 360. A value of 0 means that particles will be emitted upward, but this is true only for the gravity emitterMode. In radius emitterMode the `angle` property determines where on the `startRadius` the particles will be emitted; higher values will move the emission point counterclockwise along the radius.

```
self.angle = 0;  
self.angleVar = 0;
```

Particle Lifetime

A particle's lifetime determines how many seconds it will take to transition from start to end, where the particle will simply fade out and disappear. The `life` property sets the lifetime of individual particles. Keep in mind that the longer particles live, the more particles will be on-screen at any given time. If the total number of particles is reached, no new particles will be spawned until some existing particles have died.

```
self.life = 5.0f;  
self.lifeVar = 1.0f;
```

The `emissionRate` property directly influences how many particles are created per second. Together with the `totalParticles` property, it has a big impact on what the particle effect looks like.

```
self.emissionRate = 30;  
self.totalParticles = 250;
```

In general, you will want to balance the `emissionRate` so that it matches the particle lifetime with the `totalParticles` allowed in the particle effect. You can do so by dividing `totalParticles` by `life` and set the result as the `emissionRate`:

```
self.emissionRate = self.totalParticles / self.life;
```

TIP: By tweaking particle lifetime, the total number of particles allowed in the system, and the `emissionRate`, you can create burst effects by allowing the stream of particles to be frequently interrupted just because the number of particles on-screen is limited and new particles are emitted relatively quickly. On the other hand, if you notice undesirable gaps in your particle stream, you need to either increase the number of allowed particles or preferably reduce the lifetime and/or emission rate. In that case, you should use `emissionRate = totalParticles / life`.

Particle Color

Each particle can transition from a starting color to an end color, creating the vibrant colors particle effects are known for. You should at least set the `startColor` in a particle effect; otherwise, the particles may not be visible at all since the default color is black. The colors are of type `ccColor4F`, a struct with four floating-point members: `r`, `g`, `b`, and `a`, corresponding to the colors red, green, and blue, as well as the alpha channel, which determines the color's opacity. The value range for each of these members goes from 0 to 1, with 1 being the full color.

If you want a completely white particle color, you'd set all four `r`, `g`, `b`, and `a` members to 1. If you want a red color, you only need to set the `r` and `a` values to 1.0f. If you want blue and then set `b` and `a` to 1.0f. Note that the `a` value is the alpha transparency of the color. If you leave it at the default value of 0.0f, the color will be completely translucent and thus not visible.

```
// startColor is mostly red and fully opaque
startColor.r = 1.0f;
startColor.g = 0.25f;
startColor.b = 0.12f;
startColor.a = 1.0f;
// startColor has no variance (plus/minus 0.0f)
startColorVar.r = 0.0f;
startColorVar.g = 0.0f;
startColorVar.b = 0.0f;
startColorVar.a = 0.0f;
// endColor is a fully opaque black color
endColor.r = 0.0f;
endColor.g = 0.0f;
endColor.b = 0.0f;
endColor.a = 1.0f;
// endColorVar specifies a full variance range for color blue
// the end of lifetime color of a particle will be randomly between black and blue
endColorVar.r = 0.0f;
endColorVar.g = 0.0f;
endColorVar.b = 1.0f;
endColorVar.a = 0.0f;
```


Particle Blend Mode

Blending refers to the computation a particle's pixels go through before being displayed on-screen. The property `blendFunc` takes a `ccBlendFunc` struct as input, which provides the source and destination blend modes:

```
self.blendFunc = (ccBlendFunc){GL_SRC_ALPHA, GL_DST_ALPHA};
```

Blending works by taking the red, green, blue, and alpha of the source image (the particle) and mixing it with the colors of any images that are already on-screen when the particle is rendered. In effect, the particle blends in a certain way with its background, and `blendFunc` determines how much and what colors of the source image are blended how much and with which colors of the background.

The formula to determine the final pixel color on-screen is as follows:

```
(source color * source blend function) + (destination color * destination blend function)
```

Let's assume our example source pixel has the RGB values (0.1, 0.2, 0.3) and the destination pixel has the RGB values (0.4, 0.5, 0.6). Both color values are multiplied by the blend function, the simplest being `GL_ONE`, which equals 1.0. That means the resulting pixel's color will be as follows:

```
(0.1 * 1 + 0.4 * 1, 0.2 * 1 + 0.5 * 1, 0.3 * 1 + 0.6 * 1) = (0.5, 0.7, 0.9)
```

The `blendFunc` property has a very profound effect on how particles are displayed. By using a combination of the following blend modes for both source and target, you can create rather bizarre effects or simply cause the effect to render as black squares. There's lots of room for experimentation.

- `GL_ZERO`
- `GL_ONE`
- `GL_SRC_COLOR`
- `GL_ONE_MINUS_SRC_COLOR`
- `GL_SRC_ALPHA`
- `GL_ONE_MINUS_SRC_ALPHA`
- `GL_DST_ALPHA`
- `GL_ONE_MINUS_DST_ALPHA`

You'll find more information on the OpenGL blend modes and details about the blend calculations in the OpenGL ES documentation at www.khronos.org/opengles/documentation/opengles1_0/html/glBlendFunc.html.

TIP: Since it's hard to imagine which blend functions will create which results with varying images, I'd like to point you to an article that describes the most common blend operations with example images: www.machwerx.com/2009/02/11/glblendfunc/.

Even more interesting is the Visual glBlendFunc tool developed by Anders Riggelsen: www.andersriggelsen.dk/OpenGL/. With any HTML5-compatible browser you can play with various images and blend functions to see the results instantly.

Note that you can also modify the `blendFunc` property of other cocos2d nodes, namely, all nodes that conform to the `CCBlendProtocol` such as the classes `CCSprite`, `CCSpriteBatchNode`, `CCLabelTTF`, `CCLayerColor`, and `CCLayerGradient`.

The source and target blend modes `GL_SRC_ALPHA` and `GL_ONE` are frequently combined to create additive blending, resulting in very bright or even white colors where many particles are drawn on top of each other:

```
self.blendFunc = (ccBlendFunc){GL_SRC_ALPHA, GL_ONE};
```

Alternatively, you can simply set the `blendAdditive` property to `YES`, which is the same as setting `blendFunc` to `GL_SRC_ALPHA` and `GL_ONE`:

```
self.blendAdditive = YES;
```

The normal blend mode is set using `GL_SRC_ALPHA` and `GL_ONE_MINUS_SRC_ALPHA`, which creates transparent particles:

```
self.blendFunc = (ccBlendFunc){GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA};
```

Particle Texture

Without a texture, all particles would be flat, colored squares, as in Figure 9–3. To use a texture for a particle effect, provide one using the `CCTextureCache` method `addImage`, which returns the `CCTexture2D` for the given image file:

```
self.texture = [[CCTextureCache sharedTextureCache] addImage:@"fire.png"];
```

Particle textures look best if they are cloudlike and roughly spherical. It's often detrimental to the particle effect if the texture has high-contrast areas, resembling a particular shape or form, like the `redcross.png` from the shoot-'em-up game. This makes it easier to see individual particles because they don't blend too well with each other. Some effects can use this to their advantage, like the aforementioned stars circling the player's head. In addition, cartoonlike images or gradients work best, whereas photorealistic images tend to create an undefinable pixel mess. To demonstrate this, you'll see three different textures applied to the same particle effect in Figure 9–6.

The most important aspect of particle textures is that the image must be at most 64x64 pixels in size. The smaller the texture size, the better it is for the performance of the particle effect.

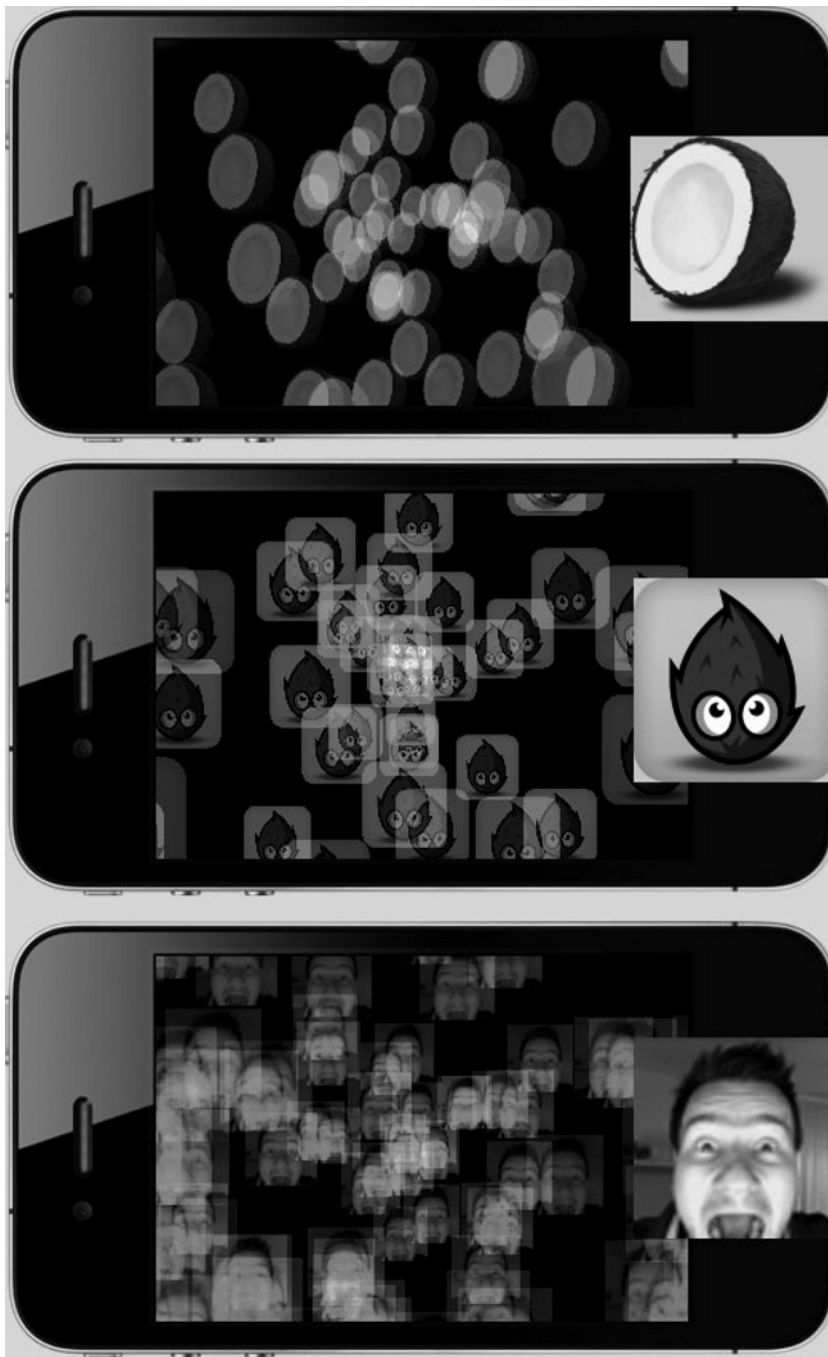


Figure 9-6. *The same particle effect using different textures. Photorealistic images don't work well.*

Particle Designer

The Particle Designer is an interactive tool for creating particle effects for cocos2d and iOS OpenGL applications. You can download the trial version at <http://particledesigner.71squared.com>.

This is an invaluable tool that will save you a lot of time creating particle effects. Its power is that you can immediately see what happens on-screen when you change a particle effect's properties, and you can share your creations with others and get inspiration from other developers' particle effects.

Introducing the Particle Designer

By default, the Particle Designer's user interface shows a visual list of particle effects. To edit a particular effect, select it and switch to the **Emitter Config** view (Figure 9–7) by either double-clicking it or clicking the **Emitter Config** button in the upper-right corner.

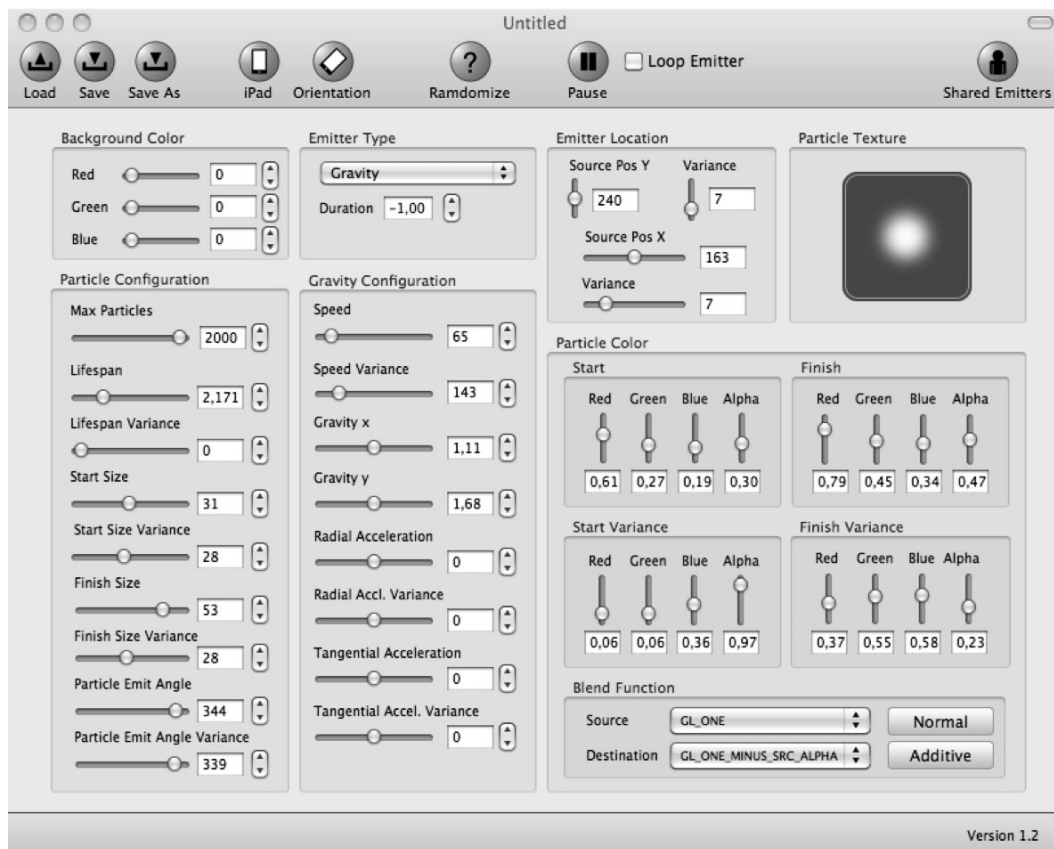


Figure 9–7. Particle Designer is full of properties you can tweak interactively. A separate window (Figure 9.8) shows the effect as you edit it.

You should recognize these parameters from the description of the self-made particle effect properties. There are only a few properties that are not available and can't be edited in Particle Designer. One is the `positionType`, and another is the `endRadiusVar` property for the radius emitter mode. The latter means you can't create particle effects that rotate outward in radius mode. But you can always load a Particle Designer effect and then tweak it in code by overriding certain properties, like setting the `positionType` to `kCCPositionTypeFree`, as you'll do later in Listing 9-4. This is just a minor nuisance compared to actually seeing on-screen how your effect changes as you move the sliders.

NOTE: The Particle Designer developers are currently working on Particle Designer 2.0, which will feature a redesigned and less crowded user interface. They also plan to create specialized controls to allow you to change both value and variance using a single gauge-like control. So if your Particle Designer looks quite different from the one in Figure 9.7, you're using version 2.0. If version 2.0 is not available as you read this, you should know that the upgrade to version 2.0 will be free for existing customers.

The only unusual control is the Particle Texture. There is no button to load an image, and double-clicking the field doesn't do anything either. The trick here is that the Particle Texture box only accepts images that are dragged and dropped onto it. Just drag any image from Finder over to this box, and the box will turn green, signaling that it will accept that image. Once you drop the image, it will be used by the particle effect.

CAUTION: Particle Designer will warn you if you drop an image that is larger than 512x512 pixels. It will use the image anyway but will scale the image down to 512x512 pixels, regardless of its original aspect ratio. Also, note that the previous limitation of 64x64 pixel textures has been removed with Particle Designer 1.3.

The Particle Designer preview window in Figure 9-8 looks just like the iPhone Simulator. It can also be set to the iPad screen size, and the orientation can be changed by clicking the **iPad/iPhone** and **Orientation** buttons in Particle Designer's menu bar, to the right of the **Load**, **Save**, and **Save As** buttons. By clicking and dragging inside the preview window, you can move the particle effect around, which helps you see how the effect might look on moving objects.

Notice that the Background Color settings are not part of the actual effect. They will change the background color of the preview window. This is useful if your game has bright colors and you want to design a dark or dim effect and still be able to see what you're doing.



Figure 9–8. *The Particle Designer has a preview window that looks like the iPhone Simulator. You can also set it to the iPad screen size and change its orientation. If you click and drag inside the screen, you can even move the particle effect around to see what it might look like on moving objects.*

If you lack inspiration, you can always make use of the **Randomize** button. You can also ponder about the meaning of the word *randomize*, which is how it's spelled in the Particle Designer. The Urban Dictionary tells me that *random* is a cooler form of random. So, I'm guessing the developers just thought their randomizer to be extra cool. Well, it's definitely inspiring even though it doesn't randomize all of the available properties. For example, **Randomize** will never change the emitter type, the emitter location, and many emitter type-specific parameters.

Once you found your inspiration, you'll want to slide the sliders and watch what happens in the preview window. Take your time and tweak an effect until you like it. Careful, though, because it's a very captivating, even mesmerizing, activity, and you'll easily find yourself making new particle effects just for the fun of it.

CAUTION: Be careful when designing particle effects! First, keep in mind that your game has to calculate and render a lot of other things, too. If the effect you're currently designing runs at 60 FPS in Particle Designer's preview window, that doesn't mean it won't kill your framerate when you use it in your game. Always test new particle effects in your game and keep an eye on the framerate. Also, make sure to run these tests on a device, in particular on older devices! Your game's performance in the iPhone/iPad Simulator is often misleading and thus must be regarded as completely irrelevant. The same goes for the Particle Designer preview window.

Using Particle Designer Effects

I'm assuming that, hours later, you've made the perfect particle effect and now you'd like to use that in cocos2d. I made mine, and the first step is to save the particle effect. When you click the **Save** or **Save As** button in Particle Designer, you'll be presented with a dialog as shown in Figure 9–9.



Figure 9–9. Saving a particle effect from Particle Designer requires you set the file format to cocos2d. Embedding the texture into the plist file is optional.

For the saved particle effect to be usable by cocos2d, you must set the **File Format** setting to **cocos2d (plist)**. You can also check the **Embed texture** box, which will save the texture into the plist file. The benefit is that you only have to add the plist file to your Xcode project; the downside is that you then can't change the effect's texture without loading the particle effect back into Particle Designer.

After saving the effect, you have to add the effect plist and, if you didn't embed the texture, the effect's PNG file to your Xcode project's Resources group. In the ParticleEffects03 project, I've added both variants, one effect with a separate PNG texture and another effect that has the texture embedded in the plist file.

Listing 9–4 shows how I modified the `runEffect` method to load the Particle Designer effects.

Listing 9–4. *Using a Particle Effect Created with Particle Designer*

```

-(void) runEffect
{
    // remove any previous particle FX
    [self removeChildByTag:1 cleanup:YES];

    CCParticleSystem* system;

    switch (particleType)
    {
        case ParticleTypeDesignedFX:
            system = [CCParticleSystemQuad particleWithFile:@"fx1.plist"];
            break;
        case ParticleTypeDesignedFX2:
            system = [CCParticleSystemQuad particleWithFile:@"fx2.plist"];
            system.positionType = kCCPositionTypeFree;
            break;
        case ParticleTypeSelfMade:
            system = [ParticleEffectSelfMade node];
            break;

        default:
            // do nothing
            break;
    }

    CGSize winSize = [[CCDirector sharedDirector] winSize];
    system.position = CGPointMake(winSize.width / 2, winSize.height / 2);
    [self addChild:system z:1 tag:1];

    [label setString:NSStringFromClass([system class]));
}

```

You initialize a `CCParticleSystem` with a Particle Designer effect by using the `particleWithFile` method and providing the particle effect's plist file as parameter. In this case, I chose a `CCParticleSystemQuad` since it performs well on all iOS devices. You can also leave this decision to `cocos2d` by using the `ARCH_OPTIMAL_PARTICLE_SYSTEM` keyword in place of an actual class name:

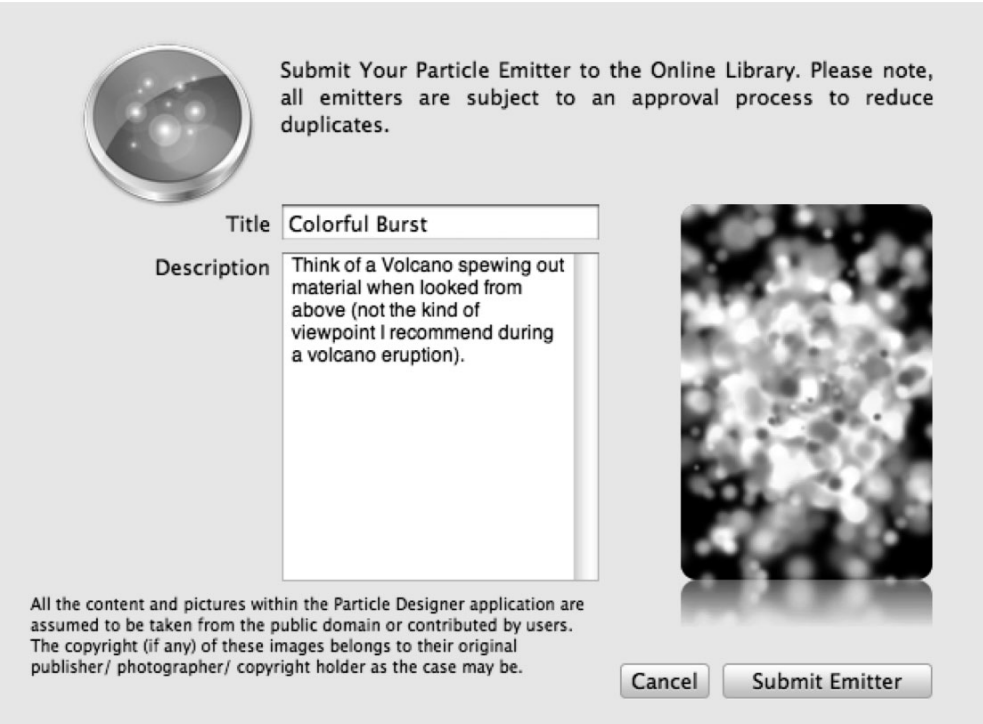
```
system = [ARCH_OPTIMAL_PARTICLE_SYSTEM particleWithFile:@"fx1.plist"];
```

CAUTION: Particle Designer effects must be initialized with either `CCParticleSystemQuad` or `CCParticleSystemPoint`. Even though `CCParticleSystem`, the parent class for the aforementioned subclasses, implements the `particleWithFile` method, it won't display anything unless you use one of the subclasses, either `Quad` or `Point` particle systems, when loading Particle Designer effects.

As a side note, I moved the two lines that position the particle system node at the center of the screen outside of the switch case to avoid duplicating that code. As I mentioned before, it's always good practice to minimize the amount of code duplication, and this is one area where I've seen developers simply copying and pasting what's already there.

Sharing Particle Effects

What's very cool about Particle Designer is that you can share your creations with other Particle Designer users. From the Particle Designer menu, simply choose **Share** and then **Share Emitter** to open a dialog that lets you enter a title and description for your particle effect, as shown in Figure 9–10.



Submit Your Particle Emitter to the Online Library. Please note, all emitters are subject to an approval process to reduce duplicates.

Title

Description

All the content and pictures within the Particle Designer application are assumed to be taken from the public domain or contributed by users. The copyright (if any) of these images belongs to their original publisher/ photographer/ copyright holder as the case may be.

Figure 9–10. By submitting a Particle Effect to the Online Library, you can share your creations with other users.

CAUTION: As the message in Figure 9.8 indicates, you should be careful to only upload artwork for which you have the rights to share and distribute or for which you are the copyright holder. Otherwise, you risk violating someone's copyright or may be violating a nondisclosure or other agreement if you work under contract.

In Figure 9–11, you can see the effect I just submitted in the lower-right corner.

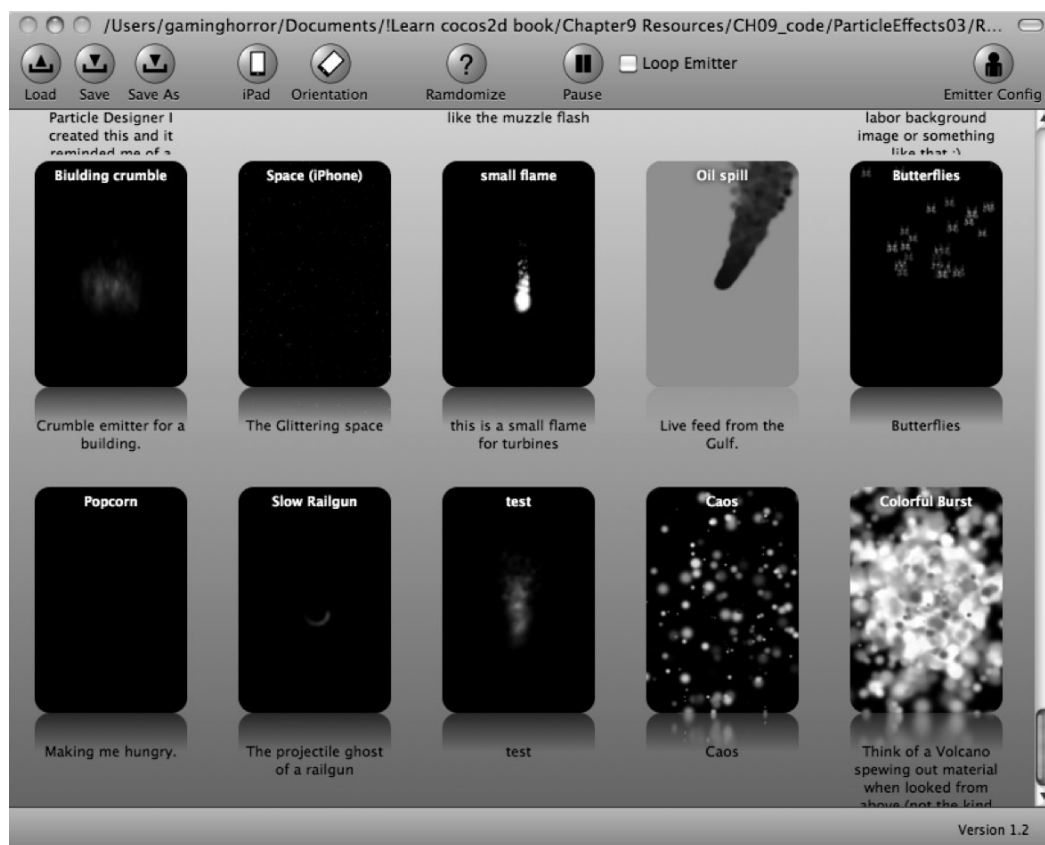


Figure 9-11. The submitted effect quickly appears in the Online Library. The effect I just submitted is the one in the bottom-right corner. Apparently my description was too long to fit in the display area.

Shared particle effects may not always be perfect for your requirements, but they often provide good starting points for your own effects. They help you achieve the desired effect much faster, and, at the very least, they can be an inspiration. I encourage you to scroll through the list of effects and try as many as you can to gain a good sense of what's possible, what looks good, and what just doesn't work.

Shoot em Up with Particle Effects

I'd love to see some of these effects in the game! Let's take the shoot-'em-up game to a new level. You'll find the results in this chapter's ShootEmUp04 project along with added sound effects, as well as in Figure 9-12.

In the `EnemyEntity` class, the `gotHit` method is the perfect place to add destructive particle explosions, as Listing 9-5 shows. I decided the boss monster should have its own particle effect, mostly because it's so big. And purple.

Listing 9–5. Adding an Explosion Effect to the Shoot 'em Up Game

```

-(void) gotHit
{
    hitPoints--;
    if (hitPoints <= 0)
    {
        self.visible = NO;

        // Play a particle effect when the enemy was destroyed
        CCParticleSystem* system;

        if (type == EnemyTypeBoss)
        {
            system = [ARCH_OPTIMAL_PARTICLE_SYSTEM ↵
                particleWithFile:@"fx-explosion2.plist"];
        }
        else
        {
            system = [ARCH_OPTIMAL_PARTICLE_SYSTEM ↵
                particleWithFile:@"fx-explosion.plist"];
        }

        // Set some parameters that can't be set in Particle Designer
        system.positionType = kCCPositionTypeFree;
        system.autoRemoveOnFinish = YES;
        system.position = self.position;

        [[GameScene sharedGameScene] addChild:system];
    }
}

```

The particle effects `fx-explosion.plist` and `fx-explosion2.plist` must be added as resources to the Xcode project. The particle system is initialized as before. Since the particle effect should and must be independent from the enemy that creates it, a few preparations are necessary. First, the `autoRemoveOnFinish` flag is set to YES so that the effect automatically removes itself. This works because both explosions run only for a short time. The effect also needs the current position of the enemy so that it's displayed at the correct position.

I add the particle effect to the `GameScene` because the enemy itself can't display the particle effect. To start, it's invisible. It also might be respawned very soon, which would interfere with the particle effect. But most importantly, all `EnemyEntity` objects are added to a `CCSpriteBatchNode`, which does not allow you to add anything but `CCSprite` objects. If the particle effect were added to the `EnemyEntity` object, a runtime exception would be inevitable.

As you play the game with the new particle effects, you may notice that the first time one of these effects displays, the game play stops for a short moment. That's because `cocos2d` is loading the particle effect's texture—a rather slow process, whether the texture is embedded into the plist, as in this case, or provided as a separate texture. To avoid that, I've added a preloading mechanism to the `GameScene`: the `init` method now calls the `preloadParticleEffect` method for each particle effect used during game play:

```

// To preload the textures, play each effect once off-screen

```

```
[self preloadParticleEffects:@"fx-explosion.plist"];
[self preloadParticleEffects:@"fx-explosion2.plist"];
```

The `preloadParticleEffects` method simply creates the particle effect. Because the returned object is an autorelease object, its memory will be freed automatically. But the texture it loads will still be in the `CCTextureCache`.

```
-(void) preloadParticleEffects:(NSString*)particleFile
{
    [ARCH_OPTIMAL_PARTICLE_SYSTEM particleWithFile:particleFile];
}
```

If you chose not to embed the texture inside the particle effect plist file, you can preload the particle effect textures simply by calling the `CCTextureCache` `addImage` method:

```
[[CCTextureCache sharedTextureCache] addImage:particleFile];
```



Figure 9–12. *I killed the boss, and I can't see a darn thing. But those particles are so beautiful!*

Summary

This chapter was truly a visual joy ride! The stock effects provided by `cocos2d` give a good indication of what the particle effect is able to deliver. They're quick and easy to use.

But it was also excruciating to create a particle effect in source code. There are so many properties to tweak; some are exclusive to specific emitter modes; some have misleading names, and they aren't exactly straightforward to figure out. With the explanation for each property, however, you should now have a good understanding how a particle effect is put together and what the most important parameters are.

We then saw how particle effects can shine with Particle Designer. This tool is extremely helpful and lots of fun to work with it. Suddenly, when you can move sliders and see the results immediately on-screen, it changes your whole view on particle effects, and even more so since you can share your creations with others and experiment with other designers' effects.

Finally the Shoot 'em Up game got a makeover and now plays particle explosions when enemies are destroyed. This makes for a much more lively game.

In the next chapter I'll set the Shoot 'em Up game aside for a bit to tell you all you need to know about tilemaps.

Working with Tilemaps

In the next two chapters, I'll introduce you to the world of tile-based games. Whether you've been playing games since the age of classic role-playing games like *Ultima* or you've just recently joined your Facebook friends in *Farmville*, I'm sure you've already played a game that uses the tilemap concept for displaying its graphics.

In tilemap games, the graphics consist of a small number of images, called *tiles*, that align with each other; placing them on a grid allows us to build rather convincing game worlds. The concept is very attractive because it conserves memory as compared to drawing the whole world as individual textures, while still allowing a lot of variety.

This chapter will introduce general tilemap concepts by using the simplest tilemaps of all: *orthogonal tilemaps*. They are most often built from square, rarely from nonsquare rectangular tiles, and typically display the world in a top-down fashion. I'll discuss the various display styles of tilemaps, and the next chapter will focus on isometric tilemaps, building on the tilemap programming basics that you'll learn in this chapter.

What Is a Tilemap?

Tilemaps are 2D game worlds made of individual tiles. You can create large world maps with just a handful of images that all have the same dimensions. This means tilemaps are very efficient at conserving memory for large maps (game worlds). It's no wonder that they first appeared in the early days of computer games. Many classic role-playing games used square tiles to create fantastic fantasy worlds. These games looked a bit like the tilemap in Figure 10–1, which is also a perfect example of an orthogonal tilemap. It looks like an aerial view, looking straight down. For this reason, orthogonal tilemaps usually look flat.

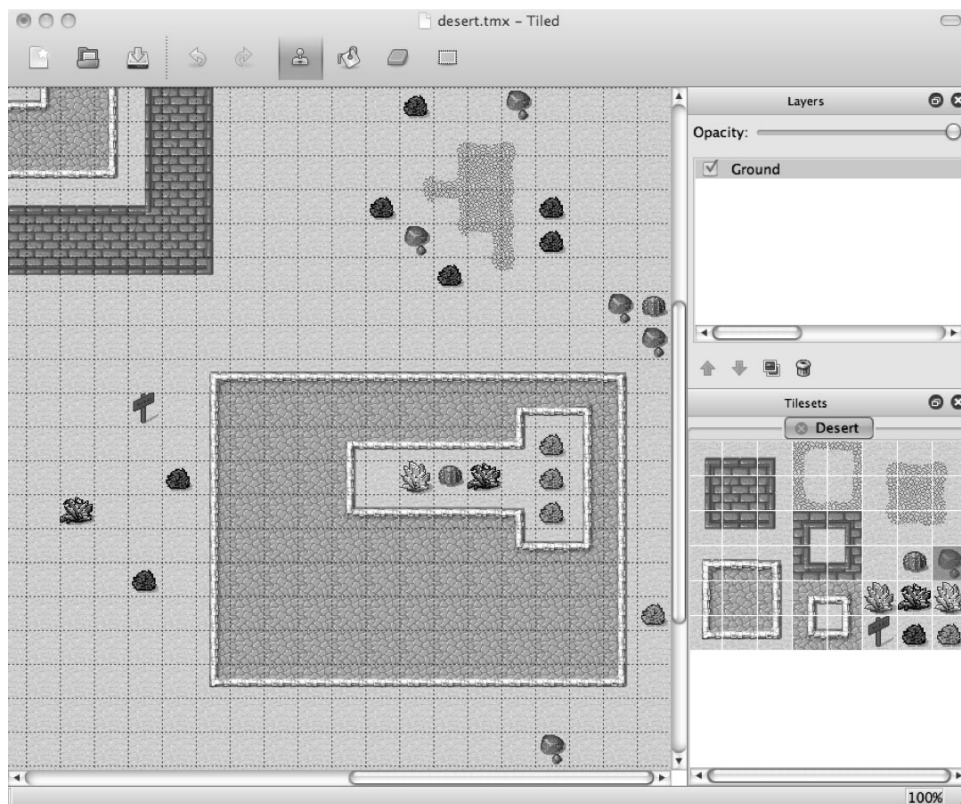


Figure 10–1. An orthogonal tilemap in Tiled (Qt) Map Editor

Tiles in an orthogonal tilemap actually don't need to be square; you can create orthogonal tilemaps from rectangular images as well. Those were most often used by Asian role-playing games, such as *Dragon Quest*. While still using an orthogonal perspective, it allowed the designers to create objects that are seemingly taller than wide. This enabled the designers to create the illusion of volumes, like houses, by painting them as several tiles and allowing game characters to be partially obstructed by tiles.

That method really came to shine with *Ultima 6* and in particular *Ultima 7*. By skewing the perspective with which tiles were drawn, the effect came close to isometric tilemaps while still being orthogonal tilemaps. Using this method, the designers were able to create the illusion of depth, as you can see in Figure 10–2.

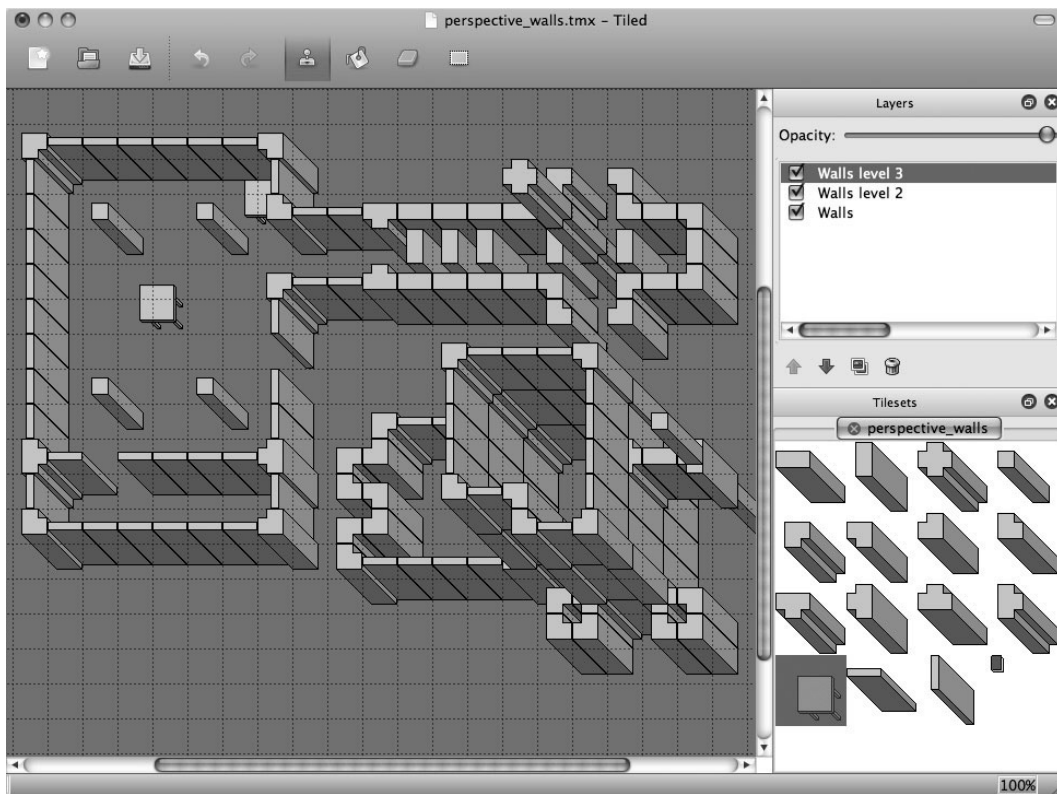


Figure 10–2. An orthogonal tilemap with perspective tiles designed to be created in multiple layers to give the impression of depth. This tilemap style has come to fame by *Ultima 7*.

Isometric tilemaps take this one step further by not just drawing the tiles in a certain perspective but also by rotating the tiles by 45 degrees. Isometric tilemaps are very effective in tricking our minds into believing that there really is a third dimension in this world, even though all of the images are still essentially flat. Isometric tilemaps achieve this impression of depth by using tile images that are drawn as diamond shapes (rhombuses) and allowing tiles closer to the viewer to draw over the tiles further away from the viewer. See Figure 10–3 for an example of an isometric tilemap, which I’ll be discussing in the next chapter in greater detail.

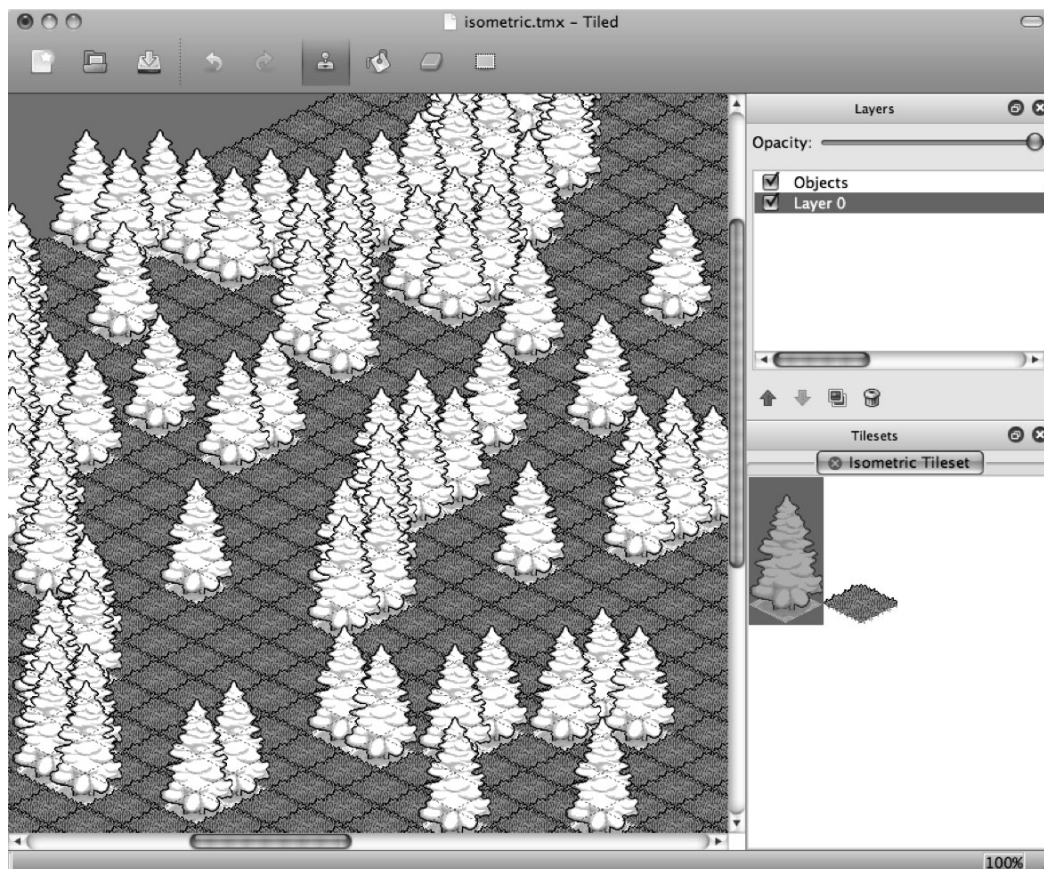


Figure 10–3. An isometric tilemap in Tiled (Qt) Map Editor

The tilemaps in Figures 10–2 and 10–3 prove that tilemaps don’t need to be flat-looking. The layering or stacking of tiles can also be used with some games that allow players to interact with the game world, as many of the Farmville fan videos show to great effect. Several Farmville users have used nothing but crop fields to build houses and even tall skyscrapers just by stacking tiles. They make use of an optical illusion that’s possible with isometric tilemaps.

Editing tilemaps is usually done with an editor, and the one directly supported by cocos2d is called Tiled (Qt) Map Editor. Tiled is free, is open source, and allows you to edit both orthogonal and isometric tilemaps with multiple layers. Tiled also enables you to add trigger areas (Object Layer), which can be used in a game to trigger certain actions when a character enters the area. They also serve as a way to add arbitrary positions to the map so that you can, for example, define spawn locations. By editing tile properties, you can determine what type of tile it is. This can also be used to block characters from entering certain tiles or, for example, to take damage when moving over a lava tile.

NOTE: Qt refers to Nokia's Qt framework, on which Tiled is built. Since there is also a defunct Java version of Tiled, it's important to make this differentiation by writing Tiled (Qt). The Java version is no longer updated but contains a few extra features that may be worth checking out. But in this and the following chapter, I will be using and discussing Tiled (Qt).

Preparing Images with TexturePacker

Before you start working with the Tiled (Qt) Map Editor, you need to prepare the tilemap graphics. The set of tile images for a tilemap is commonly referred to as *tileset*. Technically it's just a texture atlas containing tile images.

In the Tilemap01 project for this chapter, you'll find a number of square tile images in the Assets/tiles folder. Add all of these tile images to TexturePacker and then uncheck Allow Rotation, set Algorithm to Basic, and set Sort by to Name. These settings ensure that the tiles are ordered and aligned correctly for Tiled. The resulting tileset in TexturePacker should look similar to Figure 10–4.

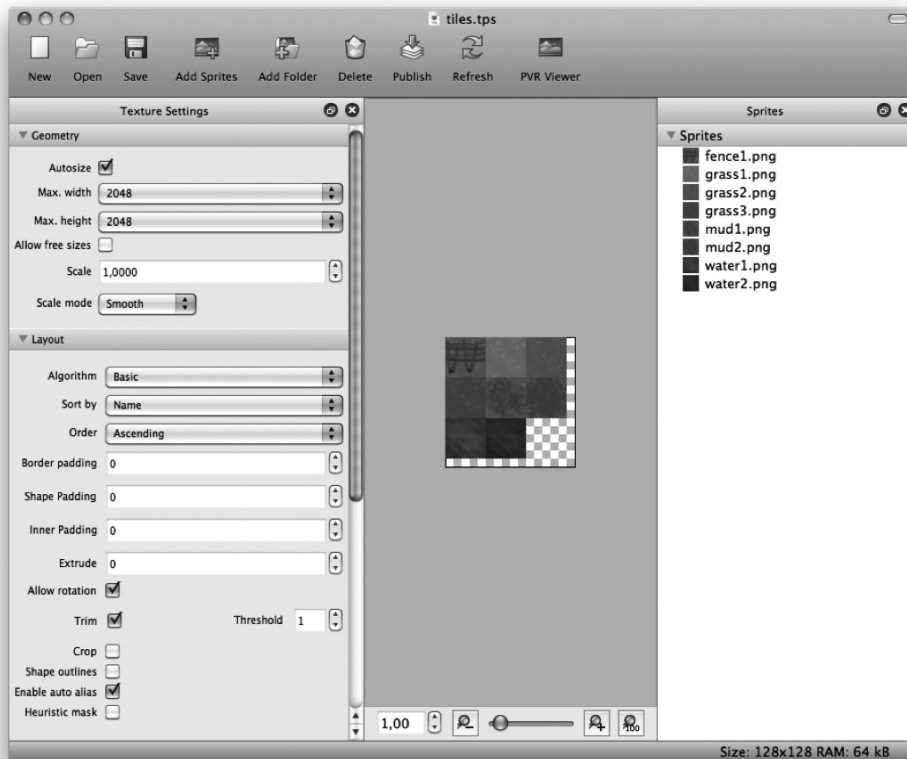


Figure 10–4. Creating a texture atlas of a few square tiles using TexturePacker

For Tiled it is crucial that the tiles stay at the same position□ this is why the sorting by name is so important□ since Tiled refers to individual tiles in the tileset by position and offset only. That means that if tiles in the texture shift places, the tilemap in Tiled using this tileset texture will look completely different. The tilemap still refers to the same tile positions in the tileset, but instead of a grass tile there could now be a water tile at that position, for example.

NOTE: TexturePacker will get a new algorithm called **TileMap** a special mode for creating tilesets for tilemaps, allowing you to manually arrange the tiles within the grid. Please visit www.texturepacker.com for availability and more information regarding this new feature.

Tiled (Qt) Map Editor

The most popular tool to create tilemaps for use with cocos2d is the Tiled (Qt) Map Editor that I've already mentioned in the preceding sections. The TMX files it creates are natively supported by the cocos2d game engine. TMX files are simply XML files that, if need be, you could edit with a text editor.

Tiled is available as a free download, and at the time of this writing, version 0.7 is the most current. You can download Tiled from its home page, at www.mapeditor.org.

If you would like to support the development of Tiled, consider making a donation to the project's developer, Thorbjørn Lindeijer. You can donate to the project here: http://sourceforge.net/donate/index.php?group_id=161281.

Creating a New Tilemap

The first thing after you've downloaded, extracted, installed, and started Tiled is to go to the View menu and check both the Tilesets and Layers items. This will show the list of layers and the current tileset on the right side of the Tiled window. Then choose File □ **New** to create a new tilemap. This will bring up the New Map dialog pictured in Figure 10-5.

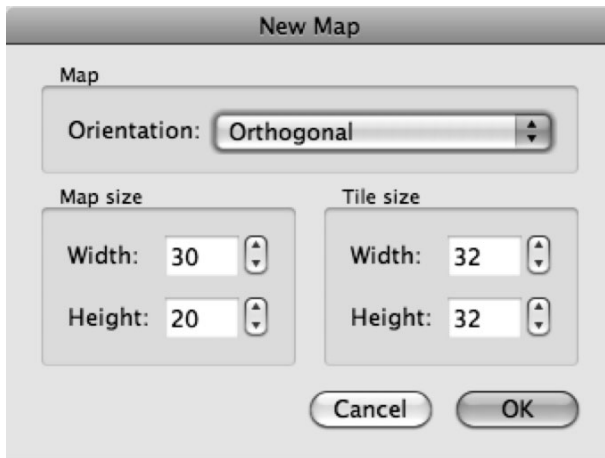


Figure 10–5. *Creating a new tilemap in Tiled*

Currently, Tiled supports orthogonal (rectangular) tilemaps as well as isometric tilemaps. The map size is given in tiles, not pixels. In this case, the new map will be 30×20 tiles, with the tile images having 32×32-pixel dimensions because that’s the size of the tile images. It’s crucial that the tile size matches the size of your tile images, or they will not align.

The new map will be completely empty, and there’s no tileset loaded that you can draw from. You can add a tileset by selecting **Map ▾ New Tileset** from Tiled’s menu. This will open the New Tileset dialog (shown in Figure 10–6), where you can browse for the proper tileset image. A tileset is just a name for an image containing multiple tiles with equal spacing, so you could also call it a texture atlas containing only images of the same size.

NOTE: The function **Map ▾ Add External Tileset** is used only to import a previously exported tileset to share the same tileset with multiple TMX maps. You can export a tileset by right-clicking the tileset view in the lower-right corner of the Tiled window and then select **Export Tileset As**.

I will use the `dg_grounds32.png` tileset. These tiles were drawn by David E. Gervais and published under the Creative Commons License, meaning you are free to share and remix his work as long as you give credit to him. You can download more of his tilesets from this web site: <http://pousse.rapriere.free.fr/tome/index.htm>.

In Figure 10–6, I have already added the `dg_grounds32.png` tileset image by locating it through the Browse button in the Resources folder of the Tilemap01 project. If you check the ☐ **Use transparent color** check box, transparent areas will simply be drawn in pink (the default color). You can leave this box unchecked for now since the tiles in use have no transparent areas.

The tile width and height are the dimensions of individual tiles in the tileset. They should match the tile size of 32×32 pixels, which you set when creating the new map. The

Margin and Spacing settings determine how many pixels away from the image border the tiles are and how many pixels of space are between them. In the case of the `dg_grounds32.png`, there is no spacing at all, so I set both values to 0.

If you had your tiles aligned by TexturePacker to create a tileset texture, you must enter the pixel-padding value used by TexturePacker in the Margin and Spacing fields. By default TexturePacker uses a padding of 0 pixels.

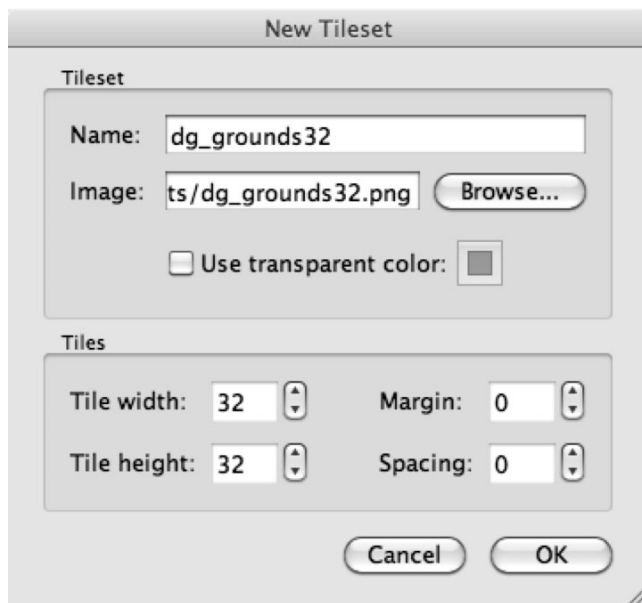


Figure 10–6. Creating a new tilesheet from an image file

When loading a new tilesheet image, make sure the tilesheet image is already located in your project's resource folder. You should then also make sure to save the tilemap TMX file to the same folder where the tilesheet image used by the tilemap is located. Otherwise, cocos2d might fail to load the tilesheet image; trying to load the TMX file will then cause a runtime exception. The culprit is that TMX files reference the tilesheet image relative to the location the TMX file is saved to. If they are not both in the same folder, cocos2d may be unable to find the image because the folder structure is not preserved when the app is installed in the simulator or on the device.

TIP: TMX files are plain XML files, so you might want to peek inside if you're curious. If you see the image file referenced with parts of a path, then cocos2d is unlikely to load the referenced image file. The image reference should list just the image file name without any path components, like so: `<image source="tiles.png"/>`.

Designing a Tilemap

With the tileset loaded, you'll be faced with a blank map, an invitation for your creativity to come up with great ideas for a tilemap. What's even better is to get rid of the blank tilemap as the very first step. It's very helpful to start the tilemap with a default floor tile. I selected the Bucket Fill tool and picked a bright grass tile so that my tilemap is now a lush meadow sort of. You can see it in Figure 10-7.

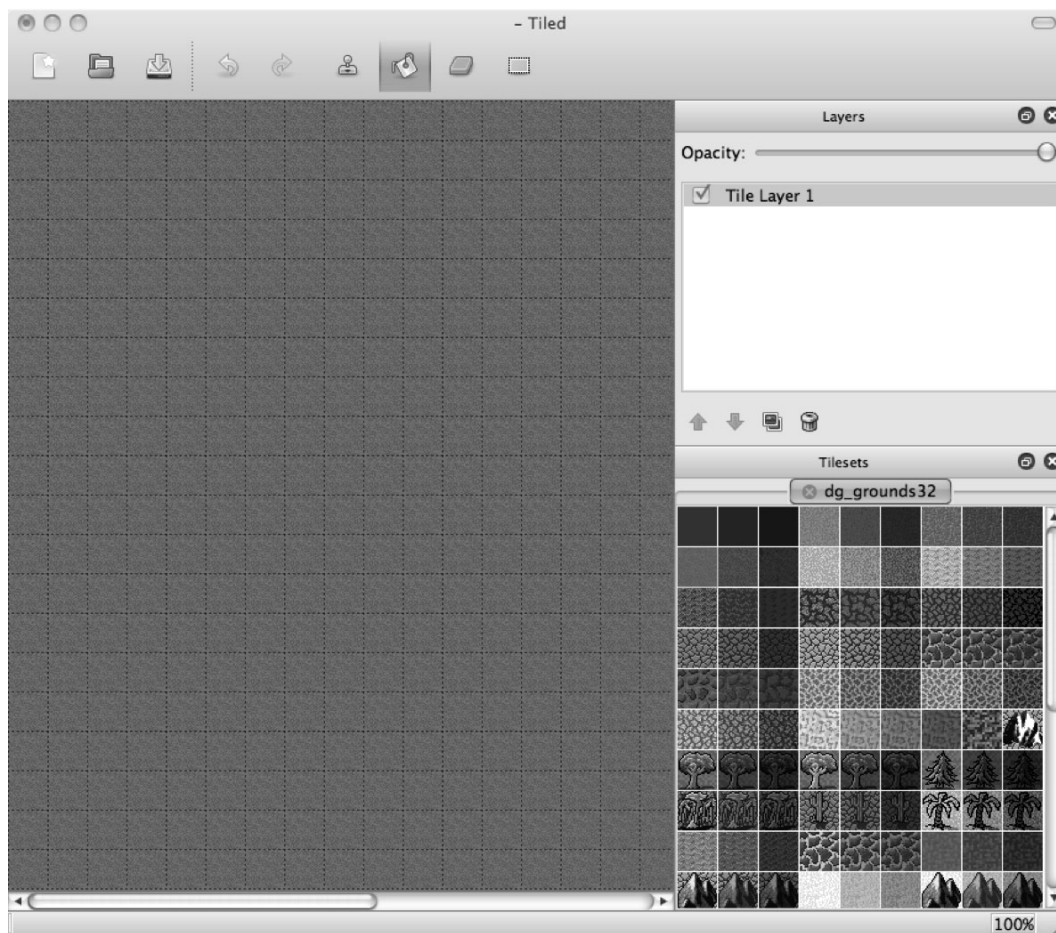


Figure 10-7. An empty map with the *dg_grounds32* tileset loaded, waiting for your inspiration

TIP: If you ever notice something missing from Tiled's window, check the **View** menu. You can hide and unhide the Tilesets, Layers, and History views. This is important to know because it's the only way to bring back these views in case you have clicked the X button on one of these views.

Tiled uses four modes for editing the tilemap, indicated by the four rightmost icons on the toolbar. They are Stamp Brush (hotkey B), which allows you to draw the current selection in the tileset; Bucket Fill (hotkey F), which fills areas of connected, identical tiles; Eraser (hotkey E), which erases tiles; and Rectangular Select (hotkey R), with which you can select a range of tiles and then copy and paste the selection.

You can also zoom the tilemap. If you have a mouse with a mousewheel, simply hold down the Command key and spin the scroll wheel to zoom in and out. Alternatively, you can also zoom in and out by pressing the Command key with either the plus and minus sign, respectively.

You'll spend most of your time picking a tile from the tileset and drawing it onto the tilemap with the Stamp Brush selected. Placing tile by tile, you'll create your tile-based game world.

You can also edit tiles in multiple layers by adding more layers in the Layers view. From the menu, choose **Layer ▾ Add Tile Layer** to create a new layer for tiles. Using multiple tile layers allows you to switch out areas of the tilemap in cocos2d. In the TileMap01 example project, I'm using it to switch parts of the map between winter and summer.

You can also choose **Layer ▾ Add Object Layer** to add a layer for adding objects. There are two types of objects; the regular objects are simply rectangles that you can place, and the other type are tile objects that allow you to freely place a tile anywhere on the map, without snapping to the tile grid. You can use the rectangle objects to annotate the map with custom information, such as spawn points, teleport locations, or trigger areas. The tile objects are most commonly used to place smaller things like swords, flowers, candles, and other items directly onto the tile world.

To work with objects, there are additional buttons in the Tiled toolbar: Select Objects (hotkey S), Insert Objects (hotkey O), and Insert Tile Objects (hotkey T). To insert a rectangle object, click the **Insert Objects** icon and click in the tilemap world to create point objects (rectangles with zero width and height), or click and drag down and to the right to create a rectangle. To insert a tile object, click the **Insert Tile Objects** icon, select a tile from the tileset, and then click the tilemap world to add a new tile object.

Some functionality in Tiled is hidden in context menus. For example, the rectangle objects I just mentioned can be deleted by right-clicking them in the Tilemap view and selecting **Remove Object**. Note that you also need to have the object layer highlighted in the Layers list view for the context menu to appear.

You can also edit properties of objects, layers, and tiles by right-clicking them and clicking the corresponding **Properties** menu item. One use for that is to create an additional tile layer by using **Layer ▾ Add Tile Layer**. This will be a layer used to tell the game about certain properties of tiles. I called it `GameEventLayer` because it will be used to define trigger areas for certain game events.

With `GameEventLayer` selected, choose **Map ▾ New Tilesheet** and load `game-events.png` from the same folder as `dg_grounds32.png`. It contains only three tiles. Right-click one, select **Tile Properties**, and add the `isWater` property, as shown in Figure 10–8.

CAUTION: Keep in mind that each tile layer creates some overhead, especially if you set tiles in multiple layers at the same location. This will cause both layers to be drawn and can adversely affect the game's performance. It is recommended to keep the number of tile layers to a minimum. Two to four tile layers should be sufficient for most games. Be sure to take a look at your game's framerate on the device after you have added a new tile layer and drawn on it.

You should also be aware that Tiled allows you to add more than one tileset for a layer. However, cocos2d supports only one tileset per layer.



Figure 10–8. Adding a tile property

You can then draw over the tilemap using the tile to whose properties you just added the `isWater` property. Ideally, draw it over the river. If you want to see the tiles underneath what you're drawing, you can use the Opacity slider for the `GameEventLayer` in the Layers view. Or, click the layer's check box to hide or unhide everything drawn on this particular layer.

Just be sure to enable all layer check boxes before saving the TMX tilemap. Cocos2d does not load layers that are unchecked in Tiled.

When you're done with this, you should have a tilemap similar to the one in Figure 10–9. Save it in the same `TileMap01 Resources` folder where the tileset images are.

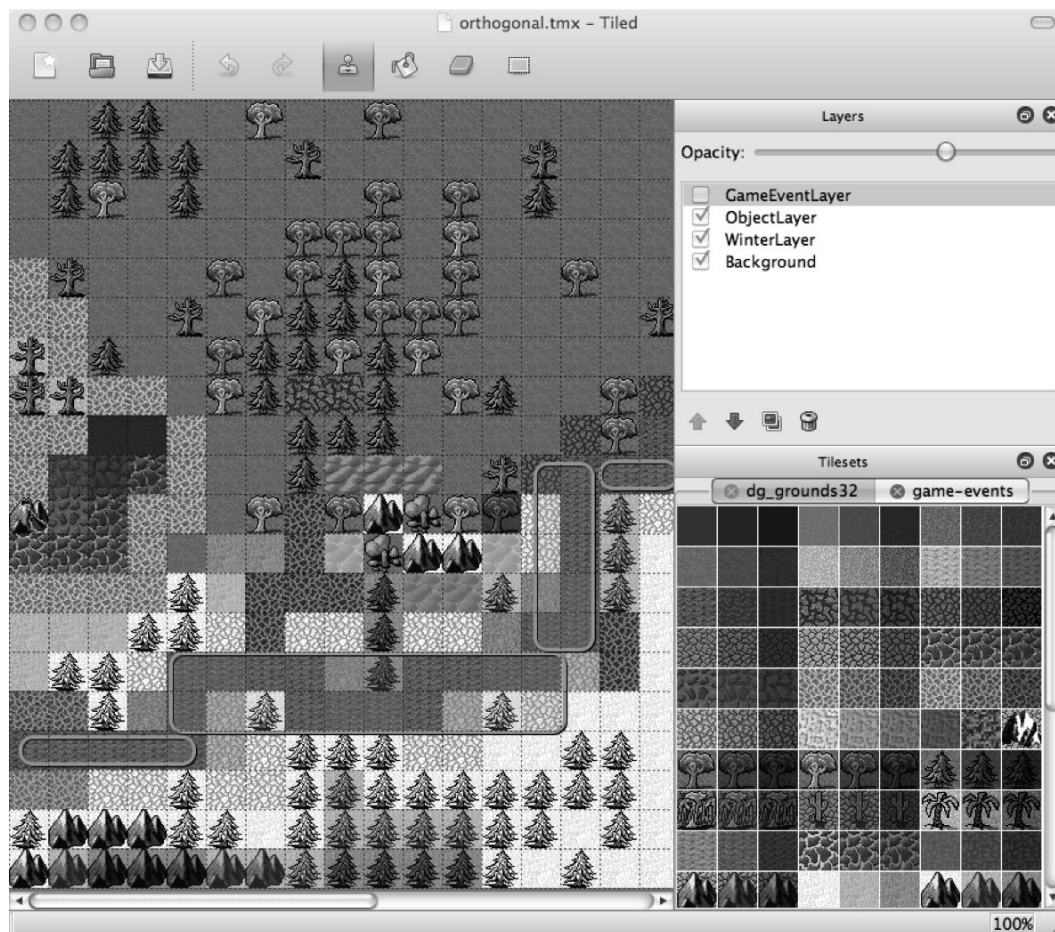


Figure 10–9. A completed tilemap with three tile layers and an object layer

Using Orthogonal Tilemaps with Cocos2d

To use TMX tilemaps with cocos2d, you first have to add the TMX file and the accompanying tileset image files as resources to your Xcode project. In the TileMap01 project, I added `orthogonal.tmx` along with the tilesets `dg_grounds32.png` and `game-events.png`. Loading and displaying the tilemap is very straightforward; the following code is from the `init` method of the `TileMapLayer` class:

```
CCTMXTiledMap* tileMap = [CCTMXTiledMap tiledMapWithTMXFile:@"orthogonal.tmx"];
[self addChild:tileMap z:-1 tag:TileMapNode];

CCTMXLayer* eventLayer = [tileMap layerNamed:@"GameEventLayer"];
eventLayer.visible = NO;
```

The `CCTMXTiledMap` class is initialized with the name of the TMX file and then added as a child with a tag so that it can be retrieved later. A member variable would of course work

just as well. The next step is to retrieve the CCTMXTiledMap used for game events by using the `layerNamed` method and providing the name of the layer as it was named in Tiled. Because the game events layer will be used only as hints for code to determine properties of certain tiles, this layer should not be rendered at all. Note that if you uncheck the layer in Tiled, it won't be displayed, but you will also not have access to its tiles and tile properties either.

If you run the project now, you'll see a tilemap just like in Figure 10-10.



Figure 10-10. *The orthogonal tilemap in the iPhone Simulator*

Right now you can't do anything with the tilemap, but I'd like to change that. Moving on to the `TileMap02` project, I'd like to be able to find the `isWater` tiles. I've added the `ccTouchesBegan` method, as shown in Listing 10-1, in order to determine the tile that the player is touching.

Listing 10-1. *Determining a Tile's Properties*

```
-(void) ccTouchesBegan:(NSSet *)touches withEvent:(UIEvent *)event
{
    CCNode* node = [self getChildByTag:TileMapNode];
    NSAssert([node isKindOfClass:[CCTMXTiledMap class]], @"not a CCTMXTiledMap");
    CCTMXTiledMap* tileMap = (CCTMXTiledMap*)node;

    // Get the position in tile coordinates from the touch location
    CGPoint touchLocation = [self locationFromTouch:[touches anyObject]];
    CGPoint tilePos = [self tilePosFromLocation:touchLocation tileMap:tileMap];

    // Check if the touch was on water (e.g., tiles with isWater property)
    bool isTouchOnWater = NO;

    CCTMXLayer* eventLayer = [tileMap layerNamed:@"GameEventLayer"];
    int tileGID = [eventLayer tileGIDAt:tilePos];

    if (tileGID != 0)
    {
        NSDictionary* properties = [tileMap propertiesForGID:tileGID];
        if (properties)
        {
            NSString* isWaterProperty = [properties valueForKey:@"isWater"];
            isTouchOnWater = ([isWaterProperty boolValue] == YES);
        }
    }
}
```

```

    }

    // Decide what to do depending on where the touch was
    if (isTouchOnWater)
    {
        [[SimpleAudioEngine sharedEngine] playEffect:@"alien-sfx.caf"];
    }
    else
    {
        // Get the winter layer and toggle its visibility
        CCTMXLayer* winterLayer = [tileMap layerNamed:@"WinterLayer"];
        winterLayer.visible = !winterLayer.visible;
    }
}

```

The CCTMXTiledMap is retrieved as usual. The location of the touch is first converted into screen coordinates and then used to retrieve the tilePos containing the indices into the tilemap at that specific screen location. I'll get to the tilePosFromLocation method in a minute. For now, just know that it returns the index of the touched tile.

At this point, I'd like to introduce the concept of global identifiers (GIDs) for tiles, which are unique integer numbers assigned to each tile used in a tilemap. The tiles in a map are consecutively numbered, starting with 1. A GID of 0 represents an empty tile. With the tileGIDat method of the CCTMXLayer, you can determine the GID number of the tile at the given tile coordinates.

Next, the CCTMXLayer named GameEventLayer is obtained from the tilemap. This is the layer where I defined the isWater tile and drew it over the river tiles. The tileGIDat method returns the unique identifier for this tile. If the identifier happens to be 0, it means there is no tile at this position on this layer□ in that case, it's already clear that the touched tile can't be an isWater tile.

The CCTMXTiledMap has a propertiesForGID method, which returns an NSDictionary if there are properties available for the tile with the given identifier (GID). This NSDictionary contains the properties edited in Tiled (see Figure 10–8). The dictionary stores any key/value pairs as NSString objects. If you want to see what's in a particular NSDictionary for debugging purposes, you can use a CLOG statement like this:

```
CLOG(@"NSDictionary 'properties' contains:\n%@", properties);
```

This will print out a line similar to the following in the Debugger Console window:

```

2010...08-30 19:50:52.344 Tilemap[978:207] NSDictionary 'properties' contains:
{
    isWater = 1;
}

```

You'll be dealing with a variety of NSDictionary objects while working with tilemaps. Logging its contents allows you to peek inside any NSDictionary, or any iPhone SDK collection class, for that matter. This will come in handy at times.

Each property in an NSDictionary can be retrieved by its name through the NSDictionary method valueForKey, which returns an NSString. To get a bool value from the NSString, you can simply use the NSString's boolValue method. In much the same

way, you can retrieve integer and floating-point values using NSString's intValue and floatValue methods, respectively.

At the end of ccTouchesBegan, I check whether the touch was on water, and if so, a sound is played. Otherwise, I retrieve the WinterLayer and toggle its visible property by negating it. Changing seasons has never been this simple! The effect should illustrate how you can use multiple layers in Tiled to achieve changes on a global scale without having to load a completely separate tilemap.

For more local changes to individual tiles, you can make use of the removeTileAt and setTileGID methods to remove or replace tiles of a specific layer during game play:

```
[winterLayer removeTileAt:tilePos];
[winterLayer setTileGID:tileGID at:tilePos];
```

Locating Touched Tiles

I mentioned the tilePosFromLocation method earlier, and I'll repeat the two relevant lines here:

```
// Get the position in tile coordinates from the touch location
CGPoint touchLocation = [self locationFromTouch:[touches anyObject]];
CGPoint tilePos = [self tilePosFromLocation:touchLocation tileMap:tileMap];
```

First, the position of the touch is mapped to screen coordinates. I've done this before, but since you'll be needing this code a lot, I've provided it in Listing 10-2 for your reference.

Listing 10-2. *Determining the Position of a Touch*

```
-(CGPoint) locationFromTouch:(UITouch*)touch
{
    CGPoint touchLocation = [touch locationInView: [touch view]];
    return [[CCDirector sharedDirector] convertToGL:touchLocation];
}
```

With the touch location converted to screen coordinates, the tilePosFromLocation method is called. It gets both the touch location and a pointer to the tileMap as parameters. The method in Listing 10-3 contains a bit of math, which I'll explain in a second—hold your breath:

Listing 10-3. *Converting Location to Tile Coordinates*

```
-(CGPoint) tilePosFromLocation:(CGPoint)location tileMap:(CCTMXTileMap*)tileMap
{
    // Tilemap position must be offset, in case the tilemap is scrolling
    CGPoint pos = ccpSub(location, tileMap.position);

    // Cast to int makes sure that result is in whole numbers
    pos.x = (int)(pos.x / tileMap.tileSize.width);
    pos.y = (int)((tileMap.mapSize.height * tileMap.tileSize.height - pos.y) /
        tileMap.tileSize.height);

    CCLOG(@"touch at (%.0f, %.0f) is at tileCoord (%i, %i)", location.x, location.y,
        (int)pos.x, (int)pos.y);
}
```

```

    NSAssert(pos.x >= 0 && pos.y >= 0 && pos.x < tileMap.mapSize.width &&
        pos.y < tileMap.mapSize.height, @"%@: coordinates (%i, %i) out of bounds!",
        NSStringFromSelector(_cmd), (int)pos.x, (int)pos.y);

    return pos;
}

```

Still with me? If you've worked with tilemaps before, this bit of code should be familiar, but if not, you may be at a loss. I'll explain. The first thing this method does is subtract the current `tileMap.position` from the touch location. The upcoming `Tilemap03` example project adds tilemap scrolling, so the tilemap's position will most likely not be at 0,0.

To make the viewpoint scroll further up (north) and to the right (east), you actually have to change its position by negative amounts. That is because the tilemap starts at position 0,0, which positions the map's bottom-left corner at the very bottom left of the screen. The tilemap's 0,0 point coincides with the screen's 0,0 point initially. If you were to move the tilemap to position 100,100, it would seem as if the viewpoint were moving toward the left and down. The common mistake is to assume that you're moving the viewpoint, which you are not. The tilemap layer is what's moving, and to scroll further toward the center of the tilemap, you have to offset the tilemap by negative values.

The rest is simple math: to get the proper offset from the tilemap (whose position we know is always negative), we have to subtract the touch location and `tileMap.position`. The concrete numbers reveal that subtracting a negative number is actually an addition:

```
location(240, 160) ... tileMap.position(-100, -100) = pos(340, 260)
```

With the tilemap layer moved -100,-100 pixels away from the screen's 0,0 point and the touch being at 240,160 pixels on the screen, the total offset of the touch location from the tilemap's position is 340,260 pixels away from the current `tileMap.position`.

With the scrolling offset taken into account, we can get the tile coordinates for the tile at this location into the tilemap. At this point, you have to consider that the tile coordinates' 0,0 tile is at the top-left corner of the tilemap. Contrary to screen coordinates, where the 0,0 point (point of origin) is at the lower-left corner, the tilemap coordinates start at the upper-left corner. Figure 10-11 shows the x,y coordinates of a series of tiles. The screenshot was made with the Tiled Java version by enabling **View ▢ Show Coordinates**, which is a feature that isn't available yet in the Tiled Qt version.

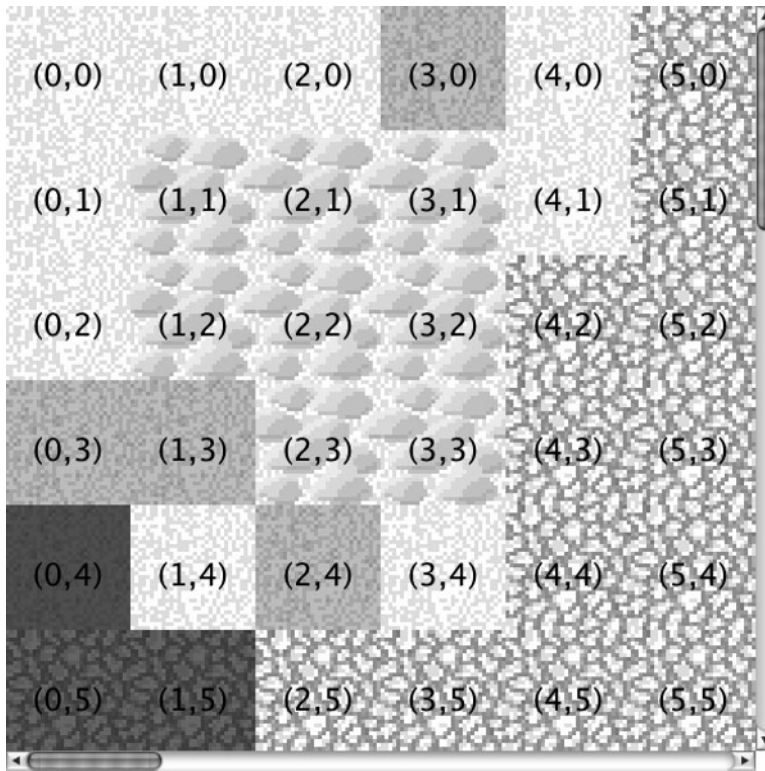


Figure 10-11. *The coordinate system of an orthogonal tilemap*

So as not to get confused, this is the line of code that calculates the tile coordinate's x position:

```
pos.x = (int)(pos.x / tileMap.tileSize.width);
```

The `tileMap.tileSize` property is the size of the tiles in the tileset, which in this case is `32×32` (see also Figure 10-6). If the touch were at the 340 x position, the calculation would reveal the following:

$$340 / 32 = 10.625$$

That can't be right, though. We're looking for a tile's x coordinate, which is never a fractional number! The reason is, of course, that the touch was somewhere inside the tile we're looking for (that is, inside a `32×32` square area). The simple trick of casting the result to an int value will get rid of the fractional part and assign this to `pos.x`:

```
pos.x = (int)10.625 // pos.x == 10
```

Casting to an int will remove the fractional part. You can safely get rid of the fractional part because it's simply not relevant—actually it's harmful. If you didn't cast away the fractional part but used the noninteger coordinate, in this example 10.625, to try to retrieve the tile at a tile coordinate 10.625, you'd receive a runtime error because there is only a tile at x coordinates 10 and 11, not at 10.625.

A slightly more complicated calculation is used to get the tile's y coordinate:

```
pos.y = (int)((tileMap.mapSize.height * tileMap.tileSize.height - pos.y) /
    tileMap.tileSize.height);
```

Note that the parentheses are important to make sure that the division is done last. In actual numbers, this calculation may be easier to understand. As shown in Figure 10–5, the `tileMap.mapSize` is 30×20 tiles, and as I mentioned earlier, `tileMap.tileSize` is 32×32 pixels. The calculation then looks like this:

```
pos.y = (int)((20 * 32 ... 260) / 32)
```

Multiplying `tileMap.mapSize.height` with `tileMap.tileSize.height` returns the full height of the tilemap in pixels. This is necessary because the tilemap starts counting y coordinates from top to bottom, whereas screen y coordinates count from bottom to top. By calculating the bottommost y coordinate of the tilemap and subtracting the current y position 260 from that, you get the correct y position of the touch into the tilemap, in pixels. And because it is a pixel coordinate, you need to divide by the `tileSize.height` and then cast down to an `int` value to get the tile's y coordinate.

The `CCL0G` and `NSAssert` lines are helpful for seeing the results of the calculation in the Debugger Console window, as well as ensuring that tile coordinates never take on illegal values. It's both a learning tool and an insurance policy.

An Exercise in Optimization and Readability

Since the tilemap's size never changes, you can optimize the calculation a little to get the y tile coordinate by adding a member variable to the class's `@interface`, which will be used to store the tilemap's height in pixels:

```
float tileMapHeightInPixels;
```

You can then make the calculation to get the `tileMapHeightInPixels` just once in the `init` method, right after the tilemap is loaded:

```
CCTMXTiledMap* tileMap = [CCTMXTiledMap tiledMapWithTMXFile:@"orthogonal.tmx"];
tileMapHeightInPixels = tileMap.mapSize.height * tileMap.tileSize.height;
```

Then you can rewrite the calculation, saving a multiplication every time you call the `tilePosFromLocation` method:

```
pos.y = (int)((tileMapHeightInPixels - pos.y) / tileMap.tileSize.height);
```

It may not win any awards for best optimization ever—it's only a very tiny improvement in performance. But every bit counts, and it does make the calculation easier to read by taking away complexity and putting a readable variable name in its place.

Working with the Object Layer

The `orthogonal.tmx` tilemap I've created as an example tilemap for this chapter also contains an object layer, fittingly named `ObjectLayer`. You can create object layers in Tiled by choosing **Layer ▸ Add Object Layer**. Then you can click inside the tilemap and draw

rectangles. I think the name *object layer* is a bit unfortunate and misleading, because most games will use these rectangles as points of interest and trigger areas, and not as actual objects.

In the Tilemap03 project, I've added a bit more code to the `ccTouchesBegan` method to interact with the object layer. Listing 10–4 shows the relevant part of the code, which follows directly after the `isWater` check:

Listing 10–4. Detecting If a Touch Was Inside an ObjectLayer Rectangle

```
// Check if the touch was within one of the rectangle objects
CCTMXObjectGroup* objectLayer = [tileMap objectGroupNamed:@"ObjectLayer"];

bool isTouchInRectangle = NO;
int numObjects = [objectLayer.objects count];
for (int i = 0; i < numObjects; i++)
{
    NSDictionary* properties = [objectLayer.objects objectAtIndex:i];
    CGRect rect = [self getRectFromObjectProperties:properties tileMap:tileMap];

    if (CGRectContainsPoint(rect, touchLocation))
    {
        isTouchInRectangle = YES;
        break;
    }
}
```

Because object layers are a different kind of layer, you can't get them via the `layerNamed` method of the tilemap. The object layer in cocos2d is the class `CCTMXObjectGroup`, another unfortunate naming mishap, since Tiled refers to it as an *object layer*, not an *object group*. In any case, you can get the `CCTMXObjectGroup` for the object layer named simply `ObjectLayer` by using the tilemap's `objectGroupNamed` method and specifying the object layer's name as defined in Tiled.

Next, I iterate over the `objectLayer.objects` `NSMutableArray`, which contains a list of `NSDictionary` items. Sound familiar? Yes, these are the same `NSDictionary` properties returned by the tilemap's `propertiesForGID` method, as shown earlier—except that the contents of these `NSDictionary` items are given by Tiled and not user editable. They simply contain the coordinates for each rectangle. The method `getRectFromObjectProperties` returns the rectangle:

```
-(CGRect) getRectFromObjectProperties:(NSDictionary*)dict
tileMap:(CCTMXTileMap*)tileMap
{
    float x, y, width, height;

    x = [[dict valueForKey:@"x"] floatValue] + tileMap.position.x;
    y = [[dict valueForKey:@"y"] floatValue] + tileMap.position.y;
    width = [[dict valueForKey:@"width"] floatValue];
    height = [[dict valueForKey:@"height"] floatValue];

    return CGRectMake(x, y, width, height);
}
```

The keys `x`, `y`, `width`, and `height` are set by `Tiled`. I simply retrieve them from the `NSDictionary` via `valueForKey` and use the `floatValue` method to convert the values from `NSString` to actual floating-point numbers. The `x` and `y` values need to be offset with the `tileMap`'s position, because the rectangles need to be moving along with the `tilemap`. At the end, a `CGRect` is returned by calling the `CGRectMake` convenience method.

The remaining code in `ccTouchesBegan` then simply checks whether the touch location is contained in the `rect` via `CGRectContainsPoint`. If it is, the `isTouchInRectangle` flag is set to `true`, and the `for` loop is aborted by using the `break` statement. There's no need to check another rectangle for containing the touch location. At the end of `ccTouchesBegan`, the `isTouchInRectangle` flag is then used to decide whether to play a particle effect at the touch location. So, this code is creating an explosion particle effect whenever you touch inside a rectangle:

```
if (isTouchOnWater)
{
    [[SimpleAudioEngine sharedEngine] playEffect:@"alien-sfx.caf"];
}
else if (isTouchInRectangle)
{
    CCParticleSystem* system = [CCQuadParticleSystem particleWithFile:@"fx-explosion.plist"];
    system.autoRemoveOnFinish = YES;
    system.position = touchLocation;
    [self addChild:system z:1];
}
```

Drawing the Object Layer Rectangles

When you run the `Tilemap03` project, you'll notice that the object layer rectangles are drawn over the `tilemap`, as shown in Figure 10–12. This is not a standard feature of `tilemaps` or object layers. Instead, the rectangles are drawn using OpenGL ES code. Every `CCNode` has a `...(void) draw` method that you can override to add custom OpenGL ES code. I tend to use this a lot to debug my code visually by drawing lines, circles, and rectangles that may be used for collision and distance tests, among other things. In this case, it's very useful to actually see where the object layer areas are. Visualizing such information beats looking up and comparing coordinates in the debugger. Our minds are much better at assessing visual information than at comparing and calculating numbers. Use this to your advantage!



Figure 10–12. The tilemap with object layer rectangles displayed using OpenGL ES code

The `...(void) draw` method just needs to be in the class, and it will be called automatically every frame. However, you should refrain from using the draw method to modify properties of nodes, because this can interfere with drawing the nodes. Listing 10–5 shows the draw method of the `TileMapLayer` class.

Listing 10–5. Drawing *ObjectLayer* Rectangles

```
-(void) draw
{
    CCNode* node = [self getChildByTag:TileMapNode];
    NSAssert([node isKindOfClass:[CCTMXTileMap class]], @"not a CCTMXTileMap");
    CCTMXTileMap* tileMap = (CCTMXTileMap*)node;

    // Get the object layer
    CCTMXObjectGroup* objectLayer = [tileMap objectGroupNamed:@"ObjectLayer"];

    // Make the line 3 pixels thick
    glLineWidth(3.0f);
    glColor4f(1, 0, 1, 1);

    int numObjects = [[objectLayer objects] count];
    for (int i = 0; i < numObjects; i++)
    {
        NSDictionary* properties = [[objectLayer objects] objectAtIndex:i];
        CGRect rect = [self getRectFromObjectProperties:properties tileMap:tileMap];
        [self drawRect:rect];
    }

    glLineWidth(1.0f);
    glColor4f(1, 1, 1, 1);
}
```

First, I get the tilemap by its tag and then the `CCTMXObjectGroup` by using the `objectGroupNamed` method. I then set the line width to 3 pixels by using the OpenGL ES method `glLineWidth` and set the color to purple by using `glColor4f`. This affects line thickness and color of all subsequent lines drawn with OpenGL ES—not just in the current method but possibly other nodes that use OpenGL ES code for drawing (for example, any of the convenience methods for drawing lines, circles, and polygons defined in cocos2d's `CCDrawingPrimitives.h` header file). That is why I reset `glLineWidth`

and `glColor4f` after I'm done drawing. It is good style in OpenGL code to leave its state like you found it; otherwise, it might alter the way other draw code produces its output. OpenGL is a state machine, so every setting you change is remembered and may affect subsequent drawing methods. To avoid this, any OpenGL settings you change should be set back to a safe default after you're done drawing.

NOTE: Code inside the `-(void) draw` method is always drawn at a z-order of 0. It is also drawn before all other nodes at z-order 0, which means that any OpenGL ES code will be overdrawn by other nodes if they are also at z-order 0. In the case of the object layer draw code, I had to add the `tileMap` at a z-order of `.1` for the rectangles to be drawn over the tilemap.

Just like before, I iterate over all object layer objects and get their properties from `NSDictionary` to get the `CGRect` of that object, which is then passed to the `drawRect` method. Unfortunately, `cocos2d` omitted this particular convenience method, but it's easy enough to add using `ccDrawLine`, as Listing 10-6 shows.

Listing 10-6. Drawing a Rectangle

```
-(void) drawRect:(CGRect)rect
{
    // The rect is drawn using four lines
    CGPoint pos1, pos2, pos3, pos4;
    pos1 = CGPointMake(rect.origin.x, rect.origin.y);
    pos2 = CGPointMake(rect.origin.x, rect.origin.y + rect.size.height);
    pos3 = CGPointMake(rect.origin.x + rect.size.width,
        rect.origin.y + rect.size.height);
    pos4 = CGPointMake(rect.origin.x + rect.size.width, rect.origin.y);

    ccDrawLine(pos1, pos2);
    ccDrawLine(pos2, pos3);
    ccDrawLine(pos3, pos4);
    ccDrawLine(pos4, pos1);
}
```

For each corner of the rectangle, a `CGPoint` is created, which is then used in four `ccDrawLine` methods to draw the lines between the corners of the rectangle. You may want to remember this method and put it in a safe place, because you'll probably need it again.

Note that the `draw` and `drawRect` methods are enclosed in `#ifdef DEBUG` and `#endif` statements. This means that the object layer rectangles will not be drawn in release builds, because I only need them for debugging and illustration purposes—the end user should never see them.

```
#ifdef DEBUG
-(void) drawRect:(CGRect)rect
{
    ...
}

-(void) draw
{

```

```

    ...
}
#endif

```

Scrolling the Tilemap

The best part comes last: scrolling. It's actually straightforward because only the CCTMXTileMap needs to be moved. In the Tilemap04 project, I've added the call to the centerTileMapOnTileCoord method in ccTouchesBegan right after obtaining the tile coordinates of the touch:

```

-(void) ccTouchesBegan:(NSSet *)touches withEvent:(UIEvent *)event
{
    ...
    // Get the position in tile coordinates from the touch location
    CGPoint touchLocation = [self locationFromTouches:touches];
    CGPoint tilePos = [self tilePosFromLocation:touchLocation tileMap:tileMap];

    // Move tilemap so that the touched tile is at the center of the screen
    [self centerTileMapOnTileCoord:tilePos tileMap:tileMap];
    ...
}

```

Listing 10–7 shows the centerTileMapOnTileCoord method, which moves the tilemap so that the touched tile is at the center of the screen. It also stops the tilemap from scrolling further if any tilemap border already aligns with the screen edge.

Listing 10–7. Centering the Tilemap on a Tile Coordinate

```

-(void) centerTileMapOnTileCoord:(CGPoint)tilePos tileMap:(CCTMXTileMap*)tileMap
{
    // Center tilemap on the given tile pos
    CGSize screenSize = [[CCDirector sharedDirector] winSize];
    CGPoint screenCenter = CGPointMake(screenSize.width * 0.5f,
    screenSize.height * 0.5f);

    // Tile coordinates are counted from upper-left corner
    tilePos.y = (tileMap.mapSize.height - 1) - tilePos.y;

    // Point is now at lower-left corner of the screen
    CGPoint scrollPosition = CGPointMake(-(tilePos.x * tileMap.tileSize.width),
    -(tilePos.y * tileMap.tileSize.height));

    // Offset point to center of screen and center of tile
    scrollPosition.x += screenCenter.x - tileMap.tileSize.width * 0.5f;
    scrollPosition.y += screenCenter.y - tileMap.tileSize.height * 0.5f;

    // Make sure tilemap scrolling stops at the tilemap borders
    scrollPosition.x = MIN(scrollPosition.x, 0);
    scrollPosition.x = MAX(scrollPosition.x, -screenSize.width);
    scrollPosition.y = MIN(scrollPosition.y, 0);
    scrollPosition.y = MAX(scrollPosition.y, -screenSize.height);

    CCAction* move = [CCMoveTo actionWithDuration:0.2f position: scrollPosition];
    [tileMap stopAllActions];
    [tileMap runAction:move];
}

```

After obtaining the center position of the screen, I modify the `tilePos y` coordinate because tilemap coordinates are counted from top to bottom (see Figure 10–11), while screen coordinates increase from bottom up. In effect, I convert the `tilePos y` coordinate as if it were counted from bottom up. In addition, I subtract 1 from the map's height to account for the fact that tile coordinates are counted from 0. In other words, if the map's height were 10, only the tile coordinates 0 to 9 would be valid.

Next, the `scrollPosition CGPoint` is created, which will become the position the tilemap will be moved to. The first step is to multiply the tile coordinates with the tilemap's `tileSize`. You may be wondering why I negate the `tilePosInPixels` coordinates. It's simply because if I want the tiles to move from top right to bottom left, I have to move the tilemap down and to the left by decreasing the coordinates.

The next big block modifies the coordinates of the `scrollPosition` to center the tile on the screen's center point. You also need to take into account the center of the tile itself, which is why half the `tileSize` is deducted from the `screenCenter` offset.

By using the Objective-C language's `MIN` and `MAX` macros, it is ensured that the `scrollPosition` is kept within the bounds of the tilemap, so as to not reveal anything past the borders of the tilemap. `MIN` and `MAX` return the minimum and maximum values of their two parameters, respectively, and are a more compact and readable solution than conditional assignments using `if` and `else` statements.

Finally, a `CCMoveTo` action is used to scroll the tilemap node so that the touched tile is centered on the screen. The result is a tilemap that scrolls to the tile that you tap. You can use the same method to scroll to a tile of interest—for example, the player's position.

TIP: As for the player character itself, you'll find an implementation in the next chapter about isometric tilemaps. You can apply the same principle to orthogonal tilemaps. And this Cocos2D forum thread will get you started with pathfinding on orthogonal tilemaps, source code included: www.cocos2d-iphone.org/forum/topic/19463.

If you're interested in a complete, working, and ready-made solution for a hack-and-slash game on orthogonal tilemaps, including a great tutorial, I recommend looking at Nate Weiss' iPhone Game Kit: www.iphonegamekit.com.

Summary

You should now have a fair understanding of what tilemaps are and how to work with Tiled Map Editor to create a tilemap with multiple layers and properties that can be used by your game.

Loading and displaying a tilemap with `cocos2d` is a simple task that quickly grows in complexity when it comes to obtaining tile and object layers, modifying them, and reading their properties. You also learned how to determine the tile coordinate of a

touch location and how to use tile coordinates to scroll the tilemap so that it centers the touched tile on the screen.

I even got you acquainted with custom drawing and a bit of OpenGL ES code to render the object layer rectangles on the tilemap for debugging purposes.

Isometric Tilemaps

With isometric tilemaps you can get the best of both worlds—using two-dimensional graphics to achieve a three-dimensional look. This is the reason isometric tilemap games are so widely popular. Isometric tilemap games started to get a strong foothold in the late 1990s but slowly disappeared with the increasing 3D-rendering performance of desktop computers and consoles. They reemerged with force in recent years in mobile games and social web games, where 3D rendering is very costly or not even available. Examples range from classic computer role-playing games like *Ultima VII* and *Diablo* to current Facebook-hype *Farmville* and many of its official and unofficial companion games.

Isometric games allow you to create believable game worlds that seem to have spatial depth with relatively simple graphics and tools. In addition, 2D graphics require far less powerful devices than real 3D computer graphics.

Figure 11–1 shows an example of the isometric tilemap game we’ll build in this chapter. You’ll control a ninja character who sneaks around in this world, avoiding collisions with walls and mountains. The ninja will also be able to hide behind certain objects, such as trees and cacti.



Figure 11–1. *An isometric tilemap game*

NOTE: All tilesets used in this chapter were created by David E. Gervais and published under the Creative Commons license. You can download more of his work at <http://pousse.rapriere.free.fr/tome/index.htm>.

Designing Isometric Tile Graphics

So you understand how to design isometric graphics, I will first introduce the concept of projection. In a 3D world, you can look at the objects from all sides, because 3D worlds can freely project the 3D world onto your two-dimensional screen from any angle and position. This conversion from a three-dimensional world onto a two-dimensional screen is called *perspective projection*. Taking a photograph is also a form of perspective projection of the real world onto a 2D image. Both projections retain the perspective of the viewer's point of view.

In an isometric tilemap world, each individual tilemap image is already a projection of a seemingly three-dimensional object onto a flat surface. This projection is usually performed with a special form of parallel projection called *isometric projection*. The image then becomes more or less skewed, but our minds still recognize it as a three-dimensional object.

TIP: If you want to learn more about the various projection techniques and technical details of each, I recommend browsing the section about parallel projection on Wikipedia at http://en.wikipedia.org/wiki/Parallel_projection.

In terms of tilemaps, if you take a look at Figure 11–2, you can see the concrete steps for creating an isometric projection of an orthogonal image. The square is first rotated by 45 degrees and then scaled down along its y-axis to give it its typical isometric diamond shape.

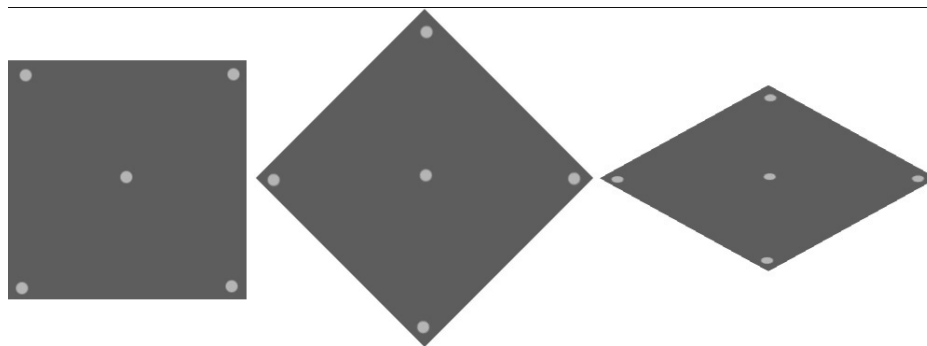


Figure 11–2. An orthogonal turned isometric by rotating it by 45 degrees and then compressing it vertically

However, Figure 11–2 is just the theoretical approach to illustrate the projection of the isometric shape. You can't turn an orthogonal image into an isometric image by simply rotating and compressing it, because the rotation would affect the image content. It would just look flat and very wrong, just like Figure 11–3.

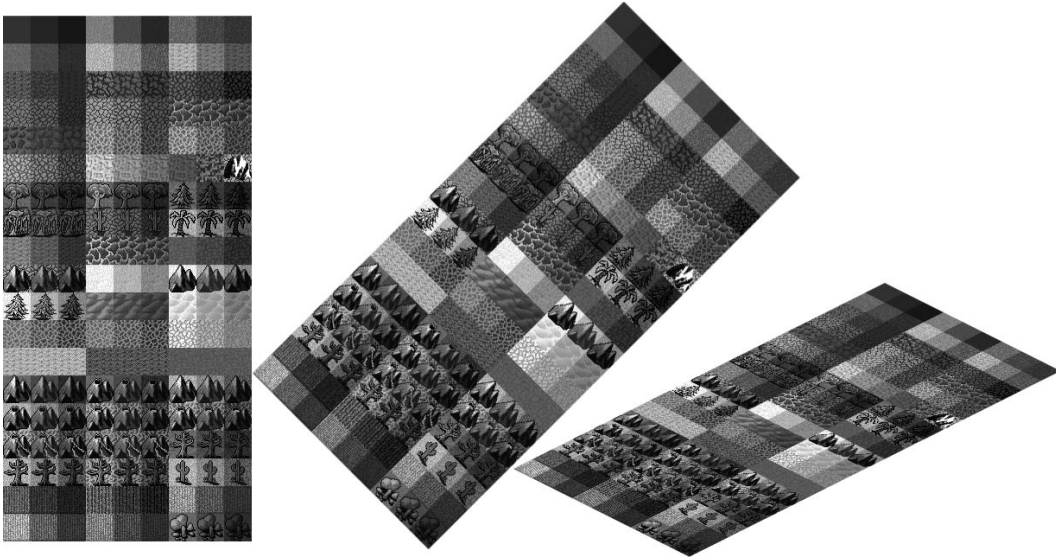


Figure 11–3. *Turning an orthogonal tileset into an isometric tileset it's not that simple!*

Instead, consider the diamond shape created in Figure 11–2 as your drawing canvas of the floor. The simplest isometric tiles you can design are flat ground tiles. Just fill the diamond shape with a certain pattern, and you get yourself usable isometric tiles. Figure 11–4 shows a number of flat-colored isometric tiles laid out next to each other, creating a ground floor pattern. Ground floor tiles are not impressive and look very flat. Yet they are essential as the game world's background layer.

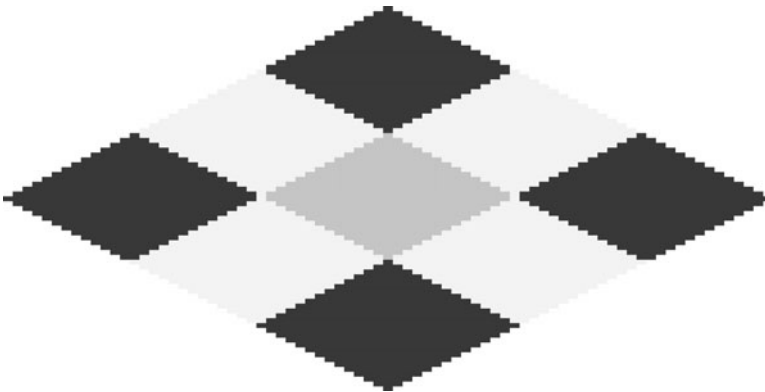


Figure 11–4. *Ground floor isometric tiles have no depth. They are used as solid surface areas.*

To add actual visual depth to an isometric tilemap, you need to have object tiles that extend beyond the diamond shape. The most commonly used approach is to draw three-dimensional objects as if they were viewed at a 45-degree angle and then draw them up and over the diamond shape, typically extending no more than one tile above. In the example in Figure 11–5, you can see this quite nicely by looking at the doorway. The door arch is drawn mostly over the isometric tile above the one that the door's frame is standing on. This gives the door its visual depth.

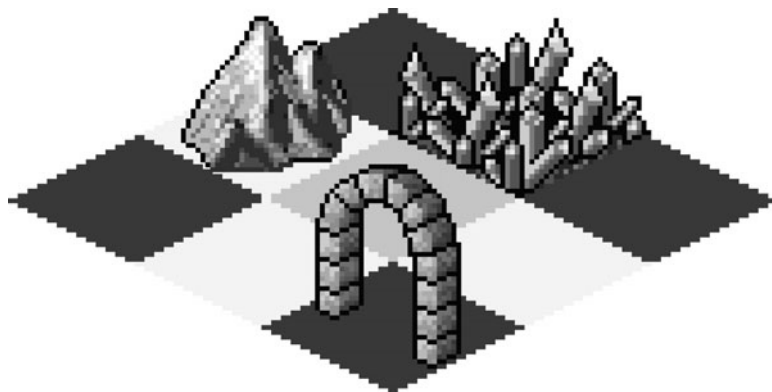


Figure 11–5. Add depth by drawing objects up to twice as high as the diamond shape.

Isometric tilemaps allow object tiles to overlap one another because the tiles are drawn from back to front, which means that object tiles closer to the viewer will always be drawn over tiles behind them, adding to the feeling of depth. But this approach requires careful design of individual tiles and the tilemap itself, because too much overlap or overlapping the wrong tiles can quickly destroy the illusion of depth.

As a good practice, try not to overlap object tiles that have wildly different shapes but use the same or similar color palette. In the case of Figure 11–5, for example, you would not want to place the crystal tile directly behind the doorway. The loss in contrast and merging outlines of these tiles could easily destroy the perception of depth.

Likewise, although you can create isometric object tiles that span much higher than twice the tile height, it's very hard to create a convincing 3D look if objects appear very high because the player will see only part of the tilemap. If you were building a huge castle whose walls span a dozen tiles high and the player approached them from below, the walls could easily be mistaken for a large section of ground floor. You can even end up creating optical illusions like the drawings of M. C. Escher because the isometric tiles do not get smaller the farther away from the screen they are. So, there's always a fine line between what works and what doesn't when designing isometric tiles and tilemaps.

Figure 11–6 shows a finely crafted isometric tilesheet named `dg_iso32.png`, which contains a good variation of ground floor tiles; object tiles such as walls, trees, and houses; and adornment objects or items that can be placed on any ground tile. The tiles in this set are each 54x49 pixels in size. The height can be chosen arbitrarily; it can be more or less than 49 pixels and depends on how much overlap between tiles you like in your tilemap.

The actual height of the diamond shape is 27 pixels. This will become important when you create the tilemap in Tiled (Qt) Map Editor.

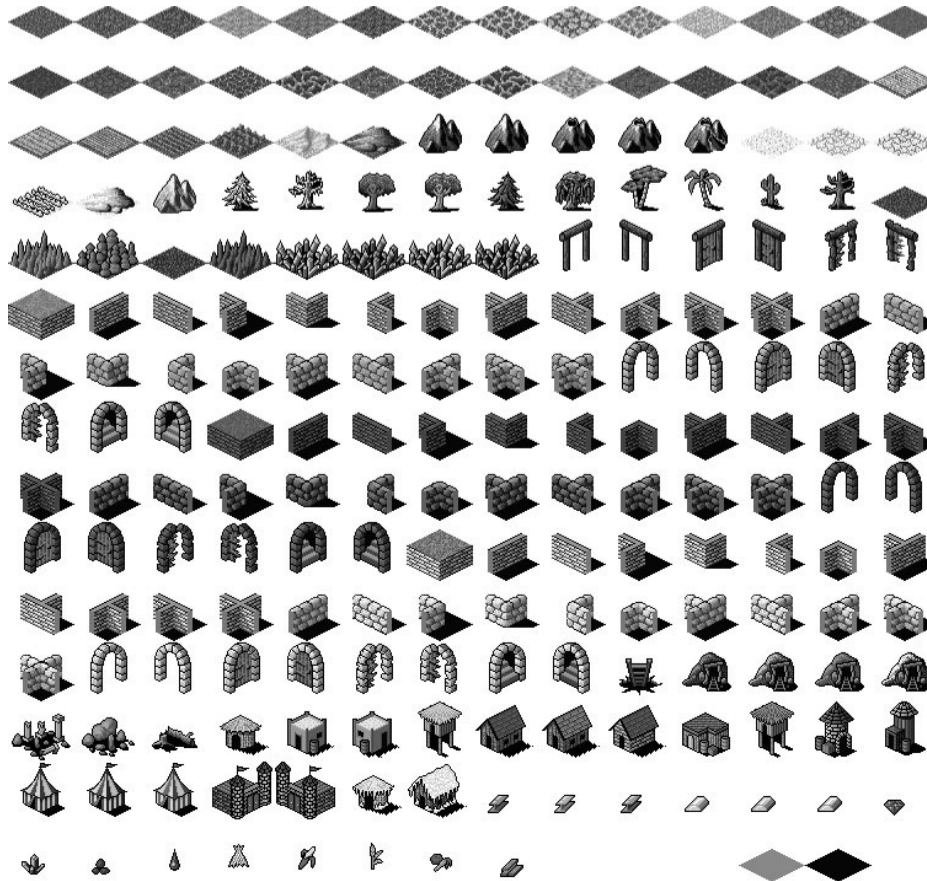


Figure 11–6. David Gervais's finely crafted isometric tileset

Isometric Tilemap Editing with Tiled

I'll use the Tiled Map Editor once again to create the isometric tilemap. The basic tilemap editing is the same as with orthogonal maps, but there are certain crucial steps to correctly set up a new isometric tilemap and load an isometric tileset.

Creating a New Isometric Tilemap

Open Tiled and choose **File** ► **New** to open the New Map dialog in Figure 11–7. The orientation should obviously be set to Isometric, and the map size is set to 30 tiles wide and high, just right for our example project. The odd thing here is the tile size width and height, which seem to be off a bit. I already mentioned that the individual tiles in `dg_iso32.png` are 54x49 pixels. The size of the diamond shape, which you have to

consider when laying down tiles, is 54x27 pixels. Yet the tile size in the New Map dialog is 52x26. This is because David Gervais designed his tiles to require overlapping of 2 pixels in the horizontal and 1 pixel in the vertical direction in order to close all gaps between the tiles.

This offset is on purpose, because isometric tiles are often designed to overlap each other a little. In this case, the size of the tiles in the Tiled isometric map must be 2 pixels less wide and 1 pixel less high than the actual size of the diamond shapes in the tileset. Other isometric tilesets may require different offsets, or even no offset at all.

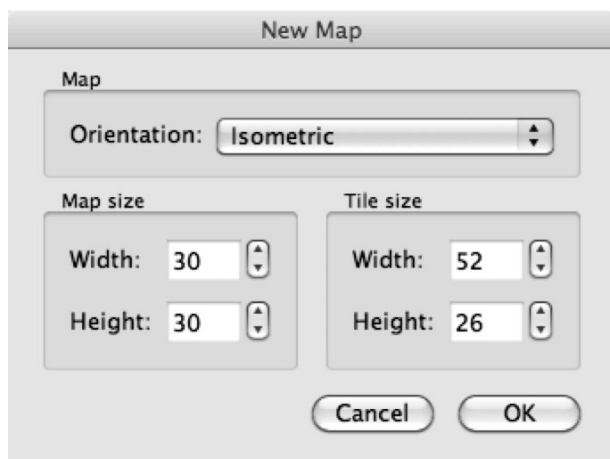


Figure 11–7. Create a new isometric tilemap in Tiled.

If you see any artifacts like the ones in Figure 11–8, you have set the wrong tile size when creating a new isometric map with David Gervais' tileset. You can find this erroneous tilemap as `isometric-no-offset.tmx` in the Tilemap05 project's resources folder, for illustration purposes.

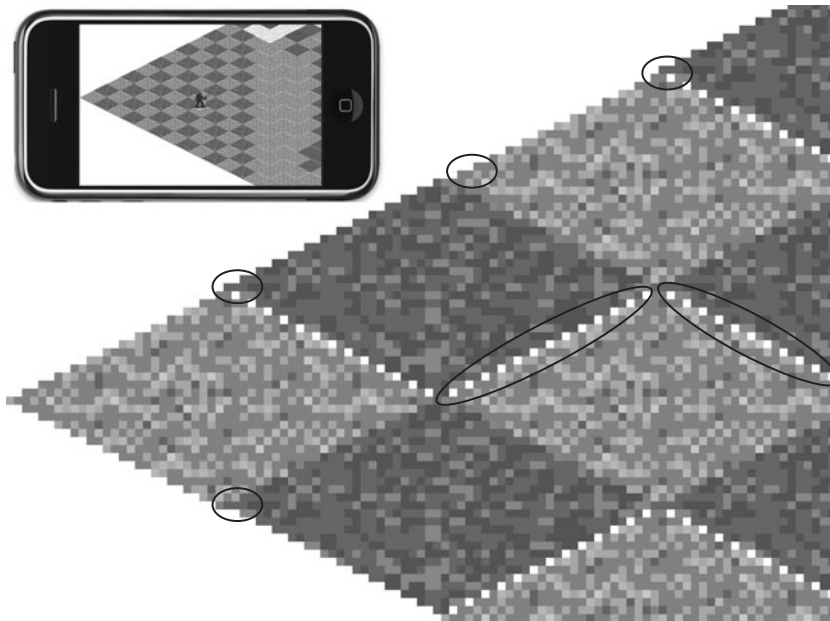


Figure 11-8. Artifacts like these indicate a tile-size offset problem.

If you did make a mistake and picked incorrect offsets and you don't want to lose the tilemap you've just spent hours designing or if you have other reasons to want to tweak the tilemap size or tileset size, there's a simple way to do this, but it requires manipulating the TMX file directly since there's no such option in Tiled itself.

The following trick makes it easy to experiment with various offsets until you get it just right. Close Tiled if it's currently running and then select the TMX file in your Xcode project; you'll see that it's displayed as a plain-text XML file. At the beginning of the file, you'll find the map section:

```
<map version="1.0" orientation="isometric" width="30" height="30" tilewidth="54"
    tileheight="27">
```

You can edit the `tilewidth` and `tileheight` parameters until you've found the correct offsets for the tilemap. Likewise, if you're having problems determining the tile size of the isometric tileset you're using, you can modify the `tilewidth` and `tileheight` parameters of the tileset(s):

```
<tileset firstgid="1" name="dg_iso32" tilewidth="54" tileheight="49">
  <image source="dg_iso32.png"/>
</tileset>
```

Just make sure to reload the TMX file in Tiled after you made any manual changes to it, because Tiled will not automatically update the file.

Creating a New Isometric Tileset

Next you need to load a tileset containing isometric tiles. For this chapter I will be using the `dg_iso32.png` tileset image found in the `Tilemap05` project's resources folder. In Tiled, choose **Map > New Tileset** and browse to the `dg_iso32.png` file.

Notice that Tiled will set default tile width and height according to the settings in the New Map dialog, shown in Figure 11-7. For isometric tilemaps, the defaults will always need to be corrected because of the overlap of isometric tiles. As I mentioned earlier, the `dg_iso32.png` tileset uses a tile width of 54 pixels and a tile height of 49 pixels. Note that you have to use the full canvas height of the isometric tiles, not the diamond shape height of 27 pixels. Figure 11-9 shows the correct setup for this tileset.

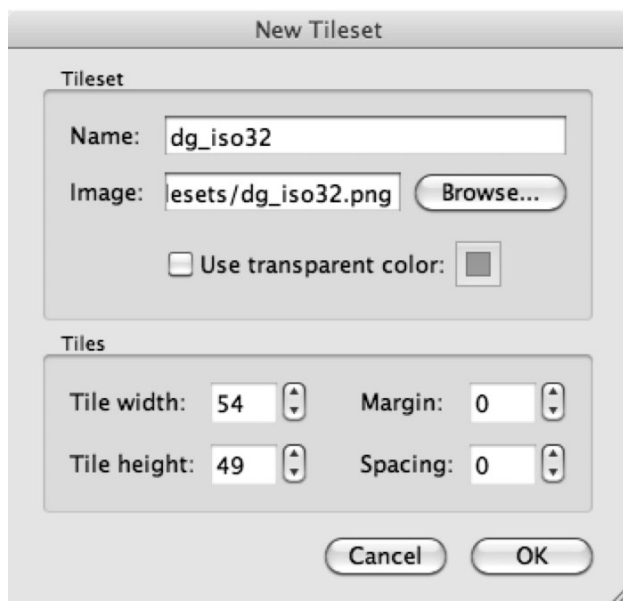


Figure 11-9. Adding the tileset with width 54 pixels and height 49 pixels

Laying Down Some Ground Rules

The most important rule for designing isometric maps is that you're going to need at least two layers so that game characters can walk behind certain tiles. One layer is for flat ground objects and floor tiles, and the other is for all other objects that either overlap other tiles or are not fully opaque, such as items. In the tileset of Figure 11-6, the first two rows are ground tiles and need to be placed on the ground floor, whereas in row 3 the mountains as well as almost all tiles in row 4 and after need to be placed on the Objects layer.

In Tiled, add two new layers via **Layer > Add Tile Layer** and name them Ground and Objects. Make sure that the Objects layer is drawn on top of the Ground layer. When

designing your tilemap, you should take great care to lay down only fully opaque, flat floor tiles on the Ground layer. All other tiles have to be placed on the Objects layer.

Cocos2d has issues with properly displaying game characters and other sprites behind partially occluding tiles in tilemaps, unless you apply the following steps. As one part of the solution, a special property named `cc_vertexz` needs to be added to tiled layers. I'll explain the solution in more detail shortly; for now, select the Ground layer and click **Layer ► Layer Properties**. Add a new property named `cc_vertexz` and set its value to `-1000`. Do the same with the Objects layer but instead of entering `-1000`, enter the string **automatic**, as in Figure 11–10.

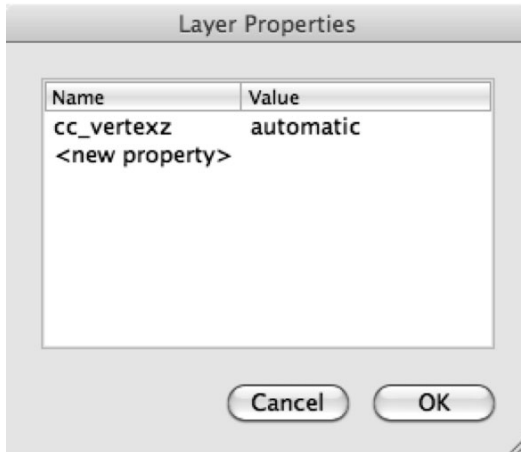


Figure 11–10. *The Objects layer needs the `cc_vertexz` property set to automatic.*

Now you can spend some time designing a nice-looking tilemap, or you can simply load the one I designed in the Tilemap05 project. Be sure to put only floor tiles on the Ground layer and only overlapping and transparent tiles on the Objects layer. When you're done, you should have a nice-looking tilemap like the one in Figure 11–11.

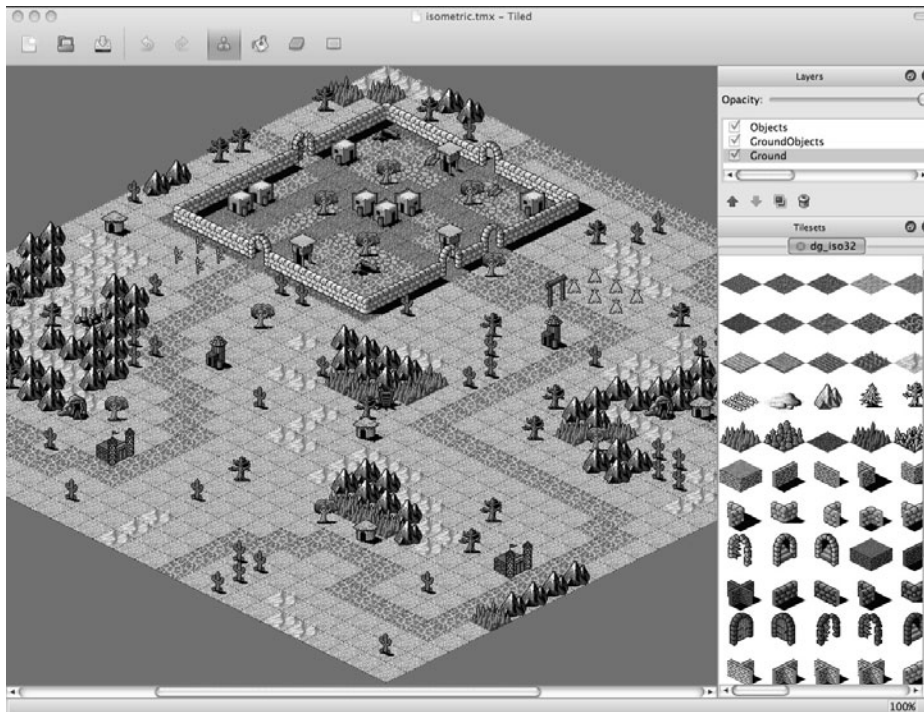


Figure 11–11. A completed isometric tilemap in Tiled using David Gervais's tileset

Isometric Game Programming

Let's use this isometric tilemap in a cocos2d game. As you might expect, some things will have to change compared to working with orthogonal tilemaps. In particular, you need to set up cocos2d properly to allow isometric tiles to partially occlude game characters. Determining a touched tile also requires different code than from orthogonal tilemaps, and when scrolling, you can no longer stop scrolling at the borders of the tilemap because the tilemap itself has a diamond shape.

Loading the Isometric Tilemap in Cocos2d

This is easy. Compared to orthogonal tilemaps, you don't need to change anything except to load the isometric.tmx file instead of orthogonal.tmx.

```
CCTMXLoadedMap* tileMap = [CCTMXLoadedMap tiledMapWithTMXFile:@"isometric.tmx"];
[self addChild:tileMap z:-1 tag:TileMapNode];
tileMap.position = CGPointMake(-500, -300);
```

I do set the isometric tilemap's position to -500, -300 right away, assuming the tilemap size is 30x30 tiles. This approximately centers the screen on the lower corner of the small village on the north of the tilemap in Figure 11–11. I did this to illustrate the following point

about properly setting up Cocos2D for isometric tilemaps, and in Figure 11–12 you can see there's something obviously wrong with the tilemap.

Setup Cocos2d for Isometric Tilemaps

If you followed the creation of the tilemap thus far and you set the `cc_vertexz` properties on the Ground and Objects layers in Tiled as described earlier, the resulting tilemap may look like the one in Figure 11–12. Somehow, the Ground layer is zoomed far out, and tiles from the Objects layer seem to be floating in midair. It looks like a scary place to be.



Figure 11–12. *Without 2D projection, the ground layer will render incorrectly.*

The way to fix this and to enable proper rendering of overlapping sprites requires cocos2d to be initialized in a different way from how the cocos2d Application template in Xcode sets things up. It initializes cocos2d in a standard way, which is fine for most games but fails to work properly with isometric tilemap games. In the Tilemap05 project, the cocos2d startup code is replaced by the code in Listing 11–1 with the important changes highlighted.

Listing 11–1. Manually Initializing cocos2d's EAGLView

```
window = [[UIWindow alloc] initWithFrame:[UIScreen mainScreen] bounds]];
if ([CCDirector setDirectorType:kCCDirectorTypeDisplayLink] == NO)
    [CCDirector setDirectorType:kCCDirectorTypeNSTimer];

CCDirector *director = [CCDirector sharedDirector];
[director setAnimationInterval:1.0/60];
EAGLView *glView = [EAGLView viewWithFrame:[window bounds]
                                pixelFormat:kEAGLColorFormatRGB565
                                depthFormat:GL_DEPTH_COMPONENT24_OES];
[director setOpenGLView:glView];
[window addSubview:glView];
[window makeKeyAndVisible];

// this fixes the zoomed out ground layer:
[director setProjection:kCCDirectorProjection2D];
```

You have to change two things: first you need to enable the OpenGL depth buffer to allow a more fine-grained control over the z-ordering of objects. Second, the CCDirector has to use a 2D projection to work with the depth buffer.

NOTE: The actual initialization code in cocos2d projects created from a project template may differ from Listing 11.1. But if you look at the initialization code line by line, you'll see that essentially the same code is run; however, it may be in a different order, or there may be other additional code added in between. The initialization code of cocos2d changes frequently, practically with every new major release, so it's not a problem if your initialization code looks different. For example, I did not include the `RootViewController` class initialization code in all previous projects, because we haven't had a use for it. But I'll get to that in detail in Chapter 15, where we'll mix cocos2d with Cocoa Touch.

You first create a `UIWindow` and then decide on a `CCDirector` type to use and set the animation interval to 60 frames per second. This is the default behavior.

The `EAGLView` line is important because for overlapping tiles to render properly, a depth buffer must be specified with the `depthFormat` parameter. In this case, it's `GL_DEPTH_COMPONENT24_OES`, which creates a depth buffer of 24 bits. To conserve memory, you can also use a 16-bit depth buffer, which may also be sufficient.

Depth buffering allows OpenGL to determine whether a certain pixel is in front or behind another pixel so it can decide whether to actually draw the new pixel or discard it. This comes at a cost of additional memory usage—around 500KB for a 24-bit depth buffer—but it allows sprites and tiles to correctly overlap one another.

After the `glView` has been created, it's assigned to the `CCDirector`, added as a subview to the window, and then the window is made visible.

The other really important line in the initialization is `setProjection`, which puts cocos2d in 2D projection mode. This changes a couple of OpenGL parameters that affect the way cocos2d renders nodes. In this case, it fixes the issue in Figure 11-12 where the ground floor is not rendered as expected, with the result in Figure 11-13. But it also enables you to finely tune the z-order of sprites by using the `vertexZ` property rather than the `zOrder` property of sprites.

By default, cocos2d's z-ordering of nodes is based on the z value when you're adding a node via the `addChild` method. Nodes with a lower z-order are drawn before nodes with a higher z-order. Nodes that have the same z-order are drawn in the order that they've been added to the node hierarchy, meaning nodes added last will be drawn over nodes added before that node. This allows cocos2d to order nodes without using a depth buffer.

If you enable depth buffering, you can also use the `vertexZ` property to change each node's draw order freely. Cocos2d requires this freedom in order to let the `cc_vertexz_tilemap` property work its magic. We'll later manipulate the `vertexZ` property of the player character to correctly draw the player sprite in front of or behind other isometric tiles.



Figure 11–13. *With 2D projection, the ground layer is displayed as expected.*

Locating an Isometric Tile

The next thing to do is to determine from a touch location the coordinates of the touched tile. This is done in the `Tilemap06` project.

If you refer to Figure 10-11 in the previous chapter, you'll recall that the tilemap indices of orthogonal tilemaps have their origin point (0, 0) at the top-left corner. Now, with isometric tilemaps, there is no top-left corner anymore. The tilemap itself is rotated by 45 degrees, which makes the topmost tile the point of origin. Figure 11-14 illustrates this well. Tiles toward the bottom right have increasing X coordinates, while Tiles toward the bottom left have increasing Y coordinates. The bottommost tile then has the coordinates 29, 29 in a map consisting of 30x30 tiles.

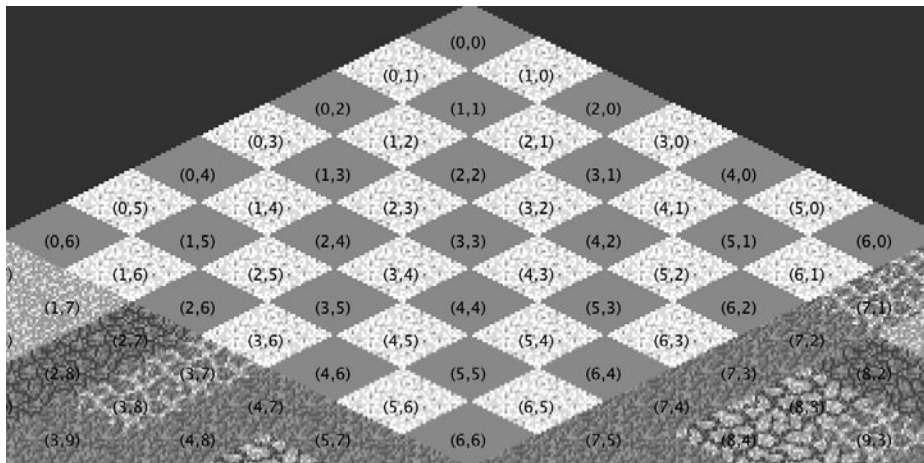


Figure 11–14. *The coordinate system of an isometric tilemap*

This may seem strange at first, but if you lean your head a bit to the right, you may notice that the tile coordinates follow the exact same order as in orthogonal tilemaps, except that the whole map is rotated by 45 degrees.

You can now straighten your head again, because I need you to focus on the modified `tilePosFromLocation` method, which calculates the touched tile coordinates from a touch location on the screen. As Listing 11–2 shows, it's a wee bit more complex than the orthogonal counterpart.

Listing 11–2. Calculating the Tile Coordinates from a Touch Location

```
-(CGPoint) tilePosFromLocation:(CGPoint)location tileMap:(CCTMXTileMap*)tileMap
{
    // Tilemap position must be subtracted, in case the tilemap position is scrolling
    CGPoint pos = ccpSub(location, tileMap.position);

    float halfMapWidth = tileMap.mapSize.width * 0.5f;
    float mapHeight = tileMap.mapSize.height;
    float tileWidth = tileMap.tileSize.width;
    float tileHeight = tileMap.tileSize.height;

    CGPoint tilePosDiv = CGPointMake(pos.x / tileWidth, pos.y / tileHeight);
    float inverseTileY = mapHeight - tilePosDiv.y;

    // Cast to int makes sure that result is in whole numbers
    float posX = (int)(inverseTileY + tilePosDiv.x - halfMapWidth);
    float posY = (int)(inverseTileY - tilePosDiv.x + halfMapWidth);

    // make sure coordinates are within isomap bounds
    posX = MAX(0, posX);
    posX = MIN(tileMap.mapSize.width - 1, posX);
    posY = MAX(0, posY);
    posY = MIN(tileMap.mapSize.height - 1, posY);

    return CGPointMake(posX, posY);
}
```

Subtracting the tilemap position to take scrolling of the tilemap into account is the same as in the orthogonal version of this method. Next I create a number of variables just to make the code a bit more readable and have less to type, and then I divide the map size width by half. I then create a `CGPoint` `tilePosDiv`, which is the pixel location within the tilemap divided by the tilemap's width and height, and an `inverseTileY` variable, which is simply the inverse of the tilemap's Y coordinates. This inversion is necessary because the tilemap Y coordinates count from top down, whereas screen Y coordinates count from bottom up.

Now I get to actually calculating the X, Y coordinates of the touched tile. The calculation starts with the inverse Y coordinate, which will be in the range of 0 to 29 for a tilemap that has a height of 30 tiles. It defines the vertical position in the tilemap from which we'll be looking for the x and y tile coordinates horizontally.

This becomes clearer if you look at Figure 11–4 and locate tile coordinate (3,3). You'll notice that when you move on a horizontal line to the left of tile coordinate (3,3), the x coordinates decrease while the y coordinates increase: (2,4), (1,5), (0,6). Similarly, if you move to the right of tile coordinate (3,3), the x coordinates increase while the y coordinates decrease: (4,2), (5,1), (6,0).

That means you can get both x and y tile coordinates from the `inverseTileY` position. In the case of the x tile coordinate, you add the `tilePosDiv.x` coordinate and then subtract `halfMapWidth`. For the y tile coordinate, you subtract the sum of `tilePosDiv.x` and `halfMapWidth` from `inverseTileY`.

NOTE: I'll spare you the details of the mathematical concepts behind this calculation, since you can apply the code as is and don't need to change anything. If you are interested in understanding the intricate details of isometric projection and the mathematics behind it, I recommend reading the excellently illustrated article by Herbert Glarner at www.gandraxa.com/isometric_projection.aspx.

By applying the Objective-C MIN and MAX macros, I ensure that the returned tile coordinate is within the bounds of the tilemap. In other words, it will return coordinates from (0, 0) to (29, 29) for the 30x30 tilemap used by the isometric tilemap projects.

Scrolling the Isometric Tilemap

With the `tilePosFromLocation` method updated to work with isometric tilemaps, the `Tilemap06` project continues by implementing isometric tilemap scrolling, using the tile coordinates returned from the `tilePosFromLocation` method. Just as in the orthogonal tilemap project, this is done using the `centerTileMapOnTileCoord` method, shown in Listing 11–3.

Listing 11–3. *Scrolling the Screen to Center on a Specific Tile Coordinate*

```
-(void) centerTileMapOnTileCoord:(CGPoint)tilePos tileMap:(CCTMXTileMap*)tileMap
{
    // center tilemap on the given tile pos
    CGSize screenSize = [[CCDirector sharedDirector] winSize];
    CGPoint screenCenter = CGPointMake(screenSize.width * 0.5f,
                                       screenSize.height * 0.5f);

    // get the ground layer
    CCTMXLayer* layer = [tileMap layerNamed:@"Ground"];
    NSAssert(layer != nil, @"Ground layer not found!");

    // internally tile Y coordinates are off by 1
    tilePos.y -= 1;

    // get the pixel coordinates for a tile at these coordinates
    CGPoint scrollPosition = [layer positionAt:tilePos];

    // negate the position to account for scrolling
    scrollPosition = ccpMult(scrollPosition, -1);

    // add offset to screen center
    scrollPosition = ccpAdd(scrollPosition, screenCenter);

    // move the tilemap
    CCAction* move = [CCMoveTo actionWithDuration:0.2f position:scrollPosition];
```

```
[tileMap stopAllActions];  
[tileMap runAction:move];  
}
```

First, the screen center position is determined as before. Then I want to use the convenience method of the layer, `positionAt`, which returns a screen position for a tile coordinate. To do so, I get the Ground layer and assert that it exists. It doesn't matter which layer you use, as long as all layers use the same size tiles.

Before calling the `positionAt` method, I have to subtract 1 from the tile Y coordinate to fix a persistent offset problem. Seasoned programmers may be worried that using a tile Y coordinate of 0 and subtracting 1 from it could lead to an invalid index and thus a disastrous crash. But in this case, the `positionAt` method doesn't use the tile coordinates as indices, and it works with any tile coordinate, even negative ones.

The `positionAt` method returns the pixel position of the given tile coordinate within the tilemap and stores it in the `scrollPosition` variable. This method isn't specific to isometric tilemaps; it works for all tilemap types: orthogonal, isometric, and hexagonal. Internally, `cocos2d` checks which type of tilemap is currently being used and then uses the appropriate calculation, since they differ in profound ways. If you are interested in the specific implementation of each of these calculations, take a look at the methods `positionForOrthoAt`, `positionForIsoAt`, and `positionForHexAt` in the `CCTMXLayer.m` implementation file.

Because the tilemap may be scrolling, in which case it will have a negative position, the `scrollPosition` is multiplied by -1, negating it. After that I just add the `screenCenter` position to it, and I know where to scroll to. The move action is the same as before and moves the tilemap so that the touched tile is centered on-screen.

This World Deserves a Better End

Because of the diamond-shaped nature of isometric tilemaps, it's inevitable that the scrolling tilemap will reveal parts outside of the tilemap, as shown in Figure 11-15. Indeed, the `tilePosFromLocation` method ensures that the returned tile coordinate is always within bounds, so you can use that safely even if the player touches outside the tilemap. But if you don't want the player to see the end of your isometric tilemap world, you'll have to use a trick.

Open Tiled and load the `isometric.tmx` file from the `Tilemap06` project's resource folder. What you want to do is to add a border around the existing map and fill it with tiles that give the impression of an impassable area. In Tiled, use **Map ► Resize Map** to bring up the Resize dialog shown in Figure 11-16. You need to add 10 tiles to each side of this tilemap to completely fill the border. Depending on the tile size, you will have to experiment to find the minimum number of tiles that need to be appended. In this case, enter **50** as the new width and height and also enter **10** in the offset boxes. This makes the tilemap 20x20 tiles larger and also moves everything you've edited previously to the center so that you end up with a border of 10 tiles on each side.



Figure 11–15. *It's the end of the world as we know it (and it feels wrong).*

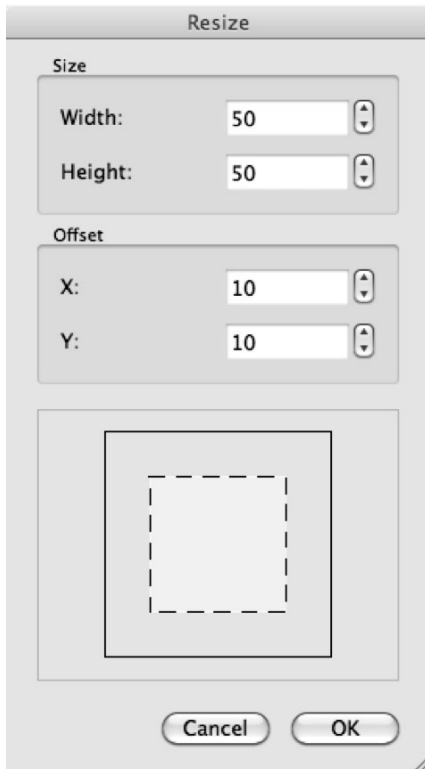


Figure 11–16. *Resizing the map in Tiled to add a border*

You can now fill this border area to give the impression of an area of the map that is simply impassable. It helps to choose a darker ground tile to hint to the player that this area can't be entered, and of course you should add impenetrable objects onto the Objects layer and around the border of the playable area. Your result should look something like Figure 11–17. I saved my version into the resources folder of the Tilemap07 project and named it `isometric-with-border.tmx`.

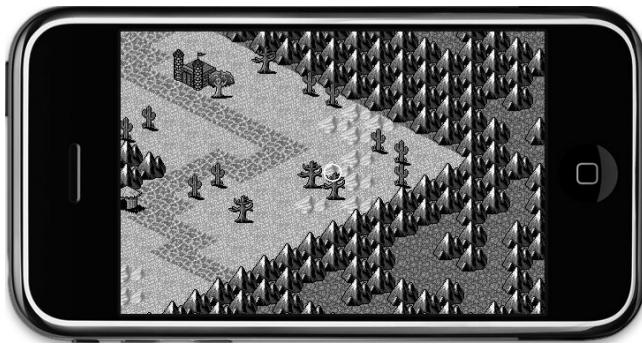


Figure 11–17. *A convincing impassable map border area*

NOTE: The impassable area in Figure 11.17 does look quite repetitive and boring. You may be tempted to add more detail to that area, but that is a double-edged sword. On one side, more details and variations within the impassable area will make it look better. On the other, it can also fool the player into thinking about and, even worse, spending time trying to reach that one spot in the impassable area that looks like it could be visited. The player might assume it's a secret area and he simply has to figure out how to get there. If you have a player thinking like this, it's bad for your game. You don't want to tempt the player into trying things that are absolutely impossible to achieve. It just wastes his time, and it ends in frustration.

The Tilemap07 project also implements the code that prevents you from scrolling outside the playable area by defining the inner tile coordinates of the playable area. I added two `CGPoint` variables, `playableAreaMin` and `playableAreaMax`, to the `TileMapLayer` class:

```
@interface TileMapLayer : CCLayer
{
    CGPoint playableAreaMin, playableAreaMax;
}
```

The playable area variables are initialized with a border size of 10 tiles in the `init` method of the class:

```
const int borderSize = 10;
playableAreaMin = CGPointMake(borderSize, borderSize);
playableAreaMax = CGPointMake(tileMap.mapSize.width - 1 - borderSize,
    tileMap.mapSize.height - 1 - borderSize);
```

The playable area is defined as anything within the bounds of the tile coordinates (10, 10) to (39, 39). All tiles outside this area should be considered not part of the playfield. All that remains is to update the `tilePosFromLocation` method by replacing the MIN/MAX lines to implement this rule of the playable area. Instead of keeping the tile coordinates within the bounds of the whole tilemap, you now want to keep it within the bounds of the playable area, as such:

```
posX = MAX(playableAreaMin.x, posX);
posX = MIN(playableAreaMax.x, posX);
posY = MAX(playableAreaMin.y, posY);
posY = MIN(playableAreaMax.y, posY);
```

If you try this, you'll see that only the tiles within the playable area can be centered on-screen. What's more, clicks outside the playable area are not just ignored; the tilemap scrolls as close as possible to the tile you clicked. This way, you don't destroy the player's impression of a world that seemingly extends far beyond what the player can see.

Adding a Movable Player Character

By adding a player character moving about the tilemap world, you get closer to an actual isometric game. In this case, I chose the `ninja.png` as the player character and added it to the `Tilemap08` project. The player is a class derived from `CCSprite`, aptly named `Player`. Listing 11–4 shows the header file.

Listing 11–4. The Player Class Interface

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

```
@interface Player : CCSprite
{
}
```

```
+(id) player;
```

```
@end
```

The `+(id) player` method in Listing 11–5 is the static autorelease initialize, which also initializes the sprite with the `ninja.png` file.

Listing 11–5. The Player Class Implementation

```
#import "Player.h"
```

```
@implementation Player
```

```
+(id) player
{
    return [[[self alloc] initWithFile:@"ninja.png"] autorelease];
}
```

```
@end
```

You then create the player in the `TileMapLayer` class's `init` method:

```
CGSize screenSize = [[CCDirector sharedDirector] winSize];
```

```
// Create the player and add it
player = [Player player];
player.position = CGPointMake(screenSize.width / 2, screenSize.height / 2);
```



```
// approximately position player's texture to best match the tile center position
player.anchorPoint = CGPointMake(0.3f, 0.1f);
[self addChild:player];
```

The player's position is set to the center of the screen on purpose. Since you already have a method that allows you to center a specific tile on the screen, centering the player sprite on the screen as well makes it behave as if it were moving across the tilemap, when in fact it always remains at the same position. You don't have to move the player sprite at all!

The player's anchorPoint is offset a little from its default of (0.5f, 0.5f) to (0.3f, 0.1f) to approximately center the sprite's feet on the center position of the tile. Otherwise, it might look wrong because all other game objects like trees and cacti have their roots, literally speaking, at the center of the tile. So, it's only natural to try to place the player's feet at that position as well.

If you try this now, even though the player sprite never moves, it looks as if the player is walking about the tilemap world. Perfect!

Well, not quite. If you move over mountains and walls and trees and buildings, the player sprite is always drawn in front of them.

Enabling the Player to Move Behind Tiles

To allow the player to be partially hidden by object tiles in front of him, such as buildings, walls, trees, and whatnot, you have to change his vertexZ value as he moves around on the map. At the start of this chapter, when you created the Objects layer in Tiled, you gave it a property named `cc_vertexz` and set it to automatic. This instructed cocos2d to assign consecutive vertexZ values to the tiles in that layer. Figure 11–18 shows you which vertexZ values the tiles are assigned in a tilemap that's 50x50 tiles in size. This is different from the tile indices shown in Figure 11–14 because the vertexZ values increase in both X and Y directions. You could say that vertexZ values decrease with each horizontal row of the tilemap.

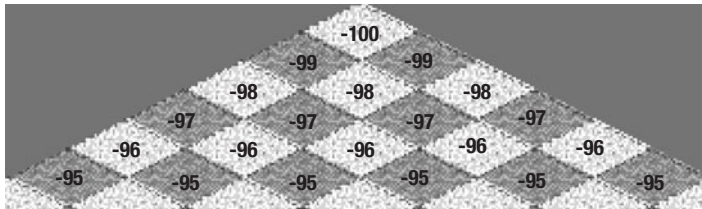


Figure 11–18. The vertexZ values of tiles in the 50x50 tilemap

This is reflected in code by the `updateVertexZ` method added to the Player class:

```
-(void) updateVertexZ:(CGPoint)tilePos tileMap:(CCTMXTileMap*)tileMap
{
    float lowestZ = -(tileMap.mapSize.width + tileMap.mapSize.height);
    float currentZ = tilePos.x + tilePos.y;
    self.vertexZ = lowestZ + currentZ - 1;
}
```

The lowest vertexZ value is simply the sum of the map size width and height in the negative. Likewise, you can get the difference of any tile coordinate in the tilemap to the lowest vertexZ value, which is the tile at position 0, 0. It's the sum of the X and Z coordinates of that position. For example, the tile at position 2, 2 is $2 + 2 = 4$ less than the lowest vertexZ value. If you add the two, you get $-100 + 4 = -96$. Since the player sprite is added to the TileMapLayer after the tilemap, it will render on top of tiles with the same vertexZ value. Because of this, I also subtract 1 so that the end result is a vertexZ value of -97 if the player is standing on the tile coordinate 2, 2.

To make this code work, you also have to define the updateVertexZ method in the Player class's interface:

```
@interface Player : CCSprite
{
}

+(id) player;
-(void) updateVertexZ:(CGPoint)tilePos tileMap:(CCTMXTiledMap*)tileMap;
@end
```

And then you should call the updateVertexZ method every time the tilemap is moved, which is done in the ccTouchesBegan method of the TileMapLayer class:

```
-(void) ccTouchesBegan:(NSSet *)touches withEvent:(UIEvent *)event
{
    CCNode* node = [self getChildByTag:TileMapNode];
    NSAssert([node isKindOfClass:[CCTMXTiledMap class]], @"not a CCTMXTiledMap");
    CCTMXTiledMap* tileMap = (CCTMXTiledMap*)node;

    CGPoint touchLocation = [self locationFromTouches:touches];
    CGPoint tilePos = [self tilePosFromLocation:touchLocation tileMap:tileMap];

    [self centerTileMapOnTileCoord:tilePos tileMap:tileMap];

    // fix the player's Z position
    [player updateVertexZ:tilePos tileMap:tileMap];
}
```

If you try this now, you'll see that the ninja player will hide behind walls, trees, and other large objects, like a good ninja does.

Moving the Player, Tile by Tile

So far, the player (actually, the screen) moves faster the further away from the center the screen is touched. The player also moves across the tiles freely, but he really should be moving from tile to tile in only four directions. The Tilemap09 project changes the control mechanism to one that allows you to move the player in one of the four allowed directions as long as you keep a finger on the screen. The move direction depends on where you touch the screen relative to the player.

This requires some additions to the TileMapLayer interface, shown in Listing 11-6.

Listing 11–6. *The TileMapLayer Class Interface*

```
typedef enum
{
    MoveDirectionNone = 0,
    MoveDirectionUpperLeft,
    MoveDirectionLowerLeft,
    MoveDirectionUpperRight,
    MoveDirectionLowerRight,

    MAX_MoveDirections,
} EMoveDirection;

@interface TileMapLayer : CCLayer
{
    CGPoint playableAreaMin, playableAreaMax;

    Player* player;

    CGPoint screenCenter;
    CGRect upperLeft, lowerLeft, upperRight, lowerRight;
    CGPoint moveOffsets[MAX_MoveDirections];
    EMoveDirection currentMoveDirection;
}
```

The EMoveDirection enum is later used to determine in which direction the player intends to walk, with MoveDirectionNone signaling no movement. Let's look at changes in the implementation of the TileMapLayer class's init method in Listing 11–7.

Listing 11–7. *Initializing the Player's Movement Directions*

```
// divide the screen into 4 areas
screenCenter = CGPointMake(screenSize.width / 2, screenSize.height / 2);
upperLeft = CGRectMake(0, screenCenter.y, screenCenter.x, screenCenter.y);
lowerLeft = CGRectMake(0, 0, screenCenter.x, screenCenter.y);
upperRight = CGRectMake(screenCenter.x, screenCenter.y, screenCenter.x,
    screenCenter.y);
lowerRight = CGRectMake(screenCenter.x, 0, screenCenter.x, screenCenter.y);

moveOffsets[MoveDirectionNone] = CGPointZero;
moveOffsets[MoveDirectionUpperLeft] = CGPointMake(-1, 0);
moveOffsets[MoveDirectionLowerLeft] = CGPointMake(0, 1);
moveOffsets[MoveDirectionUpperRight] = CGPointMake(0, -1);
moveOffsets[MoveDirectionLowerRight] = CGPointMake(1, 0);

currentMoveDirection = MoveDirectionNone;

// continuously check for walking
[self scheduleUpdate];
```

The four CGRect variables, upperLeft, lowerLeft, upperRight, and lowerRight, divide the screen into four quadrants, each of which is the touch area to move the player in the desired direction. Thus, a touch in the lower-right area of the screen will move the player to the right and down along the tilemap.

The `moveOffsets` array contains a `CGPoint` for each move direction that, when added to the current tile coordinate, will return the next tile coordinate in that direction. The `currentMoveDirection` variable simply holds which direction the player is moving toward, and you need a `scheduleUpdate` to continuously check whether the player still wants to move.

The `ccTouchesBegan` method (Listing 11–8) has changed to simply check which quadrant of the screen received the touch and then sets the `currentMoveDirection`. The newly added `ccTouchesEnded` method sets the `currentMoveDirection` back to `MoveDirectionNone`.

Listing 11–8. Moving the Player Based on Touch Location

```
-(void) ccTouchesBegan:(NSSet *)touches withEvent:(UIEvent *)event
{
    // get the position in tile coordinates from the touch location
    CGPoint touchLocation = [self locationFromTouches:touches];

    // check where the touch was and set the move direction accordingly
    if (CGRectContainsPoint(upperLeft, touchLocation))
    {
        currentMoveDirection = MoveDirectionUpperLeft;
    }
    else if (CGRectContainsPoint(lowerLeft, touchLocation))
    {
        currentMoveDirection = MoveDirectionLowerLeft;
    }
    else if (CGRectContainsPoint(upperRight, touchLocation))
    {
        currentMoveDirection = MoveDirectionUpperRight;
    }
    else if (CGRectContainsPoint(lowerRight, touchLocation))
    {
        currentMoveDirection = MoveDirectionLowerRight;
    }
}

-(void) ccTouchesEnded:(NSSet *)touches withEvent:(UIEvent *)event
{
    currentMoveDirection = MoveDirectionNone;
}
```

The gist of the work has now been moved to the update method, which is scheduled to be called every frame:

```
-(void) update:(ccTime)delta
{
    CCNode* node = [self getChildByTag:TileMapNode];
    NSAssert([node isKindOfClass:[CCTMXTiledMap class]], @"not a CCTMXTiledMap");
    CCTMXTiledMap* tileMap = (CCTMXTiledMap*)node;

    // if the tilemap is currently being moved, wait until it's done moving
    if ([tileMap numberOfRunningActions] == 0)
    {
        if (currentMoveDirection != MoveDirectionNone)
        {
            CGPoint tilePos = [self tilePosFromLocation:screenCenter tileMap:tileMap];
```

```

        CGPoint offset = moveOffsets[currentMoveDirection];
        tilePos = CGPointMake(tilePos.x + offset.x, tilePos.y + offset.y);
        tilePos = [self ensureTilePosIsWithinBounds:tilePos];

        [self centerTileMapOnTileCoord:tilePos tileMap:tileMap];
    }
}

// continuously fix the player's Z position
CGPoint tilePos = [self floatingTilePosFromLocation:screenCenter tileMap:tileMap];
[player updateVertexZ:tilePos tileMap:tileMap];
}

```

The tilemap has a running action only when it is moving, so I give it a new move action only if it has no move action currently running and the `currentMoveDirection` isn't `MoveDirectionNone`. The `tilePosFromLocation` is no longer retrieved from the screen touch location, but instead the `screenCenter` position is used. Since the player is always centered on the screen, this is a convenient shortcut to the tile coordinate at the center of the screen.

The `moveOffsets` array returns a `CGPoint` that is added to `tilePos` to get the new tile coordinate we intend to move to. Because this can be outside the playable area, the new tile coordinate is run through the `ensureTilePosIsWithinBounds` method. It's the same code we used before to keep the tile coordinate within the playable area but refactored into a separate method to avoid duplicating this code. Lastly, the `centerTileMapOnTileCoord` method is called to move and center the screen on the desired tile coordinate, which also adds the move action.

With the player now moving across the tilemap tile by tile, we can keep updating the player's `vertexZ` value. Previously, the `vertexZ` value was set to the target tile coordinate immediately, which caused the player to be drawn below all object tiles he was moving over. By continuously updating the `vertexZ` value as the player moves across the tilemap, his `z` position is now more accurate and removes any overlap glitches from the previous `Tilemap08` project.

NOTE: When you're moving the player through an arc, you'll notice that as he moves through the arc he'll suddenly appear in front of the arc or disappear behind it, depending on the direction he's moving. This is an unavoidable side effect of 2D isometric tilemaps. You can reduce this effect only by drawing your archways higher than any character in your game. You could also split the arc into three tiles so that only the middle section is passable, whereas the two sides of the archway are regarded as blocking tiles.

Stop Player on Collisions

Lastly, we don't want the player to walk over walls and mountains. He may be a ninja, but he's not *that* good. To solve that problem, add a new layer in Tiled via **Layer ► Add Tile Layer** and name it `Collisions`; then move the `Opacity` slider just above the `Layers` list to

about the middle. Now pick a tile from the tileset whose color is a strong contrast to the tilemap, because we'll use it to draw collision areas over the tilemap, and they should be easily recognizable despite having a low opacity.

I chose one of the purple tiles. Right-click the tile of your choice, and select **Tile Properties** from the context menu. Note that this command has no equivalent in the Tiled menu; tile properties can be accessed only by right-clicking a tile. In the Tile Properties dialog shown in Figure 11–19, add a property named `blocks_movement` and set the value to 1. Actually, I'm going to ignore the value in code; it's only important that the `blocks_movement` value exists.

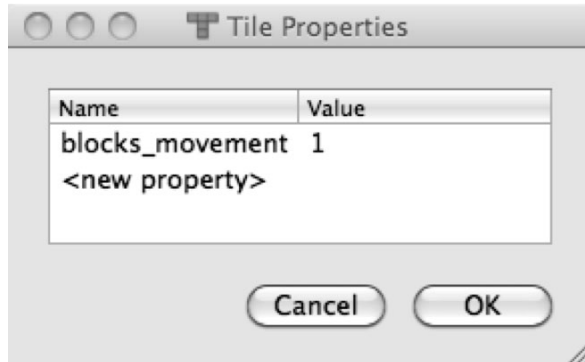


Figure 11–19. Add a `blocks_movement` tile property

With the Collisions layer selected, draw on the tilemap with the tile that has the `blocks_movement` property set. Place a tile everywhere you don't want the player to move onto, for example, walls, mountains, houses, and so on.

The tilemap `isometric-with-border.tmx` in the `Tilemap10` project is already prepared with a collision layer. The collision layer is only for checking whether a tile can be moved on and should not be displayed in the game, so the first thing you do in the `init` method of the `TileMapLayer` class is to set this layer invisible (see Listing 11–9).

Listing 11–9. *Hiding the Collisions Layer*

```
CCTMXTileMap* tileMap = [CCTMXTileMap tiledMapWithTMXFile:
    @"isometric-with-border.tmx"];
[self addChild:tileMap z:-1 tag:TileMapNode];

CCTMXLayer* layer = [tileMap layerNamed:@"Collisions"];
layer.visible = NO;
```

To check whether a certain tile coordinate is blocked, I've added the `isTilePosBlocked` method to the `Tilemap10` project, as shown in Listing 11–10.

Listing 11–10. *Determine Whether a Tile Is Blocked*

```
-(bool) isTilePosBlocked:(CGPoint)tilePos tileMap:(CCTMXTileMap*)tileMap
{
    CCTMXLayer* layer = [tileMap layerNamed:@"Collisions"];
    NSAssert(layer != nil, @"Collisions layer not found!");
```

```

    bool isBlocked = NO;
    unsigned int tileGID = [layer tileGIDAt:tilePos];
    if (tileGID > 0)
    {
        NSDictionary* tileProperties = [tileMap propertiesForGID:tileGID];
        id blocks_movement = [tileProperties objectForKey:@"blocks_movement"];
        isBlocked = (blocks_movement != nil);
    }

    return isBlocked;
}

```

The code first tries to get a tile at the given tile coordinate from the Collisions layer. If there is no tile there, the tileGID will be 0, and we can safely assume that this tile is not blocked. But if there is a valid tileGID at the tilePos coordinate, the tileMap is queried for the properties of the tile, which returns an NSDictionary object. If the dictionary's objectForKey method returns a valid object for the key named blocks_movement, the tile is blocked.

The place to check for collision is in the update method, as shown in Listing 11–11.

Listing 11–11. Checking for Collision in the update Method

```

-(void) update:(ccTime)delta
{
    // if the tilemap is currently being moved, wait until it's done moving
    if ([tileMap numberOfRunningActions] == 0)
    {
        if (currentMoveDirection != MoveDirectionNone)
        {
            CGPoint tilePos = [self tilePosFromLocation:screenCenter tileMap:tileMap];

            CGPoint offset = moveOffsets[currentMoveDirection];
            tilePos = CGPointMake(tilePos.x + offset.x, tilePos.y + offset.y);
            tilePos = [self ensureTilePosIsWithinBounds:tilePos];

            if ([self isTilePosBlocked:tilePos tileMap:tileMap] == NO)
            {
                [self centerTileMapOnTileCoord:tilePos tileMap:tileMap];
            }
        }
    }
}

```

Before moving the tilemap, the isTilePosBlocked method is called to see whether the player can actually move there. If the destination tile coordinate is not blocked, he will move; otherwise, he won't.

Adding More Content to the Game

So far, we have a game where you guide a character through an isometric tilemap world. Hiding behind trees and avoiding collisions are just the foundation for a game set in this world. What if you want to add more actors to the world, whether enemies or nonplayer characters (NPCs)?

In principle, you animate them just like you'd move the player, except that the player is centered on-screen, whereas NPCs can be anywhere on the tilemap. Still, you only need to determine which direction the NPC should walk toward and then move him like the layer is moved in the `centerTileMapOnTileCoord` method. The only difference is that the actions are run on the NPC, and the directions need to be reversed, since you aren't moving the layer; the NPC is moving on the layer.

As soon as you have NPCs wandering around, the next step is to ask how you can get them to move from A to B while avoiding obstacles and finding the shortest route. The answer to that is the A* pathfinding algorithm, which is an industry standard and has been adapted and tweaked for many situations. Tile-based games are ideal candidates for this particular pathfinding algorithm, since actor positions are usually restricted to the tile coordinates. For an in-depth introduction to the A* pathfinding algorithm, and honestly to a lot of game programming topics in general, you must visit Amit's A* pages at <http://theory.stanford.edu/~amitp/GameProgramming/>.

And you'll want to visit Amit's Game Programming Information pages. He links to articles concerning artificial intelligence and tile-based games, including procedural world generation. A lot of the articles may seem dated, but, in fact, most of them are timeless and are still valuable sources of information. Check them out at www-cs-students.stanford.edu/~amitp/gameprog.html.

Summary

In this chapter, you learned what's special about isometric tilemaps, how isometric tiles are designed, and how to create a tilemap with a perceived depth. You learned how to create and improve such an isometric tilemap with Tiled by adding an impassable border and preventing collisions.

You also learned the techniques necessary to set up a tilemap for use with cocos2d and how to set up cocos2d itself with 2D projection and a depth buffer for correct rendering of overlapping tiles and sprites.

Finally, we added a player whose sprite is correctly clipped depending on whether it is in front or behind tiles. You can also move the player around tile by tile by tapping and holding the screen relative to the player sprite to make him go in that direction. That he will do unless the direction is blocked by a mountain, wall, or any other movement-blocking tile that you set in Tiled.

So far, you've worked with games that needed to be controlled and animated in discrete steps. You were responsible for implementing all of the actor's movement and rotation as well as checking for collisions. In the next two chapters, I'll introduce you to physics engines, which allow you to lean back as you watch your game's objects bounce around and collide with each other all by themselves. If this is the first time you've worked with a physics engine, it will be a magical experience. Hold onto your hat!

Physics Engines

Physics engines are what drive popular iOS games like Angry Birds, Stick Golf, Jelly Car, and Stair Dismount. They allow you to create a world that feels more dynamic and lifelike.

Cocos2d is distributed with two physics engines: Box2D and Chipmunk. Both are designed to work only in two dimensions, so they're a perfect fit for cocos2d.

In this chapter, you'll learn the basics of both physics engines, and along the way you'll probably come to appreciate one more than the other. I'll briefly explain the differences between the two physics engines, but for the most part it's a choice based on personal preference.

If you've never worked with a physics engine before, I'll also give you a quick introduction to their basic concepts and key elements.

Basic Concepts of Physics Engines

You can think of a physics engine as an animation system for game objects. Of course, it's up to the game developer to connect and synchronize game objects like sprites with the physics objects, called *rigid bodies*. They are called that because physics engines animate them as if they were stiff, nondeformable objects. This simplification allows physics engines to calculate a large number of bodies.

There are generally two types of bodies: *dynamic* (moving) and *static* (immovable) objects. The differentiation is important because static bodies never move and should never be moved and the physics engine can rely on certain optimizations based on the fact that static bodies never collide with each other.

Dynamic bodies, on the other hand, collide with each other and with static bodies. They also have at least three defining parameters in addition to their position and rotation. One is *density*, or *mass*—in other words, a measure of how heavy an object is. Then there is *friction*—how resistant or slippery the dynamic body is with respect to moving over surfaces. Finally, there is *restitution*, which determines the bounciness of the object. While impossible in the real world, physics engines can create dynamic bodies

that will never lose momentum as they bounce, or even gain speed every time they bounce off of some other body.

Both dynamic and static bodies have one or more shapes that determine the area the body encompasses. Most often the shape is a circle or a rectangle, but it can also be a polygon, a number of vertices forming any complex shape, or merely a straight line. The shapes of a body determine where other bodies and their shapes collide. And in turn, each collision generates *contact points*—the points where the two bodies' shapes intersect. These contact points can be used to play particle effects or add scratch marks at exactly the places where the bodies have collided.

Dynamic bodies are animated by the physics engine through applying forces, impulses, and torque instead of setting their position and rotation directly. Modifying position and rotation directly is advised against, because physics engines make certain predictions that no longer hold true if you manually reposition bodies.

Finally, bodies can be connected together by using a selection of *joints*, which limit the movement of connected bodies in various ways. Some joints may have motors, which for example can act as the drive wheel of a car or as friction for the joint so that the joint tries to snap back to its original position.

Limitations of Physics Engines

Physics engines have their limits. They have to take shortcuts because the real world is prohibitively complex to simulate. The use of rigid bodies is such an example. In some extreme cases, physics engines may not be able to catch every collision— for example, when bodies are moving very fast, in which case they can tunnel through each other. While this has been proven to happen in quantum physics, real-world objects that we can actually see with our own eyes have yet to show that effect.

Rigid bodies can sometimes penetrate each other and get stuck, especially if they are constrained by joints that hold two or more bodies together and limit their range of motion. This can lead to undesirable trembling motions of the bodies as they struggle to move apart while keeping their joint constraints satisfied. You may have seen this even in modern games; for example, the ragdolls of dead actors in a first-person shooter game can sometimes be seen in very unnatural body positions or limbs not coming to rest at all. Such events continue to amaze and amuse players.

And of course there can be game-play issues. With rigid bodies, you never know what will happen given enough players interacting with them. Eventually some players may manage to block themselves and trap themselves in a dead-end situation, or they may figure out how to exploit the physics simulation to be able to move to areas they shouldn't be able to reach.

The Showdown: Box2D vs. Chipmunk

Cocos2d is distributed with two physics engines: Box2D and Chipmunk. How should you choose between them?

In a lot of cases, this boils down to a matter of taste. Most developers argue along the lines of the programming language in which the physics engines are implemented: Box2D is written entirely in C++, while Chipmunk is written in C.

You may favor Box2D over Chipmunk simply because of its C++ interface. Being written in C++ has the added advantage that it integrates better with the likewise object-oriented Objective-C language. You may also appreciate that Box2D uses fully written-out words throughout, as opposed to the many one-letter abbreviations common in Chipmunk. In addition, Box2D makes use of operator overloading so that you can, for example, add two vectors simply by writing the following:

```
b2Vec2 newVec = vec1 + vec2;
```

Box2D has a few features that Chipmunk doesn't offer. For example, it has a solution for fast-moving objects (bullets) that solves the tunneling problem I mentioned earlier.

If you're not very familiar with C++, you may find the steep learning curve of the C++ language daunting. To that end, the Chipmunk physics engine may be more welcoming to you if you're more familiar with C language syntax or prefer a lightweight implementation of a physics engine that is easier to pick up and learn. Its being part of the cocos2d distribution for many months longer than Box2D has also spawned more tutorials and forum posts about Chipmunk, although Box2D tutorials are catching on.

One warning ahead of time: Chipmunk uses *C structures*, which expose internal fields. If you're experimenting and don't know what certain fields are used for and they're not documented, that means you should not change them because they are used only internally.

There is also the popular Chipmunk *SpaceManager*, which adds an easy-to-use Objective-C interface to Chipmunk. SpaceManager also makes it easy to attach cocos2d sprites to bodies and adds debug drawing, among other things. You can download Chipmunk SpaceManager here: <http://code.google.com/p/chipmunk-spacemanager>.

In terms of functionality, you can safely choose either engine. Unless your game relies on one particular feature that one physics engine has and the other doesn't, you can use either one to great effect. Especially if you have no familiarity with either physics engine, feel free to choose the one that appeals to you more based on the language and coding style.

I will now introduce you to the basics of both physics engines for the rest of this chapter so that you can decide for yourself which one appeals to you more. In the next chapter, you'll learn how to build a playable pinball game with bumpers, flippers, and lanes built with Box2D and the VertexHelper tool.

Box2D

The Box2D physics engine is written in C++. It was developed by Erin Catto, who has given presentations about physics simulations at every Game Developers Conference (GDC) since 2005. It was his GDC 2006 presentation that eventually led to the public release of Box2D in September 2007. It has been in active development ever since.

Because of its popularity, the Box2D physics engine is distributed with cocos2d. You can create a new project using Box2D by choosing the cocos2d Box2D application template from Xcode's **File** ► **New Project** dialog. This project template adds the necessary Box2D source files to the project and provides a test project in which you can add boxes that bounce off each other, as shown in Figure 12–1. They also fall according to gravity, depending on how you're holding your device.



Figure 12–1. *The PhysicsBox2D example project*

CAUTION: Because the Box2D physics engine is written in C++, you have to use the file extension `.mm` instead of `.m` for all your project's implementation files. This tells Xcode to treat the implementation file's source code as either Objective-C++ or C++ code. With the `.m` file extension, Xcode will compile the code as Objective-C and C code and won't understand Box2D's C++ code, which will result in numerous compile errors for every line of code that uses or makes a reference to Box2D. So if you're getting a lot of errors, check that your implementation files all end in `.mm`, and if not, rename them.

Documentation for Box2D is available in two places. First, you can read the Box2D manual online at www.box2d.org/manual.html, which introduces you to common concepts and shows example code. The Box2D API reference is distributed with Box2D itself, which you can download at <http://code.google.com/p/box2d>. You can also find the Box2D API reference in the Box2D version distributed with the book's source code. The API reference is in the folder `/Physics Engine Libraries/Box2D_v2.1.2/Box2D/Documentation/API`—to view it, locate and open the file `index.html` in that folder.

If you like Box2D, you should also consider donating to the project; you can do so via the **Donate** button on its home page: www.box2d.org.

The World According to Box2D

Because the example project provided by cocos2d is quite complex, I decided to break it down into smaller pieces and re-create the example project step by step but not without adding some extras and variations.

Listing 12–1 shows the HelloWorldScene header file from the PhysicsBox2D01 project.

Listing 12–1. *The Box2D Hello World Interface*

```
#import "cocos2d.h"
#import "Box2D.h"
#import "GLes-Render.h"

enum
{
    kTagBatchNode,
};

@interface HelloWorld : CCLayer
{
    b2World* world;
}

+(id) scene;

@end
```

It's fairly standard, except that it includes the Box2D.h header file and adds a member variable of type b2World. This is the physics world—think of it as the container class that will store and update all physics bodies.

The Box2D world is initialized by creating a new b2World object in the HelloWorldScene's init method, as shown in Listing 12–2.

Listing 12–2. *Initializing the Box2D World*

```
b2Vec2 gravity = b2Vec2(0.0f, -10.0f);
bool allowBodiesToSleep = true;
world = new b2World(gravity, allowBodiesToSleep);
```

Remember that Box2D is written in C++. To instantiate one of Box2D's classes, you have to add the new keyword in front of the class's name. If Box2D were written in Objective-C, the equivalent line might look like this:

```
world = [[b2World alloc] initWithGravity:gravity allowsSleep:allowBodiesToSleep];
```

In other words, the new keyword in C++ is equivalent to sending the alloc message to an Objective-C class followed by an init message. That of course means you also have to deallocate the Box2D world. In C++ this is done using the delete keyword:

```
-(void) dealloc
{
```

```
    delete world;  
    [super dealloc];  
}
```

The Box2D world in Listing 12–2 is initialized with an initial gravity vector and a flag that determines whether the dynamic bodies are allowed to fall asleep.

Sleeping bodies? It's a trick that allows the physics simulation to quickly skip over objects that do not need processing. A dynamic body goes to sleep when the forces applied to it have been below a threshold for a certain amount of time. In other words, if the dynamic body is moving and rotating very slowly or not at all, the physics engine will flag it as sleeping and won't apply forces to it anymore—that is, unless an impulse or force applied to the body is strong enough to make the body move or rotate again. This trick allows the physics engine to save time by not processing the bodies that are at rest. Unless all of your game's dynamic bodies are in constant motion, you should enable this feature by setting the `allowBodiesToSleep` variable to `true`, as in Listing 12–2.

The gravity passed to Box2D is a `b2Vec2` struct type. It's essentially the same as a `CGPoint`, because it stores `x` and `y` float values. In this case, and fortunately for us in the real world too, gravity is a constant force. The `0, -10` vector is constantly applied to all dynamic bodies, making them fall down, which in this case means toward the bottom of the screen.

Restricting Movement to the Screen

World setup, check. What next? Well, we should limit the movement of the Box2D bodies to within the visible screen area. For that, we'll need a static body. The simplest way to create a static body is by using the world's `CreateBody` method and an empty body definition:

```
// Define the static container body, which will provide the collisions at screen borders  
b2BodyDef containerBodyDef;  
b2Body* containerBody = world->CreateBody(&containerBodyDef);
```

Bodies are always created through the world's `CreateBody` method. This ensures that the body's memory is correctly allocated and freed. The `b2BodyDef` is a struct that holds all the data needed to create a body, such as position and the body type. By default, an empty body definition creates a static body at position `0, 0`.

NOTE: The `&containerBodyDef` variable is passed with a leading `&` (ampersand) character to the `CreateBody` method. That's C++ for Give me the memory address of `containerBodyDef`. If you look at the definition of the `CreateBody` method, it requires a pointer passed to it: `b2World::CreateBody(const b2BodyDef *def);`. Since pointers store a memory address, you can get that address of a nonpointer variable by prefixing it with the ampersand character.

The body itself won't do anything. To make it enclose the screen area, you'll have to create a shape with four sides:

```
// Create the screen box sides by using a polygon assigning each side individually
b2PolygonShape screenBoxShape;
int density = 0;

// Bottom
screenBoxShape.SetAsEdge(lowerLeftCorner, lowerRightCorner);
containerBody->CreateFixture(&screenBoxShape, density);

// Top
screenBoxShape.SetAsEdge(upperLeftCorner, upperRightCorner);
containerBody->CreateFixture(&screenBoxShape, density);

// Left side
screenBoxShape.SetAsEdge(upperLeftCorner, lowerLeftCorner);
containerBody->CreateFixture(&screenBoxShape, density);

// Right side
screenBoxShape.SetAsEdge(upperRightCorner, lowerRightCorner);
containerBody->CreateFixture(&screenBoxShape, density);
```

You may notice the missing declarations for the corner variable names. I'll get to them in a moment. First I'd like you to focus on how the `b2PolygonShape screenBoxShape` variable is reused. Each `SetAsEdge` method call is followed by a call to `containerBody->CreateFixture()`, which uses the `->` operator to denote that `containerBody` is a C-style pointer and passes `&screenBoxShape`. Since Box2D makes a copy of `screenBoxShape`, you can safely reuse the same shape to create all four sides enclosing the screen area without modifying or overriding the previous lines. Since the body is a static body (the default setting for Box2D bodies), density doesn't matter and is set to 0.

NOTE: The `b2PolygonShape` class has a `SetAsBox` method that looks like it might make the definition of the screen area easier by simply providing the screen's width and height. However, that would make the inside of the body a solid object, and any dynamic body added to the screen would actually be contained inside the solid shape. This would make the dynamic bodies try to move away from the collision, possibly at rapid speeds. The sides need to be created separately in order to make only the sides of the screen solid.

Now we'll move on to the missing variable declarations. Notice that the screen width and height is divided by a `PTM_RATIO` constant to convert them from pixels to meters:

```
// For the ground body we'll need these values
CGSize screenSize = [CCDirector sharedDirector].winSize;
float widthInMeters = screenSize.width / PTM_RATIO;
float heightInMeters = screenSize.height / PTM_RATIO;
b2Vec2 lowerLeftCorner = b2Vec2(0, 0);
b2Vec2 lowerRightCorner = b2Vec2(widthInMeters, 0);
b2Vec2 upperLeftCorner = b2Vec2(0, heightInMeters);

b2Vec2 upperRightCorner = b2Vec2(widthInMeters, heightInMeters);
```

Why meters, and what is `PTM_RATIO`? Box2D is optimized to work best with dimensions in the range of 0.1 to 10 meters. It is tuned for the metric system, so all distances are

considered to be meters, all masses are in kilograms, and time is measured in—quite oddly—seconds. If you're not familiar with the meters, kilograms, and seconds (MKS) system, don't worry—you don't have to meticulously convert yards into meters and pounds into kilograms. The conversion to meters is just a way to keep the distance values for Box2D in the desirable range of 0.1 to 10, and the masses used by bodies do not resemble real-world masses anyway. The masses of bodies will often need to be tweaked by feel rather than by using realistic weights.

You should try to keep the dimensions of objects in your world as close to 1 meter as much as possible. That is not to say that you can't have objects that are smaller than 0.1 meters or larger than 10 meters, but you may run into glitches and strange behavior if you create relatively small or large bodies.

The `PTM_RATIO` is defined like this:

```
#define PTM_RATIO 32
```

It is used to define that 32 pixels on the screen equal 1 meter in Box2D. A box-shaped body that's 32 pixels wide and high will be 1 meter wide and high. A body that's 4×4 pixels in size will be 0.125×0.125 meters in Box2D, while a relatively huge object of 256×256 pixels will be 8×8 meters in Box2D. The `PTM_RATIO` allows you to scale the size of Box2D objects down to the dimensions within which Box2D works best, and a `PTM_RATIO` of 32 is a good compromise for a screen area that may be as large as 1024×768 pixels on the iPad.

Converting Points

Note that the `b2Vec2` struct is different from `CGPoint`, which means you cannot use a `CGPoint` where a `b2Vec2` is required, and vice versa. In addition, Box2D points need to be converted to meters and back to pixels. To avoid making any mistakes, such as forgetting to convert from or to meters or simply making a typo and using the x coordinate twice, it's highly recommended to wrap this repetitive code into convenience methods like these:

```
-(b2Vec2) toMeters:(CGPoint)point
{
    return b2Vec2(point.x / PTM_RATIO, point.y / PTM_RATIO);
}

-(CGPoint) toPixels:(b2Vec2)vec
{
    return ccpMult(CGPointMake(vec.x, vec.y), PTM_RATIO);
}
```

This allows you to write the following code to easily convert between `CGPoint` and pixels to `b2Vec2` and meters:

```
// Box2D coordinates to Cocos2D point coordinates
b2Vec2 vec = b2Vec2(200, 200);
CGPoint pointFromVec = [self toPixels:vec];

// Cocos2D point coordinates to Box2D coordinates
```

```
CGPoint point = CGPointMake(100, 100);
b2Vec2 vecFromPoint = [self toMeters:point];
```

Adding Boxes to the Box2D World

With a static body containing the objects within screen boundaries, all that's missing is something to be kept within the screen boundaries. How about little boxes, then?

I've added David Gervais' orthogonal tileset image `dg_grounds32.png` to the Resources folder of the `PhysicsBox2D01` project. The tiles are 32×32 pixels, so they'll make perfect 1×1-meter boxes. Listing 12-3 is the code in the `init` method that adds the texture and creates a couple boxes. It also schedules the update method, which is needed to update the box sprite positions, and it enables touch so that the user can tap the screen to create a new box.

Listing 12-3. Adding an Initial Set of Boxes

```
// Use the orthogonal tileset for the little boxes
CCSpriteBatchNode* batch = [CCSpriteBatchNode
    batchNodeWithFile:@"dg_grounds32.png"
    capacity:150];

[self addChild:batch z:0 tag:kTagBatchNode];

// Add a few objects initially
for (int i = 0; i < 11; i++)
{
    [self addNewSpriteAt:CGPointMake(screenSize.width / 2, screenSize.height / 2)];
}

[self scheduleUpdate];
self.isTouchEnabled = YES;
```

The `addNewSpriteAt` method shown in Listing 12-4 is part of the `cocos2d Box2D` application template project but slightly modified to make use of all the tiles in the tileset.

Listing 12-4. Adding a New Dynamic Body with a Sprite

```
-(void) addNewSpriteAt:(CGPoint)pos
{
    CCSpriteBatchNode* batch =
        (CCSpriteBatchNode*) [self getChildByTag:kTagBatchNode];

    int idx = CCRANDOM_0_1() * TILESET_COLUMNS;
    int idy = CCRANDOM_0_1() * TILESET_ROWS;
    CGRect tileRect = CGRectMake(TILESIZE * idx, TILESIZE * idy, TILESIZE, TILESIZE);
    CCSprite* sprite = [CCSprite spriteWithBatchNode:batch rect:tileRect];
    sprite.position = pos;
    [batch addChild:sprite];

    // Create a body definition and set it to be a dynamic body
    b2BodyDef bodyDef;
    bodyDef.type = b2_dynamicBody;
    bodyDef.position = [self toMeters:pos];
    bodyDef.userData = sprite;
```

```

b2Body* body = world->CreateBody(&bodyDef);

// Define a box shape and assign it to the body fixture
b2PolygonShape dynamicBox;
float tileInMeters = TILESIZE / PTM_RATIO;
dynamicBox.SetAsBox(tileInMeters * 0.5f, tileInMeters * 0.5f);

b2FixtureDef fixtureDef;
fixtureDef.shape = &dynamicBox;
fixtureDef.density = 0.3f;
fixtureDef.friction = 0.5f;
fixtureDef.restitution = 0.6f;
body->CreateFixture(&fixtureDef);
}

```

First, a sprite is created from the `CCSpriteBatchNode` by using `CCSprite's` `spriteWithBatchNode` initializer and supplying a `CGRect` that is 32×32 pixels in size, to randomly pick one of the tileset's tiles as the sprite's image.

Then a body is created, but this time the `b2BodyDef` type property is set to `b2_dynamicBody`, which makes it a dynamic body that can move around and collide with other dynamic bodies. The previously created sprite is assigned to the `userData` property. Later, when you are iterating over the bodies in the world, this allows you to quickly access the body's sprite.

The body's shape is a `b2PolygonShape` set to a box shape that is half a meter in size. The `SetAsBox` method creates a box shape that is twice the given width and height, so the coordinates need to be divided by 2—or, as in this case, multiplied by 0.5f to create a box shape whose sides are 1 meter wide and high.

The dynamic body also needs a fixture that contains the body's essential parameters—first and foremost the shape but also the density, friction, and restitution—which influence how the body moves and bounces around in the world. Consider the fixture to be a set of data used by bodies.

Connecting Sprites with Bodies

The box sprites won't follow their physics bodies automatically, and the bodies won't do anything unless you regularly call the `Step` method of the Box2D world. You then have to update the sprite positions by taking the body position and angle and assigning it to the sprite. This is done in the update method shown in Listing 12–5.

Listing 12–5. *Updating Each Body's Sprite Position and Rotation*

```

-(void) update:(ccTime)delta
{
    // Advance the physics world by one step, using fixed time steps
    float timeStep = 0.03f;
    int32 velocityIterations = 8;
    int32 positionIterations = 1;
    world->Step(timeStep, velocityIterations, positionIterations);

    for (b2Body* body = world->GetBodyList(); body != nil; body = body->GetNext())
    {

```

```

    CCSprite* sprite = (CCSprite*)body->GetUserData();
    if (sprite!= NULL)
    {
        sprite.position = [self toPixels:body->GetPosition()];
        float angle = body->GetAngle();
        sprite.rotation = CC_RADIANS_TO_DEGREES(angle) * -1;
    }
}

```

The Box2D world is animated by regularly calling the Step method. It takes three parameters. The first is `timeStep`, which tells Box2D how much time has passed since the last step. It directly affects the distance that objects will move in this step. For games, it is not recommended to pass the delta time as `timeStep`, because the delta time fluctuates, and so the speed of the physics bodies will not be constant. This effect rears its ugly head when the device may be taking a tenth of a second to do background processing, such as sending or receiving an e-mail in the background. This can make all physics objects move large distances in the next frame. In a game, you'd rather have the game stop for a tenth of a second and then carry on where you left off. Without a fixed time step, the physics engine would try to cope with a short interruption by moving all objects based on the time difference. If the time difference is large, the objects will move a lot more in a single frame, and that can lead to them suddenly moving a large distance.

The second and third parameters to the Step method are the number of iterations. They determine the accuracy of the physics simulation and also the time it takes to calculate the movement of the bodies. It's a trade-off between speed and accuracy. In the case of position iterations, you can safely err on the side of speed and require only a single iteration—more position accuracy is not normally needed in games unless you experience objects not coming to rest, colliding in unnatural ways, or missing collision entirely. Velocity is more important, however; a good starting point for velocity iterations is eight. More than ten velocity iterations have no discernable effect in games, but just one to four iterations won't be enough to get a stable simulation. The fewer the velocity iterations, the more bumpily and restlessly the objects will behave. I encourage you to experiment with these values.

After the world has advanced one step, the `for` loop iterates over all of the world's bodies using the `world->GetBodyList` and `body->GetNext` methods. For each body, its user data is returned and cast to a `CCSprite` pointer. If it exists, the body's position is converted to pixels and assigned to the sprite's position so that the sprite moves along with the body. Likewise, the body's angle is obtained; because that measurement is in radians, it's converted to degrees using cocos2d's `CC_RADIANS_TO_DEGREES` method and multiplied by `-1` to rotate the sprites in the same direction as the body.

Collision Detection

Box2D has a `b2ContactListener` class, which you are supposed to subclass if you want to receive collision callbacks. The following code refers to the `PhysicsBox2D02` project.

Create a new class in Xcode and name it `ContactListener`. Then rename the implementation file to `ContactListener.mm` so it has the file extension `.mm`. Then you will be able to use both C++ and Objective-C code in the same file. Listing 12–6 shows the `ContactListener` header file.

Listing 12–6. *The ContactListener Class's Interface*

```
#import "Box2D.h"

class ContactListener : public b2ContactListener
{
private:
    void BeginContact(b2Contact* contact);
    void EndContact(b2Contact* contact);
};
```

It's a C++ class, so the class definition is a little different. Note the trailing semicolon after the last bracket. It's a common error to forget that semicolon. The `BeginContact` and `EndContact` methods are defined by `Box2D` and get called whenever there is a collision between two bodies.

In the implementation, I merely change the sprite's colors to magenta while the two bodies are in contact and set it back to white when they are no longer in contact, as shown in Listing 12–7.

Listing 12–7. *Checking for the Beginning and Ending of Collisions*

```
#import "ContactListener.h"
#import "cocos2d.h"

void ContactListener::BeginContact(b2Contact* contact)
{
    b2Body* bodyA = contact->GetFixtureA()->GetBody();
    b2Body* bodyB = contact->GetFixtureB()->GetBody();
    CCSprite* spriteA = (CCSprite*)bodyA->GetUserData();
    CCSprite* spriteB = (CCSprite*)bodyB->GetUserData();

    if (spriteA != NULL && spriteB != NULL)
    {
        spriteA.color = ccMAGENTA;
        spriteB.color = ccMAGENTA;
    }
}

void ContactListener::EndContact(b2Contact* contact)
{
    b2Body* bodyA = contact->GetFixtureA()->GetBody();
    b2Body* bodyB = contact->GetFixtureB()->GetBody();
    CCSprite* spriteA = (CCSprite*)bodyA->GetUserData();
    CCSprite* spriteB = (CCSprite*)bodyB->GetUserData();
```

```

    if (spriteA != NULL && spriteB != NULL)
    {
        spriteA.color = ccWHITE;
        spriteB.color = ccWHITE;
    }
}

```

b2Contact contains all the contact information, including two sets of everything suffixed with A and B. These are the two contacting bodies; no differentiation is made as to which is colliding with the other—they are both simply colliding with each other. If, for example, you have an enemy colliding with a player's bullet, you would want to damage the enemy, not the bullet. It is up to you to determine which is which. Also keep in mind that the contact methods may be called multiple times per frame, once for each contact pair.

It's a bit convoluted to get to the sprite from the contact through the fixture to the body and then get the user data from that. The Box2D API reference certainly helps you find your way through the hierarchy, and with a little experience this will become second nature.

To actually get the ContactListener connected with Box2D, you have to add it to the world. In HelloWorldScene, import the ContactListener.h header file and add a ContactListener* contactListener to the class as a member variable:

```

#import "cocos2d.h"
#import "Box2D.h"
#import "GL ES-Render.h"

#import "ContactListener.h"

...

@interface HelloWorld : CCLayer
{
    b2World* world;
    ContactListener* contactListener;
}

...

@end

```

In the init method of HelloWorldScene, you can then create a new ContactListener instance and set it as the contact listener for the world:

```

contactListener = new ContactListener();
world->SetContactListener(contactListener);

```

What remains is to delete the contactListener in the dealloc method to free its memory:

```

-(void) dealloc
{
    delete contactListener;
    delete world;

    [super dealloc];
}

```

Now the boxes in the PhysicsBox2D02 project will be tinted purple whenever they touch other boxes.

Joint Venture

With joints, you can connect bodies together. The type of joint determines which way the connected bodies are connected. In this example method, I create four bodies total. Three are dynamic bodies connected to each other using a *revolute joint*, which keeps the bodies at the same distance but allows them to rotate 360 degrees around each other. If you find it hard to imagine how these objects might behave, you should try them in the PhysicsBox2D02 project. In this case, working code can explain it better than words or a static image could. The fourth body is a static body to which one of the dynamic bodies is also attached using a revolute joint.

```
-(void) addSomeJoinedBodies:(CGPoint)pos
{
    // Create a body definition and set it to be a dynamic body
    b2BodyDef bodyDef;
    bodyDef.type = b2_dynamicBody;

    // Position must be converted to meters
    bodyDef.position = [self toMeters:pos];
    bodyDef.position = bodyDef.position + b2Vec2(-1, -1);
    bodyDef.userData = [self addRandomSpriteAt:pos];
    b2Body* bodyA = world->CreateBody(&bodyDef);
    [self bodyCreateFixture:bodyA];

    bodyDef.position = [self toMeters:pos];
    bodyDef.userData = [self addRandomSpriteAt:pos];
    b2Body* bodyB = world->CreateBody(&bodyDef);
    [self bodyCreateFixture:bodyB];

    bodyDef.position = [self toMeters:pos];
    bodyDef.position = bodyDef.position + b2Vec2(1, 1);
    bodyDef.userData = [self addRandomSpriteAt:pos];
    b2Body* bodyC = world->CreateBody(&bodyDef);
    [self bodyCreateFixture:bodyC];

    // Create the revolute joints
    b2RevoluteJointDef jointDef;
    jointDef.Initialize(bodyA, bodyB, bodyB->GetWorldCenter());
    bodyA->GetWorld()->CreateJoint(&jointDef);

    jointDef.Initialize(bodyB, bodyC, bodyC->GetWorldCenter());
    bodyA->GetWorld()->CreateJoint(&jointDef);

    // Create an invisible static body and attach body A to it
    bodyDef.type = b2_staticBody;
    bodyDef.position = [self toMeters:pos];
    b2Body* staticBody = world->CreateBody(&bodyDef);

    jointDef.Initialize(staticBody, bodyA, bodyA->GetWorldCenter());
    bodyA->GetWorld()->CreateJoint(&jointDef);
}
```

b2BodyDef is reused for all bodies; only the position is modified for each body, and a random CCSprite is created and assigned as userData. The addRandomSpriteAt method contains the code that creates a sprite from the CCSpriteBatchNode, as discussed earlier. Since in the addSomeJoinedBodies method there are now several sprites needed, it made sense to refactor the creation of a sprite into the method addRandomSpriteAt.

b2RevoluteJointDef is filled with data by using the Initialize method providing two bodies to connect to each other and a coordinate where the joint is located. By using one body's GetWorldCenter coordinate, that body will be centered on the joint and allowed only to rotate around itself.

The joint is created by the CreateJoint method of the b2World class. Even though the HelloWorldScene class in the PhysicsBox2D02 project has a b2World member variable, I wanted to illustrate that you can also get the world through any body—it doesn't matter which one—by using the body's GetWorld method. This is good to know, because in the ContactListener discussed earlier, you do not have a b2World member variable, so you'll have to get the b2World pointer through one of the contact bodies.

Chipmunk

The Chipmunk physics engine was developed by Scott Lembcke of Howling Moon Software. Chipmunk was actually inspired by an early version of Box2D, before it was a full-fledged physics engine. You can download Chipmunk from its Google Code site. If you like Chipmunk, you should also consider donating to the project, which you can do via the **Donate** button from the same site: <http://code.google.com/p/chipmunk-physics>. The Chipmunk documentation is located on Scott's home page, at <http://files.slembcke.net/chipmunk/release/ChipmunkLatest-Docs>. And if you need help, you can find that in the Chipmunk forums: www.slembcke.net/forums.

Objectified Chipmunk

There are actually two Objective-C wrappers available for Chipmunk, and more are being worked on. I don't discuss them in this book, but you should be aware of them and try them.

Scott's Objective-C wrapper is part of the Chipmunk distribution but works only in the iPhone Simulator. To be able to deploy it to the iPhone, you have to buy his Chipmunk Objective-C wrapper on the Howling Moon Software web site, at <http://howlingmoonsoftware.com/objectiveChipmunk.php>.

It's always best to try before you buy, so you can follow this tutorial on how to use the Objective-C wrapper for the iPhone Simulator: <http://files.slembcke.net/chipmunk/tutorials/SimpleObjectiveChipmunk>.

The alternative is Chipmunk SpaceManager, written by Robert Blackwood, which comes with a free Objective-C wrapper for Chipmunk. However, its main purpose is to make integration of Chipmunk with cocos2d easier. If you would like to try Chipmunk

SpaceManager, you can download it from the following link, where you can also donate to the project should you like it: <http://code.google.com/p/chipmunk-spacemanager>.

The SpaceManager API reference can be found on Robert's home page at www.mobile-bros.com/spacemanager/docs.

Both the Chipmunk and SpaceManager distributions, including their respective documentation files, are also in this chapter's source code folder, in the Physics Engine Libraries subfolder.

Chipmunks in Space

For the Chipmunk tutorial, I'll be building the same project as before. I'll start with the PhysicsChipmunk01 project and the initial setup of the Chipmunk physics engine. Listing 12–8 shows the HelloWorldScene header file.

Listing 12–8. *The Chipmunk HelloWorld Interface*

```
#import "cocos2d.h"
#import "chipmunk.h"

enum
{
    kTagBatchNode = 1,
};

@interface HelloWorld : CCLayer
{
    cpSpace* space;
}

+(id) scene;

@end
```

There's nothing unusual here, except for the `cpSpace` member variable. Instead of *world*, Chipmunk calls its world a *space*. It's a different term for the same thing. The Chipmunk space contains all the rigid bodies.

Chipmunk is initialized in HelloWorldScene's `init` method as follows:

```
cpInitChipmunk();

space = cpSpaceNew();
space->iterations = 8;
space->gravity = CGPointMake(0, -100);
```

The very first thing you must do before using any Chipmunk methods is to call `cpInitChipmunk`. After that, you can create the space with `cpSpaceNew` and set the number of iterations—in this case eight. This is the same iteration count we used for velocity iterations in the Box2D example's update method. Chipmunk knows only one type of iteration—the `elasticIterations` field is deprecated and should no longer be used. I mention this in case you are already familiar with Chipmunk. You may get away with fewer than eight iterations if your game does not allow objects to stack; otherwise,

you may find that stacked objects never get to rest and keep jittering and sliding for a long period of time.

Notice how Chipmunk can use the same `CGPoint` structure used in the iPhone SDK. Chipmunk internally uses a structure called `cpVect`, but in cocos2d you can use both interchangeably. I use a `CGPoint` to set the gravity to -100 , which means downward acceleration that is roughly the same as that used in the Box2D project.

Of course, the space also needs to be released in the `dealloc` method; this is done by calling `cpSpaceFree` and passing the space as a parameter:

```
-(void) dealloc
{
    cpSpaceFree(space);
    [super dealloc];
}
```

Boxing-In the Boxes

To keep all the boxes within the boundaries of the screen, you need to create a static body whose shape defines the screen area. First, the variables for the screen's corners are defined:

```
// For the ground body we'll need these values
CGSize screenSize = [CCDirector sharedDirector].winSize;
CGPoint lowerLeftCorner = CGPointMake(0, 0);
CGPoint lowerRightCorner = CGPointMake(screenSize.width, 0);
CGPoint upperLeftCorner = CGPointMake(0, screenSize.height);
CGPoint upperRightCorner = CGPointMake(screenSize.width, screenSize.height);
```

Contrary to Box2D, you do not have to take any pixel-to-meter ratio into account. You can use the screen size in pixels as it is to define the corner points and to work with Chipmunk bodies in general.

Next you'll create the static body by using the `cpBodyNew` and passing `INFINITY` for both parameters, which makes the body a static body. Those parameters are mass and inertia, and with them being set to the `INFINITY` value, this body isn't going to go anywhere.

```
// Create the static body that keeps objects within the screen area
float mass = INFINITY;
float inertia = INFINITY;
cpBody* staticBody = cpBodyNew(mass, inertia);
```

NOTE: Mass and inertia in Chipmunk are comparable to density and friction in Box2D. The difference between inertia and friction is that the former determines the resistance of a body to start moving, while the latter determines how much motion a body loses when it is in contact with other bodies.

Next, you'll define the shape that goes with the body and makes up the screen borders, as shown in Listing 12–9.

Listing 12–9. Creating the Screen Border Collisions

```

cpShape* shape;
float elasticity = 1.0f;
float friction = 1.0f;
float radius = 0.0f;

// Bottom
shape = cpSegmentShapeNew(staticBody, lowerLeftCorner, lowerRightCorner, radius);
shape->e = elasticity;
shape->u = friction;
cpSpaceAddStaticShape(space, shape);

// Top
shape = cpSegmentShapeNew(staticBody, upperLeftCorner, upperRightCorner, radius);
shape->e = elasticity;
shape->u = friction;
cpSpaceAddStaticShape(space, shape);

// Left side
shape = cpSegmentShapeNew(staticBody, lowerLeftCorner, upperLeftCorner, radius);
shape->e = elasticity;
shape->u = friction;
cpSpaceAddStaticShape(space, shape);

// Right side
shape = cpSegmentShapeNew(staticBody, lowerRightCorner, upperRightCorner, radius);
shape->e = elasticity;
shape->u = friction;
cpSpaceAddStaticShape(space, shape);

```

The `cpSegmentShapeNew` method is used to create four new line segments to define the sides of the screen area. The `shape` variable is reused for convenience, but it requires you to set *elasticity* (which is the same as restitution) and friction after each call to `cpSegmentShapeNew`. Then each shape is added to the space as a static shape via the `cpSpaceAddStaticShape` method.

NOTE: In Chipmunk you will have to work with one-letter fields like `e` and `u` regularly. Personally, I find that this makes it hard to pick up Chipmunk because you don't immediately grasp the meaning of these fields, and you have to refer to the Chipmunk documentation more often than necessary.

Adding Ticky-Tacky Little Boxes

To add boxes to the world, I used the same code in the `init` method of `HelloWorldScene` as in the `Box2D` example. Refer to Listing 12–3 if you'd like to refresh your memory.

I'll go straight to creating the dynamic body for new boxes, which is what the `addNewSpriteAt` method does (Listing 12–10).

Listing 12–10. Adding a Body with a Sprite, Chipmunk Style

```

-(void) addNewSpriteAt:(CGPoint)pos
{
    float mass = 0.5f;
    float moment = cpMomentForBox(mass, TILESIZE, TILESIZE);
    cpBody* body = cpBodyNew(mass, moment);

    body->p = pos;
    cpSpaceAddBody(space, body);

    float halfTileSize = TILESIZE * 0.5f;
    int numVertices = 4;
    CGPoint vertices[] =
    {
        CGPointMake(-halfTileSize, -halfTileSize),
        CGPointMake(-halfTileSize, halfTileSize),
        CGPointMake(halfTileSize, halfTileSize),
        CGPointMake(halfTileSize, -halfTileSize),
    };

    CGPoint offset = CGPointZero;
    float elasticity = 0.3f;
    float friction = 0.7f;

    cpShape* shape = cpPolyShapeNew(body, numVertices, vertices, offset);
    shape->e = elasticity;
    shape->u = friction;
    shape->data = [self addRandomSpriteAt:pos];
    cpSpaceAddShape(space, shape);
}

```

The dynamic body for the box is created with the `cpBodyNew` method with the given mass and a *moment of inertia*. The moment of inertia determines the resistance of a body to move, and it's calculated by the helper method `cpMomentForBox`, which takes the body's mass and the size of the box—in this case `TILESIZE`—which makes it a 32×32-pixel box.

The body's position, `p`, is then updated, and the body is added to the space via the `cpSpaceAddBody` method. Note that contrary to Box2D, you do not have to convert pixels to meters; you can work with pixel coordinates directly.

Then a list of vertices are created, which will become the corners of the box shape. Because the corner positions are positioned relative to the center of the box we're creating, they are all half a tile's size away from the center. Otherwise, the box shape would be twice as big as the tile.

The `cpPolyShapeNew` method then takes the body as input, the vertices array, and the number of vertices in the array, as well as an optional offset, which is set to `CGPointZero` in this case. Out comes a new `cpShape` pointer for the box shape. The shape's elasticity and friction are set to values that give a similar behavior to the Box2D boxes, and after the sprite is set as user data to the data field, the shape is added to the space via `cpSpaceAddShape`.

The `addRandomSpriteAt` method in Listing 12–11 simply creates the `CCSprite` object that goes along with the new dynamic body.

Listing 12–11. Creating New Box Objects with Random Images

```

-(CCSprite*) addRandomSpriteAt:(CGPoint)pos
{
    CCSpriteBatchNode* batch = (CCSpriteBatchNode*)[self getChildByTag:kTagBatchNode];

    int idx = CCRANDOM_0_1() * TILESET_COLUMNS;
    int idy = CCRANDOM_0_1() * TILESET_ROWS;
    CGRect tileRect = CGRectMake(TILESIZE * idx, TILESIZE * idy, TILESIZE, TILESIZE);
    CCSprite* sprite = [CCSprite spriteWithBatchNode:batch rect:tileRect];
    sprite.position = pos;
    [batch addChild:sprite];

    return sprite;
}

```

Updating the Boxes Sprites

Just like with Box2D, you have to update the sprite's position and rotation to be in line with their dynamic body's position and rotation every frame. Again, this is done in the update method:

```

-(void) update:(ccTime)delta
{
    float timeStep = 0.03f;
    cpSpaceStep(space, timeStep);

    // Call forEachShape C method to update sprite positions
    cpSpaceHashEach(space->activeShapes, &forEachShape, nil);
    cpSpaceHashEach(space->staticShapes, &forEachShape, nil);
}

```

Just as with Box2D, you have to advance the physics simulation using a step method. In this case, it's `cpSpaceStep`, which takes the space and a `timeStep` as input. A fixed time step works best, and just like in Box2D, it's highly recommended you use a fixed time step as opposed to passing the delta time. As long as the framerate doesn't fluctuate heavily (it really shouldn't anyway), using a fixed-time-step approach works very well.

The `cpSpaceHashEach` method calls the C method `forEachShape` for, well, each of the shapes. Or, more accurately, the `cpSpaceHashEach` method calls `forEachShape` for each active shape (dynamic body) and then for each static shape (static body). With the third parameter, you can pass an arbitrary pointer as a parameter to the `forEachShape` method, but because it's not needed in this case, it is set to `nil`. And even though this example project doesn't have static shapes with sprites assigned to them, it nevertheless calls the method for static shapes, just in case you want to be adding some static shapes with a sprite.

The `forEachShape` method is a callback method that's written in C. In the example project, you can find it at the top of the `HelloWorldScene.m` file, outside the `@implementation`. Although it's not strictly necessary to place the method outside the `@implementation`, it signals that this method isn't part of the `HelloWorldScene` class. The method is defined as `static`, which effectively makes it a global method, as Listing 12–12 shows.

Listing 12–12. Updating a Body Sprite's Position and Rotation

```
// C method that updates sprite position and rotation:
static void forEachShape(void* shapePointer, void* data)
{
    cpShape* shape = (cpShape*)shapePointer;
    CCSprite* sprite = (CCSprite*)shape->data;
    if (sprite != nil)
    {
        cpBody* body = shape->body;
        sprite.position = body->p;
        sprite.rotation = CC_RADIANS_TO_DEGREES(body->a) * -1;
    }
}
```

The signature for methods passed to `cpSpaceHashEach` is strictly defined; the method must take two parameters, and both are void pointers. The second parameter would be the data pointer passed as third parameter to `cpSpaceHashEach`. For both `shapePointer` and data pointer, you have to know what kind of object they're pointing to; otherwise, disaster will strike in the form of `EXC_BAD_ACCESS`.

In this case, I know that `shapePointer` is going to point to a `cpShape` struct, so I can safely cast it and then access the shape's data field to get its `CCSprite` pointer. If the sprite is a valid pointer, I can get the body from the shape and use that to set the position and rotation of the sprite to that of the body. As with `Box2D` before, the rotation has to be converted from radians to degrees first and then multiplied by -1 to correct the direction of the rotation.

A Chipmunk Collision Course

Collisions in Chipmunk are also handled by C callback methods. In the `PhysicsChipmunk02` project, I've added the `contactBegin` and `contactEnd` static methods (in Listing 12–13), which do exactly the same as their `Box2D` counterparts: change the color of boxes that are in contact to magenta.

Listing 12–13. Collision Callbacks a la Chipmunk

```
static int contactBegin(cpArbiter* arbiter, struct cpSpace* space, void* data)
{
    bool processCollision = YES;

    cpShape* shapeA;
    cpShape* shapeB;
    cpArbiterGetShapes(arbiter, &shapeA, &shapeB);

    CCSprite* spriteA = (CCSprite*)shapeA->data;
    CCSprite* spriteB = (CCSprite*)shapeB->data;
    if (spriteA != nil && spriteB != nil)
    {
        spriteA.color = ccMAGENTA;
        spriteB.color = ccMAGENTA;
    }

    return processCollision;
}
```

```

}

static void contactEnd(cpArbiter* arbiter, cpSpace* space, void* data)
{
    cpShape* shapeA;
    cpShape* shapeB;
    cpArbiterGetShapes(arbiter, &shapeA, &shapeB);

    CCSprite* spriteA = (CCSprite*)shapeA->data;
    CCSprite* spriteB = (CCSprite*)shapeB->data;
    if (spriteA != nil && spriteB != nil)
    {
        spriteA->color = ccWHITE;
        spriteB->color = ccWHITE;
    }
}

```

The `contactBegin` method should return YES if the collision should be processed normally. By returning NO or 0 from this method, you can also ignore collisions. To get to the sprites, you first have to get the shapes from the `cpArbiter`, which just like `b2Contact` holds the contact information. Via the `cpArbiterGetShapes` method and passing two shapes as out parameters, you get the colliding shapes from which you can then retrieve the individual `CCSprite` pointers. If they are both valid, their color can be changed.

As with Box2D, these callbacks don't get called by themselves. In the `HelloWorldScene` `init` method, right after the space is created, you must add the collision handlers using the `cpSpaceAddCollisionHandler` method:

```

unsigned int defaultCollisionType = 0;
cpSpaceAddCollisionHandler(space, defaultCollisionType, defaultCollisionType, ~
    &contactBegin, NULL, NULL, &contactEnd, NULL);

```

The default collision type for shapes is 0, and because I don't care about filtering collisions, both collision type parameters are set to 0. You can assign each body's shape an integer value to its `collision_type` property and then add collision handlers that are called only if bodies of matching collision types collide. This is called *filtering collisions* and is described in the Chipmunk manual, at <http://files.slembcke.net/chipmunk/release/ChipmunkLatest-Docs/#cpShape>.

The next four parameters are pointers to C callback methods for the four collision stages: *begin*, *pre-solve*, *post-solve*, and *separation* (the same as the `EndContact` event in Box2D). These serve the same purpose as the corresponding callbacks in Box2D. Most of the time you'll be interested only in the *begin* and *separation* events.

I pass NULL for *pre-solve* and *post-solve*, because I'm not interested in handling these. You can use these methods to influence the collision or to retrieve the collision force in the *post-solve* step. The final parameter is an arbitrary data pointer you can pass on to the callback methods if you need it. I don't, so I set it to NULL as well.

With that, you have a working collision callback mechanism.

Joints for Chipmunks

The Chipmunk example project also needs its own implementation of `addSomeJoinedBodies`. The setup is more verbose than for Box2D, as shown in Listing 12–14. You'll recognize most of the code as setting up static and dynamic bodies— if you find that code familiar, feel free to skip to the end where the joints are created.

Listing 12–14. *Creating Three Bodies Connected with Joints*

```
-(void) addSomeJoinedBodies:(CGPoint)pos
{
    float mass = 1.0f;
    float moment = cpMomentForBox(mass, TILESIZE, TILESIZE);

    float halfTileSize = TILESIZE * 0.5f;
    int numVertices = 4;
    CGPoint vertices[] =
    {
        CGPointMake(-halfTileSize, -halfTileSize),
        CGPointMake(-halfTileSize, halfTileSize),
        CGPointMake(halfTileSize, halfTileSize),
        CGPointMake(halfTileSize, -halfTileSize),
    };

    // Create a static body
    cpBody* staticBody = cpBodyNew(INFINITY, INFINITY);
    staticBody->p = pos;

    CGPoint offset = CGPointZero;
    cpShape* shape = cpPolyShapeNew(staticBody, numVertices, vertices, offset);
    cpSpaceAddStaticShape(space, shape);

    // Create three new dynamic bodies
    float posOffset = 1.4f;
    pos.x += TILESIZE * posOffset;
    cpBody* bodyA = cpBodyNew(mass, moment);
    bodyA->p = pos;
    cpSpaceAddBody(space, bodyA);

    shape = cpPolyShapeNew(bodyA, numVertices, vertices, offset);
    shape->data = [self addRandomSpriteAt:pos];
    cpSpaceAddShape(space, shape);

    pos.x += TILESIZE * posOffset;
    cpBody* bodyB = cpBodyNew(mass, moment);
    bodyB->p = pos;
    cpSpaceAddBody(space, bodyB);

    shape = cpPolyShapeNew(bodyB, numVertices, vertices, offset);
    shape->data = [self addRandomSpriteAt:pos];
    cpSpaceAddShape(space, shape);

    pos.x += TILESIZE * posOffset;
    cpBody* bodyC = cpBodyNew(mass, moment);
    bodyC->p = pos;
    cpSpaceAddBody(space, bodyC);
}
```



```

shape = cpPolyShapeNew(bodyC, numVertices, vertices, offset);
shape->data = [self addRandomSpriteAt:pos];
cpSpaceAddShape(space, shape);

// Create the joints and add the constraints to the space
cpConstraint* constraint1 = cpPivotJointNew(staticBody, bodyA, staticBody->p);
cpConstraint* constraint2 = cpPivotJointNew(bodyA, bodyB, bodyA->p);
cpConstraint* constraint3 = cpPivotJointNew(bodyB, bodyC, bodyB->p);

cpSpaceAddConstraint(space, constraint1);
cpSpaceAddConstraint(space, constraint2);
cpSpaceAddConstraint(space, constraint3);
}

```

In this example, I'm creating a pivot joint with `cpPivotJointNew`, which is the same as the `b2RevoluteJoint` used in the Box2D example. Each joint is created with the two bodies that should be connected to each other and one of the bodies' center position as the anchor point. The `cpPivotJointNew` method returns a `cpConstraint` pointer, which you'll have to add to the space using the `cpSpaceAddConstraint` method.

Summary

In this chapter, you learned the basics of the two physics engines distributed with cocos2d: Box2D and Chipmunk. You now have two working examples of these physics engines at your disposal, which should help you decide which one you'd like to use.

You learned how to set up a screen area that contains all the little boxes created from a tilemap as dynamic bodies. You now also know the basics of detecting collisions and how to create joints to connect bodies together in both physics engines.

In the next chapter, you'll be making a game that uses the Box2D physics engine.

Pinball Game

I'd like to put the Box2D physics engine to good use, so in this chapter you'll be making an actual pinball game. Pinball tables are all about using our physical world and turning that into a fun experience. With a physics engine, however, you're not just limited to real-world physics.

Some elements of the pinball table, such as bumpers and balls, can be created by simply choosing the right balance of friction, restitution, and density. Others need joints to work— a revolute joint for the flippers and a prismatic joint for the plunger. And of course, you'll need lots of static shapes that define the collision polygons of the table.

Since it would be impractical to define collision polygons in source code, at least for the level of detail necessary to build a believable pinball table, I'll introduce another useful tool: PhysicsEditor. With that, you can create collision polygons by simply drawing the vertices or, even faster, let PhysicsEditor trace the shape's outlines with a single mouse click.

At the end of this chapter, you'll have a fully playable pinball game, as depicted in Figure 13–1.

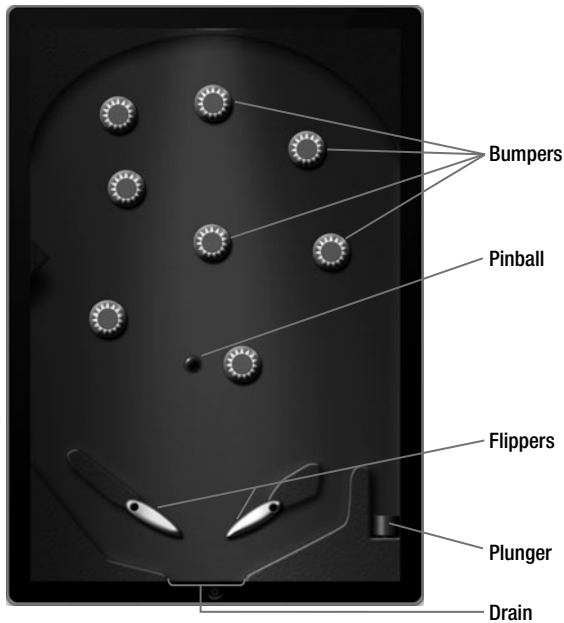


Figure 13–1. *The pinball table built in this chapter*

Shapes: Convex and Counterclockwise

Let's start with the requirements of collision polygons. The first thing you need to be aware of when defining collision polygons for Box2D and Chipmunk is that these engines expect the collision polygons to have the following properties:

- Vertices defined in a counterclockwise fashion
- Polygons as convex shapes

A *convex* shape is a shape where you can draw a straight line between any two points without the line ever leaving the shape. This is opposed to *concave* shapes, where a straight line between two points can be drawn such that the line is not entirely contained within the shape. See Figure 13–2 for an illustration of the difference between convex and concave shapes.

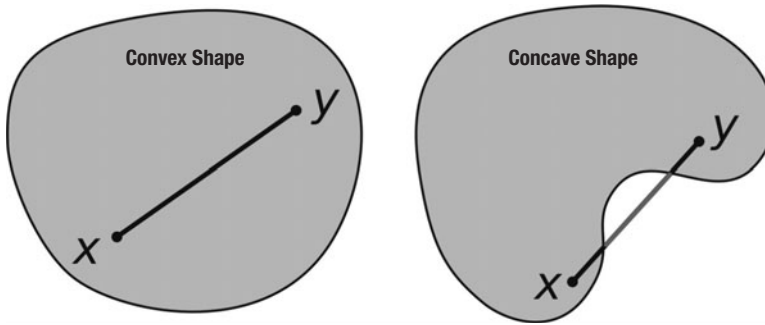


Figure 13–2. *Convex and concave shapes*

Defining the vertices of a convex shape in a counterclockwise fashion can be illustrated by drawing a convex shape in your mind. You place one vertex anywhere and then go left to place another. Then go down and to the right, and you will have drawn a rectangle in a counterclockwise fashion. Or place another vertex and then go right, up, and then left, and you will have drawn a counterclockwise shape. It doesn't matter where you start with the first vertex, but it's very important to follow the counterclockwise orientation of vertices.

Fortunately, if you're working with PhysicsEditor, you don't have to care about polygon vertex order (orientation) or whether the polygon is convex or concave. PhysicsEditor automatically takes care of that for you transparently. PhysicsEditor will split concave shapes into one or more convex shapes. The physic objects loader shipped with PhysicsEditor then assigns all the shapes to a single Box2D body. It's still good practice to try to avoid shapes to be split in order to have as few collision shapes per body as possible to get the best performance.

TIP: How do you know if you made a mistake and accidentally created a clockwise-oriented or concave collision shape? Well, every physics engine reacts differently. Some will tell you up front by throwing an error. But in the case of Box2D, if a moving body hits a collision shape that is not well formed, the moving body will simply stop moving when it gets close to that shape. If you ever see that effect happening in your Box2D game, check the nearby collision polygons.

Working with PhysicsEditor

Armed with the knowledge about properly defining collision polygons, it's time to check out the PhysicsEditor tool, which you can download from www.physicseditor.de. After opening the downloaded PhysicsEditor disk image and dragging PhysicsEditor.app to your application's folder, you are ready to run PhysicsEditor (see Figure 13–3). In the PhysicsEditor disk image, you will also find a folder named Loaders, which contains the loader code (shape cache) for Box2D and Chipmunk plist files created by PhysicsEditor. You will be using the GB2ShapeCache class in the example projects of this chapter in order to load the shapes created by PhysicsEditor.

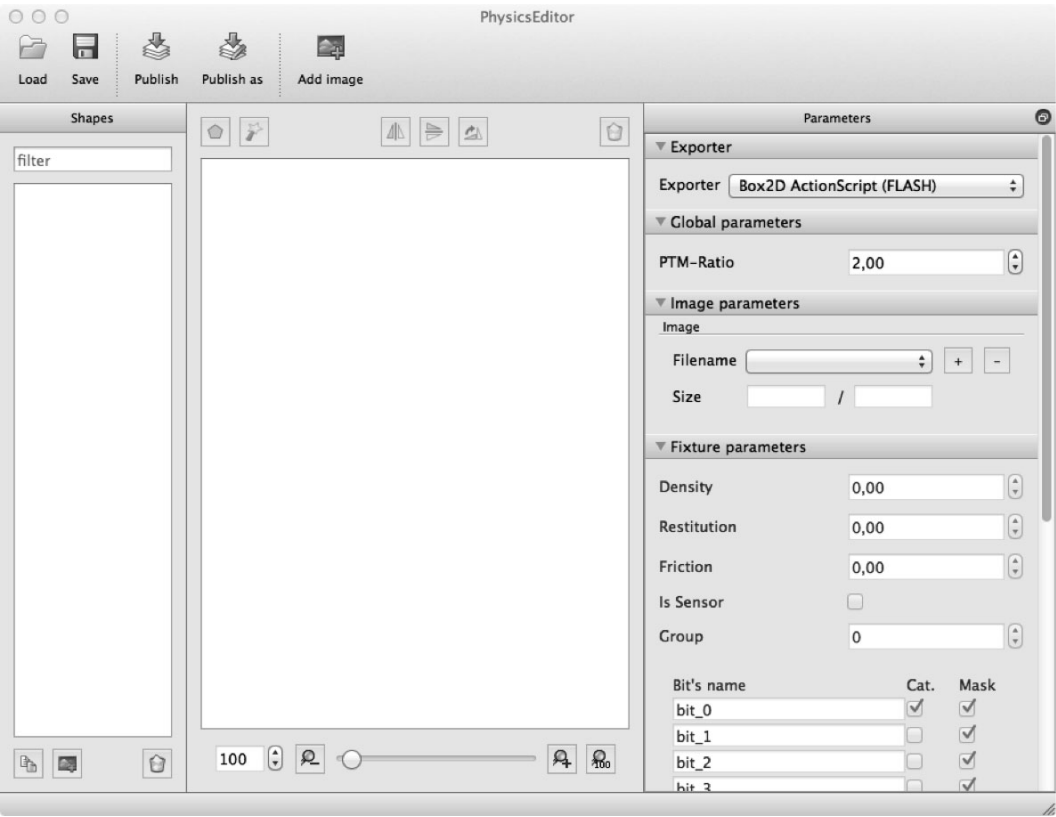


Figure 13–3. *The PhysicsEditor application*

You should now drag and drop the PNG files located in the PhysicsBox2D03 project’s Assets/pinball folder onto the leftmost pane in PhysicsEditor, which is labeled **Shapes**.

NOTE: You will be using only the HD resolution images to create physic shapes. It is not necessary to create separate HD and SD resolution physics shapes. The physics simulation world is independent from the graphical representation of the objects and thus independent of the screen resolution.

The first thing you should modify in PhysicsEditor are the settings for the exporter. PhysicsEditor can export to several game engines, supports both Box2D and Chipmunk physics engines, and even allows you to create your own custom export format. To write files compatible with cocos2d, you must set the **Exporter** setting on the rightmost pane to **Box2D generic (PLIST)**. It is important to set the exporter first since it enables or disables some features of the PhysicsEditor GUI depending on the targeted physics engine’s capabilities.

Next you should set the **PTM-Ratio** setting to 240. This value's unit is in pixels per meter, meaning 240 pixels will equal 1 meter in the Box2D physics simulation world. The dimensions of the Box2D physics world matters because Box2D is optimized to work best with objects of 1 to 10 meters in size. You can easily run a simulation with larger or smaller objects. However, Box2D loses precision and can show odd behavior when you have very large (tens or even hundreds of meters) or very small objects (small or tiny fractions of a meter).

Since we are using high-resolution images in PhysicsEditor, the **PTM-Ratio** of 240 will create a pinball table that is 4 meters high (960 Retina resolution pixels divided by 240) and 2.6 meters wide (640 Retina resolution pixels divided by 240). The actual pixels to meter ratio in cocos2d will be half the **PTM-Ratio** setting of PhysicsEditor, in this case 120 pixels per meter. This is because the cocos2d coordinate system is in points, which means both the standard-resolution display and Retina displays have the same dimension: 320x480 points. A point equals 1 pixel on standard displays and 2 pixels on Retina displays. The size of the table in the Box2D physics world is unaffected by the actual screen resolution. If you work only with standard resolution images and have Retina support disabled in your app, the **PTM-Ratio** setting in PhysicsEditor will equal that in Cocos2D.

Defining the Plunger Shape

Let's start with the plunger, the spring that kicks the ball into play. Select the plunger image on the leftmost pane and click the **Add polygon** button on the center view's toolbar. This creates a new triangle shape on the center working area and highlights it. Since we need a rectangular shape, double-click one of the sides to add a fourth vertex. In case you added too many vertices, you can right-click or Option-click a vertex and choose to **Delete point** to remove it.

You should move the four vertices to the four corners of the plunger by clicking and dragging them. Create a rectangular shape that encompasses the entire plunger, including the spring. Make the rectangle cover the complete plunger, including the spring. This will avoid problems when the ball by accident falls below the plunger's head.

You may have noticed the little bluish circle with the + inside. This is the anchor point of the shape, which coincides with the anchor point of the shape's sprite. Later, when we position the shape in cocos2d, the shape's anchor point will be centered on the coordinates that we provide.

You will want to move the anchor point to the bottom center of the plunger image to make it easier to position the plunger. You can drag the blue circle, but in many cases, you will want to place it very accurately. In those cases, you can modify the anchor point's absolute pixel position or the relative position in the **Parameters** pane under **Image Parameters**.

In Figure 13–4, you'll see the plunger shape being edited.

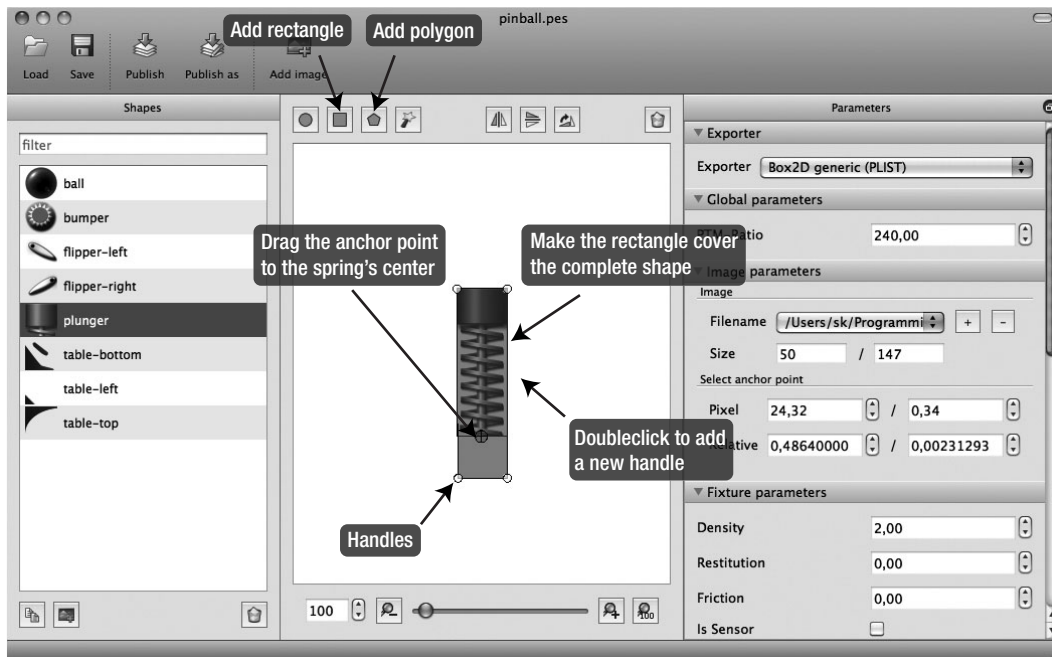


Figure 13–4. *Manually defining a shape in PhysicsEditor*

Instead of creating the shape from a polygon, you could have also used the **Add Rectangle** button. But then I wouldn't have been able to tell you how to add or delete vertices and how to drag them.

You can and should also set the collision bits for the plunger under the **Fixture parameters** section in the **Parameters** pane. The collision bits allow you to define which shapes collide and which don't. The collision bit settings will be used to prevent the plunger from colliding with any shapes but the ball.

To make working with collision bits easier, you can change the names of collision bits. The collision bit names are not exported; they serve only to remind you what each bit is used for. By default, the bits are called `bit_0` to `bit_15`. You will want to change the names of the first five bits to `Ball`, `Bumper`, `Flipper`, `Plunger`, and `Wall`, respectively.

Box2D shapes collide with each other only when the category bit (the check box column labeled **Cat.**) and the mask bit (the check box column labeled **Mask**) of both shapes are set. Typically you will want to assign each shape to just one particular category. In the case of the plunger, make sure that only the category bit for the `Plunger` category is set. In other words, you're assigning the plunger's shape to be in the `Plunger` collision bit category. With the **Mask** check box, you then set with which other categories this shape is allowed to collide with. In the case of the plunger, you should set only the **Mask** check box for the `Ball` category, allowing the plunger to collide with the ball. Figure 13–5 shows the correct settings for the `Plunger` collision category and mask flags.

NOTE: So far, the plunger would collide with the ball, but the ball would not collide with the plunger. You'll have to keep in mind that defining collisions is a two-way process, and in this case, you'll still have to put the ball in the Ball category and check the ball's **Mask** bit that corresponds with the Plunger category to have both ball and plunger collide with each other.

You can also set the mask bit for the same category, allowing multiple objects of the same category to collide with each other. In the case of the ball shape, it would make sense to set the **Mask** bit of the Ball category in order to allow ball-to-ball collisions. This will be useful if you want to extend the pinball game to support a multiple balls on the table at the same time.

Bit's name	Cat.	Mask
Ball	<input type="checkbox"/>	<input checked="" type="checkbox"/>
Bumper	<input type="checkbox"/>	<input type="checkbox"/>
Flipper	<input type="checkbox"/>	<input type="checkbox"/>
Plunger	<input checked="" type="checkbox"/>	<input type="checkbox"/>
Wall	<input type="checkbox"/>	<input type="checkbox"/>
bit_5	<input type="checkbox"/>	<input type="checkbox"/>
bit_6	<input type="checkbox"/>	<input type="checkbox"/>
bit_7	<input type="checkbox"/>	<input type="checkbox"/>
bit_8	<input type="checkbox"/>	<input type="checkbox"/>
bit_9	<input type="checkbox"/>	<input type="checkbox"/>
bit_10	<input type="checkbox"/>	<input type="checkbox"/>
bit_11	<input type="checkbox"/>	<input type="checkbox"/>
bit_12	<input type="checkbox"/>	<input type="checkbox"/>
bit_13	<input type="checkbox"/>	<input type="checkbox"/>
bit_14	<input type="checkbox"/>	<input type="checkbox"/>
bit_15	<input type="checkbox"/>	<input type="checkbox"/>
value (hex)	0008	0001
Calculated value	All	All
	None	None
	Inv.	Inv.

Figure 13–5. Collision parameters for the plunger

Below the **Cat.** and **Mask** columns, you'll find the buttons **All**, **None**, and **Inv.**, which allow you to check all, uncheck all, or invert the check boxes' checked status. They are helpful to avoid clicking possibly dozens of check boxes one after another.

Defining the Table Shapes

The pinball table consists of three separate shapes named *table-bottom*, *table-left*, and *table-top*. The table background image is split up to make it easier to edit its shapes and to make it easier to create different pinball layouts without having to replace the entire image.

Select the table-top image in the Shapes pane to start editing the shape for the topmost part of the pinball table. As you can see in Figure 13–6, it's a concave shape. With the manual method used to define the plunger, it would be difficult and error-prone to create all the vertices of this round shape manually, let alone ensuring that the resulting shape is convex. PhysicsEditor makes this a lot easier for you: it will trace the shape's outline and create a suitable shape with a single mouse click!

Above the center pane there's a magic wand icon in the toolbar called the **Shape Tracer**. Click the wand icon to open the Shape Tracer dialog shown in Figure 13–6.

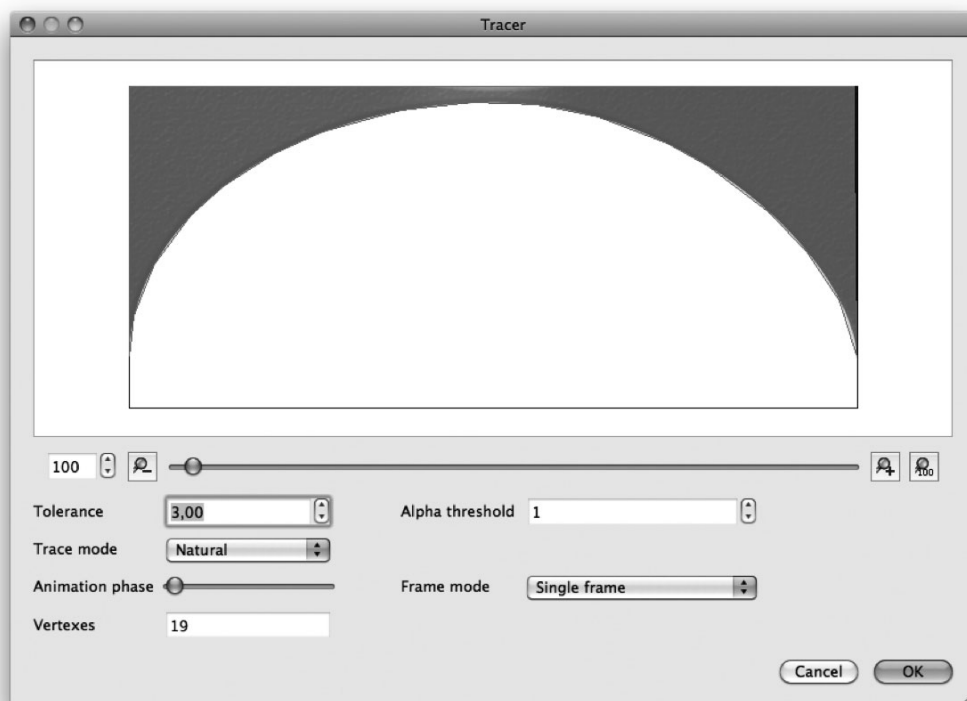


Figure 13–6. The Shape Tracer creates shapes automatically.

The Shape Tracer shows the shape's image and an overlay of the shape it is going to create when you click the **OK** button. Below the image you'll find a slider and buttons to

the left and right of the slider that control the zoom level of the image. The image zoom settings do not affect how the shape is created.

The most important setting you will want to adjust in the Shape Tracer is the **Tolerance** setting. **Tolerance** changes how accurately the image is traced to create a shape, which directly influences the number of vertices used for the shape. The number of vertices of a shape in turn influence the performance of the physics simulation. Generally, you should always strive to achieve adequate collision responses with the least amount of vertices. The more important the accurate collisions are for your game, the more vertices you will want to allow for some objects. At the same time, if you add many objects using the same shape, using a shape with fewer vertices will result in better performance.

By experimenting with the **Tolerance** setting, I found that a good compromise in this case is a **Tolerance** value of 4,0. This creates a shape with 18 vertices, and it's the highest **Tolerance** value that still traces the image's shape quite accurately. The default **Tolerance** setting of 1,0 would have created a shape with 31 vertices, so I was able to save 13 vertices. But you'll notice if you cycle through the **Tolerance** values that even a **Tolerance** value of only 1,5 will already cut down the number of vertices to 20.

TIP: The Frame Mode setting in the Shape Tracer can be used to create a shape for an animation (sequence of images). To create an animation in PhysicsEditor, you'll have to add more image files to a shape under Image Parameters in the Parameters pane, in the default PhysicsEditor window. By clicking the + button next to the Filename setting, you can add additional images to a single shape; then you can have the shape trace create a shape that is either the intersection or the union of each animation frame's shape.

When you're satisfied with the traced shape, click the OK button to close the Shape Tracer dialog. This will create the new shape. You still have to perform a quick manual tweak for the pinball table's collisions to work smoothly. Since the screen area will define the collision on the sides of the pinball table, you should drag the lower-left and lower-right corner vertices as well as the upper-left and upper-right corner vertices slightly outside the screen area and downward and upward, respectively, so that they are clearly outside the black frame border drawn around the table-top image. See Figure 13-7 for a visual hint.

By extruding these two vertices, the shape will form a soft transition to the sides of the screen borders. Without extruding these vertices, the ball might get reflected from the tips of these vertices because of slight inaccuracies that are always present in physics simulations.

Now move the anchor point (the blue circle with the + inside) to the top-left corner, preferably by using the anchor point settings in the **Parameters** pane under **Image Parameters**. Set the **Relative** values to 0,0 and 1,0, respectively, to move the anchor point to the top-left corner. This anchor point position will later allow you to align the image exactly with the screen border by simply positioning it to 0,480 in point coordinates.

NOTE: If you do not see the anchor point circle and there are no anchor point settings under **Image Parameters**, you do not have the **Exporter** setting in the **Parameters** pane set to the **Box2D generic (PLIST)** format.

The last step is to adjust the collision bits of the table-top shape. Check the category (**Cat.**) check box of the **Wall** row and then set the **Mask** check box in the **Ball** row. This will enable collisions of the pinball table with the ball. Accordingly, you'll have to set the same collision bits for the table-left and table-bottom shapes because they are all part of the **Wall** category and should collide with the **Ball** category.

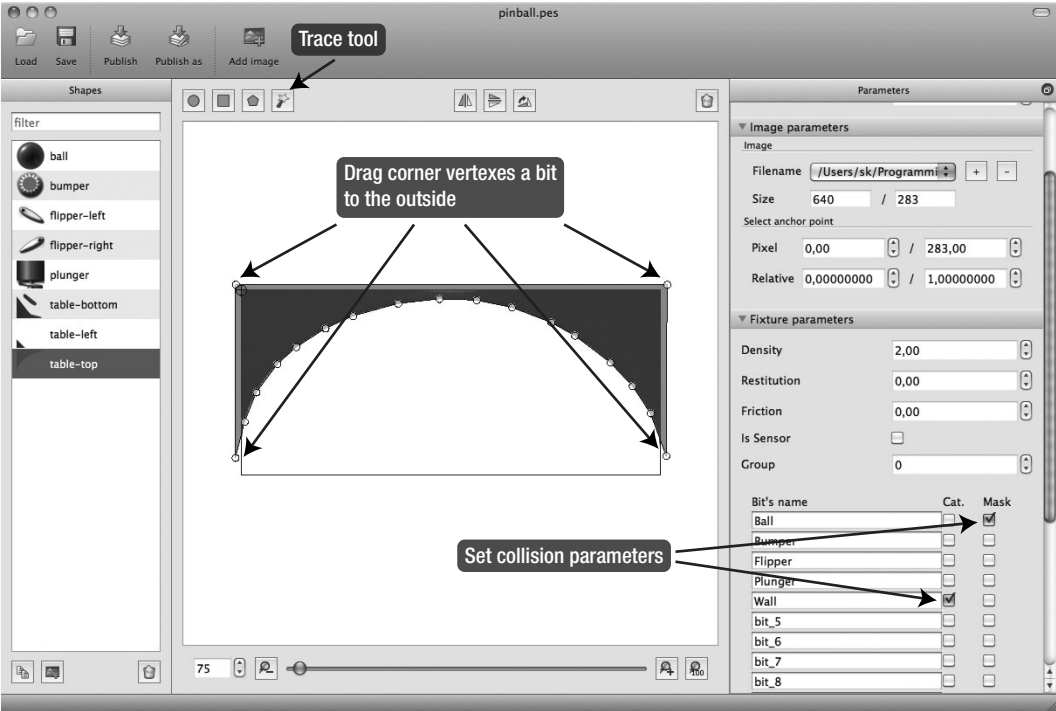


Figure 13–7. Finalizing the table's top shape

Use the Shape Tracer to create the shape for the table-left image in the same way. Under **Image Parameters**, move the anchor point to **Pixel** coordinates 0,0 and 50,0. Don't forget to set the collision bits just like earlier for table-top.

Now let's trace the shape for the table-bottom image. This requires a few additional steps because the table-bottom image actually consists of four individual, disjointed elements that need to have individual shapes. The version 1.0.4 of PhysicsEditor traces only contiguous shapes so you need to open the Shape Tracer a total of four times. Each time you open the Shape Tracer, click the part of the image that you want to trace and then click the **OK** button to create the shape. A **Tolerance** setting of 4,0 works well for all four shapes. You should end up with four shapes covering all the four elements of the

table-bottom image. Don't forget to set the collision bits exactly like you did for table-top, as shown in Figure 13-7.

Defining the Flippers

Select the image named flipper-left, and open the Shape Tracer. You will notice that the resulting shape initially suggested by the Shape Tracer does not make much sense; the shape is a lot larger than the image would suggest.

This is because the image has glow and shadow effects that create an aura around the shape itself and that is nearly invisible on a light background. To enable the Shape Tracer to create a better shape, you need to adjust the **Alpha threshold** setting. The default value is 0, which means that all pixels that are not entirely transparent will be considered when the shape is traced. In this case, we want only the fully opaque pixels to be considered for the shape. If you set the **Alpha threshold** to 254, as in Figure 13-8, you'll get a much better result.

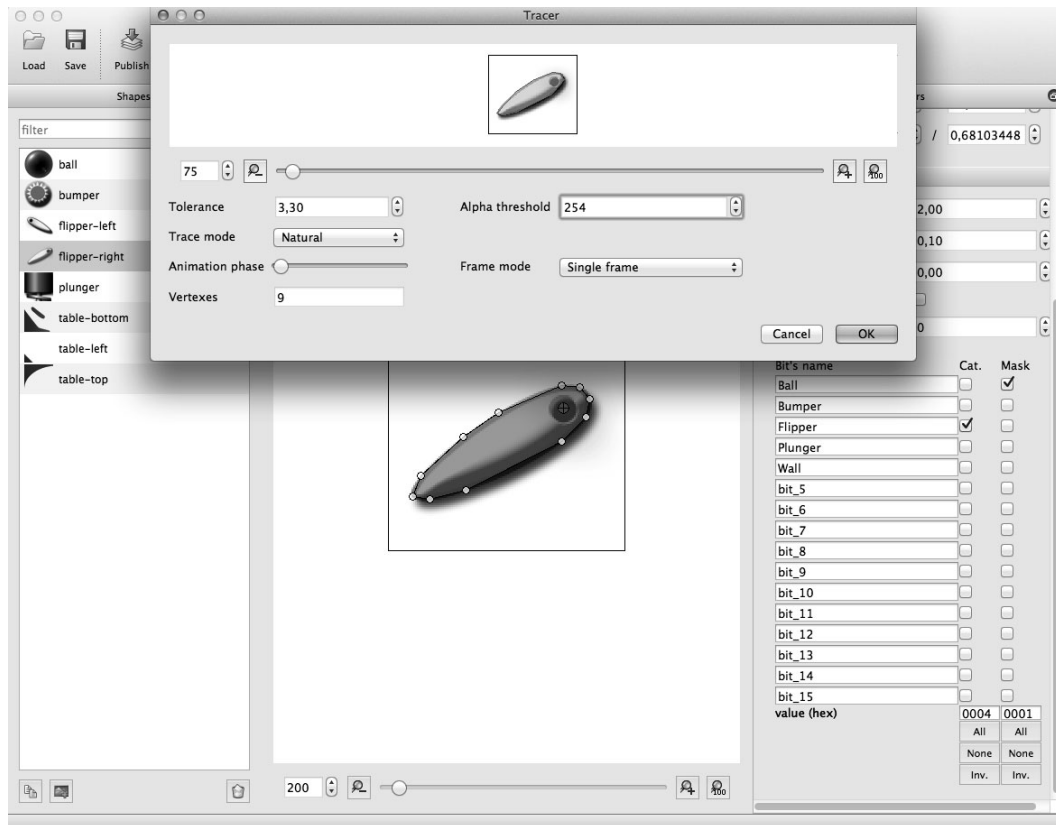


Figure 13-8. Tracing the shape of a flipper with an alpha threshold to ignore the image shadow.

TIP: If for any reason the Shape Tracer result is not what you want and neither the **Tolerance** nor the **Alpha threshold** setting allows you to fix the shape, you can still manually edit the shape after closing the Shape Tracer. Just click any vertex and drag it. You can also double-click a vertex to remove it or double-click a line segment between two vertices to add a new vertex.

Repeat the same process for the flipper-right image. And don't forget to set the collision bits for both flippers. You will want to check the **Flipper** category and the **Mask** check box in the **Ball** row. Figure 13-8 shows the correct collision bit settings.

The anchor point of the left flipper should be set to **Pixel** coordinates 27,78 and 97,79 for the right flipper.

Defining the Bumper and Ball

The ball and bumper images are both circles, so you can use the **Add Circle** command from the toolbar to create a circle shape. The circle shape has only one **vertex** that acts as a handle to resize the circle shape.

Starting with the bumper, change the circle shape's size and position so that it overlaps the solid part of the bumper while ignoring the bumper's shadow. Set the bumper's anchor point to **Pixel** coordinates 33, 42 so that the anchor point is centered on the bumper. Set the collision bits category check box next to **Bumper**, and set the **Mask** check box of the **Ball** row.

To simulate the bumper's bounce effect, you should also modify the **Fixture parameter** in the **Parameters** pane labeled **Restitution**. Restitution is the amount of elasticity with which an object repels other colliding objects. A restitution of 0 means that the collision is not elastic at all and will stop the incoming object. A restitution of 1 simulates a perfect elastic collision, allowing the colliding object to continue with the same speed after colliding.

Since we want to simulate an additional repelling force coming from the bumper, we can simply use a value of 1,5 or higher in order to have the colliding object move away from the collision at a higher speed than the speed it had when it impacted. In real life, this would be a violation of Newtonian physics and is in fact impossible. So, don't tell your physics teacher!

Lastly, create the circle shape for the ball image. Size and position the circle shape so that it overlays the entire ball image. You should set the anchor point **Relative** values to 0,5 and 0,5.

As for the ball's collision bits, you should set the category check box next to **Ball** and check all the **Mask** check boxes for the other categories: **Ball**, **Bumper**, **Flipper**, **Plunger**, and **Wall**. You can also simply click the **All** button at the bottom of the **Mask** column. Since we don't use the other bits, it doesn't matter if they are checked, too.

Some of the ball's Fixture Parameters affect how the ball behaves on the table. I set the **Density** to 8,0, **Restitution** to 0,3 and **Friction** to 0,7. These settings give the ball just a little bit of bounce. Feel free to tweak these settings and observe how the ball's behavior changes.

Save and Publish

Lastly you should save the current PhysicsEditor .pes file. You can later open the .pes file to continue editing shapes. You should not add the .pes file to your Xcode project; it is used only by PhysicsEditor.

To use the shapes in cocos2d, you will have to use the **Publish** button. This creates a .plist file that the GB2ShapeCache class distributed with PhysicsEditor can read. For the PhysicsBox2D03 project, I published the shapes as pinball-shapes.plist in the Resources folder of the project. Add this file to the Resources group in Xcode.

Programming the Pinball Game

Now we can move on to the implementation phase and actually program the pinball game. You'll learn how to lay out the table before moving on to the interactive elements like the ball, plunger, bumpers, and flippers. But before we get to that, I'd like to introduce you to the essential BodyNode class, which synchronizes a cocos2d sprite with a Box2D body.

CAUTION: Keep in mind that the pinball project uses Box2D, which is written in C++. This requires us to use the .mm file extension instead of .m for all class implementation files so that the compiler correctly switches to compiling C++ code instead of C code. Whenever you create a new Objective-C class, you will have to rename the implementation file so that it uses the .mm file extension. Otherwise, you'll see a lot of compile errors seemingly caused by the Box2D source code.

The BodyNode Class

The idea behind the BodyNode class is that you want to use a self-contained object for all of your dynamic classes. So far, we simply added the CCSprite to a body's userData field. But suppose you want to actually interact with the class represented by that sprite—for example, during one of the ContactListener methods. You couldn't, because all you had access to was the CCSprite object. To solve this problem, I created the class BodyNode in the PhysicsBox2D03 project.

BodyNode is derived from CCSprite and contains a Box2D body as an instance variable. The reference to the body allows for convenient access to it by any class that derives from BodyNode. With all classes for the pinball game elements being derived from the

BodyNode class, you have a common class to work with, which you can then further probe for its type. For example, you can use the `isKindOfClass` method to determine at runtime from any `BodyNode*` pointer which class you are working with. The `isKindOfClass` method is supported by all classes that derive from `NSObject`.

In addition, the `BodyNode` class header file includes commonly used headers such as `Box2d.h` and `Helper.h` (see Listing 13–1).

Listing 13–1. The `BodyNode` Header File

```
#import <Foundation/Foundation.h>
#import "cocos2d.h"
#import "Helper.h"
#import "Constants.h"
#import "PinballTable.h"
#import "Box2D.h"
#import "b2Body.h"

@interface BodyNode : CCSprite
{
    b2Body* body;
}

@property (readonly, nonatomic) b2Body* body;

/**
 * Creates a new shape
 * @param shapeName: Name of the shape and sprite
 * @param inWorld: Pointer to the world object to add the sprite to
 * @return BodyNode object
 */
-(id) initWithShape:(NSString*)shapeName inWorld:(b2World*)world;

/**
 * Changes the body's shape
 * Removes the fixtures of the body replacing them
 * with the new ones
 * @param shapeName name of the shape to set
 */
-(void) setBodyShape:(NSString*)shapeName;

@end
```

The `BodyNode` provides a property for its instance variable `body` so that it can be conveniently accessed. The initializer method `initWithShape` is used to initialize both the body and the sprite by using the supplied shape name. It assumes that both the name of the image and the name of the shape are identical.

CAUTION: By default TexturePacker will retain the file extension of images so that your shape might be named `plunger`, but the sprite frame name might be `plunger.png`. To fix that, you'll have to check the **Trim sprite names** check box in the TexturePacker **Output** pane. The `pinball.tps` file used in this chapter has this check box already set; it removes the `.png` extension from the sprite frame names. That way, both the TexturePacker sprite frames and PhysicsEditor shape names will be identical.

For the texture atlas, you use TexturePacker and add the `pinball` folder as a smart folder reference, meaning that TexturePacker will keep the texture atlas contents up to date with all changes to the images in the `pinball` folder. Refer to Chapter 6 for instructions on how to set up a smart folder reference.

You also need to set an additional option we did not yet use: `trim sprite names`. This feature removes the `.png` suffix from the sprite names. The advantage is that we can use the same names for the physics shapes and the sprites. The only caveat is that you'll have to refer to sprite frame names without the `.png` extension, which admittedly out of habit is sometimes easy to forget.

Listing 13–2 shows the `BodyNode` class implementation file.

Listing 13–2. *The `BodyNode` Class Implementation*

```
#import "BodyNode.h"

@implementation BodyNode

@synthesize body;

-(id) initWithShape:(NSString*)shapeName inWorld:(b2World*)world
{
    NSAssert(world != NULL, @"world is null!");
    NSAssert(shapeName != nil, @"name is nil!");

    // init the sprite itself with the given shape name
    self = [super initWithSpriteFrameName:shapeName];
    if (self)
    {
        // create the body
        b2BodyDef bodyDef;
        body = world->CreateBody(&bodyDef);
        body->SetUserData(self);

        // set the shape
        [self setBodyShape:shapeName];
    }
    return self;
}

-(void) setBodyShape:(NSString*)shapeName
{
    // remove any existing fixtures from the body
    b2Fixture* fixture;
```



```

while ((fixture = body->GetFixtureList()))
{
    body->DestroyFixture(fixture);
}

// attach a new shape from the shape cache
if (shapeName)
{
    GB2ShapeCache* shapeCache = [GB2ShapeCache sharedShapeCache];
    [shapeCache addFixturesToBody:body forShapeName:shapeName];

    // Assign the shape's anchorPoint (the blue + in a circle in PhysicsEditor)
    // as the BodyNode's anchorPoint. Otherwise image and shape would be offset.
    self.anchorPoint = [shapeCache anchorPointForShape:shapeName];
}
}

-(void) dealloc
{
    // remove the body from the world
    body->GetWorld()->DestroyBody(body);

    [super dealloc];
}

@end

```

The `BodyNode` implementation initializes the sprite with the `initWithSpriteFrameName` method. Then it calls the world's `CreateBody` method and sets `self` as the user data pointer of the body, allowing you to later access the `BodyNode` class in the `Box2D` collision callback methods.

TIP: The `CCNode` class also has a `userData` property, which you can use in the same way as `b2Body`'s `userData` field.

The `PhysicsEditor` `GB2ShapeCache` class is used to add the fixtures to the body using the provided `shapeName`. The `PhysicsBox2D03` project already includes the necessary files. If you want to make use of the `GB2ShapeCache` class in a new project, just remember to add the `GB2ShapeCache.h` and `GB2ShapeCache.mm` files from the `PhysicsEditor.dmg` disk image folder `/Loaders/generic-box2d-plist` to your Xcode project.

The key point to take away here is actually that the `BodyNode` is a `CCSprite` that manages the allocation, deallocation, and configuration of the `Box2D` body for you. It also ensures that you'll be able to access the `BodyNode` class and thus the `cocos2d` sprite wherever you normally have access only to the `Box2D` body object. Thus, the `BodyNode` class becomes the glue that brings the physics body together with the `cocos2d` sprite.

If a sprite derived from `BodyNode` goes out of scope—for example, if you remove it as a child from its `cocos2d` parent node—then `BodyNode` will take care of destroying the `Box2D` body for you.

In addition, you are able to change the sprite's body shape by calling `setBodyShape`. This removes any existing fixtures from the body and then adds the fixtures associated with the given `shapeName` from the `GB2ShapeCache`. This will change only the collision shape of the body and preserve the body's current state of motion, like its velocity, position, and rotation. Keep in mind that calling this method is time-consuming and should be done only as needed. It's certainly not a good idea to change the body shape every frame.

All classes inheriting from `BodyNode` now have to concern themselves only with setting the correct shape name and any code that is unique to the class.

The update method in the `PinballTable` class has also been rewritten to account for the body's user data change from a simple `CCSprite` class instance to a `BodyNode` class instance, since the `body->GetUserData()` method will now return a `BodyNode` object (see Listing 13-3).

Listing 13-3. *The Modified Update Method of the PinballTable Class*

```
-(void) update:(ccTime)delta
{
    int32 velocityIterations = 8;
    int32 positionIterations = 1;
    world->Step(delta, velocityIterations, positionIterations);

    // for each body, get its assigned BodyNode and update the sprite's position
    for (b2Body* body = world->GetBodyList(); body != nil; body = body->GetNext())
    {
        BodyNode* bodyNode = (BodyNode*)body->GetUserData();
        if (bodyNode != nil)
        {
            // update the sprite's position to where their physics bodies are
            bodyNode.position = [Helper toPixels:body->GetPosition()];
            float angle = body->GetAngle();
            bodyNode.rotation = -(CC_RADIANS_TO_DEGREES(angle));
        }
    }
}
```

The `world->Step()` method call has also been changed. The update method's `delta` parameter is now used in place of the previously fixed time interval in order to make the simulation framerate independent. This change will allow the physics simulation to run at the same speed even if the framerate drops.

The only drawback to updating the physics world independently of the framerate is that if there's a one-time but relatively long (more than a tenth of a second) interruption during game play—possibly caused by a background task—the physics objects may seem to jump or warp to a new location from one instant to another. If you see this behavior in your game, you will want to cap the `delta` value to a safe upper limit in order to advance the physics world no faster than the given constant value:

```
float cappedDelta = fminf(delta, 0.08f);
world->Step(cappedDelta, velocityIterations, positionIterations);
```

The effect of that code is that the physics simulation will be updated independently from the framerate, but if the framerate drops too low, the physic simulation will update at a constant speed.

Creating the Pinball Table

The pinball's table is made up of three individual images and associated shapes, named *table-top*, *table-left*, and *table-bottom*. You'll create the table using the `TablePart` class, which inherits from `BodyNode`. The `TablePart` header only adds the static initializer method `tablePartInWorld`:

```
#import "BodyNode.h"

@interface TablePart : BodyNode
{
}

+(id) tablePartInWorld:(b2World*)world position:(CGPoint)pos name:(NSString*)name;

@end
```

The `TablePart` implementation in Listing 13–4 is also rather unimpressive since the only purpose is to initialize the `BodyNode` and then set body's position to the desired location. This will also update the sprite's position automatically the next time the update method of the `PinballTable` class is executed (refer to Listing 13–3).

The `TablePart` class also sets the body type to `b2_staticBody`, which makes each `TablePart` a nonmoving object. This has two advantages, one being that `Box2D` does not need to perform certain calculations on static objects and the other is simply that objects colliding with a static body will not affect a static body's position or rotation at all.

Listing 13–4. *TablePart* Class Implementation

```
@implementation TablePart

-(id) initWithWorld:(b2World*)world position:(CGPoint)pos name:(NSString*)name
{
    if ((self = [super initWithShape:name inWorld:world]))
    {
        // set the body position
        body->SetTransform([Helper toMeters:pos], 0.0f);

        // make the body static
        body->SetType(b2_staticBody);
    }
    return self;
}

+(id) tablePartInWorld:(b2World*)world position:(CGPoint)pos name:(NSString*)name
{
    return [[[self alloc] initWithWorld:world position:pos name:name] autorelease];
}

@end
```

To create the three required `TablePart` instances (and later other pinball elements), I've created the `TableSetup` class, which creates the various `BodyNode` instances that will make up the bodies of the pinball table. `TableSetup` inherits from `CCSpriteBatchNode` to improve the rendering performance of the pinball table's elements.

This is the header file of the `TableSetup` class:

```
#import <Foundation/Foundation.h>
#import "cocos2d.h"
#import "Box2D.h"

@interface TableSetup : CCSpriteBatchNode
{
}

+(id) setupTableWithWorld:(b2World*)world;

@end
```

And since that's so unspectacular, let's turn our attention to the implementation of the `TableSetup` class of the `PhysicsBox2D03` project, which is shown in Listing 13–5.

Listing 13–5. *TableSetup* Class Implementation

```
-(id) initWithWorld:(b2World*)world
{
    if ((self = [super initWithFile:@"pinball.pvr.ccz" capacity:5]))
    {
        // add the table blocks
        [self addChild:[TablePart tablePartInWorld:world
                                position:ccp(0, 480)
                                name:@"table-top"]];
        [self addChild:[TablePart tablePartInWorld:world
                                position:ccp(0, 0)
                                name:@"table-bottom"]];
        [self addChild:[TablePart tablePartInWorld:world
                                position:ccp(0, 263)
                                name:@"table-left"]];
    }

    return self;
}

+(id) setupTableWithWorld:(b2World*)world
{
    return [[[self alloc] initWithWorld:world] autorelease];
}
```

As of now, the job of the `TableSetup` class is to create the three `TablePart` classes that make up the static background elements of the pinball table. It also sets the correct position of each `TablePart` instance, which is influenced by their shape's `anchorPoint`. For example, the `table-top` image has its shape `anchorPoint` set to the upper-left corner of the image so that positioning it at (0, 480) aligns the image and shape correctly at the top border of the screen. Since `TablePart` inherits from `BodyNode`, which inherits from `CCSprite`, it is legal and quite convenient to add these classes directly to the `TableSetup` class, which inherits from `CCSpriteBatchNode`.

You'll later extend the `TableSetup` class to add the other pinball table elements, like plunger, ball, bumpers, and flippers.

TIP: Did you notice the use of the `ccp` method? It's exactly the same as the `CGPointMake` method but simply shorter to type. Some cocos2d developers prefer to use `ccp` over `CGPointMake` simply because it's shorter to type. You'll find the entire `ccp` line of helpful math functions defined in `CGPointExtension.h`. They tend to come in handy particularly when you're developing physics games. Box2D also brings its own math functions that are defined in `b2Math.h`. The reason is that Box2D's vector classes like `b2Vec2` are C++ classes, and not C structs like `CGPoint`, meaning the Box2D data structures can generally not be used with the `ccp` methods.

The `TableSetup` class itself is initialized by the `PinballTable` class, which is based on the `HelloWorldScene` class of the Box2D project from Chapter 12. Listing 13–6 shows the interface of the `PinballTable` class.

Listing 13–6. *PinballTable Class Header File*

```
#import "cocos2d.h"
#import "Box2D.h"
#import "GL ES-Render.h"

#import "ContactListener.h"

@interface PinballTable : CCLayer
{
    b2World* world;
    ContactListener* contactListener;

    GLESDebugDraw* debugDraw;
}

+(id) scene;

@end
```

It contains references to the `ContactListener` and `GLESDebugDraw` classes. The latter I'll get to shortly; the `ContactListener` class will play a role when you're adding the pinball game's plunger. For now, let's look at the initialization of the `PinballTable` class in Listing 13–7.

Listing 13–7. *Initialization of the PinballTable Class*

```
-(id) init
{
    if ((self = [super init]))
    {
        // pre load the sprite frames from the texture atlas
        [[CCSpriteFrameCache sharedSpriteFrameCache] ←
            addSpriteFramesWithFile:@"pinball.plist"];

        // load physics definitions
```

```

[[GB2ShapeCache sharedShapeCache] addShapesWithFile:@"pinball-shapes.plist"];

// init the box2d world
[self initBox2dWorld];

// debug drawing
[self enableBox2dDebugDrawing];

// load the background from the texture atlas
CCSprite* background = [CCSprite spriteWithSpriteFrameName:@"background"];
background.anchorPoint = ccp(0,0);
background.position = ccp(0,0);
[self addChild:background z:-3];

// Set up table elements
TableSetup* tableSetup = [TableSetup setupTableWithWorld:world];
[self addChild:tableSetup z:-1];

[self scheduleUpdate];
}
return self;
}

```

The `CCSpriteFrameCache` loads the sprite frames from the texture atlas created with `TexturePacker` by loading the `pinball.plist` file.

More importantly this is followed by loading the `pinball-shapes.plist` file into the `GB2ShapeCache` class. It is important to do this first before calling any other `Box2D` method so that the shape cache's `ptmRatio` is correctly set to the value from the `PhysicsEditor` setting **PTM-Ratio**. If you recall, **PTM-Ratio** was one of the first `PhysicsEditor` setting we modified at the beginning of this chapter.

The pixel-to-meters macro `PTM_RATIO` used in Chapter 12 has been changed from a simple constant value to the following definition that you'll find in the `Constants.h` header file:

```
#define PTM_RATIO ([[GB2ShapeCache sharedShapeCache] ptmRatio] * 0.5f)
```

This improved `PTM_RATIO` macro now retrieves the points-to-meter ratio from the `GB2ShapeCache` class. And that's why you have to make sure the shape cache is initialized before using the `PTM_RATIO` macro.

The shape cache's `ptmRatio` is divided by two (multiplied by `0.5f`) because the shapes created in `PhysicsEditor` were based on the Retina resolution images, while `cocos2d's` resolution-independent point coordinates used for positioning nodes always assume the iPhone's screen resolution to be 480x320 points.

The initialization of the `Box2D` physics engine is moved to a separate `initBox2dWorld` method. The `Box2D` init code is essentially the same as in the previous chapter, with two exceptions: the static screen boundary shape that keeps dynamic objects inside the screen area defines no bottom shape, allowing dynamic bodies to fall outside the screen through the bottom. We'll need that for the ball to be able to roll into the table's drain.

In addition to that, the collision parameters for the left and right boundary are set in the init code, since these shapes aren't defined with the help of `PhysicsEditor`. The

categoryBits must be set to the Wall bit (collision bit 4 in PhysicsEditor), and the maskBits must be set to the Ball bit (collision bit 0 in PhysicsEditor). The bit values are specified in hexadecimal format, indicated by the leading 0x. You'll find that PhysicsEditor will display those hexadecimal values below the **Cat.** and **Mask** columns under **Fixture Parameters**.

The following is the `initBox2dWorld` method with the changes and additions compared to the Box2D initialization code from Chapter 12 highlighted:

```
-(void) initBox2dWorld
{
    // Construct a world object, which will hold and simulate the rigid bodies.
    b2Vec2 gravity = b2Vec2(0.0f, -5.0f);
    bool allowBodiesToSleep = true;
    world = new b2World(gravity, allowBodiesToSleep);

    contactListener = new ContactListener();
    world->SetContactListener(contactListener);

    // Define the collisions at screen borders.
    b2BodyDef containerBodyDef;
    b2Body* containerBody = world->CreateBody(&containerBodyDef);

    // for the ground body we'll need these values
    CGSize screenSize = [CCDirector sharedDirector].winSize;
    float widthInMeters = screenSize.width / PTM_RATIO;
    float heightInMeters = screenSize.height / PTM_RATIO;
    b2Vec2 lowerLeftCorner = b2Vec2(0, 0);
    b2Vec2 lowerRightCorner = b2Vec2(widthInMeters, 0);
    b2Vec2 upperLeftCorner = b2Vec2(0, heightInMeters);
    b2Vec2 upperRightCorner = b2Vec2(widthInMeters, heightInMeters);

    // Create the screen box' sides by using a polygon assigning each side individually.
    b2PolygonShape screenBoxShape;
    float density = 0.0;

    // We now only need the sides for the table:
    // left side
    screenBoxShape.SetAsEdge(upperLeftCorner, lowerLeftCorner);
    b2Fixture *left = containerBody->CreateFixture(&screenBoxShape, density);

    // right side
    screenBoxShape.SetAsEdge(upperRightCorner, lowerRightCorner);
    b2Fixture *right = containerBody->CreateFixture(&screenBoxShape, density);

    // set the collision flags: category and mask
    b2Filter collisionFilter;
    collisionFilter.groupIndex = 0;
    collisionFilter.categoryBits = 0x0010; // category = Wall
    collisionFilter.maskBits = 0x0001;    // mask = Ball

    left->SetFilterData(collisionFilter);
    right->SetFilterData(collisionFilter);
}
```

If you run the PhysicsBox2D03 project now, you'll see . . . a pinball table. Great. But how do you know that the collision shapes are properly placed and active?

Box2D Debug Drawing

This is where the `GLESDbgDraw` class distributed separately from Box2D comes in handy. It's also the reason why in Listing 13–7 all child objects are added using a negative z-order. Remember that any drawing done by OpenGL ES code in a node's draw method is drawn at a z-order of 0. If you want the OpenGL ES drawings to actually be drawn over other nodes, those nodes need to have a negative z-order.

Let's first look at the `enableBox2dDebugDrawing` method of the `PinballTable` class:

```
-(void) enableBox2dDebugDrawing
{
    // Debug Draw functions
    debugDraw = new GLESDbgDraw([[CCDirector sharedDirector] contentScaleFactor] *
        PTM_RATIO);
    world->SetDebugDraw(debugDraw);

    uint32 flags = 0;
    flags |= b2DebugDraw::e_shapeBit;
    flags |= b2DebugDraw::e_jointBit;
    // flags |= b2DebugDraw::e_aabbBit;
    // flags |= b2DebugDraw::e_pairBit;
    // flags |= b2DebugDraw::e_centerOfMassBit;
    debugDraw->SetFlags(flags);
}
```

An instance of the `GLESDbgDraw` class is created, using the pixel-to-meter ratio given by the `PTM_RATIO` macro, and then stored in the `PinballTable` member variable `debugDraw`. Since `cocos2d` uses resolution-independent point coordinates that the `GLESDbgDraw` class does not know or care about, it bypasses `cocos2d` to render the debug display with regular OpenGL calls. The current display scaling factor must also be considered so that the `GLESDbgDraw` overlay is correct for both standard and Retina resolution devices. In case you're wondering, the `contentScaleFactor` will be 2.0 for Retina devices and 1.0 for all other devices, including the iPad.

The `debugDraw` instance is then passed to the Box2D world via the `SetDebugDraw` method. You can define what to draw by setting the bits defined in `b2DebugDraw`, with the `e_shapeBit` being the most important because it draws the collision shapes of all bodies.

That alone isn't enough to render the debug output. You also have to override the draw method of the `PinballTable` class and call the `debugDraw->DrawDebugData()` method to actually draw the debug info. Since you don't want the end user to see the debug info, the draw method is enclosed in an `#ifdef DEBUG...#endif` statement so that it's visible only in debug builds:

```
#ifdef DEBUG
-(void) draw
{
    glDisable(GL_TEXTURE_2D);
    glDisableClientState(GL_COLOR_ARRAY);
```



```

    glDisableClientState(GL_TEXTURE_COORD_ARRAY);

    world->DrawDebugData();

    // restore default GL states
    glEnable(GL_TEXTURE_2D);
    glEnableClientState(GL_COLOR_ARRAY);
    glEnableClientState(GL_TEXTURE_COORD_ARRAY);
}
#endif

```

To render the debug info correctly, some of the OpenGL ES states need to be disabled and restored afterward, as you can see from the calls to `glDisable` and `glEnable` methods before and after the call to `world->DrawDebugData()`. You don't have to concern yourself with these OpenGL ES states, but if you're interested in learning more about OpenGL ES rendering, a good starting point is the OpenGL ES 1.1 reference: www.khronos.org/opengles/sdk/1.1/docs/man.

NOTE: Cocos2d 1.x versions only support OpenGL ES 1.1, and all future releases of cocos2d v1.x will continue to use OpenGL ES 1.1 in order to be compatible with all iOS devices. The next major release of cocos2d (version number 2.0) will use OpenGL ES 2.0 instead and exclusively. Note that OpenGL ES 2.0 is not available on first- and second-generation devices. In my opinion (see www.learn-cocos2d.com/2011/07/dropping-dead-opengl-es-v1), you can safely switch to cocos2d 2.0 once it's out. And when it is, you'll find the OpenGL ES 2.0 documentation at www.khronos.org/opengles/sdk/docs/man.

Adding the Ball

Can you imagine a pinball game without a pinball? I can't, so let's add one to the `PhysicsBox2D04` project and have a look at its implementation. The aptly named `Ball` class is derived from `BodyNode` and also implements the `CCTargetedTouchDelegate` protocol for experimentation purposes (see Listing 13–8).

Listing 13–8. The Ball Class's Interface

```

#import "BodyNode.h"

@interface Ball : BodyNode <CCTargetedTouchDelegate>
{
    bool moveToFinger;
    CGPoint fingerLocation;
}

+(id) ballWithWorld:(b2World*)world;

@end

```

There's the usual static initializer `ballWithWorld`, which takes a `b2World` pointer as input. Then we have the member variable `moveToFinger`, which determines whether the ball should move toward the touch location, and the `fingerLocation` `CGPoint` variable, which

specifies the actual location of the finger. We can use those to have a little fun with the ball as long as the pinball game doesn't yet have any other interactive elements to move the ball. Take a look at the `Ball` initialization and `dealloc` methods in Listing 13–9.

Listing 13–9. *The `init` and `dealloc` Methods of the `Ball` Class*

```
-(id) initWithWorld:(b2World*)world
{
    if ((self = [super initWithShape:@"ball" inWorld:world]))
    {
        // set the parameters
        body->SetType(b2_dynamicBody);
        body->SetAngularDamping(0.9f);

        // set random starting point
        [self setBallStartPosition];

        // enable handling touches
        [[CCTouchDispatcher sharedDispatcher] addTargetedDelegate:self
                                                    priority:0
                                                    swallowsTouches:NO];

        // schedule updates
        [self scheduleUpdate];
    }
    return self;
}

+(id) ballWithWorld:(b2World*)world
{
    return [[[self alloc] initWithWorld:world] autorelease];
}

-(void) dealloc
{
    [[CCTouchDispatcher sharedDispatcher] removeDelegate:self];
    [super dealloc];
}
```

Just like the `TablePart` class, the initialization begins by calling the `BodyNode` `init` method `initWithShape`, which takes care of setting up the body and sprite. Well, it almost does so, since we do have to set the body type to be a `b2_dynamicBody` to let `Box2D` know that this body should be treated as a movable object.

In addition, the angular damping value of the body is set to `0.9f`, which makes the ball's angular motion more resistant to change. This allows the ball to slide over a surface without rolling too much, which is standard behavior for heavy pinballs made of metal.

TIP: Tweaking physics values is usually a very labor-intensive aspect that requires careful consideration of each change. It's also frequently underestimated by both designers and programmers alike. That is because all physics attributes are interrelated or interdependent. For example, if you change the density (mass), friction, or restitution of one object, you'll inevitably alter the behavior of any colliding object. You'll notice that this example pinball game, while it does a fairly good job of simulating a pinball game, would still require a lot of tweaking and fine-tuning of the ball's behavior and collision responses for everything to feel just right and to be a fair and fun pinball table. PhysicsEditor can help you with the tweaking of values by providing you a single, convenient interface for editing any shape's physics parameters.

The `setBallStartPosition` method repositions the ball somewhere in the area where you'll later add the plunger. By slightly randomizing the ball's position the plunger will later shoot the ball into play more realistically, meaning unpredictably to some extent. Whenever the ball falls into the drain, the `setBallStartPosition` method is called again to place the ball back to its start position.

```
-(void) setBallStartPosition
{
    // set the ball's position
    float randomOffset = CCRANDOM_0_1() * 10.0f - 5.0f;
    CGPoint startPos = CGPointMake(305 + randomOffset, 80);

    body->SetTransform([Helper toMeters:startPos], 0.0f);
    body->SetLinearVelocity(b2Vec2_zero);
    body->SetAngularVelocity(0.0f);
}
```

The `body->SetTransform` method is used to position the ball's body and as with all bodies in this example game, the update method of the `PinballTable` class takes care of synchronizing the body's sprite with the body's position. The position in pixels must of course be converted to Box2D's meter units, which is done by using the `Helper` class' `toMeters` method. The second parameter of the `SetTransform` method is the rotation of the body.

Just changing the body's position is not enough. The body would still keep its current velocity (angular and linear) and would simply keep on moving. The last two lines of the `setBallStartPosition` method thus reset the linear and angular velocity of the body to zero. Linear velocity determines a body's speed and direction, whereas angular velocity determines how fast and in which direction the body rotates.

To actually make the ball appear on the pinball table, you'll also have to add it to the scene. This is done in the `init` method of the `TableSetup` class, by adding these lines below the initialization of the `TablePart` objects:

```
Ball* ball = [Ball ballWithWorld:world];
[self addChild:ball z:-1];
```

If you run the game now, you'd see the ball in the lower-left corner drop to the ground, and that's it. You don't have any way to move it yet, so let's add some simple ball movement code for testing the bumpers before we get to add the plunger.

Forcing the Ball to Move

So far, the ball is just dropping down, and that's it. We need a way, at least temporary, to control the ball. The `Ball` class implements the `CCTargetedTouchDelegate` and has registered itself to receive touches. Let's check what the touch delegate methods do:

```
-(BOOL) ccTouchBegan:(UITouch *)touch withEvent:(UIEvent *)event
{
    moveToFinger = YES;
    fingerLocation = [Helper locationFromTouch:touch];
    return YES;
}

-(void) ccTouchMoved:(UITouch *)touch withEvent:(UIEvent *)event
{
    fingerLocation = [Helper locationFromTouch:touch];
}

-(void) ccTouchEnded:(UITouch *)touch withEvent:(UIEvent *)event
{
    moveToFinger = NO;
}
```

These methods specify that while a finger is touching the screen, the ball moves toward the finger; and while the finger is moving, the `fingerLocation` is constantly updated.

Next, let's take a quick look at the update method of the `Ball` class:

```
-(void) update:(ccTime)delta
{
    if (moveToFinger == YES)
    {
        [self applyForceTowardsFinger];
    }

    if (sprite.position.y < -(sprite.contentSize.height * 10))
    {
        [self setBallStartPosition];
    }

    // limit speed of the ball
    const float32 maxSpeed = 6.0f;
    b2Vec2 velocity = body->GetLinearVelocity();
    float32 speed = velocity.Length();
    if (speed > maxSpeed)
    {
        velocity.Normalize();
        body->SetLinearVelocity(maxSpeed * velocity);
    }

    // reset rotation of the ball
}
```

```

    body->SetTransform(body->GetWorldCenter(), 0.0f);
}

```

I'll get to the `applyForceTowardsFinger` method next. But while we're here, notice how we check to see whether the ball has gone down the drain. The sprite's y position is compared with the sprite image's height multiplied by 10. Why the multiplication? That's just to give the impression that it takes a short moment for the ball to roll back before it reappears. If the ball has fallen down far enough outside the screen area, the `setBallStartPosition` resets the ball's position, and the fun begins anew.

The update method also ensures that the ball has a maximum speed that it will never exceed. I've chosen the `maxSpeed` value to be 6.0 merely by trial and error.

The length of the linear velocity vector of the ball's body is the ball's current speed. If the body's speed exceeds `maxSpeed`, the body should be slowed down to `maxSpeed` without changing its direction. This is achieved by first normalizing the velocity vector, which results in a vector that still points in the same direction but has a length of exactly one unit; this is called a *unit vector*. With a length of one unit, all you need to do is to multiply this unit vector by `maxSpeed` to cap the body's velocity to `maxSpeed`.

The last line simply causes the body's rotation to be reset; in other words, the body never rotates. This is to prevent the ball's sprite from rotating. Since the ball's image has a highlight and a shadow, we can only create the illusion of a light source shining on the ball if the highlight and shadow of the ball stay in place.

Since the body's world center is set as the position, the position of the body remains the same, and only its rotation is updated. Doing so does not stop the physical effect of a spinning ball colliding with or sliding along hard surfaces because that effect is calculated from the body's angular velocity, which remains unaffected. You can safely reset the ball's rotation, because from the perspective of the physics engine, the ball's shape being a circle means that the ball's surface features are the same from any direction. A circle is completely symmetrical.

Now let's have a look at the `applyForceTowardsFinger` method, which makes the ball accelerate toward the finger, as in Listing 13–10.

Listing 13–10. Accelerating the Ball Toward the Touch Location

```

-(void) applyForceTowardsFinger
{
    b2Vec2 bodyPos = body->GetWorldCenter();
    b2Vec2 fingerPos = [Helper toMeters:fingerLocation];

    b2Vec2 bodyToFingerDirection = fingerPos - bodyPos;
    bodyToFingerDirection.Normalize();

    b2Vec2 force = 2.0f * bodyToFingerDirection;
    body->ApplyForce(force, body->GetWorldCenter());
}

```

We have the two positions of the body and the finger, and then we subtract the finger position from the body position. For example, if the body's position would be at the screen center (160, 240) and the finger is touching near the upper-right corner of the

screen at (300, 450), then subtracting the body position from the finger position being the subtraction of the individual x and y coordinates results in $(300 - 160, 450 - 240) = (140, 210)$. The vector `bodyToFingerDirection` is now (140, 210). It's called the direction from `bodyPos` to `fingerPos` because if you would add `bodyToFingerDirection` to `bodyPos`, you would get to the coordinates of `fingerPos`.

NOTE: The `b2Vec2` struct makes use of a technique called *operator overloading*, which makes it possible to subtract, add, or multiply two or more `b2Vec2` structs with each other. Operator overloading is a feature of the C++ language; it's not available in Objective-C so you can't subtract, add, or multiply `CGPoint` variables this way.

So, the `bodyToFingerDirection` vector is now pointing from the body to the finger. When `Normalize` is called on the `bodyToFingerDirection` vector, it's turned into a *unit vector* as mentioned earlier. A unit vector is a vector of length 1 or one unit. This allows you to multiply it with a fixed factor, in this case doubling its length, to create a constant force vector pointing in the direction of the finger. You can then use the `ApplyForce` method of the body to apply this as an external force to the body's center. You could also use a position other than the center; however, in that case, the body would start spinning.

The end result of this is that the ball accelerates toward the point on the screen that your finger is touching. The ball will usually overshoot, slow down, and return. A little bit like the gravitational pull the sun exerts on our planets, albeit a lot more dramatically.

However, as someone interested in astronomy, I do have to correct myself. Gravity is a force that falls off by the square of the distance between two objects pulling on each other through gravity. So if you want a more realistic simulation of gravity in your game, simply replace the `applyForceTowardsFinger` code with that in Listing 13–11.

Listing 13–11. Simulating Gravitational Pull

```
-(void) applyForceTowardsFinger
{
    b2Vec2 bodyPos = body->GetWorldCenter();
    b2Vec2 fingerPos = [Helper toMeters:fingerLocation];
    float distance = bodyToFingerDirection.Length();
    bodyToFingerDirection.Normalize();

    // "real" gravity falls off by the square over distance
    float distanceSquared = distance * distance;
    b2Vec2 force = ((1.0f / distanceSquared) * 20.0f) * bodyToFingerDirection;
    body->ApplyForce(force, body->GetWorldCenter());
}
```

The multiplication by 20.0f in this case is a magic number. It's just there to make the gravitational pull noticeable enough. Now the ball will speed up more the closer it gets to your finger and will barely move if you touch the screen relatively far away from the ball.

While the `applyForceTowardsFinger` code serves only as a temporary control mechanism, you could use the gravity code in Listing 13–11 to create magnetic objects on your pinball table.

Adding the Bumpers

Now that you have a ball that you can move with your finger, let's make things a little bit more interesting by introducing bumpers to the game. What are bumpers? They're the round, mushroom-shaped objects that will force the ball away when the ball touches them.

NOTE: Sometimes people confuse bumpers with the flippers that the player controls or the usually triangular slingshots just above the flippers. If you want to refresh your memory about pinball terminology, the Wikipedia web site about pinballs should be able to help you out: en.wikipedia.org/wiki/Pinball.

Listing 13–12 shows the once again rather simple header file of the Bumper class.

Listing 13–12. The Bumper Class's Interface

```
#import "BodyNode.h"

@interface Bumper : BodyNode
{
}

+(id) bumperWithWorld:(b2World*)world position:(CGPoint)pos;
@end
```

Once more, the Bumper class is derived from BodyNode. The initialization looks very much like Listing 13–9, in which the ball was initialized, so I'll just focus on the important part in Listing 13–13.

Listing 13–13. Initializing the Bumper

```
-(id) initWithWorld:(b2World*)world position:(CGPoint)pos
{
    if ((self = [super initWithShape:@"bumper" inWorld:world]))
    {
        // set the body position
        body->SetTransform([Helper toMeters:pos], 0.0f);
    }
    return self;
}

+(id) bumperWithWorld:(b2World*)world position:(CGPoint)pos
{
    return [[[self alloc] initWithWorld:world position:pos] autorelease];
}
```

The only key ingredient for the Bumper class is to set its restitution parameter to above 1.0f—in this case you've already set it to 1.5f in PhysicsEditor. This gives any rigid body touching the surface of the bumper an impulse that is 50 percent higher than the force with which the bumper was hit. The result is something that's not possible in the real world: the impacting object increases its velocity after hitting the bumper's surface. It's

physics engine magic, and in this case it's very desirable because we save ourselves a lot of headaches in implementing the bumper's logic. Box2D does it for you.

What's left is to add some bumpers by adding the following lines in the `init` method of the `TableSetup` class. Feel free to reposition the bumpers as you desire:

```
// add some bumpers
[self addBumperAt:ccp( 76, 405) inWorld:world];
[self addBumperAt:ccp(158, 415) inWorld:world];
[self addBumperAt:ccp(239, 375) inWorld:world];
[self addBumperAt:ccp( 83, 341) inWorld:world];
[self addBumperAt:ccp(157, 294) inWorld:world];
[self addBumperAt:ccp(260, 286) inWorld:world];
[self addBumperAt:ccp( 67, 228) inWorld:world];
[self addBumperAt:ccp(183, 189) inWorld:world];
```

To make adding bumpers more convenient, the method `addBumperAt` was also added to the `TableSetup` class:

```
-(void) addBumperAt:(CGPoint)pos inWorld:(b2World*)world
{
    Bumper* bumper = [Bumper bumperWithWorld:world position:pos];
    [self addChild:bumper];
}
```

Have a look now and try how the bumpers feel in the `PhysicsBox2D04` project—pretty close to actual pinball bumpers, I think.

The Plunger

I hate to take control away from you, but for now I must. We're adding the plunger now, and being able to control the ball with your fingers might get in the way. So, go into the `Ball` class's update method and comment out the call to `applyForceTowardsFinger`:

```
if (moveToFinger == YES)
{
    // disabled: no longer needed
    // [self applyForceTowardsFinger];
}
```

Now you can add the `Plunger` class, which I've already done in the `PhysicsBox2D05` project. Listing 13–14 shows the `Plunger` class's interface, which is also derived from `BodyNode`.

Listing 13–14. The Plunger's Header File

```
#import "BodyNode.h"

@interface Plunger : BodyNode
{
    b2PrismaticJoint* joint;
}

+(id) plungerWithWorld:(b2World*)world;

@end
```


The `Plunger` class has a member variable for `b2PrismaticJoint`, which it's going to use to propel itself upward. A prismatic joint allows only one axis of movement—a telescope bar would be a good example of a prismatic joint in the real world. You can only move the smaller pipe inside the larger pipe, which allows the telescope bar to be extended and retracted but only in one direction.

Initializing the plunger is also straightforward, as Listing 13–15 shows. The plunger's physics settings were edited in `PhysicsEditor`. In particular, the friction was set to a very high value, while restitution was set to 0. This ensures that the ball is launched smoothly by remaining in close contact with the plunger during the time the plunger is propelled upward.

Listing 13–15. Initializing the Plunger

```
-(id) initWithWorld:(b2World*)world
{
    if ((self = [super initWithShape:@"plunger" inWorld:world]))
    {
        CGSize screenSize = [[CCDirector sharedDirector] winSize];
        CGPoint plungerPos = CGPointMake(screenSize.width - 13, -32);

        body->SetTransform([Helper toMeters:plungerPos], 0);
        body->SetType(b2_dynamicBody);

        [self attachPlunger];
    }
    return self;
}
```

Most interesting is the call to `attachPlunger` and the actual creation of the prismatic joint in this method, which is shown in Listing 13–16.

Listing 13–16. Creating the Plunger's Prismatic Joint

```
-(void) attachPlunger
{
    // create an invisible static body to attach joint to
    b2BodyDef bodyDef;
    bodyDef.position = body->GetWorldCenter();
    b2Body* staticBody = body->GetWorld()->CreateBody(&bodyDef);

    // Create a prismatic joint to make plunger go up/down
    b2PrismaticJointDef jointDef;
    b2Vec2 worldAxis(0.0f, 1.0f);
    jointDef.Initialize(staticBody, body, body->GetWorldCenter(), worldAxis);
    jointDef.lowerTranslation = 0.0f;
    jointDef.upperTranslation = 0.35f;
    jointDef.enableLimit = true;
    jointDef.maxMotorForce = 80.0f;
    jointDef.motorSpeed = 40.0f;
    jointDef.enableMotor = false;

    joint = (b2PrismaticJoint*)body->GetWorld()->CreateJoint(&jointDef);
}
```

First, a static body is created at the same location as the plunger's dynamic body. It will act as the larger pipe of a telescope bar held in place so that only the inner pipe may

spring upward when released. Remember physics (or *Mythbusters*, for that matter): every action has an equal and opposite reaction. To avoid the opposite reaction□ a downwards-oriented movement of the other body of the prismatic joint□ it is turned into a static, unmovable body holding the plunger in place.

The `worldAxis` restricts the prismatic joint's movement to the y-axis (in other words, up and down). The `worldAxis` is expressed as a normal vector with values from 0.0f to 1.0f; when the y-axis is set to 1.0f, the `worldAxis` becomes parallel to the y-axis. If you were to set both x- and y-axes to 0.5f, the `worldAxis` would be a 45-degree angle.

`b2PrismaticJointDef` is initialized with the `staticBody` and the world center position of the plunger's dynamic body. As anchor point for the joint the `worldAxis` is used, which restricts motion of the prismatic joint along the y-axis.

Now follows a set of parameters. The lower and upper translations define how far along the axis the plunger is allowed to move. In this case, it is allowed to move 0.35f meters upward, which is exactly 42 points or 84 pixels on Retina display devices and 42 pixels on non-Retina devices. The `enableLimit` field is set to `true` so that this movement limit is actually adhered to by the connected bodies. Since the static body won't move, the plunger will move the full extent. If both were dynamic bodies, both bodies would be able to move, which would be undesirable in this case, as mentioned earlier.

Next, I set `maxMotorForce` to 80.0f (the unit in this case is Newton-meters, which is a unit of torque). In Chipmunk this is called the body's *moment*. The `maxMotorForce` value limits the torque, or energy, of the joint's movement. The `motorSpeed` then determines how quickly, if at all, this `maxMotorForce` is reached. I determined both values merely by trial and error until it felt about right. The ball is now catapulted up and around with just about the right speed. The motor is initially disabled because I only want the plunger to go off when there's a ball touching it.

The joint is then created using the world's `CreateJoint` method and stored in the joint member variable. Since `CreateJoint` returns a `b2Joint` pointer, it has to be cast to a `b2PrismaticJoint` pointer before assignment.

Notice that it is not necessary to destroy the joint that the `Plunger` class is keeping as a member variable. The joint is automatically destroyed when either body it is attached to is destroyed, and in this case the `BodyNode`'s `dealloc` method destroys the body.

The plunger must also be added to the table. As usual, this is done in the `initWithWorld` method of the `TableSetup` class. Simply append the following code:

```
// Add plunger
Plunger *plunger = [Plunger plungerWithWorld:world];
[self addChild:plunger z:-1];
```

Creating a Universal Contact Listener

To launch the ball automatically on contact, we need some way to react to collisions. The Box2D physics engine includes the `b2ContactListener` class. To process collisions, you will have to create a custom class that inherits from `b2ContactListener` and

overrides at least one of the collision callback methods `BeginContact`, `EndContact`, `PreSolve`, and `PostSolve`.

You commonly end up with code that looks similar to Listing 13–17. In principle, you'll retrieve the two colliding bodies from the `b2Contact` class. In this particular case, each body's user data pointer holds a pointer to a `BodyNode` object, which allows you to compare the classes to decide how to further process the collision. The verbosity of the code is further increased because a contact's two bodies may be in any order even if the same objects collide frequently. That means during any contact, the plunger might be either `bodyA` or `bodyB`, while the ball would be the corresponding other body. So, you always have to test for both cases.

Listing 13–17. A Very Common but Tedious Way to Process Box2D Collisions

```
void ContactListener::BeginContact(b2Contact* contact)
{
    b2Body* bodyA = contact->GetFixtureA()->GetBody();
    b2Body* bodyB = contact->GetFixtureB()->GetBody();
    BodyNode* bodyNodeA = (BodyNode*)bodyA->GetUserData();
    BodyNode* bodyNodeB = (BodyNode*)bodyB->GetUserData();

    if ([bodyNodeA isKindOfClass:[Plunger class]] && ←
        [bodyNodeB isKindOfClass:[Ball class]])
    {
        Plunger* plunger = (Plunger*)bodyNodeA;
        // ... perform custom code for collision handling
    }
    else if ([bodyNodeB isKindOfClass:[Plunger class]] && ←
             [bodyNodeA isKindOfClass:[Ball class]])
    {
        Plunger* plunger = (Plunger*)bodyNodeB;
        // ... perform custom code for collision handling
    }
}
```

The previous approach also requires you to import the header files for each colliding `BodyNode` class. Over time, the collision-handling class would know about most game objects. Instead, you would want to have each `BodyNode` class handle the collision events that it is involved in. This keeps the code cleanly separated and easier to maintain and turns the job of the collision handling class to one of delegating the collision events to the colliding objects.

The `ContactListener` class (Listing 13–18) in the `PhysicsBox2D05` project defines two additional methods next to the regular Box2D collision callback methods to perform the delegation of collision events.

Listing 13–18. The `ContactListener` Class Definition

```
class ContactListener : public b2ContactListener
{
private:
    void BeginContact(b2Contact* contact);
    void PreSolve(b2Contact* contact, const b2Manifold* oldManifold);
    void PostSolve(b2Contact* contact, const b2ContactImpulse* impulse);
    void EndContact(b2Contact* contact);
}
```

```

void notifyObjects(b2Contact* contact, NSString* contactType);
void notifyAB(b2Contact* contact,
              NSString* contactType,
              b2Fixture* fixtureA,
              NSObject* objA,
              b2Fixture* fixtureB,
              NSObject* objB);
};

```

The regular Box2D contact methods are implemented in Listing 13–19. The `BeginContact` and `EndContact` methods simply delegate the contact information to the `notifyObjects` methods but also provide the information about whether it was a begin or end contact event by passing an appropriate `NSString` object. The reason why it's a string and not a flag or enumeration will become clear shortly. Since we do not care about the `PreSolve` and `PostSolve` events, they remain empty stubs but could be extended by also calling `notifyObjects` with the contact and an appropriate string.

Listing 13–19. Implementation of the Box2D Contact Methods

```

/// Called when two fixtures begin to touch.
void ContactListener::BeginContact(b2Contact* contact)
{
    notifyObjects(contact, @"begin");
}

/// Called when two fixtures cease to touch.
void ContactListener::EndContact(b2Contact* contact)
{
    notifyObjects(contact, @"end");
}

void ContactListener::PreSolve(b2Contact* contact, const b2Manifold* oldManifold)
{
    // do nothing
}

void ContactListener::PostSolve(b2Contact* contact, const b2ContactImpulse* impulse)
{
    // do nothing
}

```

What does the `notifyObjects` method do? Extracting the two colliding bodies and obtaining their user data pointer is similar to Listing 13–17. But take note of the differences:

```

void ContactListener::notifyObjects(b2Contact* contact, NSString* contactType)
{
    b2Fixture* fixtureA = contact->GetFixtureA();
    b2Fixture* fixtureB = contact->GetFixtureB();

    b2Body* bodyA = fixtureA->GetBody();
    b2Body* bodyB = fixtureB->GetBody();

    NSObject* objA = (NSObject*)bodyA->GetUserData();
    NSObject* objB = (NSObject*)bodyB->GetUserData();
}

```

```

    if ((objA != nil) && (objB != nil))
    {
        notifyAB(contact, contactType, fixtureA, objA, fixtureB, objB);
        notifyAB(contact, contactType, fixtureB, objB, fixtureA, objA);
    }
}

```

In this case, we simply assume the user data pointer to be a pointer to an Objective-C class derived from `NSObject`. This provides the flexibility that any object can respond to collision events, not just `BodyNode` objects. If both user data pointers are not `nil`, then the `notifyAB` method is called twice, the second time with the `A` and `B` variables switched. This will ensure that both `objA` and `objB` will receive the collision notification, and the `notifyAB` method only needs to handle one case.

The job of the `notifyAB` method is to construct the selector that should be called and, if possible, call the selector with any contact information that the receiving object might need to handle the collision. Listing 13–20 shows the implementation of the `notifyAB` method.

Listing 13–20. Implementation of the Box2D Contact Methods

```

void ContactListener::notifyAB(b2Contact* contact,
                              NSString* contactType,
                              b2Fixture* fixture,
                              NSObject* obj,
                              b2Fixture* otherFixture,
                              NSObject* otherObj)
{
    NSString* format = @"%@ContactWith%@";
    NSString* otherClassName = NSStringFromClass([otherObj class]);
    NSString* selectorString = [NSString stringWithFormat:format, contactType,
        otherClassName];
    SEL contactSelector = NSSelectorFromString(selectorString);

    if ([obj respondsToSelector:contactSelector])
    {
        Contact* contactInfo = [Contact contactWithObject:otherObj
                                                    otherFixture:otherFixture
                                                    ownFixture:fixture
                                                    b2Contact:contact];
        [obj performSelector:contactSelector withObject:contactInfo];
    }
}

```

The format string defines the general naming format of the selectors that will be called. The syntax of the selectors that `notifyAB` calls is as follows:

```
<contactType>ContactWith<otherClassName>:(Contact*)contactInfo
```

The `contactType` is the string you pass to the `notifyObjects` method, which will be either `begin` or `end` in the current implementation. The `otherClassName` string is obtained from the `NSStringFromClass` method, which takes the class of the `otherObj`. For the collision events of the ball and the plunger, the `selectorString` will be one of the following, depending on the `contactType` and the `otherObj` class name:

```
beginContactWithBall
endContactWithBall
beginContactWithPlunger
endContactWithPlunger
// and so on
```

Before performing the selector, `notifyAB` first checks whether `obj` actually responds to that selector. In the current implementation, only the `Plunger` class implements one of the selectors: `beginContactWithBall`. All other selectors will never be performed since they don't exist (yet).

The `Contact` class is also defined in `ContactListener.h` and merely acts as a container object holding any collision information that you might want to pass to receiving classes. By using a container class, the selector format does not need to change when you decide to pass more or less information, simply because the only parameter is a pointer to a `Contact` object and the information is encapsulated within the `Contact` class.

TIP: There's also a technical reason for using a container class. The `performSelector` method of the `NSObject` class knows only three variants: with zero, one, or two parameters. Since we definitely like to pass on more than two parameters to the receiving object, there's simply no other choice than to use a container class. Whenever you find the `performSelector` method limiting, remember that you can always create a container class holding any information that you'd like to pass on to the class implementing the selector.

Since `Contact` is such a simple class, Listing 13–21 shows both interface and implementation in one Listing.

Listing 13–21. *Interface and Implementation of the Contact Class*

```
@interface Contact : NSObject
{
@private
    NSObject* otherObject;
    b2Fixture* ownFixture;
    b2Fixture* otherFixture;
    b2Contact* b2contact;
}
@property (assign, nonatomic) NSObject* otherObject;
@property (assign, nonatomic) b2Fixture* ownFixture;
@property (assign, nonatomic) b2Fixture* otherFixture;
@property (assign, nonatomic) b2Contact* b2contact;

+(id) contactWithObject:(NSObject*)otherObject
      otherFixture:(b2Fixture*)otherFixture
      ownFixture:(b2Fixture*)ownFixture
      b2Contact:(b2Contact*)b2contact;

@end

@implementation Contact

@synthesize otherObject, ownFixture, otherFixture, b2contact;
```

```

+ (id) initWithObject:(NSObject*)otherObject
        otherFixture:(b2Fixture*)otherFixture
        ownFixture:(b2Fixture*)ownFixture
        b2Contact:(b2Contact*)b2contact
{
    Contact* contact = [[[Contact alloc] init] autorelease];

    if (contact)
    {
        contact.otherObject = otherObject;
        contact.otherFixture = otherFixture;
        contact.ownFixture = ownFixture;
        contact.b2contact = b2contact;
    }

    return contact;
}

- (id) retain
{
    [NSException raise:@"ContactRetainException"
                 format:@"Do not retain a Contact - it is for temporary use only!"];
    return self;
}

@end

```

The only notable aspect of the `Contact` class is that it overrides the `retain` method. This is in case a user tries to retain contact information, which is not allowed in Box2D. Directly after the call to the Box2D contact methods, the contact variable will be released by Box2D and so the `otherFixture`, `ownFixture`, and `b2contact` pointers will be invalid, and accessing them would cause a crash. Since calling `retain` indicates that the user wants to keep a reference to the `Contact` object for later use, this must be prevented by throwing an exception that will cause the program execution to halt with an error message logged to the debug console.

Responding to Contact Events

Now whenever the ball and the plunger get in contact, the `beginContactWithBall` method in the `Plunger` class will be called:

```

- (void) endPlunge:(ccTime)delta
{
    // stop the scheduling of endPlunge
    [self unschedule:_cmd];

    // stop the motor
    joint->EnableMotor(NO);
}

- (void) beginContactWithBall:(Contact*)contact
{
    // start the motor
    joint->EnableMotor(YES);
}

```

```

        // schedule motor to come back, unschedule in case the plunger is hit repeatedly
        [self unschedule:_cmd];
        [self schedule:@selector(endPlunge:) interval:0.5f];
    }

```

As soon as the ball touches the plunger, the plunger's motor is enabled, which propels the plunger and thus the ball upward. The `endPlunge` method is scheduled to stop the motor after a short time. Extra care has been taken to correctly unschedule the selectors. For example, it is very likely that the `beginContactWithBall` method is called repeatedly within a short time period, because there may be more than one contact point (Box2D reports each contact point individually); or simply, the ball might bounce a little and lose contact, but the plunger's motor ensures that the plunger will touch the ball again after a short time.

Similarly, you'll find the two methods `beginContactWithPlunger` and `beginContactWithBumper` implemented in the `Ball` class. Both types of contact will simply play a sound effect:

```

-(void) playSound
{
    float pitch = 0.9f + CCRANDOM_0_1() * 0.2f;
    float gain = 1.0f + CCRANDOM_0_1() * 0.3f;
    [[SimpleAudioEngine sharedEngine] playEffect:@"bumper.wav"
                                         pitch:pitch
                                         pan:0.0f
                                         gain:gain];
}

-(void) endContactWithBumper:(Contact*)contact
{
    [self playSound];
}

-(void) endContactWithPlunger:(Contact*)contact
{
    [self playSound];
}

```

TIP: Keep in mind that Box2D reports each individual contact of two colliding objects, causing the contact methods to be called more than once for the same two objects. In some cases, you may want to set a boolean variable to YES in order to note that a contact has already happened. The corresponding contact method should first check whether the variable is set, and if it is, skip the code. You should later set the variable back to NO in a scheduled update method to reenable contact events. By doing so, you can avoid contact code being run multiple times to avoid undesirable side effects like too many sounds played at once.

The Flippers

The final ingredients are the flippers, with which you'll control the action. The two flippers are going to be controlled by touching the screen on either the left or right side, as Listing 13–22 shows.

Listing 13–22. *The Flipper Interface*

```
#import "BodyNode.h"

typedef enum
{
    kFlipperLeft,
    kFlipperRight,
} EFlipperType;

@interface Flipper : BodyNode <CCTargetedTouchDelegate>
{
    EFlipperType type;
    b2RevoluteJoint* joint;
    float totalTime;
}

+(id) flipperWithWorld:(b2World*)world flipperType:(EFlipperType)flipperType;

@end
```

Each flipper is anchored using a `b2RevoluteJoint`. Take a look at the flipper `initWithWorld` method in Listing 13–23 to see how the flippers are created.

Listing 13–23. *Creating a Flipper*

```
-(id) initWithWorld:(b2World*)world flipperType:(EFlipperType)flipperType
{
    NSString* name = (flipperType == kFlipperLeft) ? @"flipper-left" : @"flipper-right";

    if ((self = [super initWithShape:name inWorld:world]))
    {
        type = flipperType;

        // set the position depending on the left or right side
        CGPoint flipperPos = (type == kFlipperRight) ? ccp(210,65) : ccp(90,65);

        // attach the flipper to a static body with a revolute joint
        [self attachFlipperAt:[Helper toMeters:flipperPos]];

        // receive touch events
        [[CCTouchDispatcher sharedDispatcher] addTargetedDelegate:self
                                                                priority:0
                                                                swallowsTouches:NO];
    }
    return self;
}

+(id) flipperWithWorld:(b2World*)world flipperType:(EFlipperType)flipperType
{

```

```

    return [[[self alloc] initWithWorld:world flipperType:flipperType] autorelease];
}

-(void) dealloc
{
    // stop listening to touches
    [[CCTouchDispatcher sharedDispatcher] removeDelegate:self];

    [super dealloc];
}

```

The Flipper class also registers itself with the CCTouchDispatcher to receive touch input events. The common misconception is that only the CCLayer class can receive input, but in fact CCLayer is merely conveniently preconfigured to receive touches. Any class can register itself with the CCTouchDispatcher class as a delegate, provided that the class also removes itself as a touch delegate, usually in the dealloc method.

As with the other pinball elements, the flippers are added to the TableSetup class and initialized as left and right flipper by using the EFlipperType enum from Listing 13–22.

```

// Add flippers
Flipper *left = [Flipper flipperWithWorld:world flipperType:kFlipperLeft];
[self addChild:left];

Flipper *right = [Flipper flipperWithWorld:world flipperType:kFlipperRight];
[self addChild:right];

```

NOTE: I could have used the flipper’s shape names instead, but I wanted to hide this implementation detail. No one but the Flipper class should be concerned with what the flipper’s frame and shape names are.

The attachFlipperAt method creates the revolute joints (see Listing 13–24), with a few modifications for the right flipper in order to change the direction and upper limit of the right flippers rotation. The point the flippers rotate around will be the anchor point of their shapes, which is editable in PhysicsEditor.

Listing 13–24. Creating the Flipper Revolute Joint

```

-(void) attachFlipperAt:(b2Vec2)pos
{
    body->SetTransform(pos, 0);
    body->SetType(b2_dynamicBody);

    // turn on continuous collision detection to prevent tunneling
    body->SetBullet(true);

    // create an invisible static body to attach to
    b2BodyDef bodyDef;
    bodyDef.position = pos;
    b2Body* staticBody = body->GetWorld()->CreateBody(&bodyDef);

    // setup joint parameters
    b2RevoluteJointDef jointDef;
    jointDef.Initialize(staticBody, body, staticBody->GetWorldCenter());
}

```

```

    jointDef.lowerAngle = 0.0f;
    jointDef.upperAngle = CC_DEGREES_TO_RADIANS(70);
    jointDef.enableLimit = true;
    jointDef.maxMotorTorque = 100.0f;
    jointDef.motorSpeed = -40.0f;
    jointDef.enableMotor = true;

    if (type == kFlipperRight)
    {
        // mirror speed and angle for the right flipper
        jointDef.motorSpeed *= -1;
        jointDef.lowerAngle = -jointDef.upperAngle;
        jointDef.upperAngle = 0.0f;
    }

    // create the joint
    joint = (b2RevoluteJoint*)body->GetWorld()->CreateJoint(&jointDef);
}

```

You may be wondering why the flipper's body is set as a bullet. Trust me, I'm not going to shoot flippers at you! Physics engines traditionally have a problem with detecting collisions of objects moving at high speeds since such an object can travel great distances between two collision tests, seemingly "tunneling" through other collidable objects. That may be fine for subatomic particles, but not for our flippers and the ball.

The `SetBullet` method enables a special, continuous collision detection method for fast-moving objects that takes into account the path the object must have taken between two collision tests. Thus, the bullet mode is able to detect collisions that would have otherwise been missed, at the expense of performance. The bullet mode should be used judiciously and only when absolutely needed. In the pinball game, I noticed that both the flippers and the ball would sometimes "miss" each other, so I have them both treated as fast-moving objects to get more accurate collision detection.

The static body is created to attach the flipper to an unmovable body to keep the flipper anchored in place. `b2RevoluteJointDef` uses `lowerAngle` and `upperAngle` as the rotation limits, which are in radians. I'll set the `upperAngle` to 70 degrees and convert it to radians with the `CC_DEGREES_TO_RADIANS` macro provided by `cocos2d`.

The revolute joint also has `maxMotorTorque` and `motorSpeed` fields, which are used to define the speed and immediacy of the movement of the flippers. However, contrary to the plunger, the motor is enabled all the time, and merely its direction of movement will be reversed by changing the sign of the `motorSpeed` variable. While the flippers are down, the motor will force them down so that they don't bounce when the ball hits them.

In the `ccTouchBegan` method, the location of the touch is obtained, which is validated with the `isTouchForMe` method before actually reversing the motor.

```

-(BOOL) ccTouchBegan:(UITouch*)touch withEvent:(UIEvent*)event
{
    BOOL touchHandled = NO;

    CGPoint location = [Helper locationFromTouch:touch];
    if ([self isTouchForMe:location])

```

```

    {
        touchHandled = YES;
        [self reverseMotor];
    }

    return touchHandled;
}

-(void) ccTouchEnded:(UITouch*)touch withEvent:(UIEvent*)event
{
    CGPoint location = [Helper locationFromTouch:touch];
    if ([self isTouchForMe:location])
    {
        [self reverseMotor];
    }
}

```

The `isTouchForMe` method implements the check to figure out on which side of the screen the touch was and whether the current instance of the class is the correct flipper to respond to this touch.

```

-(bool) isTouchForMe:(CGPoint)location
{
    if (type == kFlipperLeft && location.x < [Helper screenCenter].x)
    {
        return YES;
    }
    else if (type == kFlipperRight && location.x > [Helper screenCenter].x)
    {
        return YES;
    }

    return NO;
}

```

Reversing the motor speed then simply allows the flipper to spring up and to spring back down again when the touch ends and the motor speed is reversed again.

```

-(void) reverseMotor
{
    joint->SetMotorSpeed(joint->GetMotorSpeed() * -1);
}

```

The rest is just physics. If the ball is on the flipper and you touch the screen on the correct side, the flipper will be accelerated upward, pushing the ball with it. Depending on where on the flipper the ball lands, it will be propelled more or less straight upward.

Summary

In this chapter, you learned how to use the `PhysicsEditor` tool to define the collision shapes for the bodies used in the pinball game. With just the ball in place, I illustrated how you can simulate acceleration toward a point, including how to model the effects of gravity or magnetism more or less realistically.

I hope this chapter gave you an impression of how much fun physics can be, regardless of what you may have experienced in physics class. But then again, you didn't build pinball machines in physics class □ or did you?

If you'd like to go beyond this example □ for example, using more joints or taking more control of the collision process □ I'd like to refer you to the Box2D manual, at www.box2d.org/manual.html.

On the other hand, if you need more information about individual classes and structs, you should look at the Box2D API reference. It is provided in the Documentation folder of the Box2D download, which you can obtain from <http://code.google.com/p/box2d>. Since the Box2D API reference is not available online, I decided to host it myself at www.learn-cocos2d.com/box2d-api-reference/API/index.html.

To get help with Box2D, you can check out the official Box2D forums at www.box2d.org/forum/index.php and check out the Physics subsection of the cocos2d forums at www.cocos2d-iphone.org/forum/forum/7.

If you are interested in learning more about PhysicsEditor, I can recommend the PhysicsEditor blog (www.physicseditor.de/blog) in which Andreas Löw shows off some cool uses and tips and tricks for PhysicsEditor. If you have a support request for Andreas, you can simply write an e-mail to support@code-and-web.de.

Game Center

Game Center is Apple's social network solution. It enables you to authenticate players, store their scores and display leaderboards, and track and display their achievement progress. Furthermore, players can invite friends to play or choose to quickly find a match to play a game with anyone.

In this chapter, I'll introduce you not only to Game Center and the Game Kit API but also to the basics of online multiplayer programming and, of course, how to use Game Center together with cocos2d.

Since a lot of Apple's examples are intentionally incomplete, I'll be developing a `GameKitHelper` class in this chapter. This class will remove some of the complexities of Game Center programming from you. It will make it easier for you to use Game Kit and Game Center features, and it will allow you to easily reuse the same code for other games.

To configure your application for use with Game Center, you are going to use iTunes Connect. The information on the iTunes Connect web site is considered confidential Apple information, so I can't discuss it in this book. However, I will point you to Apple's excellent documentation for each step and quite frankly, setting up leaderboards and achievements on iTunes Connect is possibly the easiest aspect of Game Center.

Enabling Game Center

Game Center is the service that manages and stores player accounts and each player's friend lists, leaderboards, and achievements. This information is stored online on Apple's servers and accessed either by your game or by the Game Center app that is installed on all devices running iOS 4.1 or newer.

NOTE: Game Center is available only on devices running iOS 4.1 and newer and is not available on first-generation devices or the iPhone 3G. Game Center currently runs on iPod touch second, third, and fourth generations; iPhone 3GS and iPhone 4; and iPad; as long as they have iOS 4.1 or higher installed. The easiest way for a user to check whether their device supports Game Center is to locate the Game Center app on the device. If it exists, the device is ready for Game Center; otherwise, it is not. If the Game Center app is not available but the device is eligible for upgrading to iOS 4.1, Game Center support will become available after upgrading the device's operating system via iTunes.

If you don't have access to a Game Center-enabled device, you can still program and test Game Center features using the iPhone/iPad Simulator. With the exception of matchmaking, all Game Center features can be tested in the simulator.

On the other side, the Game Kit API is what you use to program Game Center features. Game Kit provides programmatic access to the data stored on the Game Center servers and is able to show built-in leaderboards, achievements, and matchmaking screens. But Game Kit also provides features besides Game Center—for example, peer-to-peer networking via Bluetooth and voice chat. These are the only two Game Kit features already available on devices running iOS 3.0 or newer.

The final ingredient in this mix is iTunes Connect. You set up your game's leaderboards and achievements through the iTunes Connect web site. But most importantly, it allows you to enable Game Center for your game in the first place. I'll start with that step first, so you can and should do this before you have even created an Xcode project for your game.

Your starting point for learning more about Game Center and the steps involved in creating a game that uses Game Center is at Apple's Getting Started with Game Center web site: <http://developer.apple.com/devcenter/ios/gamecenter>.

For a high-level overview of Game Center, I recommend reading the Getting Started with Game Center document: http://developer.apple.com/ios/download.action?path=/ios/getting_started_with_ios_4.1/gettingstartedwithgamecenter.pdf.

Creating Your App in iTunes Connect

The very first step is to log in with your Apple ID on the iTunes Connect web site: <http://itunesconnect.apple.com>.

Then you want to add a new application, even if it doesn't exist yet. For most fields that iTunes Connect asks you to fill out, you can enter bogus information. There are only two settings that you have to get right. The first is, obviously, to enable Game Center when iTunes Connect asks you whether the new application should support Game Center.

The other is to enter a Bundle ID (also referred to as Bundle Identifier) that matches the one used in the Xcode project. Since you don't have an Xcode project yet, you are free

to choose any Bundle ID you want. Apple recommends using reverse domain names for Bundle IDs with the app's name appended at the end. The catch is that the Bundle ID needs to be unique across all App Store apps, and there are tens of thousands of them.

For the book's example, I've chosen `com.learn-cocos2d` to be the app's Bundle ID. Since this Bundle ID is now taken by me, you will have to use your own Bundle ID. If you want, you can simply suffix it with a string of your choosing or choose an entirely new string. Just remember to use your own Bundle ID whenever I refer to the `com.learn-cocos2d` Bundle ID.

For a detailed description of how to create a new app and how to set up Game Center for an app on iTunes Connect, please refer to Apple's iTunes Connect Developer Guide: http://itunesconnect.apple.com/docs/iTunesConnect_DeveloperGuide.pdf.

Specifically, the section labeled Game Center explains in great detail how to manage the Game Center features on iTunes Connect.

Setting Up Leaderboards and Achievements

For the most part, after enabling Game Center for an app, what you'll be doing on iTunes Connect is setting up one or more leaderboards, which will hold your players' scores or times, as well as setting up a number of achievements that players can unlock while playing your game.

To access the Game Center leaderboards and achievements, you refer to them by ID. For leaderboards, you should note the leaderboard category ID strings, and for achievements the achievement ID strings, to be able to query and update the correct leaderboards and achievements.

For the purpose of this chapter, I have set up one leaderboard with a score format of Elapsed Time and a leaderboard category ID of `Playtime`. For achievements I've entered one achievement, with an achievement ID of `PlayedForTenSeconds`, that grants the player five achievement points.

Feel free to set up additional leaderboards and achievements, but keep in mind that the example code in this chapter relies on at least one leaderboard with a category ID of `Playtime` and one achievement with an achievement ID of `PlayedForTenSeconds` to exist.

Creating a Cocos2d Xcode Project

Now it is time to create the actual Xcode project. You can start the project from any cocos2d template—for example, the cocos2d HelloWorld application template. You can also use an already existing project, but you may have to upgrade to cocos2d v1.0 if your project was started with a cocos2d version prior to v0.99.5.

You can determine which version of cocos2d your project is using in three ways. First, if your project is built and you run it, one of the first lines in the Debugger Console window will read something like this:

```
2010-10-07 15:33:58.363 Tilemap[1046:207] cocos2d: cocos2d v1.0.1
```


Or, you can use the global method `cocos2dVersion` to print it out yourself—for example, using `CCLOG` in this way:

```
CCLOG(@"%@", cocos2dVersion());
```

The final option is to simply look it up. In your project, in the group containing the `cocos2d` sources, locate the file `cocos2d.m` and open it. It contains the version string in plain text:

```
static NSString *version = @"cocos2d v1.0.1";
```

If you are facing the problem of having to upgrade `cocos2d` from a previous version in an existing project, please refer to this short tutorial on upgrading `cocos2d` in an existing project with Xcode 4: www.learn-cocos2d.com/2011/05/update-cocos2d-iphone-existing-project.

TIP: If you grow tired of the error-prone and tedious upgrade process, then you might be interested in `Kobold2D`, which is discussed in Chapter 16 and available from www.kobold2d.org. With `Kobold2D`, upgrading the `cocos2d` version used in a project becomes as easy as moving the project to a different folder and adding it to the `Kobold2D` workspace.

Starting with version 0.99.5, the `RootViewController` class, which is derived from `UIViewController`, became part of every `cocos2d` project. Game Center needs a `UIViewController` to be able to show its built-in UIKit user interface, and having the `RootViewController` class available makes Game Center integration a lot easier. The `GameKitHelper` class will make use of the `RootViewController` class. And in case you're wondering, another benefit of the `RootViewController` class is that it allows you to enable autorotation in your app. You'll learn more about autorotation in Chapters 15 and 16.

Configuring the Xcode Project

The first thing you should do now is enter the Bundle ID you've entered for your app in iTunes Connect. Remember that while I'm using `com.learn-cocos2d` as the Bundle ID for the example projects, you can't use the same because it's already taken now, and Bundle IDs must be unique.

Locate the file `Info.plist` in your project's Resources folder and select it. You can then edit it in the Property List editor, as shown in Figure 14–1. You will want to set the `Bundle identifier` key to have the same value as your app's Bundle ID. In my case that will be `com.learn-cocos2d`, and in your case it will be whatever string you chose as the app's Bundle ID.

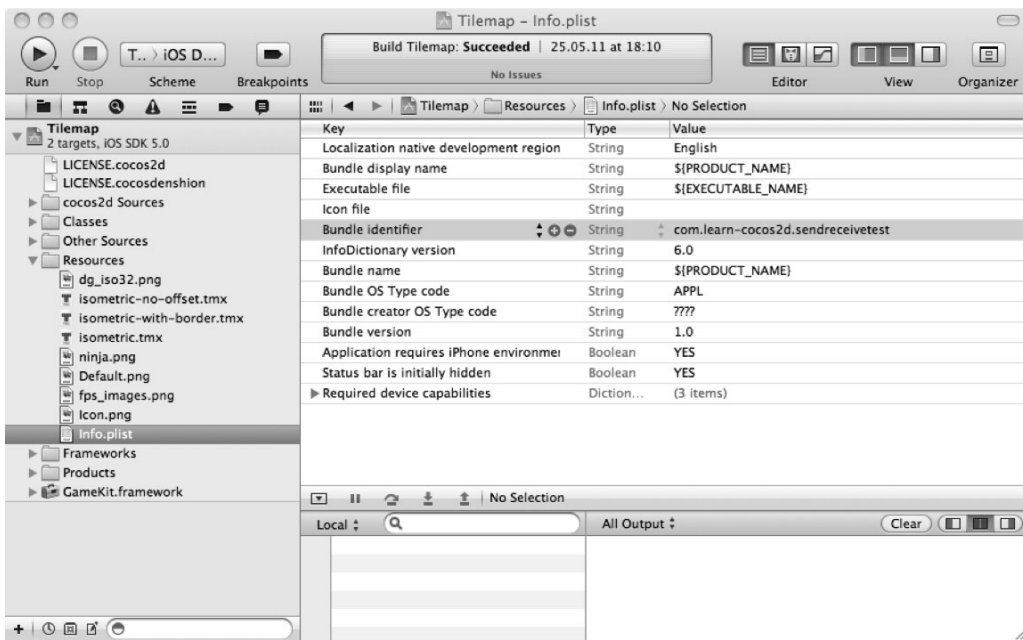


Figure 14–1. The *Bundle identifier* key must match your app's *Bundle ID*.

There are actually two ways to use Game Kit and Game Center for that matter. One is to require it, which means your app will run only on devices that support Game Center and are running iOS 4.1 or newer. However, for the examples I've written, I did not make Game Center a requirement because it's relatively easy to check whether Game Center is available and then not use it if it isn't. This allows your game to be run on older devices, just without all the Game Center features.

But if you do want to require Game Kit and Game Center to be present, you can set this in your app's Info.plist `UIRequiredDeviceCapabilities` list. By adding another key named `gamekit` with a Boolean value and checking the check box, as shown in Figure 14–2, you can tell iTunes and potential users that your app requires Game Kit and thus requires iOS 4.1 or newer.

You can learn more about iTunes requirements and the `UIRequiredDeviceCapabilities` key in Apple's Build Time Configuration documentation: <http://developer.apple.com/library/ios/#documentation/iphone/Conceptual/iphoneOSProgrammingGuide/BuildTimeConfiguration/BuildTimeConfiguration.html>.

CAUTION: If you add the `gamekit` key but later decide you don't want to make Game Kit a requirement, make sure you remove the `gamekit` entry. If you simply uncheck the `gamekit` check box, it actually tells iTunes that your app is not available on devices that do support Game Center. That's exactly the opposite of what you might expect. To actually make Game Kit an optional requirement, you'll have to remove the `gamekit` entry altogether.

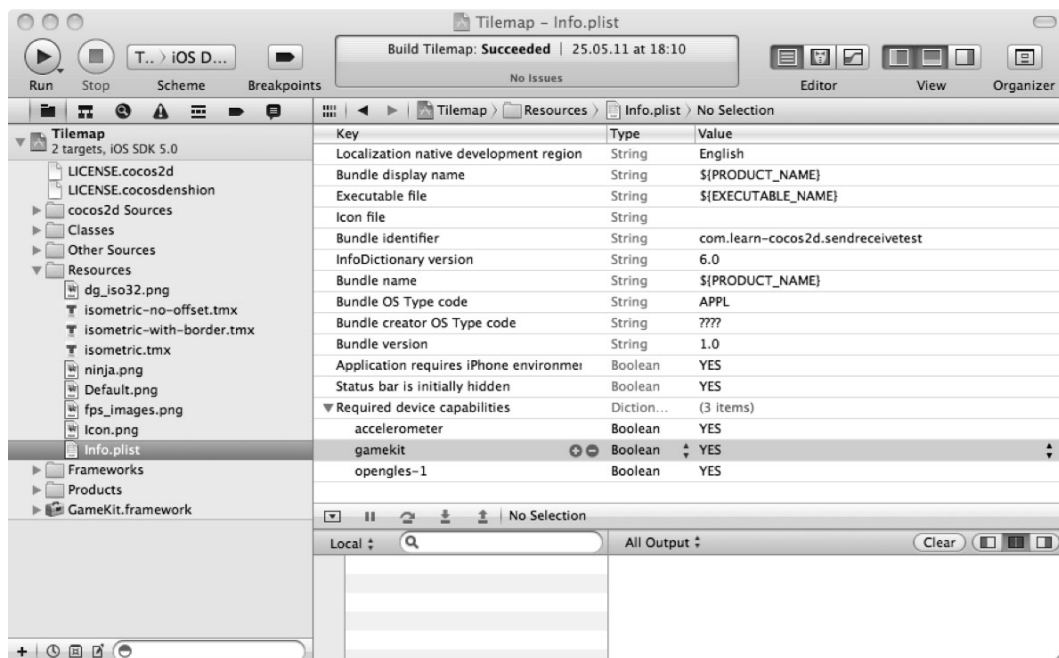


Figure 14–2. Making Game Kit a strict requirement

Next, you need to add the Game Kit framework to your Xcode project—more specifically, to the desired application target in your project in case you have several targets. Select the root entry in the Project Navigator, which is the project itself and labeled **Tilemap** in this example. Then select the appropriate target and switch to the **Build Phases** tab. Unfold the **Link Binary With Libraries** section to see the list of libraries this target is currently linked with. Below that list are two + and – buttons with which you can add or remove libraries.

To add another library, click the + button. You’ll see another list pop up like the one in Figure 14–3. Locate the **GameKit.framework** entry and click the **Add** button. Since there are a lot of libraries to choose from and they’re not always sorted alphabetically, it helps to filter the list by entering *GameKit* in the text field above the list.

The **GameKit.framework** will be added to the **Linked Libraries** list when you click the **Add** button. By default, new libraries are added as **Required**, which is displayed to the right of each library. The setting **Required** means that your app will work only on devices where the **GameKit.framework** library is available. If that is what you want and you’ve added the **gamekit** key to **Info.plist**, you can leave it at that. Otherwise, change the setting to **Optional** in order to be able to run the app even on devices that don’t have Game Center available. We can account for that with a relatively simple check in code (discussed shortly in Listing 14–3) and then disable any Game Kit features in case a device doesn’t support Game Kit.

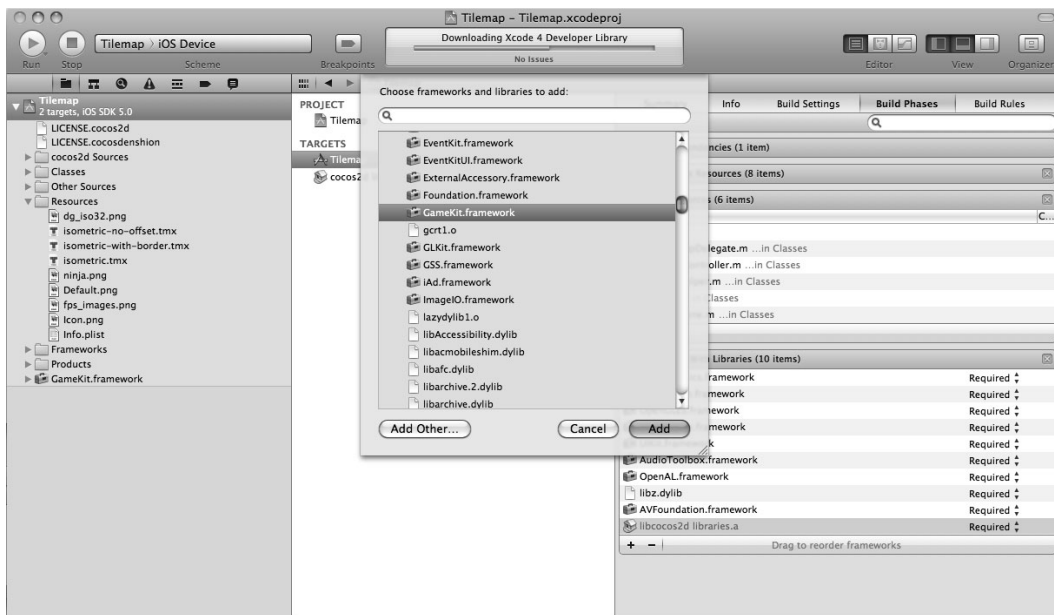


Figure 14–3. Adding *GameKit.framework*

Finally, you will want the `GameKit.h` header file to be available in all of your project's source files. Instead of adding it to each and every source file, you should add it to your project's `Prefix.pch` file. This is the precompiled header that contains header files from external frameworks to allow the project to compile faster. But it also has the added benefit that every header file added to the prefix header will make its definitions available to every source code file in the current project.

The prefix header file is always prefixed with the project's name. So, for example, in the case of the `tilemap` project, the file name is `Tilemap_Prefix.pch`, and the file can be found in the `Other Sources` group. Open the one in your project and add the `GameKit` header to it, as shown in Listing 14–1.

Listing 14–1. *Adding the GameKit Header to Your Project's Prefix Header*

```
#ifdef __OBJC__
    #import <Foundation/Foundation.h>
    #import <UIKit/UIKit.h>
    #import <GameKit/GameKit.h>
#endif
```

That's it—your app is set up for use with Game Center.

Game Center Setup Summary

To summarize, enabling Game Center for your app requires the following steps:

1. Create a new app in iTunes Connect:
 - a. Specify a Bundle ID for the new app.
 - b. Enable Game Center for this app.
2. Set up your initial leaderboards and achievements in iTunes Connect:
 - a. Note the leaderboard category IDs and achievement IDs. (Note that you will likely continue to edit and add leaderboards and achievements throughout the development of your game.)
3. Create or upgrade the Xcode project:
 - a. Make sure to use at least cocos2d v0.99.5.
4. Edit Info.plist:
 - a. Enter the app's Bundle ID in the **Bundle identifier** field.
 - b. Optionally require Game Kit by adding a Boolean value labeled `gamekit` to the `UIRequiredDeviceCapabilities` list.
5. Add the necessary Game Kit references:
 - a. Add the `GameKit.framework` linked library to each target. Change its **Type** setting from **Required** to **Weak** if Game Kit support is optional.
 - b. Add `#import <GameKit/GameKit.h>` to your project's prefix header file.

Before you proceed, make sure you have followed each step. You can always go back and make the necessary changes later. However, if you don't do all of these steps at the beginning, chances are that you will get errors or something won't work, but the associated error message won't necessarily point you to a mistake or oversight concerning one of these steps.

Common causes for Game Center to not work properly are a mismatch between the Bundle ID in the project's Info.plist file and the Bundle ID set up for your app in iTunes Connect.

Game Kit Programming

Before I get into programming Game Center with the Game Kit API, I'd like to mention the two important resources on Apple's developer web site.

There is the Game Kit Programming Guide, which provides a high-level, task-based overview of Game Kit and Game Center concepts: http://developer.apple.com/library/ios/#documentation/NetworkingInternet/Conceptual/GameKit_Guide/Introduction/Introduction.html.

For in-depth detailed information about the Game Center classes and protocols, you can refer to the Game Kit Framework Reference: http://developer.apple.com/library/ios/#documentation/GameKit/Reference/GameKit_Collection/_index.html.

The GameKitHelper Delegate

I mentioned earlier in this chapter that I will use a `GameKitHelper` class to provide easier access to Game Kit and Game Center features. Since connecting to an online server causes responses to be delayed by several milliseconds, if not seconds, it's a good idea to have a central class manage all Game Center–related features. All the Game Center examples are based on the isometric game developed in Chapter 11. You'll find the following example code in the `Tilemap11` project.

One of your game's classes can then use this functionality and register itself as a `GameKitHelper` delegate to get notified of events as they occur. To do that, the delegate must implement the `GameKitHelper` @protocol that is defined in the `GameKitHelper.h` header file (Listing 14–2). Only classes implementing this protocol can be assigned to the `GameKitHelper` delegate property to receive the protocol messages. The protocol is simply a list of method definitions that a class using the protocol must implement. If any of the methods in the protocol aren't implemented, the compiler will let you know about that.

Listing 14–2. *The GameKitHelper Header File*

```
#import "cocos2d.h"
#import <GameKit/GameKit.h>

@protocol GameKitHelperProtocol
-(void) onLocalPlayerAuthenticationChanged;
-(void) onFriendListReceived:(NSArray*)friends;
-(void) onPlayerInfoReceived:(NSArray*)players;
@end

@interface GameKitHelper : NSObject
{
    id<GameKitHelperProtocol> delegate;
    bool isGameCenterAvailable;
    NSError* lastError;
}

@property (nonatomic, retain) id<GameKitHelperProtocol> delegate;
```

```

@property (nonatomic, readonly) bool isGameCenterAvailable;
@property (nonatomic, readonly) NSError* lastError;

+(GameKitHelper*) sharedGameKitHelper;

// Player authentication, info
-(void) authenticateLocalPlayer;
-(void) getLocalPlayerFriends;
-(void) getPlayerInfo:(NSArray*)players;
@end

```

For your convenience, the `GameKitHelper` class also stores the last error in its `lastError` property. This allows you to check whether any error occurred and, if so, what kind of error, without actually receiving the Game Center messages directly. The `GameKitHelper` class is a singleton, which was described in Chapter 3, so I'll leave the singleton-specific code out of the discussion.

The remaining properties and methods will be discussed shortly. For now, let's take a look at how the `TileMapLayer` class is extended so that it can function as the delegate for `GameKitHelper`. The essential changes to the header file are importing `GameKitHelper.h` and specifying that `TileMapLayer` implements `GameKitHelperProtocol`:

```

#import "GameKitHelper.h"

@interface TileMapLayer : CCLayer <GameKitHelperProtocol>
{
    ...
}

```

Then you can set the `TileMapLayer` class to be the delegate of the `GameKitHelper` class, in the `init` method:

```

GameKitHelper* gkHelper = [GameKitHelper sharedGameKitHelper];
gkHelper.delegate = self;
[gkHelper authenticateLocalPlayer];

```

Note that you are responsible for setting the `GameKitHelper` delegate back to `nil` when appropriate—for example, shortly before changing scenes. Because `GameKitHelper` retains the delegate, it will not be released from memory, even if it otherwise would (for example, during a scene change). That would not only keep the delegate itself in memory but all of its member variables as well, including all of its children if it's a `CCNode` class.

Checking for Game Center Availability

The `GameKitHelper` class starts by checking for Game Center availability right in its `init` method (Listing 14–3). It needs to do that only once because the conditions never change while the app is running.

Listing 14–3. Testing for Game Center Availability

```
// Test for Game Center availability
Class gameKitLocalPlayerClass = NSClassFromString(@"GKLocalPlayer");
bool isLocalPlayerAvailable = (gameKitLocalPlayerClass != nil);

// Test if device is running iOS 4.1 or higher
NSString* reqSysVer = @"4.1";
NSString* currSysVer = [[UIDevice currentDevice] systemVersion];
bool isOSVer41 = ([currSysVer compare:reqSysVer
                    options:NSNumericSearch] != NSOrderedAscending);

isGameCenterAvailable = (isLocalPlayerAvailable && isOSVer41);
```

The first test is simply to check whether a specific Game Center class is available. In this case, the Objective-C runtime method `NSClassFromString` is used to get one of the Game Center classes by name. If this call returns `nil`, you can be certain that Game Center is unavailable.

But it's not quite that simple. Because Game Center was already partially available in beta versions prior to iOS 4.1, you also need to check whether the device is running at least iOS 4.1. This is done by comparing the `reqSysVer` string with the `systemVersion` string.

Once both checks are made, the results are combined using the `&&` (and) operator, so that both must be true for `isGameCenterAvailable` to become true. The `isGameCenterAvailable` variable is used to safeguard all calls to Game Center functionality within the `GameKitHelper` class. This avoids accidentally calling Game Center functionality when it is not available, which would crash the application.

Note that this is how Apple recommends to check for Game Center availability. You should not try any other methods—for example, determining the type of device your game is running on. While certain devices are excluded from using Game Center, this is already accounted for with the preceding check.

Authenticating the Local Player

The local player is a fundamental concept to Game Center programming. It refers to the player account that is signed into the device. This is important to know because only the local player can send scores to leaderboards and report achievement progress to the Game Center service. The very first thing a Game Center application needs to do is authenticate the local player. If that fails, you cannot use most of the Game Center services, and in fact Apple recommends not using any Game Center functionality unless there is an authenticated local player.

In the `GameKitHelper` `init` method, the `registerForLocalPlayerAuthChange` method is called so that `GameKitHelper` receives events concerning authentication changes for the local player. This is the only Game Center notification that is sent through `NSNotificationCenter`. You register a selector to receive the message, as shown in Listing 14–4.

Listing 14–4. Registering for Local Player Authentication Changes

```

-(void) registerForLocalPlayerAuthChange
{
    if (isGameCenterAvailable == NO)
        return;

    NSNotificationCenter* nc = [NSNotificationCenter defaultCenter];
    [nc addObserver:self
        selector:@selector(onLocalPlayerAuthenticationChanged)
        name:GKPlayerAuthenticationDidChangeNotificationName
        object:nil];
}

```

As you can see, `isGameCenterAvailable` is used here to skip the rest of the method in case Game Center isn't available. You'll notice other methods doing the same thing, and I'll refrain from repeating this in the book's code.

The actual method being called by `NSNotificationCenter` simply forwards the message to the delegate:

```

-(void) onLocalPlayerAuthenticationChanged
{
    [delegate onLocalPlayerAuthenticationChanged];
}

```

NOTE: The local player's signed-in status may actually change while the game is in the background and the user runs the Game Center app and signs out. This is because of the multitasking nature of iOS 4.0 and newer. Essentially, your game must be prepared to handle the local player logging out and some other player signing in at any time during game play. Typically, you should end the current game session and return to a safe place—for example, the main menu. But you should consider saving the current state of the game for each local player as they sign out so that when a particular local player signs back in, the game continues exactly where that player left the game.

The actual authentication is performed by the `authenticateLocalPlayer` method, in Listing 14–5.

Listing 14–5. Authenticating the Local Player

```

-(void) authenticateLocalPlayer
{
    GKLocalPlayer* localPlayer = [GKLocalPlayer localPlayer];
    if (localPlayer.authenticated == NO)
    {
        [localPlayer authenticateWithCompletionHandler: ^{
            (NSError* error)
            {
                [self setLastError:error];
            }
        }];
    }
}

```

At first glance, that's relatively straightforward. The `localPlayer` object is obtained, and if it's not authenticated, the `authenticateWithCompletionHandler` method is called. And the `NSError` object returned by the method is set to the `lastError` and . . . hey, wait a second. That's all part of the method's parameter?

Yes. These inline methods are called *block objects*. I'll tell you more about them in the next section. For now, you only need to know that the block object is a C-style method that's passed as a parameter to the `authenticateWithCompletionHandler` method. It's run only after the authentication request has returned from the server.

If you call the `authenticateLocalPlayer` method, your game will display the Game Center sign-in dialog, shown in Figure 14–4. If you have an Apple ID, you can sign in with your Apple ID and password. Or you can choose to create a new account.



Figure 14–4. Game Center sign-in dialog

But there's a third possibility□ if Game Center detects that there's already a signed-in player on this device, it will simply greet you with a □welcome back□ message. How do you sign out in that case? This is done through the Game Center app, which also exists on the iPhone/iPad Simulator for that very reason. If you run the Game Center app, select the first tab that reads either **Me** or **Sandbox**, and then click the label at the bottom that starts with **Account:**, and you'll get a pop-up dialog that allows you to view your account or sign out. After signing out through the Game Center app, the next time you run your app and it's going through the player authentication process, the sign-in dialog in Figure 14–4 will be shown again.

NOTE: If the `[GKLocalPlayer localPlayer].underage` property is set after the local player was authenticated, some Game Center features are disabled. You can also refer to the `underage` property if your game should disable optional features that are not suitable for underage players.

Now, about error handling, you'll notice that `GameKitHelper` uses the `setLastError` method wherever there's an error object returned. This allows the delegate class to check whether any error occurred through the `lastError` property. If it is `nil`, then there was no error.

However, only the last error object is kept around, and the next method returning an NSError object will replace the previous error, so it is crucial to check for the `lastError` property right away if error handling is important in that particular case. In some cases, you can safely ignore errors. They might lead only to temporary problems, like an empty friends list. Regardless, the `setLastError` message copies the new error after releasing the old one and then prints out diagnostic information so that you can always keep an eye on the kinds of errors that occur during development:

```
-(void) setLastError:(NSError*)error
{
    [lastError release];
    lastError = [error copy];

    if (lastError != nil)
        NSLog(@"GameKitHelper ERROR: %@", [[lastError userInfo] description]);
}
```

If you receive an error and would like to know more about it, you can refer to Apple's Game Kit Constants Reference, which describes the error constants defined in the `GameKit/GKError.h` header file. You can find the Constants Reference here: http://developer.apple.com/library/ios/#documentation/GameKit/Reference/GameKit_ConstantsRef/Reference/reference.html.

After the local player has successfully signed in, you can access his friend list, leaderboards, and achievements. But before I get to that, let's sidestep for a moment and review the important aspects of block objects.

Block Objects

The inline method shown in Listing 14-5 is called a *block object* and commonly referred to simply as *blocks*. You might have heard of *closures*, *anonymous functions*, or *lambda* in other languages, which are essentially the same concept. Block objects are a C-language extension introduced by Apple to make multithreaded and asynchronous programming tasks easier. In layman's terms, block objects are callback functions that can be created within other functions, assigned to variables for later use, passed on to other functions, and run asynchronously at a later time. Since a block object has read access to the local variables of the function or scope it was defined in, the block object typically requires fewer arguments than a regular callback method. Optionally, with the `__block` storage specifier, you can also allow the block object to modify variables in its enclosing scope.

TIP: Refer to Apple's Blocks Programming Topics documentation if you are interested in more details about block objects:
http://developer.apple.com/library/mac/#documentation/Cocoa/Conceptual/Blocks/Articles/00_Introduction.html.

I'll cut out the actual block object from Listing 14–5 to discuss it separately:

```
^(NSError* error)
{
    [self setLastError:error];
}
```

It looks like a method, except it has no name and it begins with a caret symbol (^). The NSError pointer is the only variable passed to it, but there can be multiple variables delimited by commas, as in this example:

```
^(NSArray* scores, NSError* error)
{
    [self setLastError:error];
    [delegate onScoresReceived:scores];
}
```

If that reminds you of a C method's parameters, you are correct. If you will, you can consider a block to be a C method whose name is ^ and can be passed to one of the many Game Kit methods taking block objects as parameters.

There are two technicalities I'd like to point out. First, local variables can be accessed in a block. But they can't normally be modified, unless they are prefixed with the `__block` keyword. Consider this code snippet:

```
__block bool success = NO;
[localPlayer authenticateWithCompletionHandler:^(
    NSError* error)
{
    success = (error == nil);
    lastError = error;
}]];
```

With blocks, it is only legal to modify a local variable declared outside the block's scope if the variable is declared with the `__block` keyword. In this case, the `success` variable is declared locally outside the block but is modified within the block, so it must be prefixed with the `__block` keyword. On the other hand, the `lastError` variable is a member variable of the class. Member variables can be modified within blocks without the use of the `__block` keyword.

Also, in the case of Game Kit, you'll be frequently passing block objects to Game Kit methods, but the block objects won't be run until a later time. You are probably used to code being executed in sequence, but in Game Kit programming it is not! The block passed to a Game Kit method is called only when the call completes a round-trip to and from the Game Center server. That takes time because data needs to be transmitted to the Game Center servers and processed, and then a result needs to be returned to the device. Only then does the block object get executed.

Let's take an example. You may find yourself tempted to write something like this:

```
__block bool success = NO;

[localPlayer authenticateWithCompletionHandler:^(
    NSError* error)
{
```

```

        success = (error == nil);
    }];

    if (success)
        NSLog(@"Local player logged in!");
    else
        NSLog(@"Local player NOT logged in!");

```

However, this example will always report that the player is not logged in. Why? Well, the execution path is such that the `authenticateWithCompletionHandler` will take your block as a parameter and store it while it sends a request to the server and waits for the response to come back. However, the execution continues right away after the `authenticateWithCompletionHandler` method, and that's where the success variable decides which log statement to print. The problem is, the success variable is still set to NO because the block hasn't been executed yet.

Several milliseconds later, the server responds to the authentication, and that triggers the completion handler — the block object — to be run. If it returns without error, the success variable is set to YES. But alas, your logging code has already been run, so the assignment has no effect.

Note that this is not a problem of block objects in general; there are methods that immediately, or even repeatedly, run a block right away before returning back to you. But in the case of almost all Game Kit methods, the block objects are used exclusively as pieces of code that will be run whenever the Game Center server has responded to a particular request. In other words, the block objects used by Game Kit are run asynchronously after an unspecified delay (and possibly not at all if the connection is interrupted).

Receiving the Local Player's Friend List

When the local player signs in or out, the `onLocalPlayerAuthenticationChanged` method is received and forwarded to the delegate. The delegate in these examples is the `TileMapLayer` class, which implements this method to ask for the local player's friend list in Listing 14–6.

Listing 14–6. *Asking for the List of Friends*

```

-(void) onLocalPlayerAuthenticationChanged
{
    GKLocalPlayer* localPlayer = [GKLocalPlayer localPlayer];
    if (localPlayer.authenticated)
    {
        GameKitHelper* gkHelper = [GameKitHelper sharedGameKitHelper];
        [gkHelper getLocalPlayerFriends];
    }
}

```

It checks whether the local player is authenticated, and if so, it calls the `getLocalPlayerFriends` method of the `GameKitHelper` class right away. Let's take a look at that in Listing 14–7.

Listing 14–7. GameKitHelper Requesting the Friends List

```

-(void) getLocalPlayerFriends
{
    GKLocalPlayer* localPlayer = [GKLocalPlayer localPlayer];
    if (localPlayer.authenticated)
    {
        [localPlayer loadFriendsWithCompletionHandler:^(
            ^([NSArray* friends, NSError* error])
            {
                [self setLastError:error];
                [delegate onFriendListReceived:friends];
            }
        )];
    }
}

```

Because the `getLocalPlayerFriends` method doesn't know when it's called or by whom, it plays things safe by checking again that the local player is actually authenticated. Then it calls the `GKLocalPlayer` class's `loadFriendsWithCompletionHandler` method, for which you'll supply another block object that is run when the server returns a list of player identifiers as strings. Unsurprisingly, this list of identifiers is stored in the `friends` array.

Once the call to `loadFriendsWithCompletionHandler` has succeeded, you can access the current player identifiers of the local player's friends through the `GKLocalPlayer` class:

```
NSArray* friends = [GKLocalPlayer localPlayer].friends;
```

Note that the `friends` array can be `nil` or not contain all friends. In the delegate that receives the `onFriendsListReceived` message, and in all other `GameKitHelper` delegate methods for that matter, you should check whether the received parameter is `nil` before working with it. If it is `nil`, you can refer to the `lastError` property of the `GameKitHelper` class to get more information about the error for debugging, logging, or possibly presenting it to the user when it makes sense to do so.

The delegate method `onFriendsListReceived` simply passes the player identifiers back to `GameKitHelper`, requesting more info about the player identifiers in the friends list:

```

-(void) onFriendListReceived:(NSArray*)friends
{
    GameKitHelper* gkHelper = [GameKitHelper sharedGameKitHelper];
    [gkHelper getPlayerInfo:friends];
}

```

That's straightforward, so let's turn our attention back to the `GameKitHelper` class's `getPlayerInfo` method. If the `playerList` array contains at least one entry, it will call the `loadPlayersForIdentifiers` static method of the `GKPlayer` class, as shown in Listing 14–8.

Listing 14–8. Requesting Players from a List of Player Identifiers

```

-(void) getPlayerInfo:(NSArray*)playerList
{
    if ([playerList count] > 0)
    {
        // Get detailed information about a list of players
        [GKPlayer loadPlayersForIdentifiers:playerList withCompletionHandler:^(

```

```

        ^(NSArray* players, NSError* error)
        {
            [self setLastError:error];
            [delegate onPlayerInfoReceived:players];
        }
    }
}

```

Again, a block object is used to handle the returned results from the server. And as always, the `lastError` property is updated before calling the delegate's `onPlayerInfoReceived` method. The `players` array should now contain a list of `GKPlayer` class instances, which the delegate then simply prints to the Debugger Console window in the absence of a proper friend list user interface:

```

-(void) onPlayerInfoReceived:(NSArray*)players
{
    for (GKPlayer* gkPlayer in players)
    {
        NSLog(@"PlayerID: %@, Alias: %@", gkPlayer.playerID, gkPlayer.alias);
    }
}

```

The `GKPlayer` class has only three properties: the player identifier, an alias, and the `isFriend` flag, which is true for all the players in this particular case. The alias is simply the player's nickname.

Leaderboards

In the `Tilemap12` project, I added functionality for posting and retrieving leaderboard scores. I hooked into the `onPlayerInfoReceived` method in the `TileMapLayer` class to submit a dummy score to Game Center, under the `Playtime` category:

```

-(void) onPlayerInfoReceived:(NSArray*)players
{
    GameKitHelper* gkHelper = [GameKitHelper sharedGameKitHelper];
    [gkHelper submitScore:1234 category:@"Playtime"];
}

```

The `submitScore` method shown in Listing 14–9 is implemented in `GameKitHelper` and calls the `onScoresSubmitted` message back to the delegate. Since there's no return parameter to pass on, it simply reports through the success value if the score was transmitted without an error.

Listing 14–9. Submitting a Score to a Leaderboard

```

-(void) submitScore:(int64_t)score category:(NSString*)category
{
    GKScore* gkScore = [[[GKScore alloc] initWithCategory:category] autorelease];
    gkScore.value = score;
    [gkScore reportScoreWithCompletionHandler:^(
        NSError* error)
    {
        [self setLastError:error];

        bool success = (error == nil);
    }
    ];
}

```

```

        [delegate onScoresSubmitted:success];
    }
}

```

The score value is of type `int64_t`, which is the same as `long`. It's a 64-bit value, so it can store an incredibly large number—one with 19 digits. That allows for more than 4 billion times greater values than a regular 32-bit integer can represent!

A temporary `GKScore` object is created and sent the autorelease message, so you don't need to release it. It's initialized with a leaderboard category identifier, which you define in iTunes Connect. In this case, the category ID is `Playtime`. The `GKScore` object also gets the score assigned, and then its `reportScoreWithCompletionHandler` method is called, which will transmit the score to the Game Center server and to the correct leaderboard.

The delegate receives the `onScoresSubmitted` message and subsequently calls the `retrieveTopTenAllTimeGlobalScores` method to get the top ten scores:

```

-(void) onScoresSubmitted:(bool)success
{
    if (success)
    {
        GameKitHelper* gkHelper = [GameKitHelper sharedGameKitHelper];
        [gkHelper retrieveTopTenAllTimeGlobalScores];
    }
}

```

The `GameKitHelper` class's `retrieveTopTenAllTimeGlobalScores` simply wraps the call to `retrieveScoresForPlayers` and feeds it with preconfigured parameters:

```

-(void) retrieveTopTenAllTimeGlobalScores
{
    [self retrieveScoresForPlayers:nil
        category:nil
        range:NSMakeRange(1, 10)
        playerScope:GKLeaderboardPlayerScopeGlobal
        timeScope:GKLeaderboardTimeScopeAllTime];
}

```

Feel free to add more wrapper methods for retrieving scores as you see fit, depending on your game's needs. Since there are a variety of ways to retrieve leaderboard scores and several filters to reduce the number of scores retrieved, it makes sense to use wrapper methods to reduce the potential for human error. Listing 14–10 shows the `retrieveScoresForPlayers` method in full.

Listing 14–10. Retrieving a List of Scores from a Leaderboard

```

-(void) retrieveScoresForPlayers:(NSArray*)players
        category:(NSString*)category
        range:(NSRange)range
        playerScope:(GKLeaderboardPlayerScope)playerScope
        timeScope:(GKLeaderboardTimeScope)timeScope
{
    GKLeaderboard* leaderboard = nil;
    if ([players count] > 0)
    {

```



```

        leaderboard = [[[GKLeaderboard alloc] initWithPlayerIDs:players] autorelease];
    }
    else
    {
        leaderboard = [[[GKLeaderboard alloc] init] autorelease];
        leaderboard.playerScope = playerScope;
    }

    if (leaderboard != nil)
    {
        leaderboard.timeScope = timeScope;
        leaderboard.category = category;
        leaderboard.range = range;

        [leaderboard loadScoresWithCompletionHandler:^(
            ^NSArray* scores, NSError* error)
        {
            [self setLastError:error];
            [delegate onScoresReceived:scores];
        }]];
    }
}

```

First, a `GKLeaderboard` object is initialized. Depending on whether the `players` array contains any players, the leaderboard may be initialized with a list of player identifiers to retrieve scores only for those players. Otherwise, the `playerScope` variable is used, which can be set to either `GKLeaderboardPlayerScopeGlobal` or `GKLeaderboardPlayerScopeFriendsOnly` to retrieve only friends' scores.

Then the leaderboard scope is further reduced by the `timeScope` parameter, which allows you to obtain the all-time high scores (`GKLeaderboardTimeScopeAllTime`), only those from the past week (`GKLeaderboardTimeScopeWeek`), or only today's scores (`GKLeaderboardTimeScopeToday`).

Of course, you also have to specify the category ID for the leaderboard; otherwise, `GKLeaderboard` wouldn't know which leaderboard to retrieve the scores from. Finally, an `NSRange` parameter allows you to refine the score positions you'd like to retrieve. In this example, a range of 1 to 10 indicates that the top ten scores should be retrieved.

Make sure that you limit the score retrieval using all these parameters (especially the `NSRange` parameter) to reasonably small chunks of data. While you could, it's not recommended to retrieve all the scores of a leaderboard. If your game is played online a lot and many scores are submitted, you might be loading hundreds of thousands or millions or billions of scores from the Game Center servers. That would cause a significant delay when retrieving scores.

With the leaderboard object set up properly, the `loadScoresWithCompletionHandler` method takes over and asks the server for the scores. When the scores are received, it calls the delegate method with `onScoresReceived`, passing on the array of scores. The array contains objects of class `GKScore` sorted by leaderboard rank. The `GKScore` objects provide you with all the information you need, including the `playerID`, the date the score was posted, and its rank, value, and `formattedValue`, which you should use to display the score to the user.

Fortunately for us, Apple provides a default leaderboard user interface. Instead of using the scores I just retrieved, I'm going to ignore them and use the `onScoresReceived` delegate method to bring up the built-in leaderboard view:

```
-(void) onScoresReceived:(NSArray*)scores
{
    GameKitHelper* gkHelper = [GameKitHelper sharedGameKitHelper];
    [gkHelper showLeaderboard];
}
```

Game Kit has a `GKLeaderboardViewController` class, which is used to display the Game Center leaderboard user interface, as shown in Listing 14–11.

Listing 14–11. Showing the Leaderboard User Interface

```
-(void) showLeaderboard
{
    GKLeaderboardViewController* leaderboardVC =
        [[[GKLeaderboardViewController alloc] init] autorelease];
    if (leaderboardVC != nil)
    {
        leaderboardVC.leaderboardDelegate = self;
        [self presentViewController:leaderboardVC];
    }
}
```

The `leaderboardDelegate` is set to `self`, which means the `GameKitHelper` class must implement the `GKLeaderboardViewControllerDelegate` protocol. The first step is to add this protocol to the class's interface, like so:

```
@interface GameKitHelper : NSObject <GKLeaderboardViewControllerDelegate>
```

Then you must implement the `leaderboardViewControllerDidFinish` method, which is used to simply dismiss the view and to forward the event to the delegate:

```
-(void) leaderboardViewControllerDidFinish:(GKLeaderboardViewController*)viewController
{
    [self dismissModalViewController];
    [delegate onLeaderboardViewDismissed];
}
```

Now there's a bit of behind-the-scenes magic going on. I've added a few helper methods to `GameKitHelper` that deal specifically with presenting and dismissing the various Game Kit view controllers making use of `cocos2d`'s root view controller. If you remember, `cocos2d` added a `RootViewController` class to all projects beginning with `cocos2d` version 0.99.5 and newer, and this controller will be used to display the Game Center views. Those methods are shown in Listing 14–12.

Listing 14–12. Using Cocos2d's Root View Controller to Present and Dismiss Game Kit Views

```
-(UIViewController*) getRootViewController
{
    return [UIApplication sharedApplication].keyWindow.rootViewController;
}

-(void) presentViewController:(UIViewController*)vc
{
    UIViewController* rootVC = [self getRootViewController];
```

```
    [rootVC presentModalViewController:vc animated:YES];  
}  
  
-(void) dismissModalViewController  
{  
    UIViewController* rootVC = [self getRootViewController];  
    [rootVC dismissModalViewControllerAnimated:YES];  
}
```

The `rootViewController` is a `UIWindow` property, and the main window used by `cocos2d` is the `keyWindow` property of the `UIApplication` class. There's only one catch: the current `cocos2d` project templates do not assign the new root view controller to the `UIWindow`'s property, so normally it's unavailable unless you expose it in the `AppDelegate` class itself. But there's a better way to make the `rootViewController` accessible to all classes without requiring to import the `AppDelegate` class. You'll have to open the project's `AppDelegate` class and add the following line in the `applicationDidFinishLaunching` method, just above the call to `[window makeKeyAndVisible]`:

```
window.rootViewController = viewController;  
[window makeKeyAndVisible];
```

After you've done this, the `cocos2d` `rootViewController` is accessible from anywhere through the `keyWindow.rootViewController` property.

The `GKLeaderboardViewController` will load the scores it needs automatically and present you with a view like the one in Figure 14–5.

NOTE: The built-in Game Kit views are presented in portrait mode. If your game uses a landscape orientation, the default views may not be ideal, because players will have to rotate their devices to view leaderboards, achievements, friends, and matchmaking views. In this case, you might have to consider writing your own user interface to display Game Center information.



Figure 14–5. *The Game Kit leaderboard view*

Achievements

In the Tilemap13 project, instead of showing the leaderboard, I'm calling the `GameKitHelper` `showAchievements` method in the `onScoresReceived` method to bring up the Achievements view (Listing 14–13).

Listing 14–13. *Showing the Achievements View*

```
-(void) showAchievements
{
    GKAchievementViewController* achievementsVC =
        [[[GKAchievementViewController alloc] init] autorelease];
    if (achievementsVC != nil)
    {
        achievementsVC.achievementDelegate = self;
        [self presentViewController:achievementsVC];
    }
}
```

This is very similar to showing the leaderboard view in Listing 14–11. Once more, the `GameKitHelper` class also has to implement the proper protocol, named `GKAchievementViewControllerDelegate`:

```
@interface GameKitHelper : NSObject <GKLeaderboardViewControllerDelegate, GKAchievementViewControllerDelegate>
```

The protocol requires the `GameKitHelper` class to implement the `achievementViewControllerDidFinish` method, which is also strikingly similar to the one used by the leaderboard view controller:

```
-(void) achievementViewControllerDidFinish:(GKAchievementViewController*)viewController  
{  
    [self dismissModalViewController];  
    [delegate onAchievementsViewDismissed];  
}
```

You can see an example of the achievements view in Figure 14–6, in which one achievement is already unlocked.



Figure 14–6. *The Game Kit achievements view*

So, what else can you do with achievements?

Obviously, you'll want to determine whether an achievement has been unlocked, and actually you'll want to report all the progress a player makes toward completing an achievement. For example, if the achievement's goal is to eat 476 bananas, then you would report the progress to Game Center every time the player eats a banana. In this example project, I'm simply checking for time elapsed, and then I report progress on the

PlayedForTenSeconds achievement. This is done in the TileMapLayer's update method, shown in Listing 14–14.

Listing 14–14. Determining Achievement Progress

```
-(void) update:(ccTime)delta
{
    totalTime += delta;
    if (totalTime > 1.0f)
    {
        totalTime = 0.0f;

        NSString* playedTenSeconds = @"PlayedForTenSeconds";
        GameKitHelper* gkHelper = [GameKitHelper sharedGameKitHelper];
        GKAchievement* achievement =
            [gkHelper getAchievementByID:playedTenSeconds];
        if (achievement.completed == NO)
        {
            float percent = achievement.percentComplete + 10;
            [gkHelper reportAchievementWithID:playedTenSeconds
                        percentComplete:percent];
        }
    }

    ...
}
```

Every time a second has passed, the achievement with the identifier PlayedForTenSeconds is obtained through GameKitHelper. If the achievement isn't completed yet, then its percentComplete property is increased by 10 percent, and the progress is reported through GameKitHelper's reportAchievementWithID method (Listing 14–15).

Listing 14–15. Reporting Achievement Progress

```
-(void) reportAchievementWithID:(NSString*)identifier percentComplete:(float)percent
{
    GKAchievement* achievement = [self getAchievementByID:identifier];
    if (achievement != nil && achievement.percentComplete < percent)
    {
        achievement.percentComplete = percent;
        [achievement reportAchievementWithCompletionHandler:^(NSError* error)
        {
            [self setLastError:error];
            [delegate onAchievementReported:achievement];
        }];
    }
}
```

To avoid unnecessary calls to the Game Center server, the achievement's percentComplete property is verified to actually be smaller than the percent parameter. Game Center does not allow achievement progress to be reduced and thus will ignore such a report. But if you can avoid actually reporting this to the Game Center server in the first place, you avoid an unnecessary data transfer. With the limited bandwidth available on mobile devices, every bit of data not transmitted is a good thing.

NOTE: Reporting an achievement's progress may fail for a number of reasons—for example, the device might have lost its Internet connection. Be prepared to save any achievements that couldn't be transmitted. Then retry submitting them periodically or when the player logs in the next time. Fortunately, the final `GameKitHelper` class contains additional code to cache achievements that failed transmission to the Game Center server. You'll find the final version of the `GameKitHelper` class in the code folder for this chapter.

This still leaves the question open: where do the achievements come from in the first place? They are loaded as soon as the local player signs in. To make this possible, I extended the block object used in `authenticateWithCompletionHandler` to call the `loadAchievements` method if there wasn't an error:

```
[localPlayer authenticateWithCompletionHandler:^(NSError* error)
{
    [self setLastError:error];
    if (error == nil)
    {
        [self loadAchievements];
    }
}];
```

The `loadAchievements` method uses the `GKAchievement` class's `loadAchievementsWithCompletionHandler` method to retrieve the local player's achievements from Game Center (Listing 14–16).

Listing 14–16. *Loading the Local Player's Achievements*

```
-(void) loadAchievements
{
    [GKAchievement loadAchievementsWithCompletionHandler:^(NSArray* loadedAchievements, NSError* error)
    {
        [self setLastError:error];
        if (achievements == nil)
        {
            achievements = [[NSMutableDictionary alloc] init];
        }
        else
        {
            [achievements removeAllObjects];
        }

        for (GKAchievement* achievement in loadedAchievements)
        {
            [achievements setObject:achievement
                               forKey:achievement.identifier];
        }

        [delegate onAchievementsLoaded:achievements];
    }];
}
```

Inside the block object, the achievements member variable is either allocated or has all objects removed from it. This allows you to call the `loadAchievements` method at a later time to refresh the list of achievements. The returned array `loadedAchievements` contains a number of `GKAchievement` instances, which are then transferred to the achievements `NSMutableDictionary` simply for ease of access. The `NSDictionary` class allows you to retrieve an achievement by its string identifier directly, instead of having to iterate over the array and comparing each achievement's identifier along the way. You can see this in the `getAchievementByID` method in Listing 14–17.

Listing 14–17. Getting and Optionally Creating an Achievement

```
-(GKAchievement*) getAchievementByID:(NSString*)identifier
{
    GKAchievement* achievement = [achievements objectForKey:identifier];

    if (achievement == nil)
    {
        // Create a new achievement object
        achievement = [[[GKAchievement alloc] initWithIdentifier:identifier] autorelease];
        [achievements setObject:achievement forKey:achievement.identifier];
    }

    return [[achievement retain] autorelease];
}
```

This is where you need to be careful. The `getAchievementByID` method creates a new achievement if it can't find one with the given identifier, assuming that this achievement's progress has never been reported to Game Center before. Only achievements that have been reported to Game Center at least once are obtained through the `loadAchievements` method in Listing 14–16. For any other achievement, you'll have to create it first. So, `getAchievementsByID` will always return a valid achievement object, but you'll only notice whether that achievement is really set up for your game when you try to report its progress to Game Center.

One curiosity here may be that I send a `retain` and an `autorelease` message to the returned achievement. This is to ensure that, in case the achievements dictionary is cleared by a call to `loadAchievements`, any achievement returned through `getAchievementsByID` remains valid. But the idea is to let `GameKitHelper` manage the achievements, so any other code shouldn't store achievement objects but just obtain them through `GameKitHelper` whenever they are needed.

You can also clear the local player's achievement progress. This should be done with great care and not without asking the player's permission. On the other hand, during development, the `resetAchievements` method in Listing 14–18 comes in handy.

Listing 14–18. Resetting Achievement Progress

```
-(void) resetAchievements
{
    [achievements removeAllObjects];

    [GKAchievement resetAchievementsWithCompletionHandler:^(
```



```

    ^(NSError* error)
    {
        [self setLastError:error];
        bool success = (error == nil);
        [delegate onResetAchievements:success];
    }];
}

```

Matchmaking

Moving on to the Tilemap14 project, we enter the realm of matchmaking—connecting players and inviting friends to play a game match together, that is. To start hosting a game and to bring up the corresponding matchmaking view, I’ve added a call to GameKitHelper’s `showMatchmakerWithRequest` method after the local player has been authenticated, as shown in Listing 14–19.

Listing 14–19. *Preparing to Show the Host Game Screen*

```

-(void) onLocalPlayerAuthenticationChanged
{
    GKLocalPlayer* localPlayer = [GKLocalPlayer localPlayer];
    if (localPlayer.authenticated)
    {
        GKMatchRequest* request = [[[GKMatchRequest alloc] init] autorelease];
        request.minPlayers = 2;
        request.maxPlayers = 4;

        GameKitHelper* gkHelper = [GameKitHelper sharedGameKitHelper];
        [gkHelper showMatchmakerWithRequest:request];
    }
}

```

A `GKMatchRequest` instance is created, and its `minPlayers` and `maxPlayers` properties are initialized, indicating that the match should have at least two and at most four players. Every match must allow for two players, obviously, and a peer-to-peer match can be played with up to four players. *Peer-to-peer networking* means that all devices are connected with each other and can send data to and receive it from all other devices. This is opposed to a server/client architecture, where all players connect to a single server and send and receive only to and from this server. In peer-to-peer networks, the amount of traffic generated grows exponentially, so most peer-to-peer multiplayer games are strictly limited to a very low number of allowed players.

NOTE: Game Center can connect up to 16 players, but only if you have a hosted server application to manage all matches using a client/server architecture. That requires a huge amount of work and know-how to set up and use properly, so I’ll leave it out of this discussion and focus only on peer-to-peer networking.

The `showMatchmakerWithRequest` method is implemented in a strikingly similar way to the code that brings up the leaderboard and achievement views, as Listing 14–20 shows.

Listing 14–20. Showing the Host Game Screen

```

-(void) showMatchmakerWithRequest:(GKMatchRequest*)request
{
    GKMatchmakerViewController* hostVC = [[[GKMatchmakerViewController alloc]
        initWithMatchRequest:request] autorelease];
    if (hostVC != nil)
    {
        hostVC.matchmakerDelegate = self;
        [self presentViewController:hostVC];
    }
}

```

Figure 14–7 shows an example matchmaking view, waiting for you to invite a friend to your match. You can also wait until Game Center finds an automatically matched player for your game, but since you’re currently developing the game, it’s rather unlikely that anyone but you will be currently playing it.

If you followed the leaderboard and achievement view examples, you’ll know that each required the `GameKitHelper` class to implement a protocol, and with matchmaking it’s no different. I also added `GKMatchDelegate` because we’re going to need it soon.

```

@interface GameKitHelper : NSObject <GKLeaderboardViewControllerDelegate,
    GKAchievementViewControllerDelegate, GKMatchmakerViewControllerDelegate,
    GKMatchDelegate>

```

The `GKMatchmakerViewControllerDelegate` protocol requires three methods to be implemented: one for the player pressing the Cancel button, one for failing with an error, and one for finding a suitable match. The latter deserves a mention:

```

-(void) matchmakerViewController:(GKMatchmakerViewController*)viewController
    didFinishMatch:(GKMatch*)match
{
    [self dismissModalViewController];
    [self setCurrentMatch:match];
    [delegate onMatchFound:match];
}

```

If a match was found, this match is set as the current match, and the delegate’s `onMatchFound` method is called to inform it about the newly found match.

Instead of hosting a match, you can also instruct Game Center to try to automatically find a match for you, as shown in Listing 14–21. If successful, the delegate receives the same `onMatchFound` message.

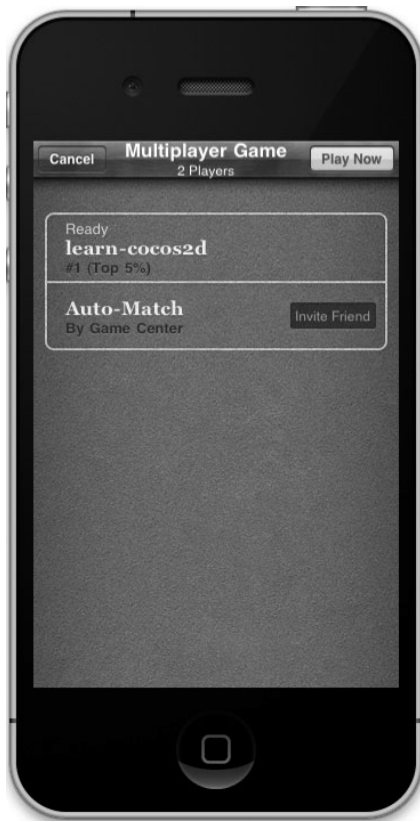


Figure 14–7. *The host game matchmaking view*

Listing 14–21. *Searching for an Existing Match*

```
-(void) findMatchForRequest:(GKMatchRequest*)request
{
    GKMatchMaker* matchmaker = [GKMatchMaker sharedMatchmaker];
    [matchmaker findMatchForRequest:request withCompletionHandler:^(
        GKMatch* match, NSError* error)
    {
        [self setLastError:error];

        if (match != nil)
        {
            [self setCurrentMatch:match];
            [delegate onMatchFound:match];
        }
    }];
}
```

While Game Center is searching for a match, you should give the user visual feedback, like an animated progress indicator, because finding a match can take several seconds or even minutes. That's where the `CCProgressTimer` class comes in handy, which I discussed in Chapter 5. You should also give your user a means to cancel the

matchmaking process, and if she does so, you should call the `cancelMatchmakingRequest` method:

```
-(void) cancelMatchmakingRequest
{
    [[GKMatchmaker sharedMatchmaker] cancel];
}
```

At this point, the match has been created, but all the players might not yet be connected to the match. As players join the game, the `match:didChangeState` method of the `GKMatchDelegate` protocol is called for each player connecting or disconnecting. Only when the `expectedPlayerCount` of the match has been counted down to 0 by the Game Kit framework should you start the match. The `GKMatch` object updates the `expectedPlayerCount` property automatically, as Listing 14–22 shows.

Listing 14–22. Waiting for All Players Before Starting the Match

```
-(void) match:(GKMatch*)match player:(NSString*)playerID
didChangeState:(GKPlayerConnectionState)state
{
    switch (state)
    {
        case GKPlayerStateConnected:
            [delegate onPlayerConnected:playerID];
            break;
        case GKPlayerStateDisconnected:
            [delegate onPlayerDisconnected:playerID];
            break;
    }

    if (matchStarted == NO && match.expectedPlayerCount == 0)
    {
        matchStarted = YES;
        [delegate onStartMatch];
    }
}
```

If at any time during your game a player drops out and the `expectedPlayerCount` property becomes greater than 0, you can call `addPlayersToMatch` to fill up the now empty space with a new player, as in Listing 14–23 (assuming that your game supports players joining a match in progress). Since there's no guarantee that a player will actually be found, you should not interrupt the game while `GKMatchmaker` is looking for a new player.

Listing 14–23. Adding Players to an Existing Match

```
-(void) addPlayersToMatch:(GKMatchRequest*)request
{
    if (currentMatch == nil)
        return;

    [[GKMatchmaker sharedMatchmaker] addPlayersToMatch:currentMatch
                                         matchRequest:request
                                         completionHandler:^(NSError* error)
    {
        [self setLastError:error];
    }];
}
```

```

        bool success = (error == nil);
        [delegate onPlayersAddedToMatch:success];
    }];
}

```

Sending and Receiving Data

Once all players are connected and the match has officially started, you can start sending and subsequently receiving data. The easiest way to do so is to send data to all players, as shown in Listing 14–24.

Listing 14–24. *Sending and Receiving Data*

```

-(void) sendDataToAllPlayers:(void*)data length:(NSUInteger)length
{
    NSError* error = nil;
    NSData* packet = [NSData dataWithBytes:data length:length];
    [currentMatch sendDataToAllPlayers:packet
                        withDataMode:GKMatchSendDataUnreliable
                        error:&error];
    [self setLastError:error];
}

-(void) match:(GKMatch*)match didReceiveData:(NSData*)data
fromPlayer:(NSString*)playerID
{
    [delegate onReceivedData:data fromPlayer:playerID];
}

```

The `sendDataToAllPlayers` method takes a void pointer as input and wraps it into an `NSData` object. You can send any data as long as you provide the correct length of that data. Typically, networked programs send structs like `CGPoint` (or any custom struct) to make this process easier, since you can then use `sizeof(myPoint)` to get the length (size in bytes) of such a data structure.

Also, to speed up transmission, most data is sent unreliably. Data that is sent frequently can especially be sent unreliably because if a packet ever gets lost, the clients simply have to wait for the next packet to arrive. If you do need every packet to arrive—for example, because it contains crucial information that is sent only once, then you should set the data mode to `GKMatchSendDataReliable`. This instructs GameKit to simply transmit the packet again if it could not be delivered. Since GameKit has to receive a return packet from clients to acknowledge that they received the packet, this adds additional traffic.

What data you should send and how often you should send it depend entirely on the game itself. The ground rule is to send as little as you can, as rarely as possible. For example, instead of transmitting each player's position every frame, you should send a packet for each movement action, because the movement in the tilemap game is always 32 pixels in one direction and done by a `CCMoveAction`. So, it's sufficient to send when the move should start and in which direction it should be, which saves a lot of traffic compared to sending each player's position every frame.

In the Tilemap16 project you'll get an introduction to sending and receiving packets over a network. The most important aspect to creating network packets is that the receiver must be able to identify the type of packet received by looking at a common header data. Typically, and this is also what the Apple documentation recommends, you will define C structs with a common struct field as the first entry for each packet. The NetworkPackets.h file defines the structs shown in Listing 14–25.

Listing 14–25. *Defining Network Packets as C Structs in NetworkPackets.h*

```
typedef enum
{
    kPacketTypeScore = 1,
    kPacketTypePosition,
} EPacketTypes;

typedef struct
{
    EPacketTypes type;
} SBasePacket;

// the packet for transmitting a score variable
typedef struct
{
    EPacketTypes type;

    int score;
} SScorePacket;

// packet to transmit a position
typedef struct
{
    EPacketTypes type;

    CGPoint position;
} SPositionPacket;
```

You'll see that all packet structs have the `EPacketTypes` type field, and it's the first field in each struct. This allows you to cast any packet to one of type `SBasePacket` to allow the receiver to inspect the packet type, and based on that, the receiver can then safely cast the struct to the actual packet.

Listing 14–26 shows an example of this. It's the `onReceivedData` method from the `TileMapScene` class.

Listing 14–26. *Receiving Packets and Determining Packet Type*

```
-(void) onReceivedData:(NSData*)data fromPlayer:(NSString*)playerID
{
    SBasePacket* basePacket = (SBasePacket*)[data bytes];

    switch (basePacket->type)
    {
        case kPacketTypeScore:
        {
            SScorePacket* scorePacket = (SScorePacket*)basePacket;
            CLOG(@"\tscore = %i", scorePacket->score);
            break;
        }
    }
}
```

```

    }
    case kPacketTypePosition:
    {
        SPositionPacket* positionPacket = (SPositionPacket*)basePacket;

        if (playerID != [GKLocalPlayer localPlayer].playerID)
        {
            CCTMXTiledMap* tileMap = ←
                (CCTMXTiledMap*)[self getChildByTag:TileMapNode];
            [self centerTileMapOnTileCoord:positionPacket->position
                tileMap:tileMap];
        }
        break;
    }
    default:
        CCLOG(@"unknown packet type %i", basePacket->type);
        break;
}
}
}

```

This code first casts the received data bytes to a pointer to an `SBasePacket` struct. If you look it up in Listing 14–25, you’ll notice that’s the struct that contains only the type field. Since we have declared that all packets must have this field at its first entry, any packet can be safely cast to `SBasePacket`. The switch statement inspects the type, and depending on the network packet, further processing is done—but not without casting the packet to the actual packet type. For example, if the `basePacket->type` is `kPacketTypeScore`, the `basePacket` will be cast to an `SScorePacket` to allow the code to access the score field.

TIP: When checking packets, it’s a good idea to add the default option. You will frequently add new packets, and from time to time you’ll forget to handle this particular packet type on the receiving end. So, logging this as an error or even throwing an exception is recommended. Otherwise, you might see bugs in your app that might be hard to track down.

Actually sending the packets is relatively easy and follows the same principle. You first create a new variable with one of the packet structs as its data type. Then you fill in each field of the struct and pass the struct to the `GameKitHelper` method `sendDataToAllPlayers`.

In Listing 14–27, the packets are created on the stack. You don’t need to allocate memory because Game Kit will make a copy of the struct and thus take over the memory management of the packet. Since `sendDataToAllPlayers` required a pointer, the packet is prefixed with the reference operator (ampersand character) `&packet` to denote that the packet variable’s address is passed instead of the packet itself.

Listing 14–27. Sending Packets via GameKitHelper

```

// TM16: send a bogus score (simply an integer increased every time it is sent)
-(void) sendScore
{
    if ([GameKitHelper sharedGameKitHelper].currentMatch != nil)

```

```

    {
        bogusScore++;

        SScorePacket packet;
        packet.type = kPacketTypeScore;
        packet.score = bogusScore;

        [[GameKitHelper sharedGameKitHelper] sendDataToAllPlayers:&packet
                                                length:sizeof(packet)];
    }
}

// TM16: send a tile coordinate
-(void) sendPosition:(CGPoint)tilePos
{
    if ([GameKitHelper sharedGameKitHelper].currentMatch != nil)
    {
        SPositionPacket packet;
        packet.type = kPacketTypePosition;
        packet.position = tilePos;

        [[GameKitHelper sharedGameKitHelper] sendDataToAllPlayers:&packet
                                                length:sizeof(packet)];
    }
}

```

The most important part of sending packets is to make sure you set the right packet type. If you assign the wrong packet type, the receiver won't know what to do with the packet. It might mistake it for a different type of packet, causing a crash because the receiver might try to access a nonexistent field. Or the receiver might simply work with unrelated data, causing all kinds of bugs. Imagine the score becoming the player's position, or vice versa. To avoid these kinds of issues, particularly if you have many different packet types, it may be helpful to create methods like `createPositionPacket` and `createScorePacket`, which you call with all the required parameters for the packet while the method itself fills in the correct packet type.

In Figure 14–8 you can see the `Tilemap16` project in action. Every time the player is moved on the iPhone, a position packet is sent over the network. The iPad is connected to the current match, receives the position packet, and moves the player character accordingly.

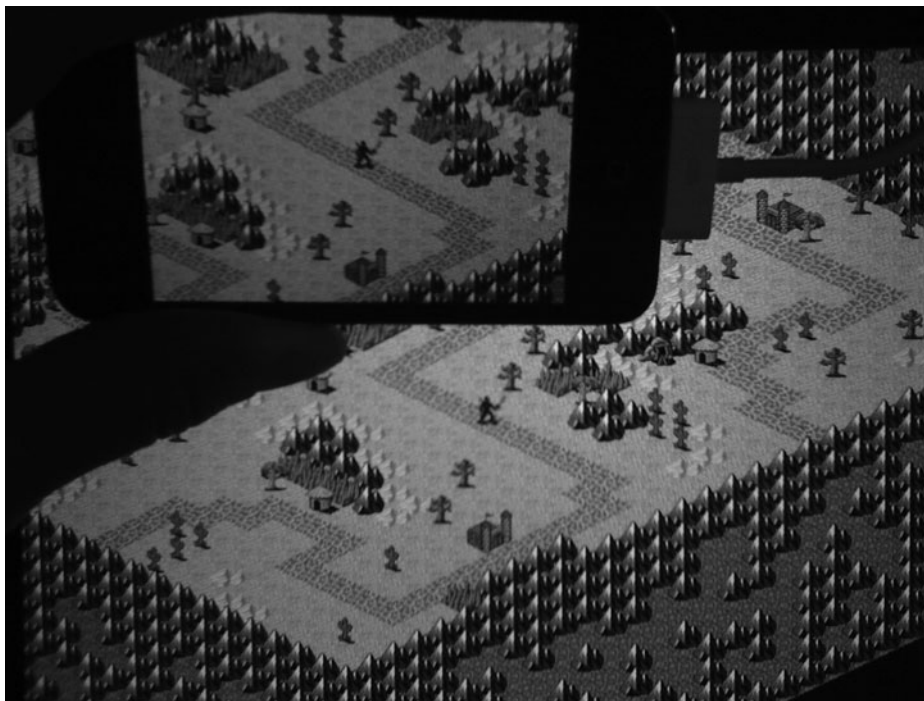


Figure 14–8. *If the player moves on the iPhone, the iPad will update its view from the position packet it received.*

Summary

I hope this chapter and the provided `GameKitHelper` class help you get comfortable with Game Center programming. Sure, network programming is no easy task, but I've laid a lot of the groundwork for you, and even block objects are no longer foreign territory for you. In particular, the checklist of tasks to enable Game Center support for your game should help you avoid a lot of the initial pitfalls faced by developers.

Over the course of this chapter, you've become comfortable using the leaderboard and achievement features of Game Center. Those alone bring your game to a new level. And with the user interface provided by Game Center, you don't even have to write your own user interface to display leaderboards and achievements.

I then introduced you to the matchmaking features of Game Center, which allow you to invite friends to join your game, find random players on the Internet, and allow them to send and receive data.

With this chapter, I already departed a little from pure `cocos2d` programming; in fact, you can apply what you just learned about Game Center to any iOS app. In the next chapter, I'll tackle another subject that's not pure `cocos2d` game programming either but frequently asked for: mixing UIKit views with `cocos2d`.

Cocos2d with UIKit Views

For most iOS developers there's a clear dividing line: if you want to program "regular" apps with no or little multimedia content, you'll be using Cocoa Touch and its UIKit framework to create the iPhone's and iPad's native user interfaces.

On the other hand, if you want to develop iOS games and multimedia applications, you want to use cocos2d and have little incentive to use anything but CCSprite and CCMenu to create your game's scenes and user interfaces.

A great number of developers are experienced only in either environment, and they'll often find it confusing to cross the border from Cocoa Touch to cocos2d, and vice versa. In almost all these cases, the programmers want to combine the best of both worlds, leveraging their existing knowledge of either Cocoa Touch or cocos2d to create hybrid applications.

Since Cocoa Touch and cocos2d work fundamentally differently and require a different mind-set, it's usually not as straightforward to create such hybrids. This chapter will help you transition in both directions. You'll learn how to add Cocoa Touch views and features to a cocos2d application; at the same time, you'll also learn how you can plug in cocos2d to an existing Cocoa Touch application.

What Is Cocoa Touch?

Cocoa Touch is the name of the application programming interface (API) used to create iOS applications. It is of course inspired by Cocoa, the API for programming Mac OS X applications.

Cocoa Touch is comprised of several frameworks such as Core Animation, Core Data, Map Kit, Store Kit, and Web Kit, just to name a few. But strictly speaking, even cocos2d is a Cocoa Touch library because the OpenGL ES framework, as well as Core Audio, OpenAL, and AV Foundation (AV stands for Audio/Video) frameworks that cocos2d is built on, are part of Cocoa Touch.

It is no wonder then that most programmers refer specifically to UIKit when they are asking about how to integrate Cocoa Touch views into cocos2d. UIKit is the framework

that provides programmers with the native iOS controls and views that are used to build the graphical user interfaces (GUIs) of iOS applications. At the same time, other frameworks such as iAd, Web Kit, Game Kit, and Map Kit add specialized views, and they are mostly built with the GUI elements provided by UIKit.

So, technically, even if programmers discuss integration issues of Game Center with cocos2d, they will often refer to the views as being part of UIKit, even though the actual view is provided by Game Kit or Web Kit, for example. For reference, here are the cocos2d forum topics tagged with UIKit and Cocoa Touch, respectively:

<http://www.cocos2d-iphone.org/forum/tags/uikit>

<http://www.cocos2d-iphone.org/forum/tags/cocoa-touch>

Using Cocoa Touch and cocos2d Together

Before we get to work with the code in this chapter, I want to step back for a moment and discuss why one would want to mix cocos2d with Cocoa Touch (UIKit views), what limitations there are, and what the differences between Cocoa Touch and cocos2d are.

Why Mix Cocoa Touch with cocos2d?

There are many good reasons to mix Cocoa Touch and cocos2d. Essentially, they all boil down to a better user experience or faster development.

For one, if you're a cocos2d programmer, you'll be adding some Cocoa Touch views to your application sooner or later, most commonly to generate some revenue with iAd or if you're writing a Game Center-enabled game. But you also might want to provide the users with a native-looking user interface, which can be designed efficiently with Interface Builder and later skinned with textures that maintain the game's look and feel so that your user interface doesn't look like the Settings app. A great example of such a skinned app is Carcassone; you'll have to look twice to see that its user interface is actually entirely made with UIKit views.

While you can make reasonably good user interfaces with cocos2d, there's simply a much greater variety of already existing controls available from UIKit that cocos2d doesn't provide. And the occasional reimplementation in cocos2d always lacks in feel and features. Sliders, on/off toggle buttons, navigation views, and tab bars can all be highly useful in designing your game's user interface, especially in those games or parts of the game where performance is not of the utmost importance.

If you are a Cocoa Touch programmer and you need some multimedia content in your game, it's much easier to rely on cocos2d to do that job and do it with high performance rather than programming it directly with OpenGL ES. After all, cocos2d shields you from OpenGL ES and provides an interface that's much easier to use.

Cocoa Touch does provide powerful graphical frameworks like Core Graphics and Core Animation. But they suffer from a major disadvantage: they are often not fast enough for

real-time games. They were designed to display and animate user interface elements, not games.

Limitations of Mixing Cocoa Touch with cocos2d

When designing your app or game that mixes Cocoa Touch views with the cocos2d view, you should be aware of some limitations. Most obviously, UIKit views are not designed for high performance, so you may notice a drop in performance, especially if you use UIKit views in fast-paced games and during game play.

For example, it is more favorable for performance to rely on CCLabelBMFont to display the score during game play than using a UITextField for the same purpose. And likewise, you should prefer to use CCMenu for the in-game pause menu button rather than using a UIButton. In menu screens, however, those performance considerations are usually not a problem, and you can see improved productivity from being able to use Interface Builder to create your menu screens.

Mixing UIKit views in a cocos2d app while supporting autorotation will also affect performance, quite severely on first- and second-generation devices. Just allowing all views to be possibly rotated can have your framerate drop to below 60 fps right away, without even rendering anything! That means you might want to leave autorotation support disabled specifically for the older devices, which is the default setting in the cocos2d project templates.

You should also be aware that any UIKit view can be either in front of the entire cocos2d view or entirely behind it. You can't have a UIKit view that is in front of some of the cocos2d scene's sprites, labels, effects, and so on, while at the same time being behind other cocos2d sprites, labels, effects, or other nodes. In other words, you cannot sandwich a UIKit view between two or more cocos2d nodes.

You can do the opposite, however, although with some limitations. You can sandwich the cocos2d view: UIKit views in the background, then a transparent cocos2d view, and then some more UIKit views in the foreground. This approach requires only a little more work setting up the view hierarchy and making the cocos2d view transparent. Imagine playing a full-motion video in the background, over which you draw cocos2d sprites, and the rest of the user interface is made up of UIKit views.

But touch input remains a problem: either the UIKit views and not the cocos2d view will receive input or those UIKit views added to the cocos2d view and the cocos2d view itself will receive input but not the views in the background. This has to do with the fact that the cocos2d view receives all touches on the screen simply because it occupies the entire screen. So, you need to write additional code to process the touches on the cocos2d view and then decide whether the cocos2d view should forward the touches, for example if the user didn't touch any of the cocos2d sprites currently displayed on screen.

Allowing all views to receive input is possible, and I'll provide you with a basic solution later in this chapter. But it is up to you to improve and adapt it for your own needs. Depending on your needs, the necessary code changes may actually be substantial and

challenging in order to fully support UIKit views both in front of the cocos2d view and behind it and have all views reacting properly to touch input.

How Is Cocoa Touch Different from cocos2d?

Let's take a look at the major differences of Cocoa Touch programming compared to working with cocos2d. One difference is the Model-View-Controller pattern common to Cocoa Touch applications but essentially missing from cocos2d. And then you also have to consider the differences caused by cocos2d's OpenGL ES view since it behaves differently in some aspects than a regular UIView.

The Model-View-Controller Pattern

Probably the first and biggest difference for programmers coming from a Cocoa Touch background is that cocos2d does not strictly adhere to the Model-View-Controller (MVC) pattern, which is commonplace in Cocoa and Cocoa Touch.

The MVC pattern divides the programming tasks into the three subsets: model, view, and controller. The model contains any algorithms that run behind the scenes and maintains the state of the world; in essence, the model represents knowledge. The view is the visual representation of the model and renders the current state of the world based on the model data. And the controller essentially provides a means for the user to interact with the world through user input, but it is also used to react to other external events such as receiving data over the network. The model, view, and controller are each separate classes to decouple the user interface from business (or game) logic.

In games, the MVC pattern can be applied, and many have attempted to do so with cocos2d. You'll find a good number of articles on the subject if you search for *cocos2d mvc*, and my personal favorite treatment of the subject is this two-part article by Bartek Wilczyński:

<http://xperiened.com.pl/blog/how-to-implement-mvc-pattern-in-cocos2d-game>

<http://xperiened.com.pl/blog/how-to-implement-mvc-pattern-in-cocos2d-gamepart-2>

For Cocoa Touch programmers, the fact that cocos2d does not follow the MVC pattern may come as a culture shock. But it's one you can work around. On the other hand, as a cocos2d programmer, you likely won't even notice that you're using MVC because the entire Cocoa Touch framework is designed for the MVC pattern. You'll happily use the controllers and views provided to you, and you'll find no problem adding the logic and algorithms (the model) into either controller or view, or both. That is also a valid pattern, albeit more tightly coupled and less maintainable in large projects.

Cocos2d's View Uses OpenGL ES

Instead of relying on UIKit for displaying its graphics, cocos2d creates an OpenGL ES view. This means cocos2d has more direct access to graphics resources and can render

its view much faster. On the other hand, it does lose some of the automatic features of UIKit applications, like autorotation.

Of course, behind the scenes, all UIKit views are also rendered by OpenGL ES; there's just a lot more stuff going behind the scenes that is needed for graphical user interfaces but is essentially a waste of performance if you want to make games. You may remember the very early games that were written entirely with UIKit, Core Graphics, and Core Animation? If not, good for you. They were often slow and unresponsive.

One immediately noticeable difference between Cocoa Touch and cocos2d is how autorotation is handled. Later in this chapter you'll learn how cocos2d implements autorotation to rotate both OpenGL ES content as well as UIKit views. Fortunately for us, this issue was solved eventually by the cocos2d engine so you can rotate both UIKit and cocos2d views, but there are some performance drawbacks. And in some cases, you still need to consider the differences in coordinate systems used by UIKit and OpenGL ES, especially if views are being rotated.

And since cocos2d is programmed to interact directly with the graphics hardware, it uses its own hierarchy of displaying graphical elements. In cocos2d that's the `CCNode` hierarchy where you can add any `CCNode`-based class to any other `CCNode`, with a `CCScene` as the very first element in that hierarchy. The UIKit framework, on the other hand, operates with a view hierarchy where you add `UIView`-based classes to another, often with a `UIWindow` as the topmost element. Both view hierarchies are incompatible, so you can't add a `UIView` to a `CCNode`, and vice versa. This is noticeable when you change from one `CCScene` to another using a `CCTransitionScene`. While the cocos2d nodes all move aside, the UIKit views will remain fixed in place unless you also move them separately and in sync with the cocos2d animation. It's actually a good idea to avoid this kind of situation in the first place.

Alert: Your First UIKit View in cocos2d

The simplest and most straightforward example for using a UIKit view with cocos2d is found in the example project `CocosWithCocoa01`. It displays a `UIAlertView` on top of the cocos2d scene created from the default cocos2d project template. To re-create the project from scratch, open Xcode and go to **File** ► **New** ► **New Project** to bring up the New Project dialog. In that dialog, select `cocos2d` under the iOS list and create the `cocos2d` project.

Let's modify the `HelloWorldLayer` class to display a `UIAlertView`. The interface in `HelloWorldLayer.h` needs only one small addition; namely, the `HelloWorldLayer` class needs to support the `UIAlertViewDelegate` protocol:

```
@interface HelloWorldLayer : CCLayer <UIAlertViewDelegate>
{
}
```

All other changes are made to the `HelloWorldLayer.m` implementation file. At the top, you first need to declare the `PrivateMethods` interface to avoid the `may not respond to`

selector compiler warning where the `addSomeCocoaTouch` method is called before the actual implementation of the method:

```
#import "HelloWorldLayer.h"
```

```
@interface HelloWorldLayer (PrivateMethods)
-(void) addSomeCocoaTouch;
@end
```

The `init` method of the `Hello World` sample is modified to use a uniformly colored background, just so you see the visual effect of the `UIAlertView`, and to call the `addSomeCocoaTouch` method. It still retains the `Hello World CCLabelTTF`, but I moved it from its center position:

```
-(id) init
{
    if ((self = [super init]))
    {
        // color background to make the UIAlertView "darkening" effect noticeable
        glClearColor(0.1f, 0.3f, 0.7f, 1.0f);

        CCLabelTTF* label = [CCLabelTTF labelWithString:@"Hello Cocos2D!"
                                                             fontName:@"Marker Felt"
                                                             fontSize:54];

        CGSize size = [[CCDirector sharedDirector] winSize];
        label.position = CGPointMake(size.width / 2, size.height / 6);
        [self addChild:label];

        [self showHelloWorldAlertView];
    }
    return self;
}
```

The `addSomeCocoaTouch` allocates a `UIAlertView` with a title, two buttons, and the message text `Hello Cocoa Touch!` For a delegate, you'll be using `self` now that you've added the `UIAlertViewDelegate` protocol to the `HelloWorldLayer` class.

Finally, you can show the alert view. Since showing the alert view retains the alert view behind the scenes, you can immediately send the release message to the `alertView` without risking a crash. Listing 15–1 shows the resulting code.

Listing 15–1. A `UIAlertView` Is Created and Added to `cocos2d's openGLView (EAGLView Class)`

```
-(void) showHelloWorldAlertView
{
    UIAlertView* alertView = [[UIAlertView alloc] initWithTitle:@"UIAlertView Example"
                                                            message:@"Hello Cocoa Touch!"
                                                            delegate:self
                                                            cancelButtonTitle:@"Well"
                                                            otherButtonTitles:@"Done", nil];

    [alertView show];
    [alertView release];
}
```

TIP: It is not necessary to add a UIAlertView to another view. This makes it very straightforward to create UIAlertView messages. The only drawback is that UIAlertView will always be drawn above everything else, and it will swallow all touches as long as it is displayed. No amount of sending views to back or reordering the view hierarchy will change that. If you need a simple solution for a pause menu, UIAlertView is your cheap and dirty friend, especially during development. But keep in mind that while touches are disabled, you'll still be receiving acceleration events, which you'll have to turn off or ignore while the UIAlertView is shown.

The HelloWorldLayer class will receive all events from the UIAlertView and can respond to them by simply implementing one or more of the UIAlertViewDelegate methods. For this example, I decided to respond to the `didDismissWithButtonIndex` message (see Listing 15-2), which is sent whenever the user taps a button, which always dismisses the UIAlertView regardless of which button was tapped. Another CCLabelTTF, with a string and color that depend on the `buttonIndex`, is added to the cocos2d scene at a random position every time the alert view is dismissed.

Listing 15-2. Responding to the UIAlertView `didDismissWithButtonIndex` Message

```
-(void) alertView:(UIAlertView*)alertView
didDismissWithButtonIndex:(NSInteger)buttonIndex
{
    NSString* message = @"Well";
    ccColor3B labelColor = ccYELLOW;
    if (buttonIndex == 1)
    {
        message = @"Done";
        labelColor = ccGREEN;
    }

    CCLabelTTF* label = [CCLabelTTF labelWithString:message
                                                                fontName:@"Arial"
                                                                fontSize:32];

    CGSize size = [[CCDirector sharedDirector] winSize];
    label.position = CGPointMake(CCRANDOM_0_1() * size.width,
                                CCRANDOM_0_1() * size.height);
    label.color = labelColor;
    [self addChild:label];

    // keep the alert view alive by bringing it up again
    [self showHelloWorldAlertView];
}
```

The `addSomeCocoaTouch` is called again whenever the alert view has been dismissed, so the alert view will keep showing up again, allowing you to add another label to the cocos2d view. You can see the result in Figure 15-1.



Figure 15–1. A UIAlertView is displayed over the cocos2d view.

Embedding UIKit Views in a cocos2d App

Next you'll be embedding more commonly used UIKit views in cocos2d. One of the simplest and most common is the UITextField, which you'll add on top of cocos2d, as you've done before. It gets more complicated when you move it to the background of cocos2d, which requires making the cocos2d view transparent.

Finally, I'll show you how you can add your Interface Builder views into a cocos2d app, instead of creating the views programmatically, and I'll also go into detail on how the RootViewController class and autorotation works.

Adding Views in Front of the cocos2d View

In the CocosWithCocoa03 project, I've added UITextField views on top of the cocos2d view. The UITextField is a simple text entry box that automatically brings up the iPhone keyboard when you tap it. I removed the UIAlertView from the addSomeCocoaTouch method in the HelloWorldLayer class implementation and added the UITextField:

```
-(void) addSomeCocoaTouch
{
    // regular text field with rounded corners
    UITextField* textField = [[UITextField alloc] initWithFrame:CGRectMake(40, 20, 200, 24)];
    textField.text = @"Regular UITextField";
    textField.borderStyle = UITextBorderStyleRoundedRect;

    // get the cocos2d view (it's the EAGLView class which inherits from UIView)
    UIView* glView = [CCDirector sharedDirector].openGLView;

    // add the text field view to the cocos2d EAGLView
    [glView addSubview:textField];

    // after that it's safe to release the textField
    [textField release];
}
```

It's important to note that the process of programmatically creating UIView classes is quite similar to how you create the UITextField. You pick the desired class derived from UIView and then call alloc and initWithFrame. Most UIView controls can be created by just providing a frame rectangle. However, you will usually have to set some properties afterward to configure the control; in this example, I've set the textField to use the rounded style as well as setting the initial text.

CAUTION: The frame rectangle is where many programmers first notice the different coordinate systems of cocos2d nodes and UIView classes. Whereas in cocos2d the origin (0, 0) is at the lower-left corner of the screen, the origin for UIView classes is at the upper-left corner of the screen. This means that the UITextField is actually 20 pixels below the top border of the screen and not 20 pixels above the bottom border. You will have to keep this in mind when working with UIView classes.

Since the UITextField, like most other UIView classes, does not have a show method, you need some other way to attach it to the view hierarchy. Since the cocos2d view is the EAGLView class, which in turn inherits from UIView, you can simply add the UITextField to the cocos2d glView as a subview. The CCDirector has an OpenGLView property, which allows you to access the cocos2d view and then call the addSubview method on it to add the textField. By default this adds the view on top of the cocos2d view.

If you try this now, you'll see a text field on your scene, and when you tap the text field, the iPhone keyboard comes up, and you can start editing text. No extra code needed. Except, the keyboard won't go away anymore.

This is by design because the Return key might be a valid key to start a new line rather than to stop editing. So, you need some way to dismiss the keyboard. To do so, open the HelloWorldLayer header file and replace the UIAlertViewDelegate protocol with the UITextFieldDelegate protocol like so:

```
@interface HelloWorldLayer : CCLayer <UITextFieldDelegate>
{
}
```

Doing so allows the HelloWorldLayer class to respond to UITextFieldDelegate methods like textFieldShouldReturn. For this to work, you must assign the HelloWorldLayer class instance to the UITextField by assigning self to the delegate property. Add the highlighted line at the end of the initialization block of the UITextField:

```
// regular text field with rounded corners
UITextField* textField = [[UITextField alloc] initWithFrame:CGRectMake(40, 20, 200, 24)];
textField.text = @" Regular UITextField";
textField.borderStyle = UITextBorderStyleRoundedRect;
textField.delegate = self;
```

Most UIKit views have this delegate method and an accompanying delegate protocol. So, if you ever wonder how you can respond to events of a certain UIView, it's done by

implementing the class's delegate protocol and responding to the appropriate message. Of course, one very common and repeated mistake you'll make (I know I do) is to forget to actually assign the delegate or assign the correct one. So, whenever a delegate method isn't being called, you should check whether you actually set the (right) class instance as the view's delegate.

In our case, the `textFieldShouldReturn` message of the `UITextFieldDelegate` protocol is sent whenever the user taps the Return key on the iPhone keyboard:

```
-(BOOL) textFieldShouldReturn:(UITextField *)textField
{
    // dismiss the keyboard
    [textField resignFirstResponder];

    // if the text is empty, remove the text field
    if ([textField.text length] == 0)
    {
        [textField removeFromSuperview];
    }

    return YES;
}
```

By sending the `resignFirstResponder` message to the `textField`, the keyboard will be dismissed. Simply as an exercise on how to remove a `UIView` from the `cocos2d` view, I've added a condition that sends the `removeFromSuperview` message to the `textField` if the `textField` is empty when the user presses Return. Notice how this entire method does not care which `UITextField` is sending the message, nor does it care where in the view hierarchy the `textField` was added. You'll take advantage of that next by adding another `UITextField`.

If you try what you have so far, you'll notice that the keyboard is dismissed when you press Return, and if you have deleted all characters from the text field, the entire text field will vanish.

TIP: Keep in mind that if it is possible that your scene changes while the user is editing text in a `UITextField`, you would have to manually send the `resignFirstResponder` message to all text fields in order to dismiss the keyboard. Otherwise, the keyboard may remain visible during and after the scene change, and the user won't be able to dismiss it anymore. To avoid this situation, it is preferable to also respond to the `textFieldDidBeginEditing` message and use that to temporarily disable any buttons or events that could change the current scene. Then reenable the buttons or events when you receive the `textFieldShouldReturn` message.

Skinning the UITextField with a UIImage

No, I'm not going to peel off the text field's skin! If you haven't heard the term *skinning* before, it basically means adding (or changing) a texture to a user interface control or

view. Essentially you'll change the native look of the control or view and replace it with your own.

In Listing 15–3 you'll be adding some more code at the bottom of the `addSomeCocoaTouch` method in order to create a second `UITextField` that uses a texture as background.

Listing 15–3. Skinning a UITextField View

```
-(void) addSomeCocoaTouch
{

    // text field that uses an image as background (aka "skinning")
    UITextField* textFieldSkinned = [[UITextField alloc] initWithFrame:CGRectMake(40, 60, 200, 24)];
    textFieldSkinned.text = @"With background image";
    textFieldSkinned.delegate = self;

    // load and assign the UIImage as background of the text field
    NSString* file = [CCFileUtils fullPathFromRelativePath:@"background-frame.png"];
    UIImage* image = [[UIImage alloc] initWithContentsOfFile:file];
    textFieldSkinned.background = image;

    [glView addSubview:textFieldSkinned];
    [textFieldSkinned release];
    [image release];
}
```

Creating the `UITextField` should be familiar, and you also add `self` as a delegate of the text field. The code that dismisses the keyboard and removes the text field when it is empty (see Listing 15–2) now works for this new `UITextField` as well.

The next part is where `cocos2d` users with little or no Cocoa Touch programming experience may have problems. You can't just add a `CCSprite` or the sprite's texture to a `UIView`. You do need a `UIImage` class for skinning Cocoa Touch views, which you can then comfortably create via `initWithContentsOfFile`. Or not? Well, the returned `UIImage` might be `nil`.

It turns out that `cocos2d` allows you to use file names without specifying a path because internally it adds the path to the application's bundle file for you. This full path to a bundle file looks something like this on an iOS device, and the path will be different when running the app in the simulator or on another device:

```
/var/mobile/Applications/ lots of letters /CocosWithCocoa.app/background-frame.png
```

Since `UIImage` and most other Cocoa Touch classes dealing with files expect the full path to the file, you will have to use the `CCFileUtils` class method `fullPathFromRelativePath` in order to create an `NSString`, which contains the full path to the file in the app bundle. Then you get a valid `UIImage`, and you can assign it to the `background` property. You can see what this looks like in Figure 15–2.



Figure 15-2. Two UITextField views with the iPhone keyboard raised

TIP: The background image of a UIView will always be scaled and stretched to fit the UIView's frame. This will often blur or otherwise distort the texture. To avoid that, you should design background images of UIViews to the exact dimensions of the UIView. Alternatively, design the texture for the largest possible size of the UIView so that even if it is scaled, it is scaled down and doesn't lose as much image quality compared to upscaling the texture.

Adding Views Behind the cocos2d View

What if you wanted to add a UIView behind the cocos2d view? For example, say you wanted to show the video camera feed in the background for an augmented reality app. You'll learn how to do so in Chapter 16, where I'm using the solution conveniently provided by cocos3d to create an augmented reality demo app.

There are a few things that you need to change to allow UIKit views in the background. You'll find these code changes in the CocosWithCocoa04 project.

Improving the View Hierarchy

First, the view hierarchy needs to be changed so that the UIKit views are not a subview of the cocos2d view, respectively; the UIKit views can't have the cocos2d view as their superview. That would always render them in front of the cocos2d view. The other required change is to make the cocos2d view transparent.

Let's start by setting up the view hierarchy so that you have some way to add views to the hierarchy so that they're behind the cocos2d view. For that change, you'll have to open the AppDelegate.m file and look for the following lines in the `applicationDidFinishLaunching` method:

```
// make the OpenGLView a child of the view controller
[viewController setView:glView];

// make the View Controller a child of the main window
[window addSubview: viewController.view];
```

```
[window makeKeyAndVisible];
```

As you can see, the cocos2d `glView` is added directly to the `UIWindow`. Well, it's not quite directly because it is first set to be the `viewController` view. The `viewController` is an instance of the `RootViewController` class provided by cocos2d to handle autorotation (more on that later). Then the `viewController.view`, which is the same as the `glView` now, is added as subview to the window.

This setup means that any view that should be rendered before the cocos2d view will be a subview of the `UIWindow`. There's only one problem with that: you lose the autorotation feature provided by the `RootViewController` class. All views added to the `UIWindow` will display in portrait orientation by default, and they won't rotate if you change the device orientation.

To overcome this problem, we need to introduce another view in between the `UIWindow` and the cocos2d view. A simple `UIView` does the job, and its only purpose is to be a dummy view that will handle the autorotation through the `RootViewController` class and will contain all the subviews, both the cocos2d view and any UIKit views:

```
// add a dummy UIView to the view controller
UIView* dummyView = [[UIView alloc] initWithFrame:[window bounds]];
[viewController setView:dummyView];
[dummyView addSubview:glView];
[dummyView release];
```

```
// make the View Controller's view a child of the main window
[window addSubview:viewController.view];
```

```
[window makeKeyAndVisible];
```

I think a picture explains best how this changes the view hierarchy. Take a look at the before and after diagram in Figure 15–3 and notice that previously you've been adding UIKit views as subviews of the cocos2d view. Now with the introduction of the dummy view, all views are at the same level so that you can have `UIViews` before and after the cocos2d view. Moreover, this gives you the ability to change the view order at runtime.

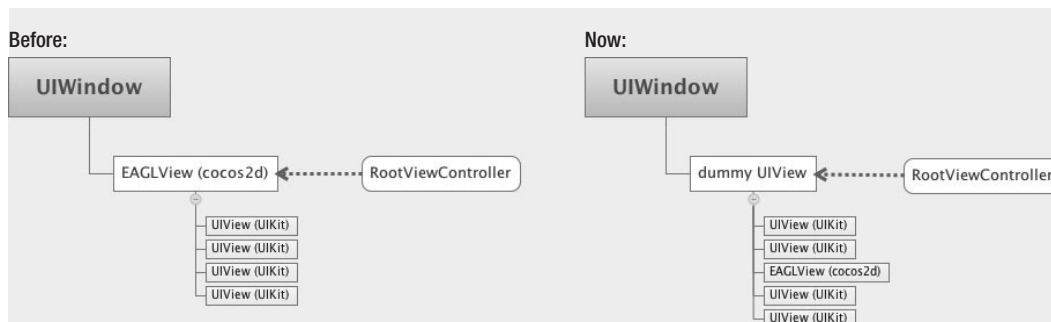


Figure 15–3. An illustration of the change made to the view hierarchy by introducing the dummy `UIView`

Moving the UITextField to the Background

Adding our UITextField views to the dummy view is straightforward. For this example, I skip over the UITextField initialization code in the `addSomeCocoaTouch` method because it doesn't change. The only change is in adding the UITextField views as subviews of the `dummyView`:

```
-(void) addSomeCocoaTouch
{
    // get the cocos2d view (it's the EAGLView class which inherits from UIView)
    UIView* glView = [CCDirector sharedDirector].openGLView;
    // The dummy UIView is the superview of the glView
    UIView* dummyView = glView.superview;

    // UITextField initialization code omitted

    // add the text fields to the dummy view
    [dummyView addSubview:textField];
    [dummyView addSubview:textFieldSkinned];

    // UITextField release code omitted
}
```

You can simply access the newly introduced dummy view because you've already added the cocos2d `glView` to it, so that makes it the `glView.superview`. The *superview* is the Cocoa term for what you would call the parent node in the cocos2d node hierarchy. You can then add the text fields to the `dummyView` instead of the `glView`.

However, you won't notice a difference if you run the project now. Since you've added the text fields after the cocos2d view, they're automatically rendered after the cocos2d view by default. This is the same behavior as in the cocos2d node hierarchy. To actually move the text fields to the back, we can either send the `sendSubviewToBack` message to all of them or, more easily, send the `bringSubviewToFront` message to the `glView`, like so:

```
// send the cocos2d view to the front so it is in front of the other views
[dummyView bringSubviewToFront:glView];
```

Note that the `sendSubviewToBack` and `bringSubviewToFront` messages are sent to the view that contains the view that should be sent to the back or front. In this case, that's the `dummyView`. If you run the project now, you will see a difference. But you won't be seeing the text fields anymore. What's the problem?

Making the cocos2d View Transparent

By default, the cocos2d view is completely opaque. Anything behind the `glView` will be obstructed because the cocos2d `EAGLView` is filled each frame with an opaque clear color. It also has its `opaque` property set to `YES`. This is easily remedied by adding the following lines to the `addSomeCocoaTouch` method:

```
// make the cocos2d view transparent
glClearColor(0.0, 0.0, 0.0, 0.0);
glView.opaque = NO;
```

The opaque flag is set to NO, and the `glClearColor` is all zero. The latter is not strictly necessary; it is sufficient to reduce the alpha channel (fourth parameter) so that the clear color is at least somewhat transparent. But for this example and in most cases, you don't want the background to be tinted or just partially opaque. You may also wonder why setting the view's opaque property to NO isn't enough to make the view transparent. The answer is simple: OpenGL ES doesn't respect that property and draws its clear color anyway.

This is only one half of the story. What's easy to forget and something you just have to know is that cocos2d's `EAGLView` has to be set up with a `pixelFormat` that actually has an alpha channel. Without the alpha channel, you can't make the cocos2d view transparent.

By default, cocos2d initialized the `EAGLView` with the `kEAGLColorFormatRGB565` pixel format. This pixel format uses 16 bits per pixel and has no alpha channel. The only other `pixelFormat` currently supported is `kEAGLColorFormatRGBA8`, which has 8 bits per color channel plus an 8-bit alpha channel, which results in 32 bits per pixel. Obviously, this has an impact on performance and memory usage because the framebuffer memory usage doubles. That's the reason why the `kEAGLColorFormatRGB565` pixel format is the default, but there's really no other choice than to use `kEAGLColorFormatRGBA8` if you want to make the cocos2d view transparent.

Open the `AppDelegate.m` file, and in the `applicationDidFinishLaunching` method look for the line that initialized the `EAGLView`. Then change that to use the `kEAGLColorFormatRGBA8` pixel format:

```
EAGLView *glView = [EAGLView viewWithFrame:[window bounds]
                      pixelFormat:kEAGLColorFormatRGBA8
                      depthFormat:0];
```

Now you can run the app again, and you'll see the `Hello Cocos2D!` labels being drawn over the text fields. There's only one issue remaining: the text fields won't respond to your touches!

Properly Propagating Touch Events via Hit Tests

The easiest way to have the views behind the cocos2d view respond to touch events is to completely disable touch input on the cocos2d view. You won't be receiving any messages from the `CCTouchDispatcher` anymore if you add this line:

```
// This will disable all touch events on the cocos2d view
glView.userInteractionEnabled = NO;
```

Now the text fields behind the cocos2d view act normally, but touch input for the cocos2d view is disabled. UIKit views, which are in front of the cocos2d view, should also work normally and respond to touches, unless you've added them to the cocos2d `glView` directly instead of the dummy view.

You may be wondering why disabling touch input on the cocos2d view is the best, or at least the easiest, option. For that, you have to understand that the cocos2d view is a `UIView` that spans the entire screen area. Although you can see through it now that you've set it up to be transparent, it still responds positively to the `UIView` `hitTest` event. After all, any touch is somewhere on the screen, and since the cocos2d view is as big as the screen and doesn't take into account what's actually displayed inside its view, it responds positively to the hit test. So, any touch that reaches the cocos2d view will be processed by it or, respectively, the `CCTouchDispatcher` class. Anything underneath the cocos2d view is cut off from receiving touch events.

Unfortunately, cocos2d does not have a built-in system to forward the `hitTest` event to its nodes in order for them to decide whether they actually need to respond to the touch. I'll present you with a solution that uses the node's bounding boxes but requires some changes to the cocos2d code.

CAUTION: Only add the following hit test code to the `EAGLView` class if you absolutely need it in your project. It will have a negative effect on performance whenever a touch event is fired, which is basically the whole time the user has at least one finger on the touchscreen. The more nodes there are in your scene, the larger the performance penalty will be.

Your first order of business, if you want to perform custom hit tests, is to make sure that `userInteractionEnabled` is set to `YES`. Please do that right away; otherwise, you might forget about the `userInteractionEnabled` property and instead start looking for a bug in the hit detection code.

Now open the `EAGLView.m` file, which you can find in the `/libs/cocos2d/Platforms/iOS` group in the Xcode project. Add the following `#import` statements at the beginning of the file, because the `EAGLView` class needs to know about these cocos2d classes:

```
#import "CCArray.h"
#import "CCScene.h"
#import "CCLayer.h"
```

Now somewhere in the implementation section of the `EAGLView` class, override the `hitTest` method shown in Listing 15–4. This method is part of the `UIView` class and gets called when the UIKit framework is trying to determine which view needs to respond to a touch event. The method either returns a `UIView` instance, which should receive the touch input, or returns `nil` to signal that the hit test was unsuccessful, in which case the UIKit framework keeps looking for other views that might want to process the touch event.

Listing 15–4. Preparing to Hit Test All cocos2d Scene Children

```
-(UIView*) hitTest:(CGPoint)point withEvent:(UIEvent*)event
{
    UIView* hitView = [super hitTest:point withEvent:event];

    if (hitView == self)
    {
        CCScene* runningScene = [CCDirector sharedDirector].runningScene;
```

```

    NSArray* sceneChildren = [runningScene children];
    CGPoint glPoint = [[CCDirector sharedDirector] convertToGL:point];

    bool hit = [self hitTestNodeChildren:sceneChildren point:glPoint];
    return (hit ? self : nil);
}

return hitView;
}

```

In this case, we first call the super implementation to receive the view the `hitTest` would normally return. In almost all cases, this will be the `EAGLView` itself, but since you can add subviews to the `EAGLView`, it might return a subview, and in this case you want to allow the subview to handle the touch.

Otherwise, the `runningScene` is obtained from the `CCDirector`, which gives you access to the `cocos2d` node hierarchy via the `children` array. Since the `hitTest` point is in Cocoa Touch coordinates, you also have to convert it to GL coordinates before passing both the `sceneChildren` and the `glPoint` to the `hitTestNodeChildren` method. If that method returns a hit, the `hitTest` responds by returning `self`. Otherwise, it lets the `hitTest` fail by returning `nil`, allowing all views behind the `cocos2d` view to take their turn and proceed with the hit testing.

The `hitTestNodeChildren` method in Listing 15–5 is more complicated and harder to understand because it uses recursion to traverse the `cocos2d` node hierarchy. In other words, the function can call itself to go even deeper into the `cocos2d` node hierarchy. Add the `hitTestNodeChildren` method just above the `hitTest` method.

Listing 15–5. Recursively Testing All Nodes to Test If Their boundingBox Contains a Given Point

```

-(BOOL) hitTestNodeChildren:(NSArray*)children point:(CGPoint)point
{
    bool hit = NO;

    if ([children count] > 0)
    {
        Class sceneClass = [CCScene class];
        Class layerClass = [CCLayer class];
        CCNode* node = nil;
        CCARRAY_FOREACH(children, node)
        {
            // check the node's children first
            hit = [self hitTestNodeChildren:[node children] point:point];

            // abort search on first hit
            if (hit)
            {
                break;
            }

            // scenes/layers are always full screen, so do not hitTest them
            if ([node isKindOfClass:sceneClass] || [node isKindOfClass:layerClass])
            {
                continue;
            }
        }
    }
}

```

```

        // check the node itself
        hit = CGRectContainsPoint([node boundingBox], point);

        // abort search on first hit
        if (hit)
        {
            break;
        }
    }

    return hit;
}

```

The first half of the `CCARRAY_FOREACH` loop simply traverses deeper into the cocos2d node hierarchy by calling the function recursively with the current node's children. If any of the recursive calls have found a hit, the loop is aborted right there.

In the second half, the actual node being iterated is checked. This performs the actual hit test by first making sure we're not testing a `CCScene` or `CCLayer` class node. The reason for this is that they both have their `boundingBox` set to the entire screen area. If you would test any of these classes, you would always `hit` them, and that is exactly what you're trying to avoid.

Now that we're sure the test is on a node with a reasonable bounding box, the actual check is as simple as testing for whether the point is inside the `boundingBox`:

```
hit = CGRectContainsPoint([node boundingBox], point);
```

Again, if there was a hit, the loop aborts, and the method returns. This is an optimization because we only ever need to find any node that responds positively to the hit test.

Obviously, this solution has some drawbacks. For one, it assumes that a node should get a touch event if the touch is inside its `boundingBox`. What it doesn't know is whether there's some kind of game state that would prevent the node from processing the touch, for example if the node is a `CCMenuItem` that is currently disabled. Or, if the touch is on a sprite that actually performs a pixel-perfect collision check, in that case the bounding box check is too broad. Moreover, the `boundingBox` is excessively large when the node is rotated because it is an axis-aligned bounding box that changes in size as the node rotates.

What you can do to alleviate this situation is to add a `hitTest` method to the `CCNode` class, which performs just the bounding box check by default but can be overridden by subclasses to perform more accurate or conditional checks. You'll find this minor code change in the `CocosWithCocoa04` project along with additional debug logging and touch detection code to help you understand the hit testing process.

Sandwiching the cocos2d View

Just for completing this test, I'd like to add another text field but in front of the cocos2d view so that we truly have a sandwiched cocos2d view with UIKit views in the back and in the front and all of them will be able to respond to touches.

The change is rather simple; just add this code at the end of the `addSomeCocoaTouch` method, and make sure you add the `textFieldFront` as subview of the `dummyView` and not the `glView`:

```
UITextField* textFieldFront = [[UITextField alloc] initWithFrame:
    CGRectMake(280, 40, 200, 24)];
textFieldFront.text = @" On top of Cocos2D";
textFieldFront.borderStyle = UITextBorderStyleRoundedRect;
textFieldFront.delegate = self;

[dummyView addSubview:textFieldFront];
[textFieldFront release];
```

Actually, you could also add the `textFieldFront` to the `glView` as a subview without any immediately noticeable change. But adding the text field to the `dummyView` allows you to reorder it in the view hierarchy at any time; for example, you could move it behind the `cocos2d` view using the `sendSubviewToBack` method of the `dummyView`. You wouldn't be able to do that if you add the view directly to `cocos2d`'s `glView`.

Check out Figure 15–4 with the result. You'll have UIKit views on top of the `cocos2d` view and behind it. The text view at the back can still be edited and manipulated as the **Cut, Copy, Paste, Replace** button shows. More importantly, despite the accompanying text field being behind the `cocos2d` view, the **Cut, Copy, Paste, Replace** pop-over button is automatically on top of the `cocos2d` view. Just how it ought to be!



Figure 15–4. *UIKit views on top and behind the cocos2d view, with input enabled for all of them*

Adding Views Designed with Interface Builder

At this point, you may be wondering how you could add a view that was designed with Apple's Interface Builder. Let's tackle this now. Code-wise, it's surprisingly simple, and you can look it up in the `CocosWithCocoa05` project if you want.

The first order of business is to create an Interface Builder resource file. In Xcode 4 you create them comfortably from within the project using the **File > New > New File** command from the menu, or right-click a group and select **New File**.

You'll be prompted to choose a template from the file template dialog. As you can see in Figure 15–5, you should create the Interface Builder file using the `UIViewController`

subclass template. This will also create the Interface Builder nib file for you and connect it with your view controller, which is essential for the view to work.

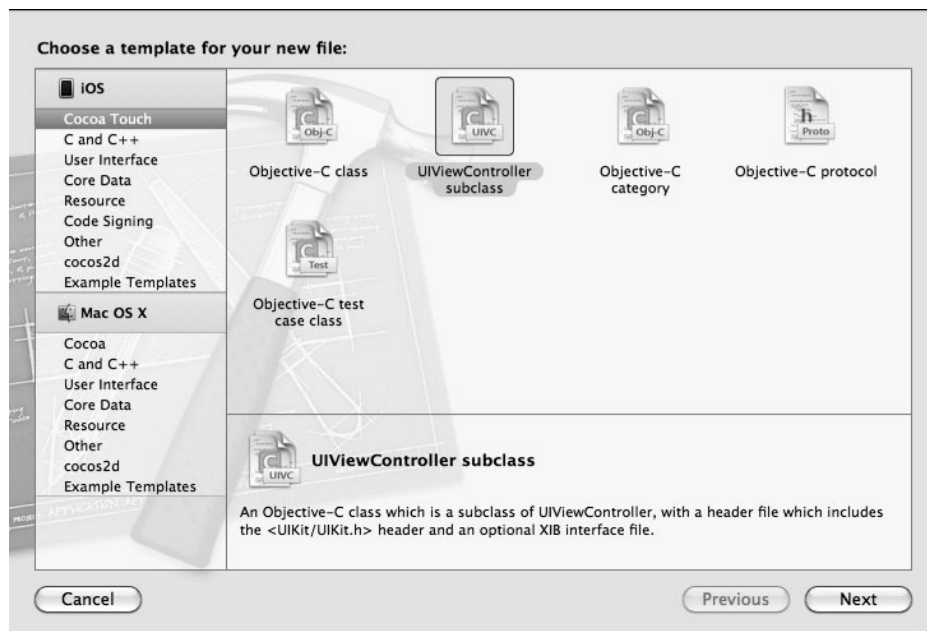


Figure 15–5. Create an Interface Builder view by creating a *UIViewController subclass*.

Make sure that the check box **With XIB for user interface** in Figure 15–6 is checked, and make sure the **Subclass of** text is the `UIViewController`. I decided to save this template using the file name `MyView.m`. You should end up with three new files in your project: `MyView.h`, `MyView.m`, and `MyView.xib`.

NOTE: The developer documentation and even the Cocoa Touch API refers to Interface Builder files as *nib files* even though they use the extension `.xib`. They used to have the `.nib` extension, and it simply stuck as a tradition even though the file extension was changed years ago. So, *nib* and *xib* are used interchangeably and refer to the same thing.

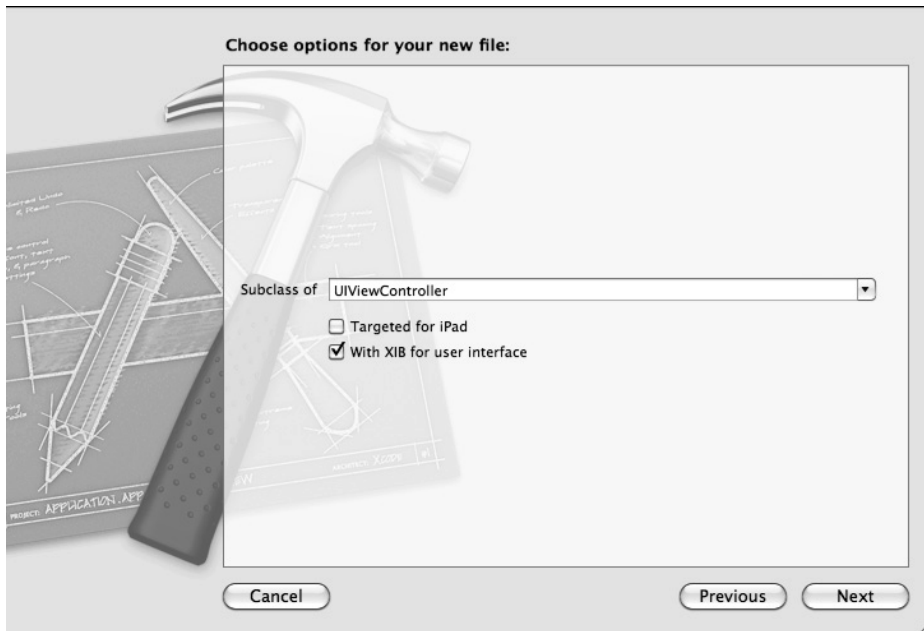


Figure 15–6. Make sure *With XIB for user interface* is checked.

If you click the `MyView.xib` file, you'll be presented with the Interface Builder, which is no longer a separate application but integrated into Xcode 4. You'll see an iPhone screen's view onto which you can drag and drop views from the Object Library, accessible via **View > Utilities > Object Library** in case it's not currently visible.

With Interface Builder, you can easily create your UIKit user interface visually. Since it's beyond the scope of the book to explain the Interface Builder workflow, I'll refer you to Apple's Xcode 4 User Guide and specifically the section on Designing User Interfaces:

<http://developer.apple.com/library/mac/#documentation/ToolsLanguages/Conceptual/Xcode4UserGuide/InterfaceBuilder/InterfaceBuilder.html>

While I'm at it, if you need a refresher or introduction to Views and Windows, take a look at Apple's View Programming Guide for iOS:

http://developer.apple.com/library/ios/#documentation/WindowsViews/Conceptual/ViewPG_iPhoneOS/Introduction/Introduction.html

NOTE: Unfortunately, you cannot use Interface Builder to design your cocos2d view. For that you will have to use a separate editor like CocoShop, CocoaBuilder, LevelHelper, or any other editing tool with cocos2d support that fits your need. Please refer to Chapter 17 for a list of cocos2d editing tools.

For now, it is sufficient to just add any views to the Interface Builder view, like sliders, buttons, labels, and whatnot. But ideally you should at least do the following: select the

main view and bring up the Attributes Inspector via **View > Utilities > Attributes Inspector**. The first attribute under Simulated Metrics is called Orientation, and you should change that to Landscape since the application is currently only capable of running in Landscape mode. If you don't do that, your views will be rotated by 90 degrees when you run the application.

The MyView class does not need to be modified; the default implementation works just fine. You can directly load the MyView.xib file by adding the following code at the end of the addSomeCocoaTouch method:

```
// add an Interface Builder view
MyView* myViewController = [[MyView alloc] initWithNibName:@"MyView" bundle:nil];
[dummyView addSubview:myViewController.view];
[dummyView sendSubviewToBack: myViewController.view]; // optional
[myViewController release];
```

Notice that the initWithNibName takes the name of the xib file as a parameter but without the .xib extension. If you add the extension, you'll receive an error message that the xib could not be loaded. The bundle parameter is nil, which means the app should look for the file in the main bundle.

Since the MyView class inherits from UIViewController, you can access the actual view with the myViewController.view property. You'll add that to the dummyView, and if you want, you can also issue an sendSubviewToBack message to put the view in the background. Lastly, and as always, you'll release the myViewController when you're done with it, since the addSubview method retains the view.

You can now create and add views designed with Interface Builder to a cocos2d app. Your result might look something like the one in Figure 15-7.



Figure 15-7. The resulting project shows the MyView.xib file designed with Interface Builder in the bottom half.

Orientation Course on Autorotation

It's about time that we discuss autorotation. It's one of the things that need special considerations when you move from a purely cocos2d-based app to one that also uses UIKit controls. This is most obvious if you run any of the previous CocosWithCocoa example projects on a first- or second-generation device. In that case, you'll see all UIKit controls oriented to the portrait mode. What's wrong?

The culprit is the code in the file `GameConfig.h` that the cocos2d project template creates. Here's the essential code of the file making use of compiler flags to change the default autorotation behavior depending on the device type and platform:

```
#define kGameAutorotationNone 0
#define kGameAutorotationCCDirector 1
#define kGameAutorotationUINavigationController 2

// 3rd generation and newer devices: Rotate using UINavigationController.
#if defined(__ARM_NEON__) || TARGET_IPHONE_SIMULATOR
#define GAME_AUTOROTATION kGameAutorotationUINavigationController

// ARMv6 (1st and 2nd generation devices): Don't rotate. It is very expensive.
#elif __arm__
#define GAME_AUTOROTATION kGameAutorotationNone

// Ignore this value on Mac
#elif defined(__MAC_OS_X_VERSION_MAX_ALLOWED)

#else
#error(unknown architecture)
#endif
```

Cocos2d defines three distinct types of autorotation support:

- `kGameAutorotationNone`
- `kGameAutorotationCCDirector`
- `kGameAutorotationUINavigationController`

Not supporting autorotation will lock the app into the orientation it was designed for. It is also the fastest mode, especially if you consider the impact on performance on ARMv6 (first- and second-generation) devices; it is considered so severe that it is disabled by default for those devices. So, why would you even support autorotation if it has an impact on performance?

The answer lies in Apple's iOS Human Interface Guidelines (HIG). Specifically, see the Handling Orientation Changes section here:

developer.apple.com/library/ios/#documentation/UserExperience/Conceptual/MobileHIG/UEBestPractices/UEBestPractices.html

The minimum recommendation to take away from this is that your app should at least support both variants of an orientation. Locking the app to just a single orientation could cause a rejection of your app during the approval process.

TIP: Shortly before the iPad was first released, Apple's iOS Human Interface Guidelines specified explicitly that all iPad apps must support rotation to all orientations. After much protest from developers, specifically from game developers whose games can't easily support all orientations, Apple has changed the wording from a `must have` to: `On iPad, strive to satisfy users' expectations by being able to run in all orientations.`

This leaves the other two autorotation modes. The difference between `kGameAutorotationCCDirector` and `kGameAutorotationUIViewController` is that the `CCDirector`-based autorotation rotates only the cocos2d view but ignores any UIKit views. Only if you have UIKit views in your app should you need to use the `UIViewController`-based autorotation mode. Mainly that's because it's the slowest mode and would simply waste performance if you did not use UIKit views.

The initial orientation is set in the `AppDelegate` class. In the `applicationDidFinishLaunching` method, you'll see a few lines of code that check the `GAME_AUTOROTATION` setting to set the initial orientation. Since the `UIViewController` expects to be started in the portrait mode regardless of which orientations your app supports, it is initially set to portrait orientation. Otherwise, the default landscape orientation is used:

```
viewController = [[RootViewController alloc] initWithNibName:nil bundle:nil];
viewController.wantsFullScreenLayout = YES;
```

```
#if GAME_AUTOROTATION == kGameAutorotationUIViewController
    [director setDeviceOrientation:kCCDeviceOrientationPortrait];
#else
    [director setDeviceOrientation:kCCDeviceOrientationLandscapeLeft];
#endif
```

The actual handling of orientation changes is the responsibility of the `RootViewController` class, which cocos2d also initializes in the `applicationDidFinishLaunching` method. Let's take a look at the `RootViewController` class implementation and specifically add the `shouldAutorotateToInterfaceOrientation` method, which is called by the UIKit framework in order to query which interface orientations are supported by this particular `UIViewController`. The method returns YES only for those orientations it supports, and it returns NO if it does not support rotation to the given `interfaceOrientation`.

```
-(BOOL) shouldAutorotateToInterfaceOrientation:⌘
    (UIInterfaceOrientation)interfaceOrientation
{
    #if GAME_AUTOROTATION == kGameAutorotationNone

        return (interfaceOrientation == UIInterfaceOrientationPortrait);

    #elif GAME_AUTOROTATION == kGameAutorotationCCDirector

        if (interfaceOrientation == UIInterfaceOrientationLandscapeLeft)
        {
            [[CCDirector sharedDirector]⌘
                setDeviceOrientation:kCCDeviceOrientationLandscapeRight];
        }
        else if (interfaceOrientation == UIInterfaceOrientationLandscapeRight)
        {
            [[CCDirector sharedDirector]⌘
                setDeviceOrientation:kCCDeviceOrientationLandscapeLeft];
        }
    }
```

```

        return (interfaceOrientation == UIInterfaceOrientationPortrait);
    #elif GAME_AUTOROTATION == kGameAutorotationUIViewController
        return (UIInterfaceOrientationIsLandscape(interfaceOrientation));
    #endif // GAME_AUTOROTATION

    return NO;
}

```

The previous code is a cleaned-up version of the code you'll find in the `RootViewController` class. I removed all comments and made the code more concise and readable, because in the `RootViewController` class it looks more daunting than it actually is.

Once again, this code uses the current `GAME_AUTOROTATION` setting set in `GameConfig.h` to pick one of three possible code paths. If autorotation support is set to `kGameAutorotationNone`, the method returns YES only if the orientation is the portrait mode. This may seem a bit strange if your app is actually using the landscape orientation. In fact, you could just as well return NO here or call the super implementation and return that value. It doesn't matter because if `kGameAutorotationNone` is used, no autorotation takes place. Of course, you can still call the `CCDirector` `setDeviceOrientation` method manually at any time to change the orientation.

If the `kGameAutorotationCCDirector` mode is in effect, the director's `setDeviceOrientation` method is called to change the device orientation to one of the supported `interfaceOrientation` modes. If you wanted to support all four orientations, for example, or both portrait orientations instead of landscape orientations, you would have to extend the code accordingly to call `setDeviceOrientation` with the corresponding and supported device orientation. Here's the code you would use if your app were designed for portrait orientations instead of landscape orientations:

```

if (interfaceOrientation == UIInterfaceOrientationPortrait)
{
    [[CCDirector sharedDirector] setDeviceOrientation:kCCDeviceOrientationPortrait];
}
else if (interfaceOrientation == UIInterfaceOrientationPortraitUpsideDown)
{
    [[CCDirector sharedDirector] ←
        setDeviceOrientation:kCCDeviceOrientationPortraitUpsideDown];
}

```

NOTE: You may have noticed that if the `interfaceOrientation` is `UIInterfaceOrientationLandscapeLeft`, the device orientation is actually set to the seemingly opposing `kCCDeviceOrientationLandscapeRight` mode. This is not a mistake but a difference in definition in the `UIInterfaceOrientation` and `UIDeviceOrientation` enums.

Setting the device orientation with the `setDeviceOrientations` method has one drawback: it won't rotate UIKit views. If you use any UIKit views in your app and you want them to autorotate, you will have to use the `kGameAutorotationViewController` setting. By default only landscape orientations are supported:

```
// support all landscape orientations
return (UIInterfaceOrientationIsLandscape(interfaceOrientation));
```

You can easily change that to support only portrait orientations:

```
// support all portrait orientations
return (UIInterfaceOrientationIsPortrait(interfaceOrientation));
```

Or you can even signal that you support all orientations:

```
// support all four orientations
return (UIInterfaceOrientationIsLandscape(interfaceOrientation) ||
        UIInterfaceOrientationIsPortrait(interfaceOrientation));
```

In the latter case, you could just return YES. But how does the view controller perform the actual autorotation? The actual rotation happens in the `willRotateToInterfaceOrientation` method in the `RootViewController` class, shown in Listing 15–6.

Listing 15–6. *Performing the Rotation of the cocos2d View*

```
#if GAME_AUTOROTATION == kGameAutorotationUIViewController
-(void) willRotateToInterfaceOrientation:(UIInterfaceOrientation)toInterfaceOrientation
        duration:(NSTimeInterval)duration
{
    CGRect screenRect = [UIScreen mainScreen].bounds;
    CGRect rect = CGRectZero;

    if (toInterfaceOrientation == UIInterfaceOrientationPortrait ||
        toInterfaceOrientation == UIInterfaceOrientationPortraitUpsideDown)
    {
        rect = screenRect;
    }
    else if (toInterfaceOrientation == UIInterfaceOrientationLandscapeLeft ||
             toInterfaceOrientation == UIInterfaceOrientationLandscapeRight)
    {
        rect.size = CGSizeMake(screenRect.size.height, screenRect.size.width);
    }

    CCDirector* director = [CCDirector sharedDirector];
    EAGLView* glView = director.openGLView;
    float contentScaleFactor = [director contentScaleFactor];

    if (contentScaleFactor != 1)
    {
        rect.size.width *= contentScaleFactor;
        rect.size.height *= contentScaleFactor;
    }

    glView.frame = rect;
}
#endif // GAME_AUTOROTATION == kGameAutorotationUIViewController
```

In essence, the cocos2d `EAGLView` obtained via `director.openGLView` gets its frame property updated with the new screen size. And that is the new screen size returned from the `[UIScreen mainScreen].bounds` property. The only thing that this code sorts out for you is that in landscape orientations the height is actually the new `EAGLView` frame's width, and vice versa. And of course, it takes the Retina display mode into account by multiplying `rect.size.width` and `rect.size.height` with the `contentScaleFactor`. You don't actually need to understand the code in full since you'll hardly ever need to change it. The only thing you should know is that the code assumes that the cocos2d view is in full-screen.

The project named `CocosWithCocoa06` has some changes regarding the `RootViewController` and autorotation. For example, it defaults to support the `UIViewController` even on first- and second-generation devices. If you have one of those, you'll see the screen autorotate, and you might want to compare its performance with the previous `CocosWithCocoa05` project. Additionally, the `RootViewController` class code is cleaned up and contains the example code to enable autorotation support for portrait or all four orientations.

Embedding the cocos2d View in Cocoa Touch Apps

Many developers don't realize it is actually possible to embed a cocos2d view in a regular application using UIKit views as its main elements. The cocos2d view doesn't even have to be full-screen!

The problem merely lies in setting up the project. I'd like to show you how it's done.

Creating a View-Based Application Project with cocos2d

Fire up your Xcode 4. Create a new project via **File** ► **New** ► **New Project**. Select the View-Based Application template, as shown in Figure 15–8, and name the project `ViewBasedAppWithCocos2D`. You'll end up with a project that has a view controller class, the corresponding `.xib` file, a `MainWindow.xib` file, and an app delegate class. If you run it right now, you'll see a blank iPhone view with just the status bar on top.

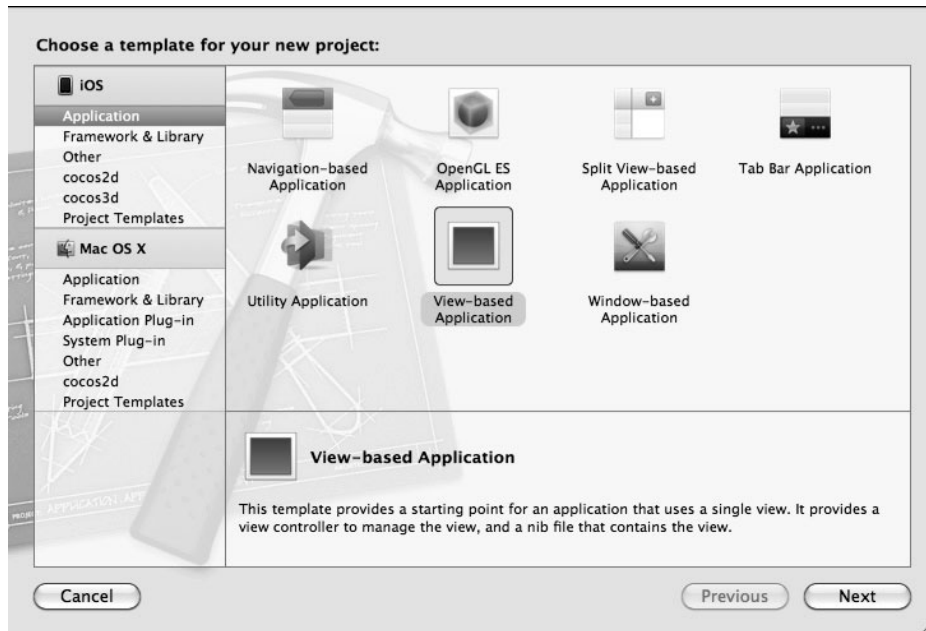


Figure 15–8. The starting point for embedding a cocos2d view is the *View-based Application* template.

TIP: You may encounter the message in Figure 15.9 if you try to run this project on a device, even if the device is properly provisioned. By default, the new project templates set the iOS Deployment Target setting to the latest iOS SDK, which may be iOS 4.3 or iOS 5.0. Those and newer iOS versions are not available for devices of the first and second generation. In such a case, click the project in the Project Navigator pane to see the list of projects and targets. Select the project again in the new view and switch to its **Info** pane. You'll see that the first setting **iOS Deployment Target** is set to iOS 4.3 or newer. Change this to iOS 3.1.3 or lower and try running the app again. If Xcode still complains with the same message as in Figure 15.9, your device is not properly provisioned, and you'll have to look into that. Possibly the provisioning profile has expired, and you may have to get a new one from <http://developer.apple.com/ios>.

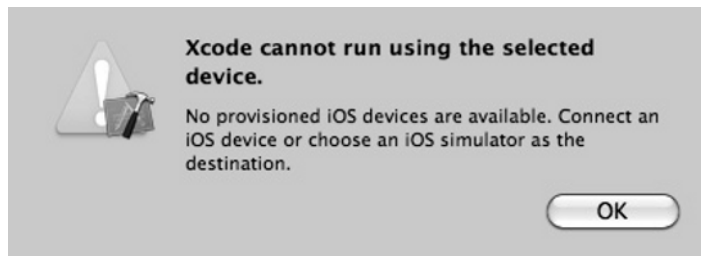


Figure 15–9. If you get this message when trying to run the app on a device, try changing the **iOS Deployment Target** setting in the project's **Info** pane to iOS 3.1.3 or lower.

The next step is to get the cocos2d source code added to the project. There are several ways to do this; I prefer to rely on the cocos2d project templates because it makes it easier to copy exactly the necessary files—no more, no less. So, in Xcode, create another project via **File > New > New Project**, and this time select one of the cocos2d project templates. If you need physics in your cocos2d view, you should choose one of the templates using a physics engine; otherwise, just pick the cocos2d template. Save the project anywhere using any name; just remember where you saved it. In fact, you can immediately close the cocos2d Xcode project after you’ve created it.

With your `ViewBasedAppWithCocos2D` project still open in Xcode 4, navigate to the cocos2d project folder using the Finder app. Locate the subfolder named `libs` in the cocos2d project folder. Drag the `libs` folder onto the `ViewBasedAppWithCocos2D` project and drop it onto the Project Navigator pane where all the project’s files are listed. If necessary, open the Project Navigator first by selecting **View > Navigators > Project** from the menu. Make sure the **Copy items into destination group’s folder (if needed)** check box is checked and the other settings also correspond to the ones you see in Figure 15–10.

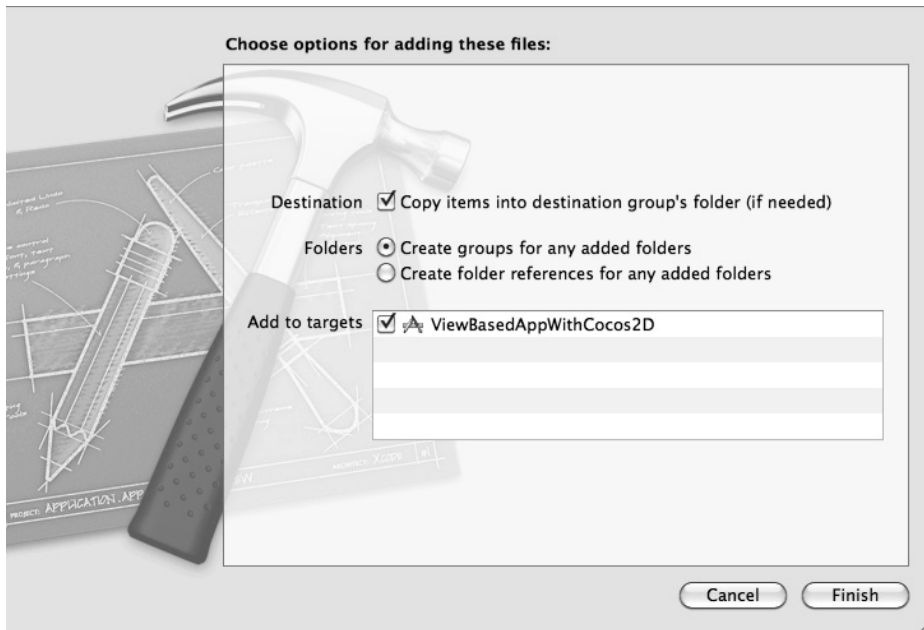


Figure 15–10. Make sure the *Copy Items* check box is set when dropping the cocos2d *libs* folder into the view-based application project.

You can’t build the project just yet because the cocos2d engine requires additional frameworks and libraries, without which you’ll only receive linker errors. To add these dependencies to the project, select the project itself in the Project Navigator (first entry in the list with the blue icon). Select the `ViewBasedAppWithCocos2D` target, navigate to the Build Phases tab, and unfold the Link Binary With Libraries pane. Then click the + button to add additional libraries. If you have trouble finding this location, take a look at Figure 15–11.

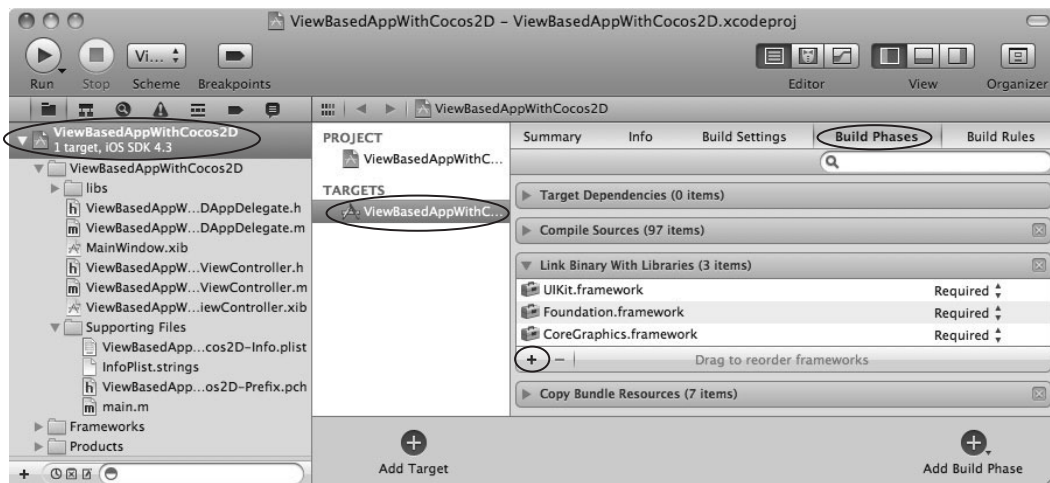


Figure 15–11. Add the missing frameworks and libraries to the target's Link Binary With Libraries Build Phase.

Once you click the + button, a list of frameworks and libraries opens. Here's the list of frameworks and libraries you will have to add. They're found in the list in the same alphabetical order as I list them here:

- AudioToolbox.framework
- AVFoundation.framework
- libz.dylib
- OpenAL.framework
- OpenGL.framework
- QuartzCore.framework

Note that you can select all libraries in one go by holding down the Command key while selecting the items in the list. The added frameworks and libraries will be added to the root of the project in the Project Navigator. You can safely move them to the Frameworks group where they belong, just to get them out of sight since you don't need to work with them.

You can now build and run the app. It has the cocos2d source code built into it, but of course without a user interface, it's the same dull and empty app as it was before. Let's change that!

Designing the User Interface of the Hybrid App

Select the ViewBasedAppWithCocos2DViewController.xib file in the Project Navigator to see the Interface Builder view. Using the Object Library (**View > Utilities > Object Library**), drag and drop the following objects onto the view. You can arrange these objects in the design area in any way you like:

- View
- Switch
- Segmented Control

Now select the newly added View object and switch to the Identity Inspector (**View** ➤ **Utilities** ➤ **Identity Inspector**). You'll notice that the first item shows the view is derived from the `UIView` class. Since this should become the cocos2d `EAGLView`, use the drop-down button to select a custom class from the list. One of the first items should be the `EAGLView` class. If not, scroll the list until you find the `EAGLView` or simply type in the name of the class. The resulting user interface mockup should look something like Figure 15–12.

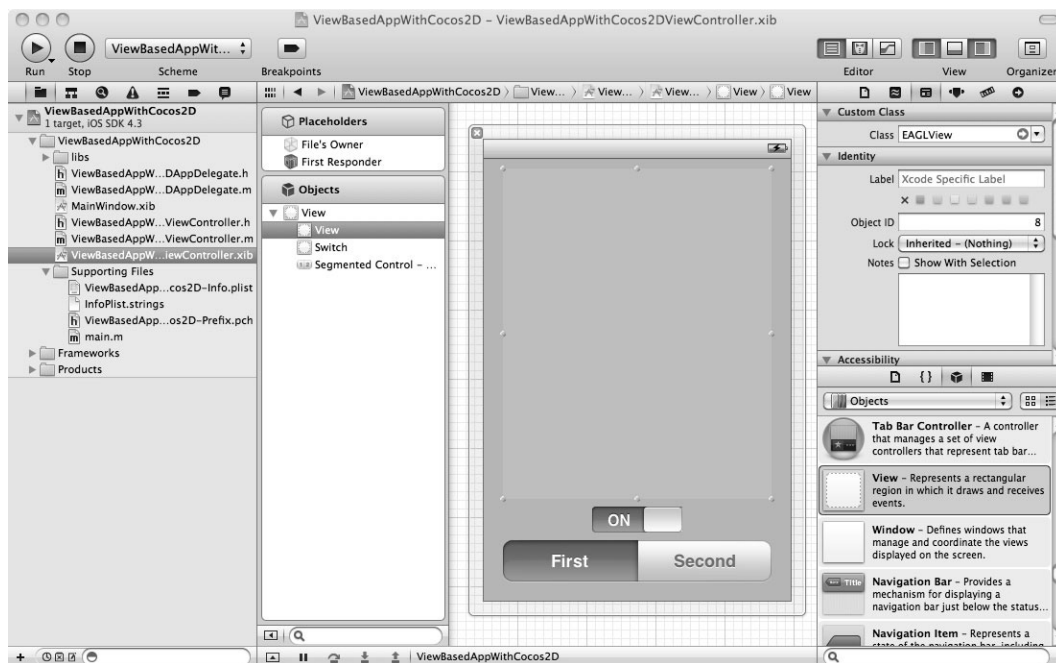


Figure 15–12. The Interface Builder view of the hybrid app's user interface

Interface Builder will automatically instantiate the `EAGLView` for you. You only need to attach the `CCDirector` with this particular view. The On/Off Switch should serve as the toggle button that turns the cocos2d view on and off.

First, some preparations. Select the `EAGLView` view and switch to the Attributes Inspector (**View** ➤ **Utilities** ➤ **Attributes Inspector**). Check the **Hidden** check box so that the view is initially hidden. With the Attributes Inspector still open, select the On/Off Switch and change its initial **State** to **Off**. You'll hide and unhide the `EAGLView` programmatically.

NOTE: The cocos2d CCDirector class can manage only one EAGLView at a time, mainly because the CCDirector class is a singleton. An app using multiple cocos2d views at the same time is not possible without significant changes to cocos2d. It was simply never designed to work with multiple views at the same time.

Now we need to make the connection from the buttons on the view to the `ViewBasedAppWithCocos2DViewController` class. The easiest way to do so is to open an Assistant Editor in Xcode 4 via **View ► Editor ► Assistant**. You can customize the layout of the Assistant Editor with one of the selections available under **View ► Assistant Layout**. The Assistant Editor will automatically display the `ViewBasedWithCocos2DViewController.h` file.

In the Interface Builder view, select the On/Off Switch and right-click it. It doesn't matter if you select it from the list or by clicking its view. The context menu that opens shows a list of events that the control sends. Click the circle next to the **Value Changed** event and drag it over to the Assistant Editor. You'll notice that it will highlight a line with the label **Insert Action** if you drag it somewhere below the class `@interface` brackets and above the `@end` statement. That's where you should drop the arrow to make the connection. A pop-up view will show up and ask you for the name of the event. I decided to call mine `switchChanged`. You can leave all the other settings at their default values and then make the connection by clicking the **Connect** button in the pop-up view.

Interface Builder has automatically created the necessary code for you in order to receive the particular event that you just connected. There's new code both in the interface and implementation sections of the `ViewBasedAppWithCocos2DViewController` class. Before you review the code changes, you should also connect the same **Value Changed** event of the Segmented Control and name it `sceneChanged`.

This concludes the user interface design part of this project. Now let's move on to hooking up cocos2d.

Start Your cocos2d Engine

If you followed the user interface design part, you'll find two empty methods called `switchChanged` and `sceneChanged` in the `ViewBasedAppWithCocos2DViewController.m` class implementation file, next to some other boilerplate code that was added by the View-based Application template.

The first step is to get cocos2d up and running. One thing that's comfortable when working with cocos2d projects created from one of the cocos2d templates is that you rarely need to add a header file to any of your classes. This is because the `cocos2d.h` file is imported in the project's prefix header. Since this is not the case in the View-based Application template, open the `ViewBasedAppWithCocos2D-Prefix.pch` file in the Supporting Files group and add the cocos2d header:

```
#import <Availability.h>

#ifdef __IPHONE_3_0
```

```
#warning "This project uses features only available in iPhone SDK 3.0 and later."
#endif

#ifdef __OBJC__
    #import <UIKit/UIKit.h>
    #import <Foundation/Foundation.h>
    #import "cocos2d.h"
#endif
```

Next you need to add a cocos2d scene class to the project. Using the **File ► Add Files to ViewBasedAppWithCocos2D ...** menu item, browse into the cocos2d project you created earlier and then locate and select both the header and implementation files of the `HelloWorldLayer` class. Make sure the **Copy items into destination group's folder (if needed)** check box is checked. Alternatively, you can also create a new cocos2d scene class from a cocos2d file template or manually. I'll be using the `HelloWorldLayer` scene throughout this example.

Import the `HelloWorldLayer` class in the `ViewBasedAppWithCocos2DViewController.m` file so that we can run it as our main cocos2d scene:

```
#import "HelloWorldLayer.h"
```

All that is left now is to actually start up cocos2d and connect it with the `EAGLView`. The `switchChanged` method in Listing 15–7 contains all the start-up code that is needed.

Listing 15–7. Setting Up the Director and Displaying the First cocos2d Scene

```
- (IBAction)switchChanged:(id)sender
{
    CCDirector* director = [CCDirector sharedDirector];

    if ([CCDirector setDirectorType:kCCDirectorTypeDisplayLink] == NO)
    {
        [CCDirector setDirectorType:kCCDirectorTypeDefault];
    }

    [director setAnimationInterval:1.0/60];
    //[director setDisplayFPS:YES];

    NSArray* subviews = self.view.subviews;
    for (int i = 0; i < [subviews count]; i++)
    {
        UIView* subview = [subviews objectAtIndex:i];
        if ([subview isKindOfClass:[EAGLView class]])
        {
            subview.hidden = NO;
            [director setOpenGLView:(EAGLView*)subview];
            [director runWithScene:[HelloWorldLayer scene]];
            break;
        }
    }
}
```

That's surprisingly little code to get cocos2d running. As usual, we pick the best possible director type, we set the animation interval, and the rest is just connecting the director with the `EAGLView`. The latter part simply goes over the list of subviews in the

view controller's view to find one that is subclassed from `EAGLView`. Once the right view is found, it's made visible and assigned to the director via `setOpenGLView`. Directly after that you can make the call `runWithScene`, and that's it.

I also break out of the loop once a scene was run as a precaution because if this loop would continue, it might find another `EAGLView`, and that would result in a crash. In fact, you'll notice if you run the app that you can change the switch button's state only once. The second time, the app crashes.

You'll notice that the `setDisplayFPS` command is commented out in this example. To enable the fps display, you will also have to add the `fps_images.png` file from any cocos2d project to your project via the **File ► Add Files To ...** command. Otherwise, the framerate counter will not be displayed.

Stop the cocos2d Engine and Restart It

You can easily start, stop and restart the cocos2d engine at any time. You just need to determine the current state of the engine. It might never have been started before, it might be running, or it might be stopped.

There's no property in cocos2d's `CCDirector` class, but you can infer the status in other ways. I prepared this example project so that this status can be detected, and depending on your project's requirements, you might have to use other mechanisms, such as keeping track of the cocos2d running status through global variables.

If the director's `openGLView` property is `nil`, you know that it has never been started before. And in this particular project, the fact that the `openGLView` is hidden tells me that cocos2d has been suspended and could be restarted. In addition to that, the switch button's state is also taken into account. The code for the `switchChanged` method has changed as follows and is shown in Listing 15–8.

Listing 15–8. *Starting, Suspending, and Stopping cocos2d*

```
- (IBAction)switchChanged:(id)sender
{
    UISwitch* switchButton = (UISwitch*)sender;
    CCDirector* director = [CCDirector sharedDirector];

    if (switchButton.on)
    {
        if (director.openGLView == nil)
        {
            if ([CCDirector setDirectorType:kCCDirectorTypeDisplayLink] == NO)
            {
                [CCDirector setDirectorType:kCCDirectorTypeDefault];
            }

            [director setAnimationInterval:1.0/60];
            //[director setDisplayFPS:YES];

            NSArray* subviews = self.view.subviews;
            for (int i = 0; i < [subviews count]; i++)
            {
```

```

        UIView* subview = [subviews objectAtIndex:i];
        if ([subview isKindOfClass:[EAGLView class]])
        {
            [director setOpenGLView:(EAGLView*)subview];
            [director runWithScene:[HelloWorldLayer scene]];
            break;
        }
    }
}
else
{
    [director startAnimation];
}

director.openGLView.hidden = NO;
}
else
{
    director.openGLView.hidden = YES;
    [director stopAnimation];
}
}

```

Notice that the director methods `startAnimation` and `stopAnimation` are used to restart and stop cocos2d. Just for the very first time, you need to call `runWithScene`. But if you wanted to run a different scene each time cocos2d is restarted, you should call `replaceScene` directly after the call to `startAnimation`. The `runWithScene` method can be called only once during the lifetime of the application and must not be used again.

Technically, the `stopAnimation` method only stops cocos2d from refreshing its view. Unless the view is hidden or obstructed by another view, the last frame cocos2d has rendered will remain as a static image in the `EAGLView`. That's why hiding the `EAGLView` is a good idea. Calling `stopAnimation` is sometimes necessary to ensure that certain UIKit views are responsive and animate smoothly, in particular all views derived from `UIScrollView`. It is good practice to call `stopAnimation` whenever you display an (almost) full-screen UIKit view, to conserve performance for the foreground view as well as conserve battery power. Once the foreground view is dismissed, you call `startAnimation` again, and the cocos2d view and director continue where they were.

TIP: If you want to see how this app behaves with autorotation, you have to make a small change to the `ViewBasedAppWithCocos2DViewController` class. Simply return YES from the `shouldAutorotateToInterfaceOrientation` method to enable rotation to all orientations. Although your app doesn't support it well (it is not designed for landscape orientation), it serves to show that the cocos2d view will be correctly autorotated even without its `RootViewController` class.

Changing Scenes

The last step to complete this project is to use the Segmented Control's buttons to change scenes in the cocos2d view. In Listing 15–9, taken from the `ViewBasedAppWithCocos2DViewController` class, the code that was added to the `sceneChanged` method is shown.

Listing 15–9. *Changing Scenes Whenever You Press a UIKit Button*

```
- (IBAction)sceneChanged:(id)sender
{
    CCDirector* director = [CCDirector sharedDirector];
    if (director.openGLView == nil || director.openGLView.hidden)
    {
        return;
    }

    UISegmentedControl* sceneChanger = (UISegmentedControl*)sender;
    int selection = sceneChanger.selectedSegmentIndex;

    CCScene* helloScene = [HelloWorldLayer scene];
    CCScene* transScene = nil;
    if (selection == 0)
    {
        transScene = [CCTransitionSlideInR transitionWithDuration:1 scene:helloScene];
    }
    else if (selection == 1)
    {
        transScene = [CCTransitionPageTurn transitionWithDuration:1 scene:helloScene];
    }
    else
    {
        transScene = [CCTransitionShrinkGrow transitionWithDuration:1 scene:helloScene];
    }

    [director replaceScene:transScene];
}
```

Since the user can press the Segmented Control buttons at any time, even before the cocos2d view is initialized, the first thing this method does is to check that the `director.openGLView` exists and is not hidden. Otherwise, the remaining code could crash the app.

The sender parameter is always the control that triggered the event. Here I assume that it's a `UISegmentedControl`. If you ever changed that, you would have to change the control's class here as well. Via the `selectedSegmentIndex`, you get the index of the currently selected button, which is then used to decide which transition to use for the new scene. I'm simply creating a new instance of the same `HelloWorldLayer` class; of course, you can also use different scene classes for each button if you want. At last, the `transScene` is used with the director method `replaceScene` to actually change the scene to the new one.

how much UIKit you want in your cocos2d app and when, where, and how you would like your cocos2d view in your UIKit app.

The trickiest aspects were making the cocos2d view transparent in order to allow UIKit views in the background as well as having a hit test method perform hit tests on cocos2d node in an attempt to allow all views to receive input, whether UIKit or cocos2d and regardless of where they are in the view hierarchy. And then there was autorotation, which cocos2d can do in two ways, but only one way using the `RootViewController` allows you to rotate UIKit views correctly.

Adding cocos2d to a UIKit app also proved to be fairly simple, even if you need to turn the cocos2d view on and off only at specific times. You may have also taken away that the cocos2d view doesn't need to be full-screen at all but can be any size, or even resized while the app is running.

But you also learned that mixing cocos2d and UIKit views is not without drawbacks, specifically performance-wise. Keep a watchful eye on your app's performance by testing it regularly on a device, particularly on first- and second-generation devices.

Kobold2D Introduction

In more than two years of working with cocos2d, I found plenty of opportunity and need to improve the cocos2d development workflow by adding helpful code snippets, by extending cocos2d classes, by enabling easier cross-platform development, by automating often-repeated tasks, by simplifying the cocos2d upgrade process, and by providing complete and accurate documentation. Eventually, this culminated in the inception of Kobold2D, a game engine that is still very much based on cocos2d-iphone but improves the workflow for cocos2d developers.

The goal of Kobold2D is to make it easier for new developers to work with cocos2d while at the same time satisfying my own needs for a professional work environment (and I hope yours too!). To be sure of that, the development of Kobold2D is strongly user-driven: if you need it (and others too), you'll get it! If you have an idea or suggestion, please vote for it on or add it to the Kobold2D feedback page: www.kobold2d.com/display/KKSITE/Kobold2D+Feedback.

Kobold2D merges the cocos2d engine with popular libraries many developers have been using with cocos2d in the past. Some of these libraries have almost become essential add-ons, including cocos3d, Wax, Lua, the new cocos2d-iphone-extensions project, SneakyInput, the Chipmunk SpaceManager, ObjectAL, and iSimulate. Kobold2D makes these libraries readily accessible and usable by all developers; you'll just have to import a library's header files and call library methods. In almost all cases, you won't have to fiddle with build settings or linker flags, and you don't need to worry about an increased app size.

Over time, Kobold2D will add more and more convenience features, starting with a simplified way to handle user input that allows you to poll the keyboard and mouse button states, provides automatic accelerometer filtering, and gives you access to gyroscope and device motion data without having to learn the Core Motion framework.

In this chapter, I'll introduce you to the key concepts of working with Kobold2D and how it changes the way you develop cocos2d-based apps. I'll guide you through the Kobold2D version of the Doodle Drop project to illustrate how easy it is to write a game that also runs on Mac OS computers and to give you an introduction on Kobold2D user

input processing. In addition, I'll provide you with a short introduction to cocos3d, the 3D add-on library for cocos2d.

You can learn more about Kobold2D and download it at www.kobold2d.com.

Benefits of Using Kobold2D

I view Kobold2D as a game development kit rather than a game engine. It's more like a Linux distribution with the cocos2d-iphone project as its "kernel" and multiple other modules tacked onto it, all ready to use after installation.

To stick with the analogy, if cocos2d-iphone were a Linux kernel with just its command-line interface, Kobold2D would be providing the graphical user interface and the essential applications that make the operating system more powerful, enjoyable to work with, and accessible to a broader user base.

At the same time, the command line is still there; it has simply become part of a bigger whole.

Kobold2D Is Ready to Use

Kobold2D will be installed like a regular application. There's no need to run a script in the Terminal app or any other such error-prone tasks. Download and run the package installer, follow the on-screen instructions, and you're done. It's quick and painless.

After installing Kobold2D, you can build one of the 15 template projects right away; most of them are based on projects we've created throughout the book. With the exception of the app start-up process, which is now simply a configuration file, everything you've learned about cocos2d in this book and elsewhere still applies.

Kobold2D Is Free

Kobold2D is free and distributed under the MIT License. All of the included libraries use either the MIT License or a compatible license.

The only exception is iSimulate, which requires the paid iSimulate app from the App Store to unlock all features. The free iSimulate Lite, for example, doesn't support the forwarding of multi-touch events to the iOS Simulator.

Kobold2D Is Easy to Upgrade

A big incentive for Kobold2D was to simplify and streamline upgrading cocos2d in existing projects. The downside of using the cocos2d Xcode project templates is that a copy of the entire cocos2d source code resides in each and every project, making the upgrade process unnecessarily complex.

Kobold2D fixes that by keeping your code separate from any library's code. If a new version of Kobold2D is released and you want your project to use the updated code, simply run the Kobold2D Project Upgrader tool of the newly installed Kobold2D version. It will scan previous Kobold2D versions and offer you to upgrade each individual project with a click of the mouse. The only task that remains for you to upgrade your code is the unavoidable code maintenance because of API changes, for example, classes that may have been renamed in cocos2d or other libraries, or library methods whose parameters have changed.

Kobold2D Provides Lib Service

A central aspect of Kobold2D is to spare developers the pain of adding third-party libraries to cocos2d projects. Correctly setting up third-party libraries often requires intricate knowledge of the build system, an intuition about how to read compiler and linker errors, and possibly even small but crucial source code changes in the right places. It can take hours, if not days, for someone else's code to compile and link successfully on all platforms: iOS (ARMv6 and ARMv7), iOS Simulator, and Mac OS X both in 32-bit and 64-bit variants.

That's what Kobold2D provides: lib service. In the first public release of Kobold2D, the following libraries are included:

- These are libraries for both iOS and Mac OS X projects:
 - Kobold2D (game engine code, Objective-C)
 - cocos2d-iphone (2D graphics, Objective-C)
 - cocos2d-iphone-extensions (utility code, Objective-C)
 - Box2D (physics, C++)
 - Chipmunk (physics, C)
 - Chipmunk SpaceManager (Chipmunk physics, Objective-C)
 - CocosDenshion (audio, Objective-C)
 - Wax (scripting, Lua)
- These are libraries available only for iOS projects:
 - cocos3d (3D graphics, Objective-C)
 - ObjectAL (audio, Objective-C)
 - SneakyInput (joystick, Objective-C)
 - iSimulate (library for iSimulate app)

Whenever an essential library such as cocos2d or cocos3d is updated, a new release of Kobold2D will follow within days, if not hours. There's no need for you to get active on the library front.

Some readers may wonder whether this means that Kobold2D apps are bloated, with all those libraries being included. You may be surprised to hear that early tests showed that Kobold2D apps are actually slightly smaller compared to cocos2d-iphone apps even though Wax and Lua are built into Kobold2D. The reason is that the Kobold2D projects are set up to allow the linker to throw away any code that isn't used in your app. That means if, for example, you don't include any of the Box2D headers, none of the Box2D library code will be linked with your app. But if you do want to start using Box2D, it's as simple as adding the Box2D.h header file to your project, and you're ready to start writing Box2D-enabled physics apps.

Kobold2D Takes Cross-Platform to Heart

By having one target for each platform (iOS and Mac OS X) by default in its project templates, Kobold2D allows you to build and run your code on both platforms from within the same Xcode project.

Developing for both iOS and Mac OS platforms is also encouraged by the Kobold2D API, which goes a long way to ensure that game engine code compiles for both platforms. For example, if you try to read the mouse button states on iOS, the code still compiles, and the result is a safe default, which in this case would simply report that no mouse button is pressed.

The Kobold2D Workspace

After downloading and installing Kobold2D, you'll find the most recent version of Kobold2D in a versioned subfolder of ~/Kobold2D, for example, ~/Kobold2D/Kobold2D-1.0. The installer will also open the Kobold2D Project Starter.app for you, which allows you to start a new Kobold2D project from one of the supplied project templates (see Figure 16–1).

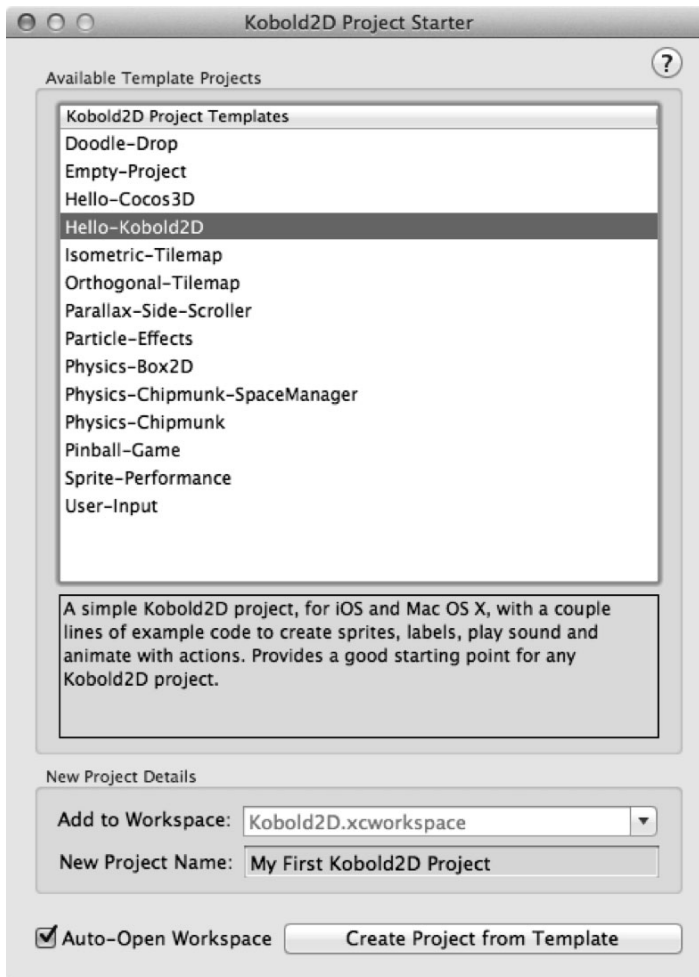


Figure 16–1. The *Kobold2D Project Starter* tool allows you to start new projects easily.

Select the Hello-Kobold2D template, enter any text in the **New Project Name** text field, and click **Create Project from Template**. You can also change the **Add to Workspace** text if you want to add the new project to a custom workspace (it will be created if necessary). By default, Kobold2D projects are added to the Kobold2D.xcworkspace.

Xcode should now open the Kobold2D workspace that contains your new project, as shown in Figure 16–2.

CAUTION: Kobold2D uses the new workspace concept of Xcode 4, which allows you to combine multiple projects in a single workspace window. If you open the .xcodeproj file of a Kobold2D project either from the most recently used list or by double-clicking it in Finder, the project will not work. You can easily spot this problem because the Kobold2D-Libraries project will be absent from the Project Navigator pane. Make sure to always open the corresponding .xcodeworkspace file that contains the .xcodeproj that you want to work with.

Consequently, Kobold2D is not compatible with Xcode 3.x since workspaces were first introduced with Xcode 4.0.

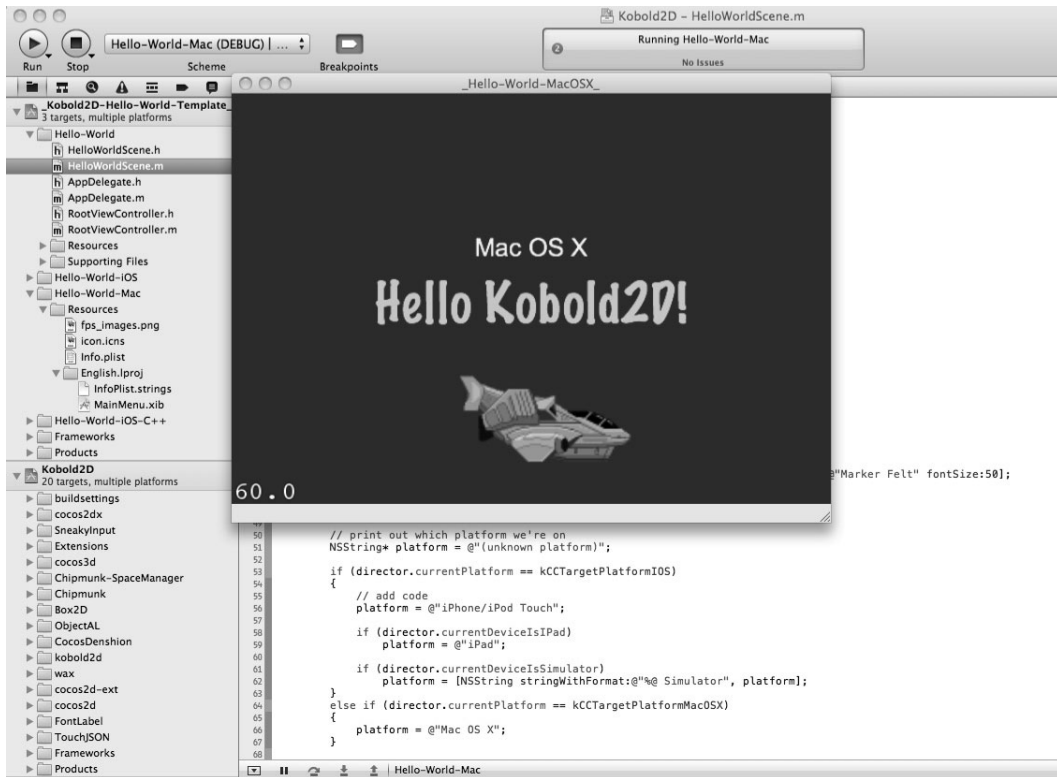


Figure 16–2. The Kobold2D Xcode 4 workspace view with the Hello Kobold2D Mac OS X project running

The Hello Kobold2D Template Project

Let's take a closer look at the Hello Kobold2D project (Figure 16–2) to demonstrate several key concepts of Kobold2D.

The Hello World Project Files

In Figure 16–3 you'll see the groups and files in the My First Kobold2D Project, which was created from the Hello-Kobold2D template project.

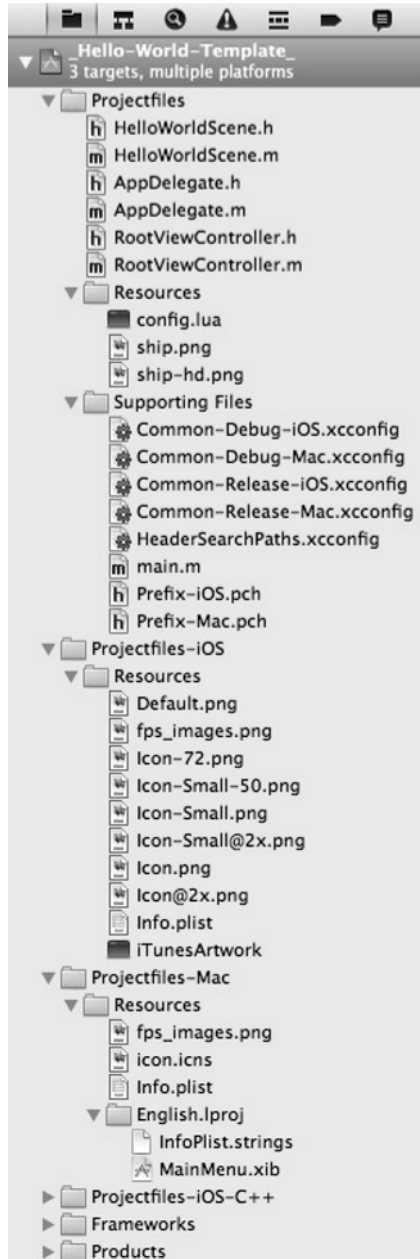


Figure 16–3. The default group structure of a Kobold2D project

The immediately noticeable difference compared to a regular cocos2d project is that three groups begin with *Projectfiles*, and there is the additional BuildSettings group. The idea behind that is to have a Projectfiles group and folder that contain the source code and resource files common to all platforms. The majority of your source code and resources will be in this folder. The additional Projectfiles-iOS and Projectfiles-Mac groups should contain only files used by that particular platform. These groups are meant as suggestions; you are of course free to structure your project's group as you see fit.

The BuildSettings group contains several .xcconfig files, which are the textual format for Xcode build settings. Normally you will not need to modify these files, but in some cases you may want to, for example, disable iSimulate or enable support for cocos3d. The huge benefit of the .xcconfig files is that they can be documented, and accordingly you'll find several notes for each build setting, what it does, what it affects, and when you might want to enable or disable it.

Almost all Kobold2D project templates provide targets for both iOS and Mac OS X, which are suffixed with -iOS and -Mac, respectively. I hope that the readily available Mac target will encourage you and other developers to consider cross-platform development from the start and to publish more apps to both Apple App Store. It's certainly a lot easier to take cross-platform development into account from the start rather than porting the app once it's finished.

How Kobold2D Launches an App

Kobold2D simplifies the start-up process, in particular how much custom code is needed in your project (by default: none). Kobold2D does all the initialization performed by the main function, the app delegate, and the root view controller behind the scenes.

Furthermore, Kobold2D allows you to easily modify the start-up settings, such as the first scene or layer to display, the device orientation, the render settings, the Mac window size, Retina support, and many more. All these settings are centralized in the config.lua script file.

Main, AppDelegate, and RootViewController

The main.m file is the entry point for each application; it contains the main method. The Kobold2D implementation simply calls its internal method KKMain:

```
#import "kobold2d.h"

int main(int argc, char *argv[])
{
    return KKMain(argc, argv, NULL);
}
```

KKMain takes both argc and argv arguments and a third, optional parameter that can be used to pass additional parameters to the start-up process if needed. Since that's rarely, if ever, needed, you can just pass NULL. KKMain hides the complexity of launching the

app for either the iOS or Mac OS platform. It also parses the config.lua file, which provides app configuration settings. I'll discuss the config.lua file in the next section.

At this point, it makes sense to show you the AppDelegate class of a Kobold2D project, or, rather, what's left of it. Here's the interface:

```
#import "KAppDelegate.h"

@interface AppDelegate : KAppDelegate
{
}

@end
```

Nothing there. Maybe there's more in the implementation part of the app delegate?

```
#import "AppDelegate.h"

@implementation AppDelegate

// Called when Cocos2D is fully setup and you are able to run the first scene
-(void) initializationComplete
{
}

@end
```

Nope. I can already hear you: dude, where's my app delegate?

Kobold2D takes care of the entire app start-up process for you and hides it within the KAppDelegate class from which the AppDelegate class inherits. In particular, KAppDelegate correctly sets up cocos2d based on the config.lua settings and encapsulates platform-specific app delegate code.

The KAppDelegate class is a regular UIApplicationDelegate on iOS and a NSApplicationDelegate class on Mac OS; you can implement (override) any of the app delegate protocol methods yourself whenever you need to do so. However, you should make sure to call the super implementation of overridden methods to ensure that KAppDelegate still performs its duties.

The only custom method that the AppDelegate class offers you is initializationComplete, which is called after the app and cocos2d have been fully initialized but before the first scene is about to be run. You can call the CCDirector runWithScene method inside initializationComplete to run a specific scene. However, it doesn't, because the config.lua file has a setting FirstSceneClassName that allows you to specify just the name of the class that cocos2d should run as its first scene.

You'll also notice the RootViewController class is equally and unimpressively minimal. The gist of the code is in Kobold2D's KKRootViewController. Once more, if you override any of the UIViewController messages, be sure to call its super implementation. Note that Mac OS targets do not use the RootViewController class.

In essence, the Kobold2D classes KKMain, KAppDelegate, and KKRootViewController provide the common functionality developers expect from these classes. Furthermore,

they provide the foundation code for any Kobold2D project, enabling you to use Lua for configuration files, among other things. You also have less code to maintain. In particular, if there's an essential change to these classes in iOS, Mac OS, or cocos2d, Kobold2D will take care of these changes or additions for you.

The Start-up Configuration File

Kobold2D always loads the Lua script file `config.lua`, which is in the Resources group of each Kobold2D project.

The `config.lua` script file returns a Lua table containing all the game's settings. A Lua table is a flexible data structure that combines the features of dictionaries (indexed by string) and arrays (indexed by value). You can create deeply nested Lua tables, which in terms of possibilities equal XML files or property lists but with a relatively simple, readable syntax with built-in error-reporting.

Lua scripts are text files, so naturally they're easier to edit than property lists, regardless of whether you're using a property list editor or editing the property list XML file directly. And Lua scripts allow you to comment on each line to explain your rationale, the range of valid values, and so on. You can't do that with property list files since property list editors don't allow you to comment on entries.

NOTE: Lua support for Kobold2D is provided by the Wax library created by Corey Johnson. Wax allows all Lua scripts to call any Objective-C method and to instantiate Objective-C classes, including cocos2d classes. However, Kobold2D avoids Wax scripting; it does not make calls to Objective-C methods from Lua. Instead, Kobold2D relies on just loading settings provided by Lua, executing Lua scripts and calling Lua functions. This is fast.

The other way around, calling Objective-C methods from Lua through Wax, is a lot slower. Wax performs a lot of Objective-C runtime method calls behind the scenes. This adds overhead that can quickly add up, especially during game play you don't want to waste precious milliseconds. Besides, even if you could live with that, the syntax of Wax scripts is too close to Objective-C to give you much benefit in terms of the number of code lines. Worse yet, Xcode has no support for Lua scripts, so there is no code completion, no syntax highlighting, no compiler warnings or errors, and no runtime debugging of Lua scripts. For those reasons, it is preferable to just use Lua for configuration files rather than attempting to write an entire game with Wax.

The `config.lua` file contains settings for just about everything you might want to tweak during the start-up process. Listing 16–1 shows an example `config.lua` file.

Listing 16–1. *The config.lua Script Contains All Start-up Settings for Kobold2D*

```

local config =
{
    KKStartupConfig =
    {
        -- load first scene from a CScene or CCLayer derived class with this name
        FirstSceneClassName = "HelloWorldLayer",

        -- set the director type, and the fallback in case the first isn't available
        DirectorType = DirectorType.DisplayLink,
        DirectorTypeFallback = DirectorType.NSTimer,

        MaxFrameRate = 60,
        DisplayFPS = YES,

        EnableUserInteraction = YES,
        EnableMultiTouch = NO,

        -- Render settings
        DefaultTexturePixelFormat = TexturePixelFormat.RGBA8888,
        GLViewColorFormat = GLViewColorFormat.RGB565,
        GLViewDepthFormat = GLViewDepthFormat.DepthNone,
        GLViewMultiSampling = NO,
        GLViewNumberOfSamples = 0,

        Enable2DProjection = NO,
        EnableRetinaDisplaySupport = YES,

        -- Orientation & Autorotation
        DeviceOrientation = DeviceOrientation.Portrait,
        AutorotationType = Autorotation.UIViewController,
        ShouldAutorotateToLandscapeOrientations = YES,
        ShouldAutorotateToPortraitOrientations = YES,
        AllowAutorotateOnFirstAndSecondGenerationDevices = NO,

        -- iAd setup
        EnableAdBanner = YES,
        LoadOnlyPortraitBanners = YES,
        LoadOnlyLandscapeBanners = NO,
        PlaceBannerOnBottom = NO,

        -- Mac OS specific settings
        AutoScale = NO,
        AcceptsMouseMovedEvents = NO,
        WindowFrame = RectMake(1024-640, 768-480, 640, 480),
    },
}

return config

```

Most of these settings should be self-explanatory and might seem familiar to you. The settings are documented in the KKStart-upConfig class in the Kobold2D API Reference and here: www.kobold2d.com/display/KKDOC/Config.lua+Settings+Reference.

For example, the FirstSceneClassName setting allows you to specify the name of a class inheriting from CScene or CCLayer (automatically wrapped inside a CScene), which will be

the first scene run by the CCDirector. You can enable iAd ad banners, change the way your app is autorotated, or provide the default window position and size for Mac builds.

The big advantage of using a Lua-based configuration file is that the entire start-up code is part of the Kobold2D code and can be updated in new versions if it needs to be. As Kobold2D matures, more settings will be added to the start-up config depending on what developers need to change or include most in their apps. In addition, you can create and use custom config.lua settings. The Hello World project provides an example of loading custom settings. I'll get to that in the next section.

TIP: You can learn more about Lua from the free *Programming in Lua* book, which is available online on the official Lua home page at www.lua.org/pil. The book is for an older version of Lua but still largely relevant. You might also want to browse the Lua reference manual at www.lua.org/manual to get a quick overview of the language. Lua has grown to be the number-one scripting language for game developers over the past ten years. It has a very small memory footprint and speed-wise often comes close to within 80 to 90 percent of the performance achievable with C programming.

The Hello Kobold2D Scene and Layer

Let's move on to the actual scene class HelloWorldLayer, which is set as the first scene via the config.lua setting `FirstSceneClassName = "HelloWorldLayer"`. You'll notice that this first scene is actually derived from CCLayer. Kobold2D realizes that and automatically wraps the HelloWorldLayer class into a CScene instance behind the scenes.

TIP: To avoid writing the repetitive `+(id) scene` method in each CCLayer class, you can simply call the `+(id) nodeWithScene` method in Kobold2D:

```
[[CCDirector sharedDirector] replaceScene:[MyGameLayer nodeWithScene]];
```

The HelloWorldLayer interface declaration is pretty unspectacular and provides only three instance variables that will later be loaded from the config.lua file:

```
#import "cocos2d.h"

@interface HelloWorldLayer : CCLayer
{
    NSString* helloWorldString;
    NSString* helloWorldFontName;
    int helloWorldFontSize;
}

@property (nonatomic, copy) NSString* helloWorldString;
@property (nonatomic, copy) NSString* helloWorldFontName;
@property (nonatomic) int helloWorldFontSize;

@end
```


Listing 16–4 shows the HelloWorldLayer implementation in its entirety. Apart from the previous call to KKConfig and the use of the CCDirector extensions and platform macros discussed earlier, it's still 100 percent cocos2d code.

Listing 16–4. Hello Kobold2D Implementation File

```
#import "HelloWorldLayer.h"

@implementation HelloWorldLayer
@synthesize helloWorldString, helloWorldFontName;
@synthesize helloWorldFontSize;

-(id) init
{
    if ((self = [super init]))
    {
        CCDirector* director = [CCDirector sharedDirector];

        CCSprite* sprite = [CCSprite spriteWithFile:@"ship.png"];
        sprite.position = director.screenCenter;
        [self addChild:sprite];

        // get the hello world string from the config.lua file
        [KKConfig injectPropertiesFromKeyPath:@"HelloWorldSettings" target:self];

        CCLabelTTF* label = [CCLabelTTF labelWithString:helloWorldString
                                                    fontName:helloWorldFontName
                                                    fontSize:helloWorldFontSize];
        label.position = director.screenCenter;
        label.color = ccGREEN;
        [self addChild:label];

        // print out which platform we're on
        NSString* platform = @"(unknown platform)";

        if (director.currentPlatformIsIOS)
        {
            // add code
            platform = @"iPhone/iPod Touch";

            if (director.currentDeviceIsIPad)
                platform = @"iPad";

            if (director.currentDeviceIsSimulator)
                platform = [NSString stringWithFormat:@"%@@ Simulator", platform];
        }
        else if (director.currentPlatformIsMac)
        {
            platform = @"Mac OS X";
        }

        CCLabelTTF* platformLabel = [CCLabelTTF labelWithString:platform
                                                    fontName:@"Arial"
                                                    fontSize:24];
        platformLabel.position = director.screenCenter;
        platformLabel.color = ccYELLOW;
        [self addChild:platformLabel];
    }
}
```

```

        glClearColor(0.2f, 0.2f, 0.4f, 1.0f);
    }

    return self;
}

@end

```

Notice the way I determine the platform (iOS, Mac OS) and device type (iPad, iOS Simulator) using director properties such as `currentPlatformIsIOS` and `currentDeviceIsSimulator`. These are some of the extensions to `CCDirector` I mentioned earlier.

You may be wondering why I haven't used preprocessor macros like `__IPHONE_OS_VERSION_MAX_ALLOWED` to determine platform and device type. First, in Kobold2D I wouldn't have used the hard-to-remember and verbose SDK macros. Instead, Kobold2D provides the simpler macros `KK_PLATFORM_IOS` and `KK_PLATFORM_MAC` to differentiate between the two supported platforms. If needed, you can also differentiate between iOS device and iOS Simulator by using the macros `KK_PLATFORM_IOS_DEVICE` and `KK_PLATFORM_IOS_SIMULATOR`.

The real reason of not using preprocessor macros and why conditional compilation with `#ifdef` should be used only as a last-resort measure is this: the compiler is your friend! Every time your code is compiled, the compiler lets you know that everything is in order or tells you whatever is technically or syntactically wrong with your code. It may be annoying at times, but the compiler is only letting you know that you made a mistake or forgot something. Allowing as much code as possible to be compiled by the compiler every time you build the code is so important for cross-platform development that the added overhead of platform and device runtime tests are entirely negligible.

Once you do cross-platform development, you'll likely spend a lot of time working on and compiling code for only one platform. Any code that is within an `#ifdef` for the other platform is invisible to the compiler, and it won't complain about errors. Now as soon as you switch targets and compile for the other platform, you'll likely run into errors that weren't there before. They may be related to code changes that you made an hour ago, a day ago, or maybe even a week ago.

Not only does it cause a lot of mental load to figure out which code change caused the error and what the correct fix will be, it's also frustrating because you'll frequently find that switching target platforms results in build failures. Either you'll spend more time than necessary building code regularly for both target platforms or you'll simply give up, maybe with the good intention of porting the project when it's finished. However, porting a completed project is much more work than developing it for both platforms from the beginning. Why? Because as long as your code is written for only one platform, your code could as well be entirely within an `#ifdef`.

Code that compiles works. At least it's technically correct. Immediate build errors are more likely to be corrected right away and easier to fix because you still have the most recent code changes in your short-term memory.

Running Hello World with iSimulate

If you run the Hello World project on the iPhone or iPad Simulator, you'll notice a network connection dialog as in Figure 16–4. You'll also notice the iAd banner that shows up because it was enabled in config.lua.

NOTE: If you see an `bannerView:didFailToReceiveAdWithError` error in your log with the message `The operation couldn't be completed. (ADErrorDomain error 1.)`, then this is most likely caused by the app not being set up for iAd. The iAd service needs to be enabled for each app and each developer in iTunes Connect. You'll find more information about enabling iAd for your app at https://itunesconnect.apple.com/docs/iTunesConnect_DeveloperGuide.pdf.



Figure 16–4. Network connection warning caused by iSimulate

The incoming network connections warning dialog is caused by iSimulate. It's nothing to worry about. The iSimulate library accepts incoming connections from the iSimulate app, which you can run on your WiFi-enabled iOS device in order to remote control the simulator. In other words, iSimulate enables you to test your game using your device but running in the simulator. All the features the simulator doesn't have, like GPS, Accelerometer, or Multi-Touch, can be simulated with the iSimulate app. It can be a real time-saver.

For example, if you have iSimulate running on your device and connected to your Mac, you will receive messages like `accelerometer:didAccelerate:` in your app even though the simulator doesn't have an accelerometer. This makes iSimulate an invaluable tool if you consider that running your app is usually a lot faster than deploying it to a device. I recommend giving it a try with Kobold2D's User-Input template project.

iSimulate is available on the App Store and normally costs \$15.99:
<http://itunes.apple.com/app/isimulate/id306908756>.

Doodle Drop for Mac with KKInput

So far, all the projects we created throughout the book were written for iOS. If you install Kobold2D, you'll notice that not only are most of the book's projects included in Kobold2D, almost all of them also have a Mac OS version.

So, what would it take to make a project like Doodle Drop from Chapter 4 work both on Mac and on iOS? Not that much actually. It turns out that by far the biggest change is related to handling user input. Thankfully, Kobold2D provides a platform-agnostic user input handler that simplifies user input dramatically by allowing you to test the state of input devices at any time in any class and method.

First, the `accelerometer:didAccelerate` event method has been removed since it is no longer needed. Instead, KKInput will be responsible for providing the app with acceleration values. We tell it to activate accelerometer input and to set the filtering factor in the `init` method of the Doodle Drop `GameLayer` class:

```
// Yes, we want to receive accelerometer input events.  
[KKInput sharedInstance].accelerometerActive = YES;  
[KKInput sharedInstance].acceleration.filteringFactor = 0.2f;
```

Enabling the accelerometer will first test whether the device supports the Core Motion framework. If so, acceleration values will be provided by Core Motion, which gives us a tiny performance benefit. In all other cases, the standard `UIAccelerometer` interface is used to obtain acceleration values. The filtering factor is a percentage that determines how responsive the game character will react to sudden changes in acceleration.

Listing 16–5 shows the modified Doodle Drop update method that includes user input handling for both platforms.

Listing 16–5. Handling User Input for Both Platforms with *KKInput*

```

-(void) update:(ccTime)delta
{
    KKInput* input = [KKInput sharedInput];
    if (isGameOver)
    {
        if (input.anyTouchEnded || ⌘
            [input isKeyDown:kKKKeyCode_Space] || ⌘
            [input isKeyDown:kKKKeyCode_Return])
        {
            [self resetGame];
        }
    }
    else
    {
        [self acceleratePlayerWithX:input.acceleration.smoothedX];

        if ([input isKeyDown:kKKKeyCode_LeftArrow])
        {
            [self acceleratePlayerWithX:-keyAcceleration];
        }
        else if ([input isKeyDown:kKKKeyCode_RightArrow])
        {
            [self acceleratePlayerWithX:keyAcceleration];
        }

        // The rest of the update code remained unchanged.
    }
}

```

The update method processing is split into handling the gameover and the rest of the code that runs while the game is commencing. If the game is over, we simply check whether any touch ended or the spacebar or Return key was pressed before resetting the game, starting over.

While the game is running, the player’s velocity is updated either based on the `input.acceleration.smoothedX` value or based on a constant `keyAcceleration` value when either the left or right arrow key is held down. The `acceleratePlayerWithX` method is shown in Listing 16–6 later in the chapter, which contains the code previously in the `accelerometer:didAccelerate` method.

`KKInput` provides you with built-in high-pass and low-pass filters via properties of the `KKAcceleration` class exposed by the `input.acceleration` property. You can access the raw acceleration, the smoothed (low-pass filtered), and the instantaneous (high-pass filtered) values. In most games, you will want to use the smoothed values, which provides steady acceleration and cancels out sudden, short-lived movements. Instantaneous acceleration values are useful whenever you want to react to sudden acceleration movements, such as shaking or quickly flipping the device.

You’ll notice that the input code does not use conditional compiling via `#ifdef`. If you run this code on Mac, the `anyTouchEnded` method is guaranteed to return NO all the time.

Likewise, when running on iOS, the `isKeyDown` method always return NO, since there's no keyboard available. And the `input.acceleration` values are all 0 on Mac OS.

If it seems wasteful to you to test for keyboard events on iOS and touch events on Mac OS, please keep in mind that the additional overhead is very small while the benefit of always compiling all your code guarantees that it continues to work for both platforms. If the platform-specific code is extensive, you can always branch it using the Kobold2D CCDirector extensions like `currentPlatformIsIOS` and `currentPlatformIsMac`.

Listing 16-6. Updating the Player's Velocity

```
-(void) acceleratePlayerWithX:(double)xAcceleration
{
    // adjust velocity based on current accelerometer acceleration
    playerVelocity.x = (playerVelocity.x * deceleration) +
        (xAcceleration * sensitivity);

    // we must limit the maximum velocity of the player sprite, in both directions
    if (playerVelocity.x > maxVelocity)
    {
        playerVelocity.x = maxVelocity;
    }
    else if (playerVelocity.x < -maxVelocity)
    {
        playerVelocity.x = -maxVelocity;
    }
}
```

Everything else requires no changes to create the Mac OS port thanks to the fact that no hard-coded positions and offsets were used. Nevertheless, it makes sense for game play reasons to restrict the Mac window size to that of an iPhone in `config.lua`:

```
WindowFrame = RectMake(300, 300, 320, 480),
```

The Lua function `RectMake` creates a rectangle with the given origin (300, 300) and size (320, 480). `RectMake` creates rectangles that are compatible with `CGRect` or `NSRect`, depending on the platform. Additional Lua utility functions are `PointMake` and `SizeMake`.

Entering the Third Dimension with cocos3d

In February 2011 the first release of the `cocos3d` add-on library made it possible to add the third dimension to `cocos2d` apps with relative ease. `Cocos3d` is modeled after the `cocos2d` API and is currently available as an early beta version, labeled v0.6.1. Sadly, it's not part of the `cocos2d-iphone` distribution, and correctly setting up a `cocos3d` project is not as straightforward as one would hope it to be. Installing Kobold2D is by far the easiest way to try `cocos3d`.

Kobold2D comes with a Hello-Cocos3D template project, which is based on the official `cocos3d` project template and includes the necessary changes to integrate it smoothly with Kobold2D.

You can find the documentation for `cocos3d` on the official web site at <http://brenwill.com/cocos3d>.

Changes to the AppDelegate Class

The entire start-up code for cocos3d is in the AppDelegate class. Although it was entirely empty in the Hello World project, the class can now be put to good use. The AppDelegate interface now has a new instance variable:

```
#import "KKAppDelegate.h"
#import "CCNodeController.h"

@interface AppDelegate : KKAppDelegate
{
    CCNodeController* nodeController;
}
@end
```

The CCNodeController is provided by cocos3d and is needed for two things. For one, it allows you to perform autorotation with cocos3d. The regular RootViewController class is not fully compatible with cocos3d; it tends to stretch the view or displaces nodes while rotating. So, cocos3d provides its own view controller solution for autorotation, unfortunately without the rotation animation.

The other benefit of the CCNodeController class is that it has built-in support for the UIImagePickerController, which enables your app to show the camera view in real time as the background of the scene. This allows you to create so-called augmented reality apps.

Figure 16–5 shows an example of my iPhone’s camera scanning the solar system poster I have above my computer on the wall. The image gives the phrase “Hello World” an entirely new meaning (depicted is the earth, *Erde* in German).

CAUTION: Check the framerate counter in Figure 16.4. My poor iPhone 3G (second-generation device) was heavily stressed by the real-time camera background. It was averaging around 20 frames per second (fps), and even without rendering the Hello World 3D model it did not go above an average somewhere between 30 to 40 fps. My iPod touch 4 did not perform that much better and averaged at about 40 frames per second, with or without the 3D model. Apparently it has enough reserves for rendering 3D models in this scenario. The performance tests were made with release builds, of course.

Despite the performance penalties, I hope you’ll come up with cool augmented reality apps. Just witnessing the cocos2d/cocos3d scene overlaying the live camera view is an inspiring experience!



Figure 16-5. Augmented reality: using the iPhone's camera as real-time background image

Setting up the cocos3d node controller to show the camera background is easy as pie. In Listing 16-7 you only need to set the variable `useNodeController` to YES, and the camera background is automatically enabled, if the feature is available. It's only one line of code that tests and enables the camera overlay:

```
nodeController.isOverlayingDeviceCamera = nodeController.isDeviceCameraAvailable;
```

This code assigns the result of the camera availability check directly to the `isOverlayingDeviceCamera` property, enabling it on every device that has a camera.

The AppDelegate implementation in Listing 16-7 uses the `initializationComplete` code to perform all that is needed to set up cocos3d. This method is perfect because it is called right after everything is set up except for the actual running scene. And since you'll be running a scene in this method either directly with the `CCDirector runWithScene` method or indirectly via the `CCNodeController's runSceneOnNode` method, the `FirstSceneClassName` setting in `config.lua` will be silently ignored.

Listing 16-7. The AppDelegate Implementation of a cocos3d Project

```
#import "AppDelegate.h"
#import "Hello3DLayer.h"
#import "Hello3DWorld.h"
#import "HelloWorldLayer.h"

@implementation AppDelegate

-(void) initializationComplete
{
```

```

CC3World* cc3World = [Hello3DWorld world];
CC3Layer* cc3Layer = [Hello3DLayer node];
cc3Layer.cc3World = cc3World;

[cc3World play];
[cc3Layer scheduleUpdate];

ControllableCCLayer* mainLayer = cc3Layer;

BOOL useNodeController = NO;
if (useNodeController)
{
    CCLayer* helloLayer = [HelloWorldLayer node];
    [mainLayer addChild:helloLayer z:-1];

    nodeController = [[CCNodeController controller] retain];
    nodeController.doesAutoRotate = YES;
    [nodeController runSceneOnNode:mainLayer];

    // Let's have some fun with the camera
    nodeController.isOverlyingDeviceCamera=nodeController.isDeviceCameraAvailable;
}
else
{
    CCScene* scene = [CCScene node];
    CCLayer* helloLayer = [HelloWorldLayer node];
    [scene addChild:helloLayer];
    [layer addChild:mainLayer];
    [[CCDirector sharedDirector] runWithScene:scene];
}
}
@end

```

The cocos3d scenes always consist of two parts. An instance of the CC3World class defines the world, loads the 3D models, and sets up the 3D scene. The CC3World is also the root class of the 3D scene and contains nodes derived from CC3Node that make up the 3D scene hierarchy.

A CC3Layer class instance is responsible for integration with the cocos2d engine. The CC3Layer class instance can be added to any cocos2d node, because it is derived from the cocos2d class CCLayerColor. In addition to that, it holds a reference to the CC3World instance and is responsible for updating the 3D world.

This explains why you need to call both `play` on the `cc3World` and `scheduleUpdate` on the `cc3Layer`. The layer's update method will be forwarded the world, which can be either in a play or pause state depending on whether the play or pause method was called last. Whenever you call pause on the CC3World, the world will not be rendered, but the last frame remains visible. Also, any cocos2d actions that are currently running on 3D nodes will continue to be updated and move the 3D nodes. To pause and resume the actions as well, you need to pause or resume each 3D node that is running an action:

```

[[CCActionManager sharedManager] pauseTarget:node];
[[CCActionManager sharedManager] resumeTarget:node];

```

The cocos3d start-up code implementation I chose allows you to switch between using the cocos3d node controller, which enables autorotation, and the live camera background. In this case, the HelloWorldLayer is added as a child to the CC3Layer and with a negative z value so it is displayed entirely behind the cocos3d layer. You could also use a positive z value of 1 or higher to display the scene in front of the cocos3d layer. Either approach works best if your app is primarily a 3D application that uses only some cocos2d elements, for example for its user interface.

The alternative to using the CCNodeController is to create the regular cocos2d scene and then add the CC3Layer as a child of the scene. Now you can easily decide which of the 2D nodes in the scene are behind or in front of the cocos3d layer by adjusting their z value. This approach is preferable if your app is primarily 2D but you want to be able to render 3D models occasionally. It will also be faster since it does not use the CCNodeController. Just to give you some ballpark numbers, my iPhone 3G renders 30 fps with the CCNodeController enabled and 50 fps without, whereas my iPod touch 4 is unaffected and maxed out at 60 fps either way.

Whichever approach you choose, you can always embed the CC3Layer in a cocos2d scene so that some cocos2d nodes are behind and others are in front of the 3D nodes. It's a simple matter of using the `addChild` method's z parameter or, respectively, the `zOrder` property. You can see a sandwiched 3D model in Figure 16-6.



Figure 16-6. You can have cocos2d nodes both in front of and behind the CC3Layer.


```

CCActionInterval* tintUp = [CCTintTo actionWithDuration:tintTime
                                red:startColor.r
                                green:startColor.g
                                blue:startColor.b];
CCActionInterval* tintCycle = [CCSequence actionOne:tintDown two:tintUp];
[helloTxt runAction:[CCRepeatForever actionWithAction:tintCycle]];
}

```

The first thing the 3D world needs is a camera through which you view the scene. The camera affects only cocos3d nodes and will not change how cocos2d nodes are displayed. The camera location property defines the position of the camera, which newcomers to the 3D world might find confusing at first. If you move the camera, all cocos3d nodes will move in the opposite direction. Imagine looking through the viewfinder of your camera and then moving the camera to the right, and you'll see all objects in the camera view moving to the right. It's the same with CC3Camera except that you have to imagine the screen of the device to be the viewfinder through which you look into the 3D world.

For reference, increasing the values for the camera's x, y, and z locations will move the camera to the right, up, and toward the viewer, respectively. This will have 3D objects move to the left, down, and away from the viewer (zoom out). Decreasing the x, y, and z location will move the camera view to the left, down, and away from the viewer. The 3D objects will move to the right, up, and toward the viewer (zoom in). Of course, you can always have the camera at a fixed location and move the 3D nodes instead if you find that easier to work with.

A CC3World is not complete without at least one light. Without light, your 3D models would be illuminated equally on all sides and would make them look flat despite being 3D. By default a CC3Light node creates a directional light, comparable to that of a flashlight. If you want to achieve the effect of a lightbulb illuminating all of its surroundings equally (the preferred choice), you need to set isDirectionallyOnly to NO.

With camera and lights set, all that is missing is action. Before we get to that, we need a 3D model to display. You can load 3D models from POD files with the CC3Node method addContentFromPODResourceFile. This will load the POD file and add it as a CC3MeshNode instance to the world. You can later get a reference to this node with the getNodeNamed method and by supplying the name of the mesh as set in the 3D modeling program. The hello-world.pod mesh has the name Hello.

TIP: Cocos3d v0.6.1 supports loading of PowerVR POD files only. Additional 3D model file loaders are listed in the road map for Q4 2011 but without specifying which file formats will be supported. Until then, you can either use POD or write your own loader code.

The POD format is an optimized binary format that most 3D modeling applications can't directly export. However, almost all of them can export to the COLLADA format, which you can then convert to POD. The developer of cocos3d Bill Hollings has an excellent tutorial about converting COLLADA files to the POD format: <http://brenwill.com/2011/cocos3d-importing-converting-collada-to-pod>.

The tutorial assumes you are using the Blender 3D content creation suite, which is very popular because it's free and open source. Many other 3D modeling programs cost four- or even five-digit sums. You can get Blender at www.blender.org and overlook some of its oddities. After all, you have just saved a big pile of money.

The method `createGLBuffers` should be called after loading all 3D mesh models. It optimizes the vertex arrays for rendering. And `releaseRedundantData` simply performs a cleanup to release some memory afterward.

The second half of the `initializeWorld` method is left to animations with `cocos2d` and `cocos3d` actions. Some additional actions are provided by `cocos3d` like `CC3RotateBy`, `CC3MoveBy`, and essentially all actions that manipulate positions needed to have versions that work in the three-dimensional coordinate system. Other actions like `CCSequence`, `CCRepeatForever`, or `CCTintTo` are from `cocos2d`. They work as always and can be mixed with `cocos3d` actions.

In essence, there are three action blocks. The first rotates the Hello World 3D model around its own axis. The second has the camera bounce back and forth. And the third action block performs a repeated color cycle on the Hello World 3D model.

Adding cocos3d to an Existing Kobold2D Project

You can always start with a regular Kobold2D project template and only later decide that you want to include `cocos3d` features in your app. There's only one thing that you'll have to do manually; otherwise, you'll get runtime errors like this unrecognized selector error:

```
-[Hello3DWorld addContentFromPODResourceFile:]: unrecognized selector sent to instance
```

This is a common error that can occur when an app links to a static library that contains Objective-C categories.

CAUTION: The often proposed simple solution to this problem is to use the `...all_load` linker flag. However, this is not recommended and specifically not for Kobold2D, because the linker would blindly include each and every library and framework whether it's being used or not. The final executable will be several megabytes larger if you use the `...all_load` flag in a Kobold2D project.

The recommended solution is to explicitly specify the library that must be forcibly included in the executable via the `...force_load` linker flag:

```
-force_load $(BUILT_PRODUCTS_DIR)/libcocos3d-ios.a
```

You will have to append this line to the build setting **Other Linker Flags** of the target that needs the cocos3d library. Depending on which other libraries are already being linked to your app, the executable size will increase by several hundred kilobytes up to around 1.5 megabytes. Since that's a size that isn't negligible, cocos3d is currently not fully integrated with Kobold2D and requires this manual step.

Summary

I hope that this chapter has given you a good impression of how working with Kobold2D will help you make games and apps easier and with more possibilities. And on the www.kobold2d.com web site, you'll have access to the API documentation of all libraries, a programming guide, the support forum, the feedback section, and a road map that allows you to see how Kobold2D development is progressing.

One of the most important features of Kobold2D is its ability to use Lua tables to define settings. You'll then be able to feed these settings directly into properties of class instances with just a single call to the `KKConfig` class. This is even more important if you work with others who need to make changes to the app but do not want to or should not have to change the source code.

Kobold2D also makes cross-platform development for iOS and Mac OS easier and provides a convenient, one-stop class for handling user input, as you saw in this section. You'll also find plenty of template projects in Kobold2D based on projects created throughout the book and subsequently ported to work on Mac OS.

New to the cocos2d world is the cocos3d add-on library. You learned about the basics of cocos3d in the discussion of the Hello Cocos3D template project. And I think you'll appreciate it that you can readily try cocos3d now and understand the basic distinction of the `CC3World` being the root node for the cocos3d nodes while the `CC3Layer` allows you to plug the 3D world anywhere into a cocos2d scene.

What's left is for you to go to www.kobold2d.com now to download the latest version, install it, and start experimenting with the provided template projects.

Out of the Ordinary

This final chapter contains no source code. I'm not even going to talk much about the cocos2d engine. Instead, I'd like to focus on where you can go after you've read this book—for example, to ask questions and to learn more. But you should also investigate which technologies may be useful to implement in your game, such as advertising, analytics, one of the many social networking libraries, and even server technology used in persistent world games.

Everything you ever wanted to know is probably somewhere on the Internet. It may simply be hiding. If you want to know where to find art, audio, and freelancers, I'll provide good starting points. And I'll continue to provide more information and links on my blog, so be sure to visit my web site, at www.learn-cocos2d.com.

I'm also giving you a glimpse into marketing and public relations in this final chapter. Those are topics that are often asked about, full of mystery and misunderstandings. They include working with a publisher and how you can benefit from such a relationship and also how to market your game and yourself.

For an independent developer, it's very important to be recognized by the community as a creative, enthusiastic game developer and to connect with the community. All of your social networking efforts will then help you promote your game simply by being able to reach out to more like-minded people. If you can build a network of followers, the success of your game will follow. A lot of people get that mixed up and think it's the other way round. It's not.

You'll also learn about the reference games and apps made available with cocos2d. They'll give you a good impression of what's possible with the cocos2d game engine and also what you can achieve as an independent developer. One of the most exciting learning tools is other people's source code, so I've included a list of commercial cocos2d source code projects that are on sale for exactly this reason.

Most of all, *out of the ordinary* should be the guiding principle for whatever you do. Create something that's different, and don't be afraid to be different.

Additional Resources for Learning and Working

The purpose of this section is to help you find answers to your questions, get support for a particular problem, obtain more source code to learn from and base your own games on, and of course introduce you to all the cocos2d tools and some of the best cocos2d reference games.

It is also the purpose of my blog at www.learn-cocos2d.com/blog. You'll find the latest updates and cocos2d game development tips and tricks in my blog where I post a new article on average once per week.

Where to Find Help

Whether you are facing a technical problem that you can't solve on your own or need more people to work on your game, you can get help. In addition, if you're looking for art, audio, or tools, I know just the places where you can find what you're looking for, or at least where you can begin your search.

TIP: If you get stuck and you don't know what else to try, it can help to just write down what the problem is, what you're trying to achieve, and what you've done so far. Most of the time it frees your mind to think of things you haven't tried yet, and more often than not it leads to a solution. If not, you now have a summary that you can post to a forum or Q&A site, which will help you get a good answer more quickly. The art of asking questions is all about making it easy for others to answer them.

Cocos2d Home Page

This may seem obvious, but if you have a question related to cocos2d, you should stop by and join the cocos2d community in the forum: <http://cocos2d-iphone.org/forum>.

In the cocos2d forum, you can ask about anything related to cocos2d. It has subforums for hot topics like audio programming, physics engines, social networks, cocos3d, and ads, as well as a general forum for Objective-C and iPhone SDK-related questions. For the most part, the cocos2d community is friendly and very helpful, and a lot of great example code and development stories have been shared on the forum.

Before asking questions, be sure to search both the forum and the official cocos2d documentation wiki: <http://cocos2d-iphone.org/wiki/doku.php>.

In addition, you can announce your newly released game in the cocos2d games forum. Don't forget to also add it to the list of games made with cocos2d. You can do this on the cocos2d Games page: www.cocos2d-iphone.org/games.

Cocos2D Central

I launched a community hub called Cocos2D Central for everything related to cocos2d. Most importantly, there are forum sections for cocos2d, the book, Kobold2D and my game kits. Cocos2D Central also hosts the official forum of cocos2d-javascript, the web browser port of cocos2d.

In the Resources section, there are several small tutorials available, and the Downloads section hosts all files I currently offer for download including the cocos2d installer and the source code for this book.

Stack Exchange Network

Forums are a great way for communities to interact with each other. But as such, they tend to get a little chatty, and searching for a specific answer can be cumbersome because the forum's content isn't strictly limited to solving problems.

That's exactly where Q&A web sites like Stack Exchange shine. You go in, ask a particular question, and get answers. Since the focus is on Q&A, it's easier to find existing answers. And if you really like a question or answer, you can vote it up so it will be listed higher on search results.

I'm regularly amazed by the show of expertise from contributors on the Stack Exchange network. This is in part thanks to the built-in badge and points system, making it very rewarding to both ask interesting questions and write thoughtful, in-depth answers. The Stack Exchange network is comprised of several free Q&A web sites, the most popular and my personal favorite being <http://stackoverflow.com>, which is about programming questions in general.

You won't find as many questions about cocos2d on Stack Overflow as you will on the official cocos2d forum, but the questions on Stack Overflow are good ones, and almost all of them get good answers. There's a bit of confusion about the use of search tags on the site, with both *cocos2d* and *cocos2d-iphone* used to tag questions regarding the iPhone version of cocos2d. This can be attributed to the success of cocos2d for iPhone, in that it has become synonymous with the name cocos2d itself. Use these two links to find all the cocos2d for iPhone-related questions on Stack Overflow:

- <http://stackoverflow.com/questions/tagged/cocos2d>
- <http://stackoverflow.com/questions/tagged/cocos2d-iphone>

Stack Exchange is expanding as a Q&A web sites network. One of the latest additions is the Game Development Q&A web site. On this site you can ask general game programming questions and about anything game development-related in general, including design, marketing, and sales. Check out the Game Development Stack Exchange site at <http://gamedev.stackexchange.com>.

Tutorials and FAQs

There are plenty of tutorials for cocos2d on the Web, but one tutorial writer clearly stands out from the crowd: Ray Wenderlich. He has written more than a dozen cocos2d tutorials and published them on his web site, at www.raywenderlich.com/tag/cocos2d. Besides cocos2d tutorials, on this site you'll also find other highly interesting iPhone SDK-related tips and tricks. In at least one particular case this is also helpful for cocos2d developers, where Ray explains how to save and load your app's data. This can be applied to saving and loading games as well (see www.raywenderlich.com/1914/how-to-save-your-app-data-with-nsencoding-and-nsfilemanager).

Then there's the wonderful Coconut College. It provides entry-level tutorials and a handholding approach. The web site's clean and professional design is both inviting and to the point. The author goes by the nickname cjl and calls himself the Dean of Coconut College but otherwise keeps a very low profile. Check out Coconut College at www.coconutcollege.net, and if you do want to reach out to the author, you can try his cocos2d forum profile: www.cocos2d-iphone.org/forum/profile/49.

The knowledge base on my Learn & Master cocos2d web site (www.learn-cocos2d.com/knowledge-base) hosts my own tutorials and FAQs. Currently I offer an Xcode tutorial, which allows you to keep the cocos2d files separate in order to be able to update cocos2d more easily. Another tutorial is about Hiero, the bitmap font editor, and includes a number of FAQs. This site is also where I blog about topics and post links of general interest to cocos2d developers. Take this article on how to upgrade cocos2d in an existing project for example: www.learn-cocos2d.com/2011/05/update-cocos2d-iphone-existing-project.

Another web site I'm running is called the Indiepinion, at www.indiepinion.com, where I blog about the indie experience, post interviews with indie developers, and generally try to encourage developers to walk the indie way.

Source Code Projects to Benefit From

Personally, I learn best by browsing through other people's code. One of the first things I did when I got cocos2d and found the documentation to be lacking was to invest in the Sapus Tongue source code project to see how cocos2d is actually being used in a game. It was very helpful for getting started quickly.

NOTE: Since Zynga acquired certain assets of Sapus Media, the Sapus Tongue Source Code and Level SVG products are no longer available. However, the source code products I'm introducing in the following sections are excellent replacements.

The advantage of commercially sold source code compared to the open source projects is that you rarely get support for the latter, and they're almost never updated, to the point that most of the open source projects you'll find are using versions of cocos2d

prior to v1.0, with one exception, the official cocos2d-iphone-extensions project: www.cocos2d-iphone.org/forum/topic/17546.

The following source code projects are all commercial offerings. I refrain from listing prices because they are subject to change.

The iPhone RPG Game Kit

Created by Nathanael M. Weiss, the iPhone Game Kit is also known as the iPhone RPG Kit. The game *Quexlor: Lands of Fate* (see Figure 17–1) is a role-playing hack-and-slash game following in the footsteps of *Diablo*, featuring a large tilemap world with multiple levels, various monsters, and plenty of items. The game kit comes with a huge amount of royalty-free graphics created by Reiner Prokein, a comprehensive *Make Your Own iPhone Game* e-book, and a publishing guide that explains in detail how to submit your game to the App Store. Obviously, you also get the game's excellently crafted source code, but that almost seems secondary.

There are no notable licensing restrictions. You get a 60-day money-back guarantee and free updates for life, although it does not specify whose life. Support is available through the support forum, where you can also share your work with other Game Kit customers.

You can find the link to the *Quexlor* game on the App Store and a whole lot more information about the iPhone Game Kit on its web site, so I suggest taking a look: www.iphonegamekit.com.



Figure 17–1. *Quexlor: Lands of Fate (iPad)* is a game made with the iPhone Game Kit.

Line-Drawing Game Starterkit

My Line-Drawing Game Starterkit is modeled after successful games like *Flight Control* and *Harbor Master*. If you like to create line-drawing games, this starter kit gets you going with drawing lines, moving objects along the paths, collision detection, and a clearly structured code base, including both iPhone and iPad versions.

Each purchase grants you a site license, which means your whole team is allowed to use the game's source code and assets. Since it's a starter kit, you are naturally allowed to make clones of the game. I also offer a 30-day money-back guarantee, and I'll be happy to help you learn the starter kit's source code and give directions on how to extend it.

The Line-Drawing Game Starterkit's product page contains the starter kit's feature list, links to the demo apps (see Figure 17-2), a code sample, and the complete documentation: www.learn-cocos2d.com/store/line-drawing-game-starterkit.

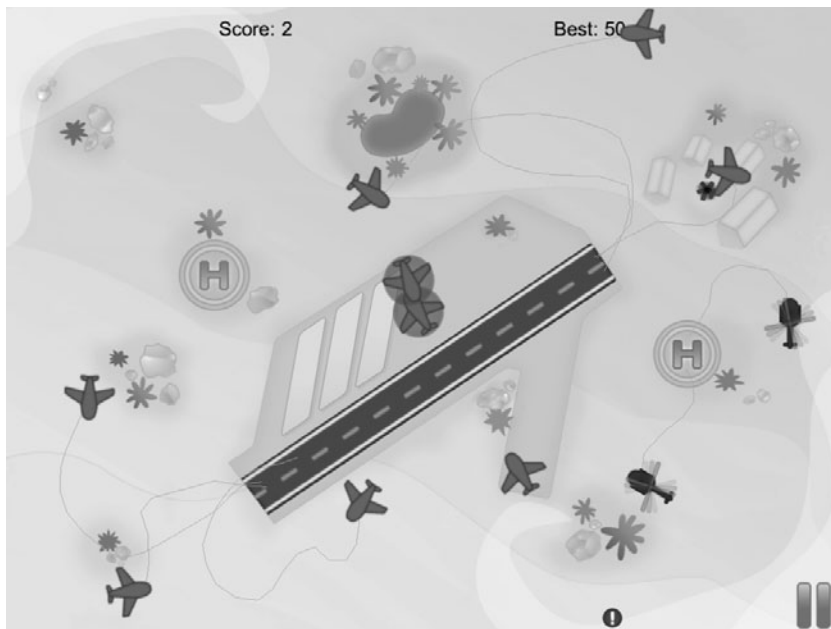


Figure 17-2. *The Line-Drawing Game Starterkit example game (iPad)*

The Platformer Game Kit

My second source code product is all about making platform games like *Sonic*, *Mario*, *Giana Sisters*, *Braid*, *Super Meat Boy*, and so on. The first version models the game play of the *League of Evil* iPhone game (see Figure 17-3) but will be extended to support other types of platform games as well, based on your feedback.

NOTE: I'm not affiliated with the developers of the League of Evil game, and I did not help develop it. I chose to re-create the League of Evil game play as the first milestone for the Platformer Game Kit because League of Evil is a fun and popular game and it contains the most important elements of a platformer game. Check out the game at www.leagueofevilgame.com.

The kit is built with Kobold2D and with cross-platform development in mind so that you can easily create and deploy your platform game to iOS devices and as a Mac OS X application. The game levels are created with the Tiled Map Editor using the free game assets supplied with the kit.

The source code is designed to be modular and flexible so that you can easily extend it with your own functionality. For example, many game parameters are editable via Lua configuration files, and ready-made behavior classes can be added or removed to individual actors to allow them to fall, jump, collide, die, shoot, animate, play sound, and so on.

Learn more about the Platformer Game Kit and get the latest development updates on www.platformergamekit.com and www.learn-cocos2d.com/store.



Figure 17–3. Create games similar to *League of Evil* (depicted here) with the Platformer Game Kit.

The Space Game Starterkit

The Space Game Starterkit evolved from Ray Wenderlich's Space Game Tutorial (www.raywenderlich.com/3611/how-to-make-a-space-shooter-iphone-game) into a commercial starter kit for making a side-scrolling shoot-'em-up game.

As with the iPhone RPG Game Kit, there's added value in the form of four epic tutorials included in the purchase. The tutorials explain a lot of the details that go into making this particular game and games in general. Coming from Ray Wenderlich, you can expect high-quality code and tutorials.

Ray's wife, Vicki, provided the artwork for the Starterkit. You can reuse and modify the artwork as you like as long as you credit Vicki Wenderlich. She runs an iPhone art blog at www.vickiwenderlich.com.

Learn more about the Space Game Starterkit, pictured in Figure 17-4, in Ray Wenderlich's store: www.raywenderlich.com/store/space-game-starter-kit.



Figure 17-4. Ray and Vicki Wenderlich's Space Game Starterkit

iUridium Source Code

If you had a Commodore 64 home computer in the 1980s or early 1990s, chances are that you've played or at least heard of Uridium. Uridium was a fast-paced space game shooter with a unique feature: you could change the flight direction of your spaceship with a smoothly animated Immelmann turn, to fly over the enemy spacecraft from end to end to complete various objectives. If you will, it was one of the early free-roaming world type of games.

Nenad Alajbegovic is porting this game to the iPhone, appropriately named iUridium. You get the entire source code for the game, which uses just about everything the cocos2d game engine has to offer, and then some. For example, Nenad implemented caching (pooling) of bullets and enemies to create a smooth game play. He also has UIKit views, Game Center leaderboards, and Facebook and Twitter integrated. The levels are created entirely as tilemaps and dynamically loaded from XML into the game's scene and layer nodes.

The license is fair, only asking you not to make a clone of iUridium but expressly allowing you to make your own side-scrolling shooter game. But you do have to replace all of the artwork, music, and sound effects with your own.

You can find all the information about iUridium, pictured in Figure 17–5, on this web site: www.iuridium.com/?page_id=2.

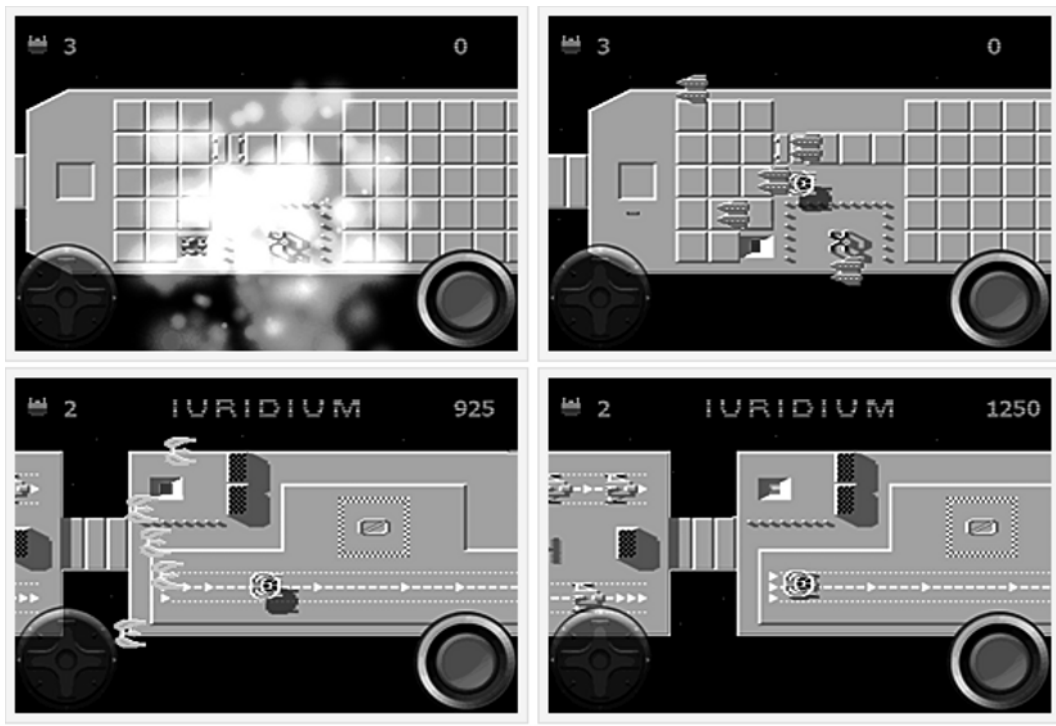


Figure 17–5. The iUridium's game source code is available for sale.

BATAK Duel Source Code

Dan Nelson is the developer of BATAK Duel for iPhone. He created this game in five months with no prior iPhone game development knowledge. He managed to integrate OpenFeint into the game, a menu systems, and of course savegames. The app is rich in visuals, both by utilizing cocos2d's particle system as well as additional effects such as lighting, smoothly scrolling credits, and a transparent pop-up keyboard.

The license is fair, only asking you not to use any of the BATAK Duel artwork, music, and sound effects. You can find more information about the BATAK Duel source code, pictured in Figure 17–6, on this web site: www.batakduel.com/blog/78.



Figure 17–6. *The source code of BATAK Duel for iPhone is for sale.*

Cocos2D Podcast

Mohammad Azam and I are recording the Cocos2D Podcast to report on recent events; to talk about hot topics; to interview game developers, tool authors, and bloggers; and to give listeners insider information and insights into game development with cocos2d.

The Cocos2D Podcast is available on cocos2dpodcast.wordpress.com, and new episodes are announced on www.learn-cocos2d.com/blog. Each episode runs between 30 to 60 minutes. Here's a selection of episode topics we've covered so far:

- Earning additional revenue using iAds, IAP, selling source code, and so on
- Michael Daley explains Particle Designer and Glyph Designer
- Ray Wenderlich talks about Cocos2d, book, workshops, and more
- Andreas Löw explains TexturePacker and PhysicsEditor

- Vladu Bogdan on LevelHelper and SpriteHelper
- Zynga acquires Cocos2d contributors
- Tools in Cocos2d iPhone game development
- Game engines and frameworks as alternatives to Cocos2d

Tools, Tools, Tools

In this book I've introduced you to what I think are the best tools for each purpose. However, there are usually always alternatives. If you also read the first edition of this book, you know that I used different tools in the first edition than in the second. In less than a year, some tools put on wings, and others essentially stopped progressing and fell behind.

Since that almost natural process is not going to change in the future and it's hard to predict which tools are here to stay and which aren't over the coming years, I wanted to share an alphabetically sorted list of currently existing tools that you can use for cocos2d without going into describing them and regardless of their current state, except of course that they need to be usable and functional at the very least.

The following tools can be used for development with cocos2d:

- **Bitmap Font Tools**
 - BMDFont (Windows): www.angelcode.com/products/bmfont
 - Fonteditor: <http://code.google.com/p/fonteditor>
 - Glyph Designer: <http://glyphdesigner.71squared.com>
 - Hiero: <http://slick.cokeandcode.com/demos/hiero.jnlp>
 - LabelAtlasCreator: www.cocos2d-iphone.org/forum/topic/4357
- **Particle Editing Tools**
 - ParticleCreator: www.cocos2d-iphone.org/forum/topic/16363
 - Particle Designer: <http://particledesigner.71squared.com>
- **Physics Editing Tools**
 - Mekanimo: www.mekanimonet.net
 - PhysicsBench: www.cocos2d-iphone.org/forum/topic/9064
 - PhysicsEditor: www.physicseditor.de
 - VertexHelper: www.cocos2d-iphone.org/archives/779
- **Scene Editing Tools**
 - CocosBuilder: <http://cocosbuilder.com>
 - Cocoshop: www.cocos2d-iphone.org/forum/topic/15668

- LevelHelper: www.levelhelper.org
- Texture Atlas Tools
 - DarkFunction Editor: <http://darkfunction.com>
 - SpriteHelper: www.spritehelper.org
 - TexturePacker: www.texturepacker.com
 - Zwoptex: zwoptexapp.com
- Tilemap Editing Tools
 - iTileMaps (iPad): www.klemix.com/page/iTileMaps.aspx
 - Tiled Map Editor: www.mapeditor.org

Cocos2d Reference Apps

The following is a list of games and apps made with cocos2d. They should serve as shining examples of what you can do with cocos2d as well as the creativity of cocos2d developers.

Instead of providing an iTunes link to each app in the book, I decided to create a post on my blog where I host all the links to the games mentioned here, and I'll update this list to include noteworthy games released after the book has been published. You can find the list of links to these apps, including other links of interest, on the Great Apps Made With cocos2d page: www.learn-cocos2d.com/2010/10/great-apps-made-with-cocos2d.

- **The Elements** (iPad) is a graphical representation of the periodic table of elements. The outstanding features are the plentiful photographs and smooth 360 animations that invite you to explore the elements that make up you, me, and the rest of the universe (excluding empty space, of which there's a lot I've been told). It's priced highly but worth every cent, and if you need an app that will let you brag about your new iPad, this is it!
- **Bloomies** is a colorful gardening game, full of bees (see Figure 17-7). If that doesn't sway you, maybe the idea of fostering and nurturing your own garden does. The flowers need your constant attention, and the game play is addictive, just like any Tamagotchi-style game. Oh, and it happens to be made by two former colleagues of mine. It's just a beautiful game, and so is their follow-up game, Super Blast.



Figure 17–7. *Bloomies from Phantoom Entertainment*

- **StickWars** is a game where you defend your castle from incoming stick figures by flicking them in the air or literally shaking them to the ground. The developer, John Hartzog, had never before worked with Objective-C or on mobile devices, but he pulled it off. StickWars remains to this date within the top 100 games and continues to be updated even a year after the initial release.
- **ZombieSmash** is also a castle defense game, except that this time hordes of zombies are attacking, and you get explosives, 16-ton weights, shotguns, and other cool items that make a bloody mess to fend them off (see Figure 17–8). Your castle is your barn, and if you can defend it, you'll be rewarded with a slow-motion animation of the final zombie losing its, err, unlife. The outstanding feature of this game is certainly the rag doll animation system that allows zombies to walk, crawl, or otherwise try to move even if they lost some of their limbs.



Figure 17–8. *Zombie Smash* by GameDoctors

- **Super Turbo Action Pig** revives the simple game play concept of a scrolling level where your character always falls down, except when you touch the screen to boost his jetpack. The extraordinary part here is that the game's graphics are extremely well made and the overall presentation of the game, the trailer, the web site, and the humor set a great example.
- Then there's **Farmville**—do I even have to explain what it's about? It's an incredibly successful Facebook game that has millions of players worldwide building their farms in an isometric landscape. It just goes to show how powerful cocos2d is if a company like Zynga uses it to port its most successful game to the iPhone. It's notable that *Zombie Farm* came out on the iPhone before *Farmville*, and it was also created with cocos2d.
- **Melvin Says There's Monsters** (iPad) is a beautifully animated cartoon kid story with professional-quality voiceovers. The story is cleverly constructed and has an insightful turning point. It's a pleasure to watch even for an adult, and it also uses cocos2d's page-flip animations very effectively. If you have an iPad and kids, it's a must have!

- **Trainyard** is an innovative puzzle game that was clearly engineered with the user in mind. It features a mode for the color-blind, is optimized to use little battery power, saves and loads the game just as the user left it, and even allows users to share puzzle solutions on the Web, using a duplicate of the game engine written in Flash. This all besides being a really innovative puzzle game where you lay tracks and combine trains to match them with colored train yards.
- **Abstract War 2.0** is a dual-stick shooter featuring colorful and vibrant geometric visuals (see Figure 17–9). It's obviously inspired by Geometry Wars on Xbox Live Arcade. It's an intense space-shooter with plenty of game modes. You can even play it in multiplayer mode via a Bluetooth connection, and it allows you to use your own iPod music.

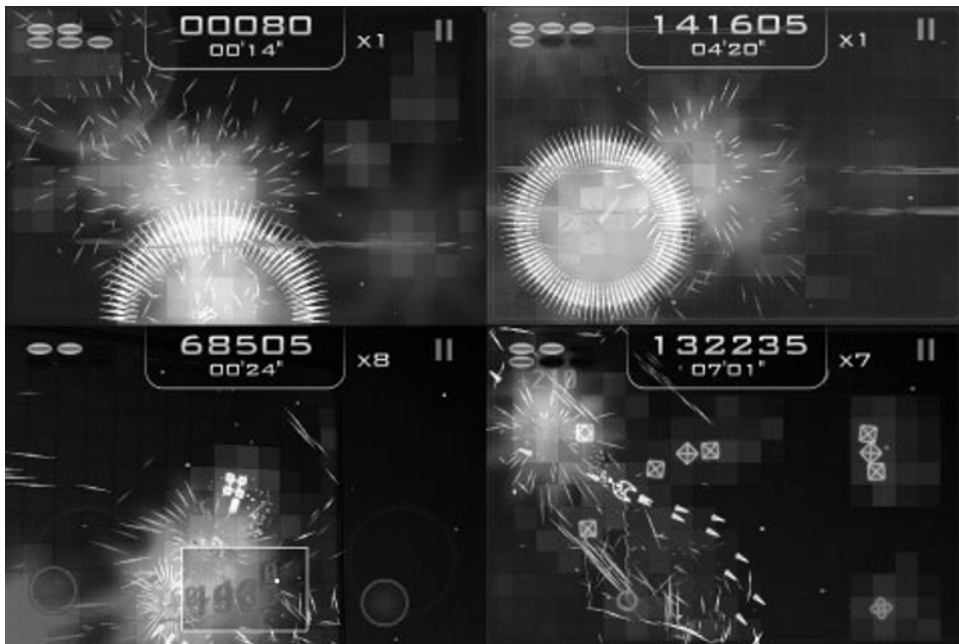


Figure 17–9. *Abstract War 2.0* by Forzefield Studios SL

- **Fuji Leaves** is an interesting music game, where dropping balls hit leaves, and depending on speed and location of impact, a sound is played. With several balls on the screen bouncing around, you can dynamically create musical scores. It's intensely fascinating to play this game, trying to come up with interesting scores and just the right placement of leaves. Before you know it, an hour has passed.

The Business of Making Games

This section covers social gaming, ads, and In App Purchases, basically, the business of making games. In this case, the focus isn't on technology; in this section, I'll talk about my experience as a professional and independent game developer and what it takes to make successful games.

At least the tips provided in this section will improve your odds to make a more successful game with the right people, and you'll learn how to market the game to your target audience.

Working with Publishers

In the early days of computer programming, developers single-handedly copied their game code onto floppy disks and shrink-wrapped it to sell it in local software and hardware stores. Those days are long gone, and with the increasing complexity of games, publishers took on the task of distributing and marketing games. This model has been the default for the past 20 years or so, but with the advent of the indie game scene in recent years, mostly driven by new technologies such as downloadable games, the Xbox Live Indie Games channel, and Apple's App Store, more and more game developers are turning to self-publishing again. So, why would you want to work with a publisher these days?

The first aspect is technical support and testing. Publishers want to make sure that every game they release does not reflect badly on them, so one of their goals is to release games with as few bugs as possible and with polished game play. If you're not used to this process, which includes the scrutiny of a quality assurance team, you'll be in for a surprise, and it may not always be a pleasant one—such as if the game's release is held back due to yet another obscure and hard-to-find bug. So, that's a bad thing? To the contrary, it forces you to work with an attention to detail that's all too often neglected and dismissed by hobbyist developers. And in the end, you'll be rewarded with a better game. In most cases, this also translates to more favorable reviews by press and players alike and thus more sales.

Working with an established publisher also reflects positively on you, simply because of the respected games already released by said publisher. I don't mean that in an ego-trip kind of way. You're not special because you're working with big-name publisher X—but you sure will learn a lot, and that's going to make you stand out from the crowd. It also improves your street cred and the chances to land a better job or contract sometime down the road.

The experiences working with a publisher don't just involve game programming and the things you'll pick up from technical support and the quality assurance process. You'll also sign a contract and get a glimpse at all the legalese and paperwork involved (it's really not that bad).

Signing a standard contract with a publisher is more like opening a bank account. Everything is ready-made, and you just have to fill in a few blanks. Expect the publisher to take anywhere between 30 percent to 70 percent of the revenue. It depends a lot on what they do, in which phase of development they come on board, and whether they support you financially.

If you allow yourself to learn from working with a publisher, it can be very insightful to get an impression of the details that need to be specified to get everything in order from a financial and legal point of view. If the publisher has a good track record with other developers, you can feel assured that they're not going to screw you over. But you definitely need to understand the terms you're signing, because if you're used to posting your sales numbers, those may now be covered by a nondisclosure agreement you signed with the publisher.

You do give up a certain amount of freedom, and you need to be able to live with it and trust the publisher to do the best job they can for those parts you're giving up. For example, if a publisher is asking you for a change in direction about certain aspects of your game, you should seriously consider it. At least most of them know what they're talking about, and they also know what's working and what isn't (this certainly isn't the case for all of them, though). However, since games is a very subjective field, publishers do have the tendency to favor proven sets of features over other risky but innovative ones□ but less so on the iOS, where publishers are more willing and able to give their developers creative freedom, if not total control over the design of the game.

In return for giving up some freedoms, they'll reward you with marketing for your game. They know the channels, and they have a direct feed to the review web sites and the press. And that's possibly another area of expertise you'll learn a lot from. The press has a certain way they like to receive and consume the information they need to write a review about your game. Your publisher knows all about it, and they'll request a few things from you that you wouldn't have considered in the first place, such as high-resolution artwork or a one-line catchphrase that really gets players interested in your game.

If you get the chance to cooperate with a publisher, my advice is to go for it at least once, no matter how much you value your creative freedom. Afterward, you'll understand much better what you're giving up and what you're receiving in return, at least in that particular case. Whatever your experience may be, the experience alone and what you'll learn should make it worthwhile (and maybe it will be rewarding enough to do it again□ but if not, you'll know why).

There are several game publishers you might want to consider contacting. Your best shot would be those who market specifically for iOS or mobile devices, and in that area two names stand out from the crowd. One is ngmoco, who published Goldfinger, We Rule, We Farm, Epic Pet Wars, and Rolando. Visit ngmoco's web site at www.ngmoco.com. The other is Chillingo, who has released games like Minigore, the Quest, Knight's Rush, and of course Angry Birds. Their web site is www.chillingo.com.

Finding Freelancers

If you believe in your game and you are ready to invest money into it, you may want to get help from professional freelancers worldwide. Besides asking for jobs in the cocos2d forum itself, you can also post a job offer on one of the more popular outsourcing web sites. Likewise, you can also offer your expertise as a freelancer to employers on these sites. There are plenty of web sites offering such services; you'll even find some that can hook you up with someone close to where you live.

I'll refer you to the ones that I know work best and have a good reputation: they are eLance (www.elance.com) and Guru (www.guru.com).

They both work on the same principle. You post a job offer, which could be a small task or an entire project. Then you receive proposals from candidates, from which you can pick one or more to do the job. Once you have received, reviewed, and approved the work, the freelancers get paid. Abuse can be reported and basically locks you out of the platform, so while you need to be able to trust people you've never worked with, the risks are minimal for both parties. To start, I recommend choosing smaller tasks. Just as if you were to get into the stock market, you want to get a good feel for how this works and what can happen by starting small.

Finding Free Art and Audio

The alternative to hiring someone to do the work for you is to find the work available on the Internet (preferably free). However, be careful about anything that's "free." I'm not saying that it might have a catch; I'm warning about the common misconception that free means "can be used freely." It may not cost you anything to get an image or audio file that's free, but that doesn't tell you anything about what you're legally allowed to do with it. That's typically where a license should come into play but often doesn't. A lot of people publish their own source code, artwork, audio, and writings for free on the Internet but forget to add a proper license file to it if the work is intended to be used by others. The problem is that by default, the author has the copyright and retains all rights to how the work can be used. If there's no explicit waiver, preferably in the form of a license, then you should not use this work (especially not in commercial products, and that includes \$0.99 apps sold via the iTunes App Store).

For reference, I'd like to point you to Funplosion Labs, which has an article listing web sites where you can get free game graphics and audio. Funplosion also disclaims this with a warning about the copyright and a link to the license agreement for each web site (see <http://funplosion.com/free-assets.html>).

CAUTION: Be wary of the General Public License (GPL), especially if it is used by a source code library that you want to use or integrate into a commercial app. Using GPL-licensed code in your own project requires you to open source your own project's source code. Not just that, but anyone is subsequently given the right to use your source code and to copy, modify, and redistribute it. Note that the Lesser General Public License (LGPL) license is not as stringent. Refer to this link for a comparison of common source code licenses:
http://developer.kde.org/documentation/licensing/licenses_summary.html.

Finding the Tools of the Trade

Sometimes you may wish you had a tool that does just *that*, whatever *that* might be. There are times as a game developer where you need to process data, modify images, or build whole worlds—things that are tedious and error prone to do in code or simply too time-consuming to do on your own because you'd rather focus on writing your game.

My tip is to use the Indie Game Tools web site, which collects, categorizes, and allows others to rate game design tools for independent game developers. The focus here isn't on expensive software used by professional studios but on low-cost and free solutions for about anything, from game engines to converters, from scripting languages to server technologies, and from asset packages to game editors. Maintained by Robert —Robc— Charney, it's the place I go to see if there's a tool available that fits my bill. Visit the site here: <http://indiegametools.com>.

Just a minor caveat: don't put too much weight on the ratings of individual tools. The number of ratings is very small, so there can be huge differences between similar tools, and the ratings may be biased by both unhappy users and proactive communities. You should leave your mark and add some of your own ratings so that over time the ratings become more accurate. In addition, some tools simply can't be compared by rating; they may have totally different uses, making it unfair to compare them based on their rankings on the Indie Game Tools web site.

Marketing

So, you made a game and submitted it to the App Store. Now what? How do people find your game in the first place?

The story starts at the beginning. The moment you begin working on your game should also be the starting point for your marketing efforts. Get a web site up and running, and post your development experiences and maybe some work-in-progress screenshots. That should be the first step to connect with other game developers.

In terms of the marketing and business aspects, I'd like to save you some time. The following link is to the Big List of Indie Marketing and Business Tips, and I have to say

that's an understatement. You'll find most of the meaningful and intelligent articles ever written on the subject on just this one page, so be sure to check it out even if you're only mildly interested in the marketing and business aspects of indie game development. Check it out at www.pixelprospector.com/indev/2010/08/the-big-list-of-indie-marketingand-business-tips.

Still interested in marketing? Good, check out the free e-book *Videogame Marketing and PR*, written by Scott Steinberg. You can get it on the book's web site: www.sellmorevideogames.com.

Marketing Your Game and Yourself

If you're really not sure whether your game is going to be a success based on how much fun it's going to be, one rather effective way to increase your chances is to make a game that's very presentable and colorful and just looks like great fun in screenshots. With the low barrier to entry on mobile devices because of the low price points, a great-looking game with little in terms of play depth more often than not will outperform and outsell a complex game with multiple game modes and many hours of potential game play. It may pain some developers because it goes against everything they believe in, but it's an unavoidable fact of life. Of course, if you can combine the two and create an innovative, fun game that just looks great, the opportunities are endless.

If you browse reviews, there's one thing you should be giving a lot of attention to. The most prominently featured screenshots are always action scenes from the game, usually with a lot of things going on at the same time. This has almost become an art form; professional developers will even develop specialized tools to stage scenes where they can take the best possible action screenshots, without having to play the game and rely on luck to get it just right. Maybe you should consider that, too. So, the best artwork you can create or pay for should definitely be a priority for you, and so should making a convincing trailer for your game. Outstanding presentation is a very important buy-in to get to talking with a publisher as well.

This advice goes for your blog as well. You do have a blog, don't you? If not, start one right now! You can get a free blog on <http://wordpress.com>, and you'll learn how to work with the most popular blogging software along the way. As your blog matures, you may want to consider hosting your blog on your own server; you can do that with WordPress as well, but with a lot more options for customization through plug-ins and themes.

Your blog is an important piece for marketing yourself. The most important aspect here is that self-marketing will indirectly benefit anything you do from now on. Blogging gets you in touch with other like-minded people, who sooner or later will be willing to help you, sometimes for free. If you are really making a name for yourself in your development community, it's good news, because it could get you attention from publishers, who follow the same channels you're working in (as opposed to most of the players of your game, which you'll have to reach out to on a different level).

And since your blog should be your public face, it should be you who's talking. Don't put on a mask and try to sound like a big corporation— for example, by saying things like what you're doing is going to revolutionize the way we play games. I call that . . . a term I won't write out in this book, but I'm sure you know what I mean.

Make sure you don't come across as cocky on your blog, but likewise don't belittle yourself. You may be a beginner, but you're learning, so focus on what you've learned. The things you do learn will seem like things that millions of other developers already know, so you might wonder whether you should really blog about them. I'm familiar with these thoughts, and frankly speaking, you'll always have doubts about whether what you write is going to be of interest to others. It is, trust me. And if it isn't, no big deal. The truth is, even though there are millions of developers who may already know about what you post, there are millions more who don't and who will be able to learn from you.

And remember to put your best skills up front and avoid blogging about your weaknesses. You may not know it all, but you can learn it. If you really want to know how multiplayer game programming can be enhanced by predicting client movements, learn about it. The web is a great resource— collect what you find, and blog about what you found and learned. Others will respect you for it. Blogging takes years of practice, so it's best to start now because it will pay off in the end— possibly in ways you can't even imagine right now.

Another important way of marketing yourself is by using Twitter. Once you have something to tell the world, you'll be happy to be able to reach out to dozens, hundreds, if not thousands of your followers at once. It's a very effective marketing tool. How to use it effectively is a matter of following simple steps. First, don't protect your tweets— it seriously limits the number of people who are going to follow you. (I certainly won't.) Then provide an interesting bio, which should include your interests, what you do, and anything else that makes you seem like an interesting person to follow. Simply using a joke, poem, or quote as your bio is a no-no. And don't forget to link to your blog! The most important thing is to tweet regularly, and tweet about things others might find interesting. Tweeting only about yourself and your products (or just retweeting other's posts) isn't going to convince many people to follow you.

Public Relations and Press Releases

If you work alone and you don't want to cooperate with a publisher, hiring a public relations (PR) firm or agent to give yourself and your game a better chance in the market may be a wise decision. But it could also be downright stupid and pointless. In the latter case, it's usually not the fault of whoever is doing the PR; it's a matter of understanding the benefits of PR— what it takes to make it work effectively— and assessing whether it's worth spending thousands of dollars on.

As an independent developer, you are likely to have a very limited budget. Even entry-level prices for PR agents are going to make your jaw drop to the floor. If that's the case, walk away and get back to working on your game. Your gut reaction is correct.

Now, if you already have a game out there and it's presentable, and you've earned not just money but also some influence with the people around you, things may be a little different. Perhaps you have some money to spare, you've learned a lot from your first game, your second one is even better, and you already have players waiting for it to come out. Can you give it a boost with the help of professional PR? In this case it's more likely.

If you take on the help of a PR agent, you should definitely try to find one who has a track record of working for the game industry—preferably with independent game developers and the mobile games niche. They're not easy to find, so ask around. Using a PR agent who isn't into game development is a waste of time and money—you need to have PR with the ability to reach out to your game's target audience. But most of all, the deciding factor should really be your game.

If you know you have something special, testers tell you so, and the game just looks gorgeous, then investing a few thousand dollars in PR might be a good choice. PR wants game reviewers to write about your game and game development sites to take notice of your special abilities. If these aren't clearly visible, professional PR won't be able to make a big difference. PR works best if you can provide gorgeous screenshots and an intriguing, funny, and captivating trailer movie.

How do you know if your game is something special? By asking the people who wouldn't hesitate to tell you what they don't like. Family and friends are typically too kind to provide the sometimes harsh criticism needed to improve a game. You should ask on forums for private beta testers and provide them with an appetizer, which could be a screenshot or a description of your game's special features, so that they'll be more interested to try your game. How you deal with criticism and feature requests, and generally how you interact with your testers, is also a matter of PR. Dealing with user feedback, and specifically criticism, is a vital skill you should hone as early as possible.

But who's to say you can't try for yourself first? There are press release services specifically for independent developers, and they cost a fraction of what a professional PR agent would charge you. And writing a press release isn't that hard if you follow the rules. The following press release services tap into exactly the right channels for game players and developers. The hottest candidates are the Indie Press Release Service, at www.gamerelease.net, and the Game Press Release Submission Service at Mitorah Games, at www.mitorahgames.com/Submit-Game-Press-Release.html.

Unsurprisingly, both services are run by independent developers themselves. Specifically, Juuso Hietalahti created the Indie Press Release Service and also runs the very insightful www.gameproducer.net blog. This blog is especially interesting if you want to learn more about production and marketing aspects.

You should also consider the Games Press web site, which is frequented by game journalists worldwide (see www.gamespress.com/about_howtosubmit.asp).

Engaging Players for More Revenue

Every platform has a number of peripheral technologies that are helpful if not essential, at least in some cases and for some developers. On the iOS platform, this includes the ever-growing list of social networking platforms to choose from, besides Apple's Game Center. Then there's server development kits that you might need for developing persistent games that ought to connect to your own server, be it to find and run matches of more than four players or simply to save characters, progress, settings, and worlds online.

And sometimes regarded dubiously, providing ads in games can create an additional revenue stream, especially for free games and lite versions. Often in conjunction with ads, you can also investigate whether it would help your game to add analytics and metrics in order to find out where players fail most often, what buttons they click the most, and how frequently they play individual game modes. This can help you tweak your game to be more fun for more players.

Engaging Your Players

The big buzzword in the game industry is clearly **social**. Whether buzz or bubble, whether it has investors, reviewers, and players look up or yawn, the social gaming component is growing stronger and has become expected. Social gaming entails anything from passively pushing updates to networks like Twitter and Facebook to directing player interaction in multiplayer games.

In addition, push notifications are a powerful tool for you to remind your users about important events happening in your app. Be it an update, new content, or just the next game you released, it gives you an additional channel to keep your users engaged with your apps and your brand.

Social Networks

Besides Apple's Game Center technology, there are a number of different social networking platforms. All are more or less similar in that they allow players to connect, post high scores, earn achievements, and do many other things, including posting game events to Twitter and Facebook—and all of them are free for both players and developers!

Since their feature sets are constantly evolving and the market for social networks is booming, the final decision is up to you. I'll list the big players here and mention some of their outstanding features.

NOTE: Most of the social networking SDKs as well as ShareKit (<http://getsharekit.com>) already include support for connecting your users with Twitter and Facebook, so you don't need to learn and implement the separately available Facebook and Twitter APIs. If all you need is access to Twitter, you should have a look at the excellent MGTwitterEngine API from Matt Gemmell, at <http://mattgemmell.com/2008/02/22/mgtwitterengine-twitter-from-cocoa>. And if you need to integrate Facebook into your app, the official Facebook iOS SDK is located on GitHub: <http://github.com/facebook/facebook-ios-sdk>.

- **OpenFeint** is the perceived leader of iOS social networking SDKs. It boasts Game Center compatibility and turn-based multiplayer features to stand out from the crowd. But first and foremost it's very popular, with an audience of players in the millions. Have a look at OpenFeint's developer portal: www.openfeint.com/developers.
- **Scoreloop** sets itself apart from the competition by offering additional revenue streams via downloadable content, sharing virtual goods and in-game currencies. It also includes analytics to determine what your players are doing. Take a feature tour on the Scoreloop home page: www.scoreloop.com.
- **Plus+** is the social networking platform created by ngmoco, a big publisher of iOS games. It's currently available only to ngmoco's development partners. Ngmoco is looking for select developers to partner with; if you think you have what it takes, then you might want to apply on its developer web site: <http://plusplus.com/developers>.
- Chillingo's **Crystal** is the other big iOS publisher's social networking platform. And just like with Plus+, access is limited to developers working with Chillingo, respectively Electronic Arts, the new owners of Chillingo. You can get information on the Crystal SDK on <http://devsupport.crystalsdk.com>.

If you are missing Agon-Online in this list, you probably haven't heard the news that it shut down on June 30, 2011. As for Geocade, it is still available at www.geocade.com but simply does not play a role among its competitors.

Socket Server Technology for Multiplayer Games

If you're looking to build a multiplayer game that requires more sophisticated server-side game logic and storage than the social networking platforms are able to offer, you will have to write your own socket server. Luckily, the difficult part of writing a server/client architecture with socket connection and other networking voodoo has already been done for you.

Hosting your online games on a socket server has several advantages. For some games, it's very important that players can't cheat, so running the most critical game logic

isolated on the server while being able to verify the data coming from clients can prevent a lot of common cheating mechanisms. A server-based approach is also the only way to write a game that supports a lot of players at once. With peer-to-peer technology, you quickly run into scalability issues, because each device's bandwidth is rather limited, and every player in a peer-to-peer match adds computational overhead to every device connected to that match. Apple's Game Kit restricts peer-to-peer connections between devices to a maximum of four players for a reason. The server hardware is ultimately more powerful and has a higher bandwidth than any device connected to it, which allows you to host more players in the same match. In addition, the server hosts the database and usually runs verifications that ensure that players haven't tampered with their app or the stream of data coming in to defeat cheats and exploits.

- **Electrotanks's Electroserver** is a feature-rich server technology used by professionals worldwide. It's priced accordingly, but it does have a free version that allows for up to 25 concurrent users to be connected with the game server. If that's sufficient for you, give it a try: www.electrotank.com.
- **SmartFoxServer** by gotoAndPlay() follows in the tracks of Electroserver. It's not as feature rich, but it's very popular among independent game developers, mostly because it has a completely free Lite version, while the Basic and Pro versions still allow up to 20 users for free. In general, the prices are more affordable by independent developers. Compare the SmartFoxServer editions on the products page: www.smartfoxserver.com/products.
- **Exit Games** follows a two-pronged approach by offering a networking component called Photon and a managed service for developing persistent online games called Neutron. Neutron natively supports the development of turn-based multiplayer games, and it can suspend and resume game sessions. Photon has a free version that supports up to 50 concurrent users per app or server, and Neutron has a free trial version. Compare features and prices on the Exit Games web site: www.exitgames.com.

Push Notification Providers

Push notifications are little alert messages popping up on your iPhone's screen, whether you're using it or not. Some apps use them to inform users about the stock market, about a sports game's events as it is being played, or simply to let you know that there's a new comic strip available. The uses are endless. The greatest power of push notifications is that it gives your users an incentive to keep using your app; they won't forget it as easily if they let the app remind them about news and events.

You may have heard of Apple's push notification service, but if not, you can learn more about it in Apple's Push Notification Programming Guide: <http://developer.apple.com/library/ios/#documentation/NetworkingInternet/Conceptual/RemoteNotificationsPG/Introduction/Introduction.html>.

Push notifications allow apps to broadcast messages to individual devices. If you implement one of the social networking platforms, you'll get this feature for free, and you don't need to care about the complexities of push notification programming. However, in some cases you might need to implement your own solution, and for that you need a push notification provider. That's a server that can communicate with Apple's servers and the iOS devices. You can write your own server with the free and open source `apns-sharp` library (see <http://code.google.com/p/apns-sharp>), or you can sign up for the Mono Push service (www.monopush.com) or Urban Airship (<http://urbanairship.com>).

iCloud

With iOS 5 Apple introduced iCloud, its cloud-based storage solution for storing documents and simple key-value data on the cloud. What cloud? Well, obviously not that white, fluffy stuff floating way above your head. Cloud storage simply means storing data on a remote server, and the *cloud* part refers to the fact that you don't know exactly which server your data is stored on, or how. You just send data to the cloud, and the cloud server stores it. The next time you want your data back, the cloud server finds it for you and delivers it back to you.

Cloud storage allows game developers to store savegames in the cloud so that the game's progress isn't locked into a particular device and app. If your app is available for both iOS and Mac OS, you could store savegames on the cloud and the user can continue where he left off on both versions and on any of his devices and computers.

Or you could save the persistent world of your Mac OS massively multiplayer game to the cloud, and the iPhone app could allow the user to change the player's inventory, sell items on the auction house, and do all the other chores while he's on the road.

You can learn more about iCloud here: <http://developer.apple.com/icloud>.

Earning More Revenue

Don't we all want to make more money? With apps, you can do so by displaying ads or by selling in-game content via In App Purchases. Both are powerful tools in particular for free apps, because apps that cost nothing get on average about ten times more downloads than if the same app would cost \$.99.

Ads and Analytics

Advertisements provide a way to generate additional revenue from apps and games. It can be very lucrative if the app is very popular, but frankly speaking, it's also easy to put off users by displaying ads, and it's much more likely to make little to no money. It's still worth experimenting with advertisements—for example, if you use them as a motivator in free versions of your app by letting your users know that the full version removes the ads—besides offering all the other cool features the players want, of course.

Regardless of whether you dislike ads, there's another reason to evaluate the ad space. That's because most advertising SDKs can also provide you with insightful metrics and

analytics. This not only includes statistical data such as which iOS version your users have installed and which device they're using but also includes custom metrics, such as which game mode is played most often and where players tend to fail more frequently. This allows you to tweak the difficulty and the user interface layout according to how the users are actually using your app and to plan upgrades and your marketing strategy.

- **iAd** by Apple is the first contender when it comes to advertising on iOS devices. You can get an overview of the iAd Network on Apple's advertising web site: <http://advertising.apple.com>.
- **AdMob for App Developers** is also a very popular ad platform among game developers. You can learn more about the iOS version of AdMob at www.admob.com/appdevs, while the Mobile Analytics product is hosted on this web site: <http://analytics.admob.com>. AdMob also offers AdWhirl (www.adwhirl.com), which allows you to display ads from a variety of ad providers.
- **Flurry Analytics** is the only product that focuses solely on metrics and analytics. Note that Flurry bought Pinch Media, and its products have merged into Flurry Analytics, in case you're wondering why I'm not mentioning the very popular Pinch Analytics. Begin your research into Flurry Analytics here: www.flurry.com/product/analytics/index.html.

In App Purchases

Even if you sell your app, you could publish a free (lite) version of your app that uses In App Purchase to unlock the full version while the user is running the app. For you it can be as little as setting a `BOOL` variable to `YES` after the purchase was made, but for the user it's so much more convenient not having to go through the App Store to make the purchase. I have no data on this, but you can expect a significant percentage of users won't make the purchase simply because it means quitting your app, waiting for the App Store to load, entering their password, and then waiting for the download to finish.

To get acquainted with In App Purchases, you can (and should) read Apple's extensive and detailed In App Purchase Programming Guide: <http://developer.apple.com/library/ios/#documentation/NetworkingInternet/Conceptual/StoreKitGuide/Introduction/Introduction.html>.

But then again, you might want to get started as quickly as possible and learn as you go. With a technology as complex and involving as In App Purchases, it can be quite frustrating, but Ray Wenderlich's tutorial will help you get up on your feet: www.raywenderlich.com/2797/introduction-to-in-app-purchases.

Since Apple is constantly improving their technology, some of the information in the tutorial may no longer apply. In such a case, it is very helpful to study the Technical Note TN2259: Adding In App Purchase to your iOS and Mac Applications: http://developer.apple.com/library/ios/#technotes/tn2259/_index.html. This is a relatively short guide that helps you overcome the typical hurdles of implementing

In App Purchases, including links to set up your contracts, tax and banking information, and a frequently asked questions list.

Summary

I hope you enjoyed learning about some of the surrounding technologies and resources that may be useful for creating your game and becoming a successful game developer, including where to get help and where to find royalty-free game art and audio.

As mentioned in this chapter, marketing and public relations will definitely need to be on your table if you want to actually earn a living from making games. I've only scratched the surface of those here, but it should help you get started.

Finally, remember to buy and use other cocos2d games in order to learn from them. As a game developer, you need to be constantly playing new games to feed your creativity with new ideas, and you also need to learn what players are saying about other developers' games. Improving your knowledge of how games are made is just as important. Purchasing one or several commercial cocos2d source code projects is certainly going to be a wise investment. In particular, if you want to support the development of Kobold2D, I recommend you look at the offerings on the Kobold2D Store: www.kobold2d.com/display/KKSITE/Kobold2D+Store.

This book ends here, but your journey has just begun. I'm sure you'll still have lots of questions, and for that reason I've launched a community web site Cocos2D Central on <http://cocos2d-central.com>. Feel free to stop by and ask me anything about cocos2d, the book, my game kits, marketing, publishers, Xcode, Objective-C, or anything else related to game development. I will continue to blog about cocos2d on the book's www.learn-cocos2d.com web site, and I'm looking forward to hearing from you.

I hope you have enjoyed reading this book as much as I did writing it. Thank you for reading it!

Index

A

acceleratePlayerWithX method, 456
acceleration.x parameter, 91
Achievements, Game Kit programming
 authenticateWithCompletionHandler,
 390
 GameKitHelper class, 387
 GameKitHelper showAchievements
 method, 387
 getAchievementByID method, 391
 loadAchievements method, 390–391
 NSMutableDictionary, 391
 onScoresReceived method, 387
 percentComplete property, 389
 PlayedForTenSeconds achievement,
 389
 progress determination, 389
 progress reporting, 389
 resetAchievements method, 391
 view, 388
addRandomSpriteAt method, 311
addSomeCocoaTouch method, 406,
 408, 414, 419, 422
addSomeJoinedBodies method, 311
addSpriteFramesWithFile, 164
addSubview method, 409
AdMob, 493
AdWhirl, 493
afconvert command, 108
alignItemsVerticallyWithPadding, 68
anchorPoint property, 65, 66
animationWithFile helper method, 156
animationWithFile method, 154, 165,
 166
animationWithFrame method, 166

AppDelegate class
 animation interval, 26
 applicationDidFinishLaunching
 method, 25
 dealloc method, 25
 device orientation, 25
 FPS display, 27
 UIApplicationDelegate protocol, 25
AppDelegate's dealloc method, 25
Apple's UIResponder class, 61
Application programming interface
 (API), 401
applicationDidFinishLaunching method,
 386, 412, 415, 424
ARMv6, 423
authenticateLocalPlayer method, 377
authenticateWithCompletionHandler
 method, 377

B

BATAK Duel source code, 475–476
Block objects, 377
 authenticateWithCompletionHandler
 method, 380
 callback functions, 378
 C-language extension, 378
 lastError variable, 379
 multithreaded and asynchronous
 programming tasks, 378
 NSError pointer, 379
Box2D
 addNewSpriteAt method, 305–306
 allowBodiesToSleep variable, 302
 b2BodyDef type property, 306
 b2PolygonShape set, 306
 b2Vec2 struct type, 302
 b2World, 301

- Box2D API reference, 300
- CCSpriteBatchNode, 306
- vs. Chipmunk, 299
- cocos2d Box2D application
 - template, 300
- collision detection
 - b2ContactListener class, 308
 - BeginContact method, 308–309
 - Box2D API reference, 309
 - ContactListener class's interface, 308
 - dealloc method, 309–310
 - EndContact method, 308–309
 - HelloWorldScene, 309
- converting points, 304
- deallocate message, 301
- Donate button, 301
- GDC, 300
- Hello World Interface, 301
- joint venture, 310–311
- movement restriction
 - b2BodyDef, 302
 - b2PolygonShape variable, 303
 - CreateBody method, 302
 - PTM_RATIO 32, 304
 - PTM_RATIO constant, 303
 - screen area, four sides, 302
 - screenBoxShape variable, 303
 - SetAsEdge method, 303
- Objective-C class, 301
- orthogonal tileset, 305
- PhysicsBox2D01 project, 305
- SetAsBox method, 306
- sleeping bodies, 302
- sprites connection, 306, 307
- update method, 305
- Box2D.h header file, 442
- bringSubviewToFront message, 414
- Bullets
 - bulletSpriteBatch method, 149
 - CCMoveTo and CCMoveBy actions, 148
 - memory allocations, 150
 - position, 148
 - Ship class, 148–150
 - ship shooting, 147

- shoot method, 151–152
- tag, 147
- z-order, 147
- bulletSpriteBatch accessor, 150

C

- CADisplayLink class, 52
- CCActionEase class, 72–74
- CCActionInstant class hierarchy, 76
- CCActionInstant subclass, 69
- CCActionInterval actions, 70
- CCActionInterval class hierarchy, 70, 71
- CCActionInterval subclass, 69
- CCAnimationCache, 155
- CCAnimation class, 154
- CCAnimationHelper class, 165
- CCCallFunc actions, 77, 78
- CCDirector class, 434, 450
- CCDirector runWithScene method, 447
- CCDirector setDeviceOrientation
 - method, 425
- CCFileUtils class, 411
- CCGridAction class, 74
- CCGrid3DAction class hierarchy, 75
- CCGrid3DAction subclass, 74
- CCInstantAction class hierarchy, 76
- CCKeyboardEventDelegate protocol, 62
- CCLabelBMFont class, 107, 403
- CCLabelTTF class, 28
- CCLabelTTF node, 41, 44
- CCLayer class, 27, 41, 43, 51, 449
- CCLayer subclass, 83
- CCLayerColor class, 460
- CCMenu, 403
- CCMenuItemSprite, 68
- CCMenuItemToggle, 68
- CCMotionStreak class, 139
- CCMouseEventDelegate protocol
 - messages, 63
- CCMoveAction, 396
- CCNode class, 27, 43, 51, 83
- CCNode-based class, 405
- CCPageTurn3D action, 76
- CCParallaxNode class
 - CCMoveBy actions, 137
 - CGPoint, 137
 - illusion of depth, 136

- parallaxing, 135–136
- parallaxRatio, 137
- ScenesAndLayers08 project, 136
- CCParticleSystem, 218–219
 - emitter duration, 225
 - emitter modes
 - emitterMode property, 226
 - gravity, 226, 227
 - radius, 227, 228
 - initWithTotalParticles method, 225
 - particle blend mode, 231
 - particle color, 230
 - particle direction, 229
 - particle lifetime, 229
 - particle position, 228
 - particle size, 229
 - particle texture, 232, 233
- Point/Quad
 - ARCH_OPTIMAL_PARTICLE_SYSTEM, 222
 - ARMv7 CPU architecture, 222
 - CCParticleSystemPoint, 222
 - CCParticleSystemQuad, 222
 - optimal particle system class, 222
 - properties, 223–224
 - totalParticles property, 225
 - variance properties, 224
- ccpDistance method, 102
- CCProgressTimer class
 - CCProgressTimer node, 134
 - CCProgressTimerType, 135
 - scheduleUpdate, 135
 - update method implementation, 135
- UserInterfaceLayer class, 134
- CCRepeatForever action, 72
- CCRepeatForever sequences, 78
- CCRibbon class, 138–139
- CCScene class, 41, 43, 51, 405, 449
- CCScene node, 27
- CCSequence action, 72
- CCSpriteFrameCache's method, 164
- CCSprite node, 41
- CCStandardTouchDelegate protocol, 131
- CCTargetedTouchDelegate protocol, 131
- CCTextureCache addImage method, 242
- CCTexture2D texture, 66
- CCTiledGrid3DAction class hierarchy, 74, 75
- CCTintTo actions, 77
- ccTouchesBegan method, 33, 62
- ccTouchesEnded method, 60
- CCTransitionFade, 58
- CCTransitionFadeTR, 58
- CCTransitionJumpZoom, 58
- CCTransitionMoveInL, 58
- CCTransitionPageTurn, 58
- CCTransitionRadialCCW, 58
- CCTransitionRotoZoom, 58
- CCTransitionScene, 115, 405
- CCTransitionSceneOriented, 58
- CCTransitionShrinkGrow class, 54, 59
- CCTransitionSlideInL, 59
- CCTransitionSplitCols, 59
- CCTransitionTurnOffTiles, 59
- CGGeometry methods, 125
- CGPoint, 91
- CGPointExtension files, 102
- CGRectContainsPoint, 133
- Chillingo's Crystal, 490
- Chipmunk SpaceManager, 299
- C++ interface, 299
- Cocoa Touch
 - API, 401
 - cocos2d
 - engine starting, 432–434
 - engine stopping and restarting, 434–435
 - Game Center, 402
 - graphical frameworks, 402
 - hybrid app's user interface, 430–432
 - iAd, 402
 - Interface Builder, 402
 - limitations, 403
 - MVC pattern, 404
 - OpenGL ES, 402–405
 - scenes changing, 436–437

- View-based Application (see View-based Application)
- Game Kit/Web Kit, 402
- GUIs, 402
- Cocos2D Central, 2
- Cocos2d game engine, 467
 - accelerometer controls, 11
 - Android devices, 7
 - API documentation, 9
 - App Store, 1
 - BATAK Duel source code, 475–476
 - bitmap font tools, 477
 - building blocks, 12
 - China Unicom, 7
 - Cocos2D Central, 2, 469
 - Cocos2D Podcast, 476–477
 - cocos2d-iphone project
 - community, 7
 - Farmville, 6
 - OpenGL ES 2.0, 6
 - Sapus Media, 6
 - Zynga, 6
 - community, 467
 - C++ version, 7
 - essentials, 11
 - Game Center, 13
 - game-programming API, 1
 - games making business
 - engaging players (see Engaging players)
 - finding freelancers, 484
 - free art and audio, 484
 - marketing (see Marketing)
 - trade tools, 485
 - working with publishers, 482–483
 - “Hello World” project, 11
 - help, 468
 - home page, 468
 - iOS
 - app developers, 10
 - CCSprite class, 5
 - Cocoa Touch user interface elements, 5
 - cocos2d developers, 11
 - community, 5
 - 2D, 4
 - free cost, 3
 - Objective-C, 3, 4
 - OpenGL ES code, 4–5
 - open source, 3
 - ParticleDesigner tools, 10
 - physics engines, 4
 - programming, 5
 - TexturePacker tools, 10
 - iPhone RPG Game Kit, 471, 474
 - isometric tilemaps, 12
 - iUridium source code, 474–475
 - Kobold2D, 13
 - Line-Drawing Game Starterkit, 472
 - Objective-C code, 7
 - parallax scrolling shooter game, 12
 - particle editing tools, 477
 - particle effects, 12
 - physics editing tools, 477
 - physics engines, 12
 - Pinball game, 13
 - Platformer Game Kit, 472–473
 - ports, 7, 8
 - prerequisites
 - Objective-C, 8–9
 - programming experience, 8
 - reference apps
 - Abstract War 2.0, 481
 - Bloomies, 478, 479
 - Elements, 478
 - Farmville, 480
 - Fuji Leaves, 481
 - Melvin Says There’s Monsters, 480
 - StickWars, 479
 - Super Turbo Action Pig, 480
 - Trainyard, 481
 - ZombieSmash, 479, 480
 - scene editing tools, 477
 - second edition, 2–3
 - shoot ’em up, 12
 - source code, 13, 470–471
 - Space Game Starterkit, 473–474
 - sprites in-depth, 12
 - Stack Exchange network, 469
 - texture atlas tools, 478
 - tilemap editing tools, 478

- tilemaps, 12
- tutorials and FAQs, 470
- UIKit views, 13
- World Wide Web, 9
- www.learn-cocos2d.com, 467
- Cocos2d games, 15
 - application creation
 - HelloWorld project, 19, 20
 - project running in iOS Simulator, 20, 21
 - Xcode project templates, 19
 - certificates and provisioning profiles, 16–17
 - changing the world
 - CCLabelTTF object, 33–34
 - CCLayer class, 33
 - getChildByTag, 33–34
 - label object access, 32–33
 - NSAssert method, 33–34
 - self.isTouchEnabled, 33
 - Cocos2d Central download, 17
 - Debug and Release, 39
 - HelloWorld application
 - AppDelegate class (see AppDelegate class)
 - files location, 22
 - HelloWorldLayer class, 27–29
 - Main.m, 23–24
 - precompiled prefix header, 24
 - resources, 23
 - iOS devices
 - hardware differences, 34, 35
 - iPhone 4, 35
 - iPod touch models, 34
 - second-generation devices, 35
 - iOS SDK, 17
 - iOS simulator, 37, 39
 - memory management
 - addChild, 31
 - autorelease message, 29–30
 - CCArray, 31
 - CCNode* myNode variable, 31
 - CCNode object, 30–31
 - cocos2d objects, 31
 - NSMutableArray, 31
 - release message, 30
 - memory usage, 36–37
 - register as iOS developer, 16
 - system requirements, 15–16
 - Xcode project templates installation, 18
- Cocos2D Podcast, 476–477
- cocos2dVersion method, 368
- Cocos3d
 - addition, 464–465
 - AppDelegate class
 - augmented reality, 458, 459
 - CC3Layer class, 460
 - CCNodeController class, 458, 461
 - CC3World class, 460
 - HelloWorldLayer, 461
 - implementation, 459–460
 - isOverlayingDeviceCamera
 - property, 459
 - node controller, 461
 - real-time camera background, 458
 - RootViewController class, 458
 - sandwiched 3D model, 461
 - useNodeController variable, 459
 - CC3Layer class, 462
 - CC3Light node, 463
 - CC3MeshNode, 463
 - CC3Node class, 462
 - CC3World class, 462
 - COLLADA format, 464
 - createGLBuffers method, 464
 - Hello-Cocos3D template project, 457
 - initializeWorld method, 462–463
 - PowerVR POD files, 464
 - releaseRedundantData, 464
- CocosDenshion audio functionality, 108
- Collision_type property, 318
- contentSize property, 98
- convertEventToGL method, 64
- cpArbiterGetShapes method, 318
- cpSpaceAddCollisionHandler method, 318
- cpSpaceHashEach method, 316
- createPositionPacket method, 399
- createScorePacket method, 399

D

David Gervais's tileset, 273, 277, 278

Doodle Drop GameLayer class, 455

DoodleDrop game, 81

- accelerometer input, 91

- collision detection, 101–102

- final version, 81

- first test run, 91

- iPad porting

- App Store, 110

- iPhone and iPad versions, 110

- Universal apps, 109

- with Xcode 3, 110–111

- with Xcode 4, 111–112

labels and bitmap fonts

- CCLabelBMFont, 104

- CCLabelTTF class, 103

- currentTime, 103

- Glyph Designer, 105–107

- score label, 103–104

- totalTime, 103

obstacles addition

- CCArray class, 97

- CCCallFuncN action, 100, 101

- CCMoveTo action, 100

- dealloc method, 96

- EXC_BAD_ACCESS error, 96

- GameScene init method, 95

- GameScene.h file, 95

- initSpiders method, 95–96

- logging statement, 100

- node's position, 99

- NSAssert, 101

- numSpidersMoved, 95

- resetSpiders method, 99

- resetting spider position, 101

- runSpiderMoveSequence

- method, 100

- spider sprite positions resetting, 98

- spider.png, 95

- spiderDidDrop method, 100

- spiderMoveDuration, 95, 100

- spidersUpdate method, 99

- tempSpider CCSprite, 96

player sprite

- accelerometer input, 89

- alien.png, 87, 89

- CCSprite* member, 89

- contentSize property, 90

- GameScene class, 88

- image height, 90

- image sizes, 90

- JPEG files, 88

- keeping a weak reference, 89

- positioning, 90

- resource files addition, 87, 88

player velocity

- acceleration and deceleration, 92

- accelerometer code, 92, 93

- accelerometer control, 95

- actions, 92

- CGPoint variable, 92

- contentSize width, 94

- current velocity, 93–94

- design parameters, 93

- filtering, 95

- GameScene init method, 93

- imageWidthHalved, 94

- (void) update:(ccTime)delta method, 93

playing audio, 107–108

project setup

- autorotation, 87

- CCNode-derived classes, 83, 84

- Classes and Resources groups, 82, 83

- cocos2d Application template, 82

- DoodleDropAppDelegate.m file, 86, 87

- GameScene class, 85

- GameScene.h, 84–86

- GameScene.m, 84–86

- HelloWorldScene class, 83, 86

- save file dialog, 83, 85

dummyView, 419

E

EAGLView, 414, 415, 431, 435

Electrotanks's Electroserver, 491

enableRetinaDisplay, 144

- Engaging players
 - earning revenue
 - ads and analytics, 492–493
 - In App Purchases, 493
 - iCloud, 492
 - iOS platform, 489
 - push notification providers, 491
 - social networks, 489–490
 - socket server technology,
 - multiplayer games, 490–491
- Entity class hierarchy
 - component classes, 209–211
- EnemyCache class
 - CCArray, 208
 - enemies initialization, 206–207
 - enemiesOfType CCArray, 207
 - init method, 205–206
 - interface, 205
 - spawnEnemyOfType method, 208
 - spawning Enemies, 207–208
 - unspectacular header file, 205
 - update method, 208
- EnemyEntity class
 - dealloc method, 204
 - EnemyType_MAX, 202
 - EnemyTypes enum, 202
 - initWithSpawnFrequency method, 203–204
 - initWithType method, 202–203
 - interface, 201
 - numberWithInt initializer, 204
 - spawn method, 204–205
 - StandardMoveComponent, 203
 - static spawnFrequency CCArray, 204
 - switch statement, 203
 - visible status, 205
- Entity, definition, 201
- isOutsideScreenArea function, 201
- screen rectangle run, 201
- setPosition code, 201
- ShipEntity, 201
- Entry-level tutorials, 470
- EPacketTypes type field, 397
- Essentials Xcode project, 41
- actions
 - CCAction class hierarchy, 69
 - CCFollow action, 69
 - CCRepeatForever action, 69
 - CCSpeed action, 70
 - ease actions, 72–74
 - grid actions, 74–76
 - instant actions, 76–77
 - interval, 70–71
 - sequences, 72
- CCDirector class
 - iOS deployment target setting, 53
 - kCCDirectorTypeDisplayLink, 52
 - kCCDirectorTypeMainLoop, 52
 - kCCDirectorTypeNSTimer, 52
 - kCCDirectorTypeThreadMainLoop, 52
 - NSTimer object, 52
 - responsibilities, 51
 - sharedDirector method, 51
 - singleton, 51
 - types, 52
- CCLabelTTF class, 66
- CCLayer class
 - accelerometer events, 62
 - adding scene, 59
 - backgroundLayer, 59
 - keyboard events, 62
 - mouse events, 63
 - touch events, 60–62
 - userInterfaceLayer, 59
- CCNode class
 - abstract class, 46
 - actions, 47
 - child nodes, 46
 - hierarchy, 44–45
 - scheduled messages (see Scheduled messages)
 - tag parameter, 46
 - z parameter, 46
- CCScene class
 - applicationDidFinishLaunching method, 54
 - CCLayer object, 54
 - CCSceneTransition, 53

- CCTransitionShrinkGrow class, 54
- pushScene method, 53
- replaceScene method, 53, 54
- runWithScene method, 53, 54
- CCSprite class
 - anchor points, 65
 - CCTexture2D texture, 64
 - texture dimensions, 65
- CCTransitionScene subclass
 - hierarchy, 57
 - popScene, 58
 - replaceScene and pushScene, 58
 - transitions, 58–59
- cocos2d scene graph
 - background layer, 42
 - CCNode class, 41
 - CCScene and CCLayer, 42
 - child node position, 44
 - exploded view, 42
 - game objects layer, 42
 - node hierarchy, 43, 44
 - shoot 'em up game, 41, 42
 - virtual joypad layer, 42
- Cocos2d test cases, 80
- menus
 - CCMenu class, 67
 - CCMenu node, 68, 69
 - CCMenuItem class hierarchy, 67
 - CCMenuItemFont, 68
 - CCMenuItemImage, 68
 - CCMenuItemLabel class, 68
 - menuItem1Touched method, 68
 - MenuScene class, 67–68
- pushing and popping scenes, 55–56
- scenes and memory, 54–55
- Singletons
 - MyManager class, 78
 - static autorelease initializer, 78
 - tight coupling, 79
- Exit Games, 491

F

- FadeIn/FadeOut actions, 129
- Farmville, 6
- fastRemoveObject method, 97
- fastRemoveObjectAtIndex method, 97

- floatValue method, 257
- Flurry Analytics, 493
- forEachShape method, 316
- frameCount parameter, 155
- fullPathFromRelativePath, 411

G

- Game building blocks, 115
 - CCNode class
 - CCMotionStreak class, 139–140
 - CCParallaxNode class, 135–137
 - CCProgressTimer class, 134–135
 - CCRibbon class, 138–139
 - game objects, CCSprite
 - composition (see Game object composition)
 - subclassing, 128–129
 - multiple layers
 - CCLayerColor, 128
 - ccTouchBegan, 125
 - ccTouchEnded, 126
 - ccTouchMoved event, 126
 - CGRectContainsPoint, 125
 - dealloc method, 122
 - GameLayer class, 123–124
 - GameLayer rotated and zoomed out, 126
 - getChildByTag method, 123
 - isTouchForMe method, 124, 125
 - LayerTagGameLayer, 122
 - levels implementation, 126–127
 - MultiLayerScene class, 121–122
 - multiLayerSceneInstance variable, 122
 - registerWithTouchDispatcher, 124
 - ScenesAndLayers04 project, 121
 - semi-singleton object, 122
 - sharedLayer method, 122
 - TargetedTouchDelegate, 124
 - TargetedTouchHandlers, 123
 - touch events, 125–126
 - UserInterfaceLayer class, 123–124
 - multiple scenes
 - adding more classes, 115

- CCDirector replaceScene
 - method, 115
- CCScene class, 118
- debugging, 117
- FirstScene dealloc method, 117
- FirstScene onExit method, 117
- init/dealloc method, 116
- initWithTargetScene method, 120
- LoadingScene class, 118–120
- LoadingScene.h file, 118
- myArray, 118
- NSMutableArray instance
 - variable, 118
- onEnter method, 117
- onExit method, 115, 118
- OtherScene CCLayer's init
 - method, 117
- ScenesAndLayers02 project, 116
- ScenesAndLayers03 project, 118
- sceneWithTargetScene method, 120
- scheduleUpdate, 119, 120
- switch statement, 120
- TargetScenes enum, 120
- Game Center, 365
 - Cocos2d Xcode project
 - Build Phases tab, 370
 - Bundle ID, 368, 369
 - Bundle identifier key, 368, 369
 - creation, 367
 - GameKit.framework addition, 370, 371
 - GameKit.h header file, 371
 - Game Kit requirement, 369, 370
 - Info.plist
 - UIRequiredDeviceCapabilities list, 369
 - prefix header, 372
 - Prefix.pch file, 371
 - Property List editor, 368
 - Tilemap_Prefix.pch file, 371
- Game Kit API, 366
- GameKitHelper class, 365
- Game Kit programming
 - achievements (see Achievements, Game Kit programming)
 - availability checking, 374
 - block objects (see Block objects)
 - GameKitHelper delegate, 373–374
 - leaderboards (see Leaderboards)
 - local player authentication (see Local player authentication)
 - local player's friend list, 380–381
 - matchmaking (see Matchmaking)
 - sending and receiving data, 396–399
- iOS 3.0, 366
- iOS 4.1, 365, 366
- iTunes Connect, 366
- leaderboards and achievements, 365–367
- player accounts and friend list, 365
- setup procedure, 372
- Game Developers Conference (GDC), 300
- GameKitHelper init method, 375, 377
- Game objects composition
 - CCNode class, 129
 - CCStandardTouchDelegate
 - protocol, 131
 - CCTargetedTouchDelegate protocol, 131
 - ccTouchBegan method, 133
 - CCTouchDispatcher class, 131
 - changed Spider class, 132–133
 - initWithParentNode method, 130
 - moveAway method, 133
 - NSObject class, 129
 - parentNode parameter, 130
 - Spider class implementation, 130–131
 - Spider class interface, 130
 - spiderSprite class, 130
- GameScene class, 103, 150
- getAchievementByID method, 391
- getChildByTag method, 46
- getLocalPlayerFriends method, 381

- GKAchievementViewControllerDelegate protocol, 387
- GKLeaderboardPlayerScopeGlobal, 384
- GKLeaderboardPlayerScoreFriendsOnly, 384
- GKLeaderboardViewControllerDelegate protocol, 385
- GKLocalPlayer class, 381
- GKMatchmakerViewControllerDelegate protocol, 393
- GKMatchSendDataReliable data mode, 396
- GKPlayer class, 381–382
- glClearColor, 415
- GLSDebugDraw class, 343
- Global identifiers (GIDs) concept, 256
- glView, 414
- glView.superview, 414
- Glyph Designer, 2
- Graphical user interfaces (GUIs), 402

H

- Handholding approach, 470
- Hello Kobold2D template project
 - AppDelegate class, 447
 - config.lua file, 448
 - config.lua script file, 448, 449
 - files, 445, 446
 - FirstSceneClassName, 447, 449
 - iSimulate, 454–455
 - KKAppDelegate class, 447
 - KKMain method, 446
 - KKStart-upConfig class, 449
 - Lua table, 448
 - main.m file, 446
 - NSApplicationDelegate class, 447
 - Objective-C method, 448
 - RootViewController class, 447
 - scene and layer
 - config.lua file, 450–451
 - cross-platform development, 453
 - custom config.lua settings, 451
 - director properties, 453
 - FirstSceneClassName, 450
 - HelloWorldLayer class, 450
 - HelloWorldSettings, 451
 - implementation file, 452–453

- KKConfig, 451, 452
- preprocessor macros, 453
- Wax library, 448
- XML files, 448
- HelloWorldLayer class, 433, 437
- Hiero Bitmap Font tool, 2
- Hiero tool, 104
- High-definition (HD) graphics, 142
- hitTest method, 416, 418
- hitTestNodeChildren method, 417
- Human Interface Guidelines (HIG), 423

I, J

- iAd, 454, 493
- Indiepinion, 470
- init method, 29
- initializationComplete, 447
- initWithContentsOfFile, 411
- initWithParentNode method, 130
- initWithSpriteFrameName method, 164
- initWithTargetScene method, 120
- Interface Builder, 402, 403, 408
 - Attributes Inspector, 422
 - CocosWithCocoa05 project, 419
 - initWithNibName, 422
 - Landscape mode, 422
 - MyView class, 422
 - MyView.xib file, 421, 422
 - nib files, 420
 - orientation, 422
 - sendSubviewToBack message, 422
 - UIViewController subclass, 420, 422
 - Xcode 4, 419, 421
- iOS 5, 2
- iPad Simulator, 111
- isGameCenterAvailable, 376
- iSimulate library, 455
- isKindOfClass method, 101
- Isometric tilemap games, 269
 - Cocos2d
 - addChild method, 280
 - cc_vertexz properties, 279
 - CCDirector, 279
 - depthFormat parameter, 280
 - EAGLView, 279
 - GL_DEPTH_COMPONENT24_OES, 280

- ground floor, 280
 - ground layer, 279
 - OpenGL depth buffer, 279
 - setProjection initialization, 280
 - tilemap loading, 278–279
 - UIWindow, 280
 - vertexZ property, 280
 - zOrder property, 280
 - 3D-rendering performance, 269
 - impassable map border area, 285, 286
 - isometric graphics
 - dg_iso32.png, 272, 273
 - door arch, 272
 - ground floor pattern, 271
 - isometric projection, 270
 - orthogonal image, isometric
 - diamond shape, 270
 - orthogonal tileset into isometric tileset, 271
 - perspective projection, 270
 - isometric tile location
 - CGPoint tilePosDiv creation, 282
 - coordinate system, 281
 - halfMapWidth coordinate, 283
 - inverseTileY variable, 282
 - Objective-C MIN and MAX macros, 283
 - tilePosFromLocation method, 282
 - isometric.tmx file, 284
 - movable player character
 - anchorPoint, 288
 - blocks_movement tile property, 293
 - ccTouchesBegan method, 291
 - centerTileMapOnTileCoord method, 292
 - CGRect variables, 290
 - collisions layer, 293
 - currentMoveDirection variable, 291
 - EMoveDirection enum, 290
 - ensureTilePosIsWithinBounds method, 292
 - isTilePosBlocked method, 293–294
 - move behind tiles, 288–289
 - MoveDirectionNone signaling, 290
 - movement directions, 290
 - moveOffsets array, 291
 - player class implementation, 287
 - player class interface, 287
 - TileMapLayer class interface, 290
 - TileMapLayer class's init method, 287
 - tilePosFromLocation, 292
 - touch location, 291–292
 - update method, 294
 - ninja character, 269
 - nonplayer characters, 295
 - playable area variables, 286
 - resize map, 284, 285
 - scrolling, 283–284
 - tiled map editor (see Tiled map editor)
 - TileMapLayer class, 286
 - tilePosFromLocation method, 284
 - isTouchEnabled property, 60
 - iTunes Connect, 365
 - iUridium source code, 474–475
- ## K
- kEAGLColorFormatRGB565 pixel format, 415
 - kEAGLColorFormatRGBA8 pixel format, 415
 - KKAcceleration class, 456
 - KK_PLATFORM_IOS macro, 453
 - KK_PLATFORM_IOS_DEVICE macro, 453
 - KK_PLATFORM_IOS_SIMULATOR macro, 453
 - KK_PLATFORM_MAC macro, 453
 - Kobold2D, 2, 3, 13, 439, 473
 - advantages
 - cocos2d-iphone, 440
 - cross-platform, 442
 - easy to upgrade, 440
 - free, 440
 - graphical user interface, 440

- Lib Service, 441–442
- Linux distribution, 440
- ready to use, 440
- cocos3d (see Cocos3d)
- Core Motion framework, 439
- Hello Kobold2D template project, 444
- libraries, 439
- Mac Doodle Drop with KKInput
 - acceleration values, 455
 - accelerometer:didAccelerate event method, 455
 - anyTouchEnded method, 456
 - filtering factor, 455
 - input.acceleration.smoothedX value, 456
 - isKeyDown method, 457
 - keyAcceleration value, 456
 - player's velocity, 457
 - PointMake function, 457
 - RectMake function, 457
 - SizeMake function, 457
 - update method, 455–456
- workspace
 - Hello-Kobold2D template, 443
 - Kobold2D Project Starter.app, 442–443
 - Xcode 4 workspace, 443, 444
- Kobold2D.xcworkspace, 443

L

Leaderboards

- AppDelegate class, 386
- GameKitHelper class, 385
- GKLeaderboard object, 384
- GKLeaderboardViewController, 386, 387
- leaderboardDelegate, 385
- loadScoresWithCompletionHandler method, 384
- onPlayerInfoReceived method, 382
- onScoresReceived method, 384
- playerScope variable, 384
- Playtime category ID, 383
- retrieveScoresForPlayers method, 383

- retrieveTopTenAllTimeGlobalScores method, 383
- RootViewController class, 385–386
- submitScore method, 382
- TileMapLayer class, 382
- user interface, 385
- leaderboardViewControllerDidFinish method, 385
- League of Evil game, 473
- Level SVG products, 470
- Levels
 - layers, 127
 - scenes, 127
- loadAchievementsWithCompletionHandler method, 390
- loadFriendsWithCompletionHandler method, 381
- Local player authentication
 - authenticateLocalPlayer method, 376
 - block objects, 377
 - error handling, 377, 378
 - Game Center sign-in dialog, 377
 - iPhone/iPad Simulator, 377
 - leaderboards, 375
 - localPlayer object, 377
 - NSNotificationCenter method, 376
 - player account, 375
 - registration, 376
- Lua configuration files, 473
- Lua functions, 448
- Lua table, 448, 451

M

Marketing

- App Store, 485
- Indie Marketing and Business Tips, 485
- public relations and press releases, 487–488
- your game and yourself, 486–487

Matchmaking

- addPlayersToMatch, 395
- cancelMatchmakingRequest method, 395
- CCProgressTimer class, 394
- expectedPlayerCount property, 395

- GKMatchDelegate, 393
- GKMatchRequest instance, 392
- host game view, 393, 394
- match searching, 394
- minPlayers and maxPlayers
 - properties, 392
- onMatchFound message, 393
- peer-to-peer networking, 392
- showMatchmakerWithRequest
 - method, 392–393
- Model-View-Controller (MVC) pattern, 404

Moment of inertia, 315

MoveBy action, 92

N

- NetworkPackets.h file, 397
- Neutron, 491
- nextInactiveBullet, 151
- Node hierarchy, 43
- NodeHierarchy project, 48
- NSArray method, 97
- NSArray subset, 97
- NSDictionary class, 391
- NSError object, 377, 378
- NSEvent class, 63
- NSMutableArray class, 97
- NSMutableArray method, 97
- NSNotificationCenter, 375
- NSString, 411
- NSString's intValue method, 257
- NSStringFromClass method, 220

O

- Object pooling, 150
- onCallFunc method, 77
- onEnter method, 115
- onEnterTransitionDidFinish, 117
- onFriendsListReceived method, 381
- onLocalPlayerAuthenticationChanged
 - method, 380
- onReceivedData method, 397
- onScoresReceived delegate method, 385
- OpenFeint, 490
- OpenGL, 128
- OpenGL ES framework, 401

- openGLView, 434
- Optimizations, 150–152
- Orientation course, autorotation
 - AppDelegate class, 424
 - ARMv6, 423
 - compiler flags, 423
 - director.openGLView, 427
 - EAGLView, 427
 - GameConfig.h file, 423, 425
 - interfaceOrientation modes, 424, 425
 - iOS HIG, 423
 - kGameAutorotationCCDirector,
 - 423–425
 - kGameAutorotationNone, 423, 425
 - kGameAutorotationUIViewController,
 - 423, 424
 - kGameAutorotationViewController,
 - 426
 - RootViewController class, 424, 425,
 - 427
 - UIViewController, 424
 - willRotateToInterfaceOrientation
 - method, 426
- Orthogonal tilemaps, Cocos2d
 - CCLOG statement, 256
 - CCTMXLoadedMap class, 254
 - CCTMXLoadedMap method, 256
 - ccTouchesBegan method, 255
 - GIDs, 256
 - iPhone Simulator, 255
 - layerNamed method, 255
 - NSDictionary method valueForKey,
 - 256
 - NSDictionary property, 256
 - NSString objects, 256
 - NSString's boolValue method, 256
 - object layer, 260, 262
 - object layer rectangles
 - ccDrawLine, 264
 - CCTMXObjectGroup, 263
 - draw and drawRect method, 264
 - objectGroupNamed method, 263
 - OpenGL ES code, 262
 - OpenGL ES codeOpenGL ES
 - code, 262

- TileMapLayer class, 263
 - (void) draw method, 262, 263
- optimization and readability, 260
- propertiesForGID method, 256
- removeTileAt method, 257
- setTileGID method, 257
- tileGIDAt method, 256
- TileMapLayer class, 254
- tilemap scrolling
 - CCMoveTo action method, 266
 - CCTMXTiledMap method, 265
 - centerTileMapOnTileCoord method, 265
 - scrollPosition CGPoint, 266
- tilePosFromLocation method, 256
- tilePos method, 256
- tile's properties, 255–256
- touched tiles location
 - CCLOG and NSAssert lines, 260
 - coordinate system, 259
 - noninteger coordinate, 259
 - tile coordinates, 257
 - tileMap.mapSize, 260
 - tileMap.position, 258
 - tileMap.tileSize property, 259
 - tilePosFromLocation method, 257
 - tileSize.height, 260

P

- Parallax scrolling, 169
 - anchorPoint property, 178–179
 - background re-creation, code
 - background CCSprite, 174
 - background images, 173
 - CCLayerColor, 174
 - CCSpriteBatchNode, 173
 - CCTextureCache, 173
 - getNodeByTag method, 173
 - getTextureByName method, 173
 - iPad version, 174
 - ParallaxBackground node, 172
 - stringWithFormat, 173
 - TexturePacker, 172
 - background stripes
 - Assets folder, individual stripes, 170, 171

- image-editing program Seashore, 170
- individual parallax layers, 169, 170
- iOS devices, fill rate, 172
- overdraw, 172
- ScrollingWithJoy01 project, 169
- TexturePacker, texture atlas, 169–171
 - z-order images, 172
- CCParallaxNode, 169
- CCTexture2D method
 - setTexParameters, 182
- flicker, 180, 181
- GL_REPEAT texture parameter, 181, 182
- infinite scrolling, 169
- numStripes, tag number, 178
- off-screen background images, 178
- parallax speed factors
 - CCArray, 175
 - dealloc method, 177
 - floatValue method, 178
 - iOS SDK collection classes, 176
 - NSNumber numberWithFloat, 176
 - ParallaxBackground class
 - header, 176
 - ScrollingWithJoy02 project, 175
 - scrollSpeed, 177
 - speedFactors initialization, 176
 - update method, 177
 - zOrder property, 177
- ParallaxBackground, 174–175
- repeatRect, 182
- update method, endless scrolling, 179–180
- Particle effects, 217
 - CCNode objects, 217
 - CCParticleExplosion, 218
 - CCParticleSystem (see also CCParticleSystem)
 - fire.png, 220
 - motionless, moving slowly and moving fast effect, 217, 218
 - particle designer

- background color settings, 235
- cocos2d and iOS OpenGL
 - applications, 234
- code duplication, 238
- Emitter Config view, 234
- endRadiusVar property, 235
- iPhone Simulator, 235, 236
- Online Library, 239, 240
- Particle Texture, 235
- particleWithFile method, 238
- plist file, 237
- positionType property, 235
- Ramdomize button, 236
- runEffect method, 237
- particle system properties, 221
- runEffect method, 218
- Shoot 'em Up, 240–242
- trial-and-error process, 217
- Peer-to-peer networking, 392
- Photon, 491
- Physics engines, 297
 - animation system, 297
 - Box2D (see Box2D)
 - Chipmunk
 - addSomeJoinedBodies, 320
 - b2RevoluteJoint, 320
 - boxing-in boxes, 313–314
 - collision course, 317–318
 - collision
 - courseaddSomeJoinedBodies, 319
 - cpConstraint pointer, 320
 - cpPivotJointNew, 320
 - cpSpaceAddConstraint method, 320
 - Howling Moon Software, 311
 - Objective-C wrappers, 311
 - space, 312
 - SpaceManager API reference, 312
 - sprite's position and rotation, 316–317
 - Ticky-Tacky little boxes, 314–315
 - contact points, 298
 - density/mass, 297
 - dynamic bodies, 297–298
 - friction, 297
 - iOS games, 297
 - joints selection, 298
 - limitations, 298
 - restitution, 297
 - rigid bodies, 297
 - static bodies, 297
- PhysicsEditor, 2
 - application's folder, 323
 - bumper and ball, 332
 - flippers, 331–332
 - plunger shape
 - Add Rectangle button, 326
 - Delete point, 325
 - Fixture parameters, 326
 - Image parameters, 325
 - manual shape definition, 326
 - Plunger collision category and mask flags, 326, 327
- PNG files, 324
- PTM-Ratio, 325
- save and publish, 333
- table shapes
 - Ball category, 330
 - black frame border, 329
 - Shape Tracer, 328
 - table-bottom shape, 328
 - table-left shape, 328
 - table-top shape, 328–330
 - Tolerance setting, 329, 330
- PhysicsEditor tools, 2
- Pinball game, 321, 322
 - applyForceTowardsFinger method, 348
 - ball class's interface, 344
 - b2_dynamicBody, 345
 - BodyNode class
 - .png extension, 335
 - CCSprite object, 333
 - GB2ShapeCache class, 336
 - header file, 334
 - implementation file, 335–336
 - initWithShape method, 334
 - initWithSpriteFrameName method, 336
 - isKindOfClass method, 334

- physics simulation, 338
 - PinballTable class, 337
 - TexturePacker, 335
 - world->Step() method, 337
 - body->SetTransform method, 346
 - bodyToFingerDirection vector, 349
 - Box2D Debug Drawing, 343–344
 - Box2D physics engine, 321
 - bumpers, 350–351
 - convex and counterclockwise shape, 322–323
 - elements, 321
 - fingerLocation method, 347
 - flippers
 - attachFlipperAt method, 361
 - ccTouchBegan method, 362
 - CCTouchDispatcher, 361
 - EFlipperType enum method, 361
 - initWithWorld method, 360–361
 - interface, 360
 - isTouchForMe method, 363
 - maxMotorTorque and motorSpeed fields, 362
 - revolute joint, 361–362
 - SetBullet method, 362
 - gravitational pull simulation, 349
 - init and dealloc methods, ball class, 345
 - PhysicsEditor (see PhysicsEditor)
 - pinball table creation
 - anchorPoint set, 339
 - CCSpriteFrameCache, 341
 - initWithWorld method, 342–343
 - PinballTable class, 340
 - pixel-to-meters macro PTM_RATIO, 341
 - Retina resolution images, 341
 - tablePartInWorld method, 338
 - tablePartInWorld methodTtablePart class implementation, 338
 - TableSetup class implementation, 339
 - plunger
 - applyForceTowardsFinger method, 351
 - contact events, 358–359
 - CreateJoint method, 353
 - enableLimit field, 353
 - header file, 351
 - initialization, 352
 - initWithWorld method, 353
 - maxMotorForce, 353
 - prismatic joint, 352
 - universal contact listener (see Universal contact listener)
 - worldAxis, 353
 - setBallStartPosition method, 346
 - TablePart objects initialization, 346
 - touch delegate methods, 347
 - unit vector, 348
 - PlayedForTenSeconds achievement ID, 367
 - player.position, 91
 - Playtime category ID, 367
 - Plus+, 490
 - popScene method, 55–56
 - Portable network graphics (PNG), 88
 - Prefix.pch header file, 23, 24
 - pushScene method, 55–56
- ## Q
- Quexlor game, 471
- ## R
- registerForLocalPlayerAuthChange method, 375
 - reportScoreWithCompletionHandler method, 383
 - RootViewController class, 368, 413
 - runWithScene method, 434, 435
- ## S
- Sapus Tongue source code project, 470
 - SBasePacket struct, 398
 - sceneChanged method, 436
 - Scene hierarchy, 41
 - sceneWithTargetScene method, 120
 - Scheduled messages
 - _cmd hidden variable, 49
 - Debugger Console, 48
 - delta parameter, 47
 - Objective-C lingo, 47

- Objective-C methods, 49
- parameters name and number, 49
- scheduleUpdate method, 48
- tenMinutesElapsed method, 49
- undeclared selectors, 48, 49
- update methods, 51
- (void) update:(ccTime)delta method, 50
- scheduleUpdate method, 48
- Scoreloop, 490
- Segmented Control buttons, 436
- selectedSegmentIndex, 436
- sendDataToAllPlayers method, 396, 398
- sendSubviewToBack message, 414
- setDeviceOrientation method, 425–426
- setDisplayFPS, 434
- setLastError method, 377, 378
- setOpenGLView, 434
- sharedGameScene accessor, 150
- sharedManager static method, 78
- Shoot 'em Up, 195
 - base Entity class, 200
 - boss monster, 213
 - BulletCache class
 - BulletCache header file, 195
 - BulletCache instance, 198
 - bullet-shooting code, 196
 - bullets reuse, 196–197
 - CCNode getChildByTag method, 196
 - CCSpriteBatchNode, 195
 - CGRectIntersectsRect method, 197
 - CGRectIsEmpty method, 198
 - interface, 195, 196
 - setDisplayFrame method, 197
 - shootBulletAt method, 197
 - update method, 198
 - CGRectIntersectsRect test, 212
 - checkForBulletCollisions method, 212
 - clean code design, 199
 - collisions with bullets, 211–212
 - component-based programming, 195
 - EnemyTypeBoss, 215
 - Entity class hierarchy (see Entity class hierarchy)
 - game's enemy characters, 199
 - HealthBarComponent, 213
 - isEnemyBulletCollidingWithRect method, 211
 - isPlayerBulletCollidingWithRect method, 211
 - Objective-C programming language, 200
 - onHit method, 214
 - plug-and-play solution, 213
 - spawn method,
 - HealthbarComponent, 215
 - update method, 214
- shootBulletFromShip, 152
- shouldAutorotateToInterfaceOrientation method, 424–435
- SimpleAudioEngine, 108
- SmartFoxServer, 491
- SneakyInput
 - AppDelegate class, 184
 - classes, 184, 185
 - cocos2d version, 183
 - ColoredCircleSprite class, 183
 - Download Source button, 183
 - GitHub, 183
 - HelloWorldScene class, 184
 - iPhone Simulator, 183
 - open source software, 183
 - SneakyInput Xcode project, 183
 - textures, 184
- SoundConverter, 108
- Sprites, 141
 - animations
 - addImage method, 153
 - CCAnimate action, 153
 - CCAnimationCache class, 153
 - CCRepeatForever action, 153
 - CCSpriteFrame, 153
 - CCTexture2D objects, 153
 - contentSize property, 153
 - HD and SD resolution, 152
 - helper category, 154–156
 - ship's animation, 152
 - texture atlas, 152–153

- batching, 141
 - CCSpriteBatchNode
 - batch rendering, 144
 - CCSprite nodes, 144, 145
 - disadvantages, 145
 - error message, Debugger
 - Console window, 145
 - functions, 141
 - Sprites01 project, 146
 - Sprites02 project, 147–148
 - Sprites03 project, 148–150
 - Sprites04 project, 150–152
 - texture, 144
 - texture atlas, 145
 - draw call, 141
 - Retina display
 - CCDirector method, 144
 - HD image, 143
 - HD graphics, 142
 - PowerVR MBX Lite, 142
 - resolution-independent
 - coordinate system, 142
 - SD images, 143
 - ship-hd.png, 143
 - subpixel rendering, 143
 - technical specifications iOS
 - devices, 142
 - texture atlas, 141
 - CCAnimation helper category
 - update, 165–166
 - CCSpriteBatchNode, 156
 - CCGRect structure, 157
 - dimensions, 166
 - LoadingScene, 166
 - memory, 166
 - purge methods, 167
 - removeUnused methods, 167
 - texture size, 156
 - TexturePacker (see TexturePacker)
 - with cocos2d, 163–164
 - SScorePacket, 398
 - Standard-definition (SD) graphics, 142
 - StandardShootComponent method, 209–211
 - startAnimation method, 435
 - stopAnimation method, 435
 - switchChanged method, 433, 434
- ## T
- textFieldFront, 419
 - TexturePacker
 - creation
 - Assets folder, 159
 - banding effect, 163
 - Border Padding settings, 161
 - color depth, 162
 - Dithering option, 163
 - GameArt folder, 159
 - HD texture, 160, 161
 - Image Format setting, 162
 - Output settings, 161
 - PNG file format, 162
 - PVR format, 162
 - RGB5551 format, 162
 - RGB565 format, 163
 - RGBA4444 format, 162
 - RGBA8888 format, 162
 - SD texture, 160
 - Shape Padding settings, 161
 - Sprites06 project, 161
 - stray pixels, 161
 - Texture Format, 162
 - transparent border pixels, 160
 - true-color image, 162
 - working process, 159, 160
 - 2D sprite-packing tool, 157
 - iPhone's PowerVR graphics chip, 157
 - preparation, 158
 - ship's animation frames, 157
 - Terminal app, 157
 - TexturePacker Pro, 157
 - TexturePacker tools, 2
 - Tiled (Qt) Map Editor
 - tilemap creation
 - dg_grounds32.png tileset, 249
 - Margin and Spacing settings, 250
 - New Map dialog, 248
 - New Tileset dialog, 249, 250
 - texture atlas, 249
 - tile size, 249
 - TMX file, 250

- tilemap design
 - blank map, 251
 - Bucket Fill (hotkey F), 252
 - choose Layer, 252
 - dg_grounds32 tileset, empty map, 251
 - edit properties, 252
 - Eraser (hotkey E), 252
 - GameEventLayer, 252, 253
 - Insert Objects (hotkey O), 252
 - Insert Tile Objects (hotkey T), 252
 - isWater property, 252, 253
 - Layers view, 252
 - Rectangular Select (hotkey R), 252
 - Remove Object, 252
 - Select Objects (hotkey S), 252
 - Stamp Brush (hotkey B), 252
 - tile layers and object layertile and object layers, 253, 254
 - TMX files, 248
 - Tiled editor, 12
 - Tiled map editor, 473
 - ground rules, 276–278
 - new isometric tileset creation, 276
 - New Map dialog, 273, 274
 - plain-text XML file, 275
 - tile-size offset problem, 274, 275
 - tilewidth and tileheight parameters, 275
 - TMX file, 275
 - TileMapLayer class, 380
 - TileMapLayer's update method, 389
 - Tilemaps, 243
 - 2D game world, 243
 - Farmville game, 246
 - isometric tilemap, 245, 246
 - multiple layers, 244, 245
 - orthogonal tilemaps, Cocos2d (see Orthogonal tilemaps, Cocos2d)
 - TexturePacker image preparation, 247–248
 - Tiled (Qt) Map Editor (see Tiled (Qt) Map Editor)
 - tiles, definition, 243
 - TileMapScene class, 397
 - TMX file format, 12
 - Touch events propagation
 - boundingBox, 417, 418
 - CCARRAY_FOREACH loop, 418
 - CCLayer class, 418
 - CCMenuItem, 418
 - CCNode class, 418
 - CCScene class, 418
 - CCTouchDispatcher class, 415, 416
 - EAGLView class, 416, 417
 - glView, 415
 - hitTest method, 417
 - hitTestNodeChildren method, 417
 - runningScene, 417
 - sceneChildren, 417
 - UIView hitTest event, 416
 - userInteractionEnabled, 416
 - Twitch-based game, 27
 - Twitter, 487
- ## U
- UIAccelerometer interface, 455
 - UIAlertViewDelegate protocol, 405, 409
 - UIApplicationDelegate class, 447
 - UIApplicationDelegate protocol, 25
 - UIImage class, 410–412
 - UIImagePickerController, 458
 - UIKit views, 3
 - CCMenu, 401
 - CCSprite, 401
 - Cocoa Touch (see Cocoa Touch)
 - cocos2d
 - addSomeCocoaTouch method, 407, 408
 - CCDirector, 409
 - CCLabelITTF, 406
 - CocosWithCocoa03 project, 408
 - didDismissWithButtonIndex message, 407
 - dummyView, 414
 - EAGLView class, 409
 - glView, 409
 - HelloWorldLayer class, 405, 407, 408
 - init method, 406
 - Interface Builder (see Interface Builder)
 - openGLView, 406

- orientation course, autorotation
 - (see Orientation course, autorotation)
 - PrivateMethods interface, 405
 - removeFromSuperview message, 410
 - resignKeyFirstResponder message, 410
 - RootViewController class, 408
 - sandwiching, 418–419
 - textFieldDidBeginEditing message, 410
 - textFieldShouldReturn message, 409, 410
 - touch events propagation (see touch events propagation)
 - UIAlertView, 405, 407
 - UIAlertViewDelegate methods, 407
 - UIAlertViewDelegate protocol, 405
 - UITextField, 408, 409, 414
 - UITextField and UIImage skinning, 410–412
 - UITextFieldDelegate protocol, 409
 - view hierarchy, 412–413
 - view transparent, 414–415
 - UISegmentedControl, 436
 - UITextField views, 403, 411, 412
 - UITextFieldDelegate protocol, 409, 410
 - UIView-based classes, 405
 - UIViewController, 368
 - UIWindow, 405, 413
 - Universal contact listener
 - BeginContact and EndContact methods, 355
 - beginContactWithBall selector, 357
 - Box2D collision process, 354
 - Box2D Contact method
 - implementation, 355–356
 - ContactListener class definition, 354
 - interface and implementation, 357–358
 - notifyAB method, 356
 - NSStringFromClass method, 356
 - retain method, 358
- ## V
- VertexHelper, 2
 - View-based Application, 437
 - AudioToolbox.framework, 430
 - AVFoundation.framework, 430
 - Build Phases tab, 429
 - cocos2d libs folder, 429
 - iOS Deployment Target, 428
 - libz.dylib, 430
 - Link Binary With Libraries pane, 429, 430
 - MainWindow.xib file, 427
 - OpenAL.framework, 430
 - OpenGL ES.framework, 430
 - Project Navigator pane, 428, 429
 - QuartzCore.framework, 430
 - template, 427, 428
 - ViewBasedAppWithCocos2D
 - project, 427–429
 - Xcode 4, 429
 - ViewBasedAppWithCocos2D-Prefix.pch file, 432
 - ViewBasedAppWithCocos2DViewContr
 - oller class, 432, 435, 436
 - ViewBasedAppWithCocos2DViewContr
 - oller.m file, 433
 - ViewBasedAppWithCocos2DViewContr
 - oller.xib file, 430
 - ViewBasedWithCocos2DViewContr
 - oller.h file, 432
 - viewController view, 413
 - Virtual joystick, 182
 - addJoystick method, 190
 - autoCenter, 190
 - cocos2d's ccpMult method, 191
 - defaultShip accessor method, 191
 - digipad/thumbstick, 182
 - digital controls, 193
 - GameScene class, 191
 - setPosition method, 192
 - setRotation:(float)rotation method, 192
 - shoot button
 - addFireButton method, 186
 - buttonRadius, 187

- CCLayer, 185
- CCGRect parameter, 186
- CCGRectZero, 186
- Debugger Console window, 187
- GameScene class, 185
- GameScene header file, 186
- InputLayer.h header file, 186
- scene method, 185, 186
- update method, 187
- skinned analog thumbstick and fire button, 182
- skinning
 - activated state image, 188
 - addFireButton method, 188–189
 - ccTime variables, 189
 - cocos2d-style static initializers, 187
 - default state image, 188
 - disabled state image, 188
 - fireButton, 188
 - pressed state image, 188
 - shooting bullets, fire button, 189

- SneakyButton autorelease initializers, 188
- SneakyButtonSkinnedBase class, 189
- SneakyExtensions class, 187, 188
- SneakyInput (see SneakyInput)
- SneakyJoystick, 190
- SneakyJoystickSkinnedBase, 190
- update method process, 191
- velocity property, 191

W

- willRotateToInterfaceOrientation method, 426

X

- .xcconfig files, 446
- Xcode 4, 2
- .xcodeworkspace file, 444

Z

- zOrder property, 175