

Development Tales of
iPhone App Masters



iPhone Advanced Projects

Joachim Bondo | Dylan Bruzenak | Steve Finkelstein | Owen Goss
Tom Harrington | Peter Honeder | Florian Pflug | Ray Kiddy
Noel Llopis | Joe Pezzillo | Jonathan Saggau | Ben Britten Smith

Preface by Glenn Cole

Apress®

www.it-ebooks.info



RELATED TITLES



The Apress series of iPhone Projects books features experienced app developers presenting their own work in their own words. You get firsthand accounts of what it takes to design, implement, and launch some of the finest applications available from Apple's iTunes App Store.

iPhone Advanced Projects, the third book in this series, tackles some advanced aspects of iPhone development. The first generation of iPhone applications has hit the App Store, and now it's time to optimize performance, streamline the user interfaces, and make every successful iPhone app just that much more sophisticated.

Your guides for this exploration of the next level of iPhone development include the following:

- **Ben Britten Smith**, discussing particle systems using OpenGL ES
- **Joachim Bondo**, demonstrating his implementation of correspondence gaming in the most recent version of his chess application, Deep Green
- **Tom Harrington**, implementing streaming audio with Core Audio, one of many iPhone OS 3 APIs
- **Owen Goss**, debugging those pesky errors in your iPhone code with an eye toward achieving professional-strength results
- **Dylan Bruzenak**, building a data-driven application with SQLite
- **Ray Kiddy**, illustrating the full application development life cycle with Core Data
- **Steve Finkelstein**, marrying an offline e-mail client to Core Data
- **Peter Honeder** and **Florian Pflug**, tackling the challenges of networked applications in WiFi environments
- **Jonathan Saggau**, improving interface responsiveness with some of his personal tips and tricks, including "blocks" and other esoteric techniques
- **Joe Pezzillo**, pushing the frontiers of iPhone OS 3's new Apple Push Notification Service (APNS) that makes the cloud the limit for iPhone apps
- **Noel Llopis**, taking mere programmers on a really advanced developmental adventure into the world of environment mapping with OpenGL ES

It's a full banquet of treats, so dig in where the morsels look most tempting. There's plenty here for every palate. Apress also offers a nourishing first course with its best-selling *Beginning iPhone 3 Development: Exploring the iPhone SDK*. And we're always on the lookout for what's new and even tastier, so feel free to share your most nourishing apps with us. We'd love to be able to add them to the next volume of iPhone Projects.

This book is for all iPhone application developers with any level of experience or coming from any development platform who wants to see how an advanced app is made. Take what you learn in this book and use it to create the next great iPhone app!

COMPANION eBook

SEE LAST PAGE FOR DETAILS ON \$10 eBook VERSION

Apress®

SOURCE CODE ONLINE

www.apress.com

US \$39.99

Shelve in
Mobile Computing/Mac ProgrammingUser level:
Intermediate

www.it-ebooks.info

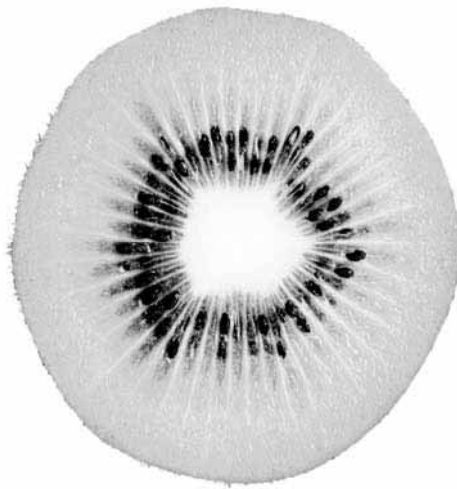
ISBN 978-1-4302-2403-7

5 399 9



9 781430 224037

iPhone Advanced Projects



Dave Mark, Series Editor

**Joachim Bondo
Dylan Bruzenak
Steve Finkelstein
Owen Goss
Tom Harrington
Peter Honeder**

**Ray Kiddy
Noel Llopis
Joe Pezzillo
Florian Pflug
Jonathan Saggau
Ben Britten Smith**

Apress®

iPhone Advanced Projects

Copyright © 2009 by Dave Mark, Joachim Bondo, Dylan Bruzenak, Steve Finkelstein, Owen Goss, Tom Harrington, Peter Honeder, Ray Kiddy, Noel Llopis, Joe Pezzillo, Florian Pflug, Jonathan Saggau, Ben Britten Smith

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-13 (pbk): 978-1-4302-2403-7

ISBN-13 (electronic): 978-1-4302-2404-4

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

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

Lead Editor: Clay Andres

Technical Reviewer: Glenn Cole

Developmental Editor: Douglas Pundick

Editorial Board: Clay Andres, Steve Anglin, Mark Beckner, Ewan Buckingham, Tony Campbell, Gary Cornell, Jonathan Gennick, Michelle Lowman, Matthew Moodie, Jeffrey Pepper, Frank Pohlmann, Ben Renow-Clarke, Dominic Shakeshaft, Matt Wade, Tom Welsh

Coordinating Editor: Kelly Moritz

Copy Editor: Kim Wimpsett

Compositor: MacPS, LLC

Indexer: Julie Grady

Artist: April Milne

Distributed to the book trade worldwide by Springer-Verlag New York, Inc., 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 <http://www.springeronline.com>.

For information on translations, please e-mail info@apress.com, or visit <http://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 <http://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 <http://www.apress.com>. You will need to answer questions pertaining to this book in order to successfully download the code.

To my lovely wife, Leonie.

—Ben Britten Smith

To my wife, Malena, who once again gave me the support I hadn't earned.

—Joachim Bondo

To everyone I know and to everyone I haven't met yet.

—Dylan Bruzenak

To all of my family and friends for their support and patience with my demanding schedule. To my loving wife, Michelle, who sustains me and encourages me to take risks. Finally, this one is for my grandmother, Asya; you will live forever in all our hearts.

—Steve Finkelstein

To the iPhone game developers on Twitter for sharing so much and being such a supportive community.

—Noel Llopis (@snappytouch on Twitter)

I'm so grateful to so many people I can't possibly hope to name them all individually, so, en masse, let me thank the blessing that is my family (especially my son), the unstoppable geniuses at Apple, the folks at Apress who patiently awaited my writing, the incredibly supportive Mac and iPhone indie developer community, all my clients and customers, my business partners and colleagues, and, of course, the great ineffable spirit of the universe that makes everything possible.

Thank you!

—Joe Pezzillo

To my family, my friends, the island "La Palma," and the one who introduced me to it.

—Florian Pflug

To Dr. Michele, who doesn't let me call her doctor. Thanks for making me type.

—Jonathan Saggau

Contents at a Glance

■ Contents at a Glance	iv
■ Contents.....	v
■ Foreword	xi
■ About the Technical Reviewer	xii
■ Preface.....	xiii
Ben Britten Smith	1
■ Everything You Ever Wanted to Know About Particle Systems	3
Joachim Bondo	37
■ Chess on the 'Net: Correspondence Gaming with Deep Green.....	39
Tom Harrington.....	63
■ Audio Streaming: An Exploration into Core Audio	65
Owen Goss	99
■ You Go Squish Now! Debugging on the iPhone.....	101
Dylan Bruzenak.....	139
■ Building Data-Driven Applications with Active Record and SQLite	141
Ray Kiddy.....	181
■ Core Data and Hard-Core Design	183
Steve Finkelstein	209
■ Smart In-Application E-mail with Core Data and Three20.....	211
Florian Pflug and Peter Honeder.....	247
■ How iTap Tackles the Challenges of Networking.....	249
Jonathan Saggau	277
■ Fake It 'Til You Make It: Tips and Tricks for Improving Interface Responsiveness.....	279
Joe Pezzillo.....	311
■ Demystifying the Apple Push Notification Service	313
Noel Llopis	345
■ Environment Mapping and Reflections with OpenGL ES.....	347
■ Index	365

Contents

■ Contents at a Glance.....	iv
■ Contents	v
■ Foreword	xi
■ About the Technical Reviewer	xii
■ Preface	xiii
Ben Britten Smith	1
■ CHAPTER 1: Everything You Ever Wanted to Know	
About Particle Systems	3
Adding Life to Your Game with Particles	5
Basic Particle Systems and You.....	7
Overview of the Sample Code	8
Basic Game Flow.....	9
The Anatomy of a Particle System	10
Code! Finally!	12
Slight Tangent About Degenerates.....	15
Back to the Code	16
Random Numbers and Initial Conditions	19
Emitting Particles	20
Tweaking Your Particle System	21
May the Force Be with Your Particles	25
Amazing Technicolor Dream Particle	28
Off on a Tangent: Lerpig.....	28
Color-Changing Particles.....	30
Summary.....	35

Joachim Bondo.....	37
■ Chapter 2: Chess on the 'Net: Correspondence	
Gaming with Deep Green	39
Deep Green, an Already Awesome Application	40
The Tasks at Hand.....	42
Inviting a Friend to a Game	43
Accepting the Invitation	43
Making a Move	43
Getting Notified	43
The Tools of the Trade.....	44
Stop Talking, Start Coding!	45
Installing the Tools	45
Coding the Web Service	47
Accepting the Challenge on the Device.....	54
Making a Move	57
Summary.....	61
Tom Harrington	63
■ Chapter 3: Audio Streaming: An Exploration into Core Audio	65
Hey, I Could Write an App to Play Music	66
MPMoviePlayerController: Hey, This Is Easy! Right?.....	66
Finding a Better Approach.....	68
The System-Sound Way	69
AVAudioPlayer: The Not-Available-in-Beta Way.....	69
Doing It the Cowboy Way with Core Audio	74
Getting Halfway There: Audio Queue Services	74
Getting the Rest of the Way There: Audio File Stream Services.....	81
Putting It All into an App.....	93
One More Thing.....	93
Launch It!	96
iPhone 3.0 and Further Work	96
Summary.....	97
Owen Goss.....	99
■ Chapter 4: You Go Squish Now! Debugging on the iPhone.....	101
Assumed Knowledge.....	102
Objective-C vs. C and C++	104
While You're Writing That Code.....	105
Custom Asserts	105
Custom Logging	107
Using #define	108
Crash!	109
Getting a Crash Log from Your Testers	109
You Have Been Saving Your dSYM Files, Right?	110
Symbolicating a Crash Log.....	110

Using atos	111
Reproducing Rare Crashes.....	112
Thread	112
System	113
Race Conditions	113
The Scientific Method of Debugging	113
Forming a Hypothesis	113
Creating a Test for Your Hypothesis.....	114
Proving or Disproving Your Hypothesis	115
Increasing the Probability of the Crash	115
So, You Have a Call Stack	115
Starting Code.....	115
What Is a Memory Stomp?	118
Identifying a Mem Stomp	122
Tools to Detect Memory Problems	123
Watching Variables	131
Link Map Files	135
Summary.....	137
Dylan Bruzenak	139
■ Chapter 5: Building Data-Driven Applications with	
Active Record and SQLite	141
A Short Road Off a High Cliff (How I Got Here)	141
Ready! Set! Wait, What? (Why I Decided to Write a To-Do Application).....	142
Data-Driven Applications on the iPhone.....	143
Active Record: A Simple Way of Accessing Data	144
Writing a Database Wrapper Around the C API: ISDatabase	144
Setting Up the Example Project.....	145
Creating and Initializing the Database	148
Opening a Database Connection	149
Making Simple Requests.....	152
More Advanced SQL	158
Preventing Duplicate Create Statements	158
Handling Parameters.....	160
Refactoring and Cleanup	162
Grouping Statements into Transactions	163
Writing a Simple Active Record Layer: ISModel	164
Maintaining the Database Connection	165
The Model Object: Grocery Item	165
How Groceries Are Mapped.....	166
Saving	168
Updating	170
Deleting.....	170
Finding Grocery Items	171
Putting It All Together.....	174

Simple Migration Handling	176
Alternative Implementations	179
Summary	180
Ray Kiddy	181
■ Chapter 6: Core Data and Hard-Core Design	183
Where Did Core Data Come From?	184
The Client Is King	184
A Very First Core Data App	185
First, Steal Code (Not Music!)	186
A View to an Object, Any Object	187
Our Very First Crash, or Perhaps Not!	193
CoreData Tutorial for iPhone OS: Managing Model Migrations	194
The Easy Migrations Are Easy	194
Adding a New Entity	197
Using Key-Value Coding to Create a Reusable Object	199
Remote Databases: It's All Net!	203
Summary	206
Steve Finkelstein	209
■ Chapter 7: mart In-Application E-mail with Core Data and Three20	211
Planning a Simple Offline SMTP Client	212
Creating the User Interface	213
Diving into Xcode	213
Setting Up Instance Variables in OfflineMailerAppDelegate.h	215
Initializing the UIApplication Delegate	217
Working with Core Data	218
Understanding the Core Data Stack	221
Adding Three20	221
Journeying Through the User Interface	224
Managing Top-Level Data with DataManager	226
Diving into Three20 and TTMessageController	228
Composing and Sending Messages	230
Creating the Core Data Model	235
Hacking SKPSMTPMessage to Support Threaded Message Sending	239
Setting Up the NSRunLoop on SKPSMTPMessage	239
Switching the Bits Back to Online Mode	241
Summary	244
Florian Pflug and Peter Honeder	247
■ Chapter 8: How iTap Tackles the Challenges of Networking	249
Meet iTap and iTap Receiver	250
iTap	251
iTap Receiver	251
How the Idea for iTap Emerged and Evolved	252

The Main Challenges	252
No Physical Buttons on the iPhone	252
Third-Party Applications Cannot Use USB or Bluetooth	253
Supporting Both Mac and PC	254
User-Friendliness Demands Autodiscovery of Computers and Devices.....	255
WiFi Networking on the iPhone from a Programmer's Perspective	255
About the Sample Code.....	256
Introducing Sockets	257
Creating a Socket.....	258
Using CFSocket to React to Networking Events.....	262
Querying the Network Configuration	264
Contacting All Devices on the Network	267
Detecting WiFi Availability.....	268
Playing by the Power Management Rules.....	269
The Networking Subsystem of iTap	271
To use Bonjour or Not to Use Bonjour	271
Using Notifications to Communicate Between Components	272
Our Custom Autodiscovery Solution.....	273
Summary.....	275
Jonathan Saggau.....	277
■ Chapter 9: Fake It 'Til You Make It: Tips and Tricks for Improving Interface Responsiveness	279
Plotting of Historical Stock Prices with AAPLot.....	280
Storing Data Between Runs	283
Using Plists to Persist Data	284
Saving Data to the iPhone Application Sandbox	285
Shipping AAPLot with Placeholder Data.....	286
Extending the App for Multiple Stock Graphs: StockPlot	288
Concurrency.....	292
NSOperation, NSOperationQueue, and Blocks	293
Installing the Plausible Blocks Compiler and Adding It to the Project.....	294
Using Blocks, NSOperation, and NSOperationQueue in StockPlot	295
Displaying Large Amounts of Data Efficiently	298
Zooming a UIScrollView	300
UIScrollView Zooming Under the Covers.....	300
Resetting Resolution in a UIScrollView after a Zoom Operation.....	301
Drawing into an Off-Screen Context	304
Observations, Tips, and Tricks	309
Summary.....	310
Joe Pezzillo	311
■ Chapter 10: Demystifying the Apple Push Notification Service.....	313
What Is the Apple Push Notification Service?	314
What You'll Need	314

Step 1: Create the Client	314
The Application Delegate	315
Handling Incoming Notifications	317
Sounds	318
Build and Go! Er, Not So Fast...	318
Step 2: Create the Certificate	319
A Walk-Through of the Program Portal Process.....	319
Back to the Portal.....	328
Add the Mobile Provisioning File for Code Signing	329
Step 3: Set Up the Server	331
A Walk-Through of What This Script Does	333
Download Server File	334
The Home Stretch	336
Wiring Up the Client	336
Additional Considerations/Advanced Topics	341
Feedback Server	341
SSL Server Connections	342
Moving from Development Sandbox to Production	342
Development vs. Ad Hoc	343
Mobile Provisioning Files	343
Debugging	343
User Experience	343
Open Source Code.....	344
Hosted Solutions	344
Summary.....	344
Noel Llopis.....	345
■ Chapter 11: Environment Mapping and Reflections	
with OpenGL ES	347
The Beginnings.....	347
First Steps: OpenGL Lighting.....	349
Turning to Environment Mapping	352
Spherical Environment Mapping Implementation	353
Combining Environment Mapping and Diffuse Textures	356
Per-Pixel Reflections	359
iPhone 3GS.....	362
Summary.....	363
Index.....	365

Foreword

Dear Readers,

We started this series of iPhone Projects books because we recognized that there is a community of iPhone developers all starting from scratch and full of enthusiasm for Apple's iPhone and iPod touch devices. The community has come a long way since we became aware of this phenomenon. For one thing, we're not all starting from scratch anymore, and this book, as does every book in this series, highlights the work of the more experienced among us.

But this enthusiasm remains a defining characteristic, along with an eagerness to learn and a willingness to share. If we were Homeric storytellers, this would be our Trojan War, an image I find particularly apt in this time of renewed gaming interest. And like the ancient poetic bards, we have some compelling stories to tell. Though, rather than warriors with shields and spears, these are tales of developer derring-do.

Our heroes are the quietly toiling, Internet-connected, basement-dwelling developers who are the stuff of iTunes App Store lore. We'll leave the modern-day mythology, Hollywood sound tracks, and CG animation to the finished applications. The chapters in this book are real-life stories of highly caffeinated work, relatively sweat-free code adventurers who dare to push the limits of a cool, little, pocket-sized, life-changing pair of devices known as the iPhone and the iPod touch. It's a dirty job, but somebody has to succeed at it.

I have worked with Dave Mark, the series editor and author of several best-selling Apress books, including *Beginning iPhone 3 Development*, to find developers who produce efficient and bug-free code, design usable and attractive interfaces, and push the limits of the technology. Dave's common-man touch, tell-it-like-it-is sense of reality, and delight at all that's cool and wonderful can be felt throughout the series.

And that brings us back to the unique quality of community among iPhone developers. Every chapter is written by a different developer with their own goals and methods, but they're all willing to share what they've learned with you. And you'll learn many things about the design and implementation of great apps, but you'll also learn that you are not alone. Every developer gets stuck, has a bad day, and experiences delays and frustrations, and the lessons learned from these setbacks are as important as the API calls and algorithms that will be part of your finished products.

And finally, we hope you'll find the apps presented in these chapters and the stories of how they came to be both interesting as human drama and as cool as the iPhone and iPod touch themselves. Happy adventuring, and send us a postcard!

Clay Andres
Apress Acquisitions Editor, iPhone and Mac OS X
clayandres@apress.com

About the Technical Reviewer

Glenn Cole has been a professional software developer for nearly three decades, from COBOL and IMAGE on the HP 3000 to Java, Perl, shell scripts, and Oracle on the HP 9000. He is a 2003 alumnus of the Cocoa Bootcamp at the Big Nerd Ranch. In his spare time he enjoys taking road trips, playing frisbee golf, and furthering his technical skills.

Preface

Getting started with iPhone application development is relatively easy thanks to online tutorials and especially to books like *Beginning iPhone Development* by Dave Mark and Jeff LaMarche. But sometimes, software is just hard.

A year and a half after receiving an iPhone as a birthday present, I am still amazed. It looks so simple and it's so easy to use, but behind it all is a world of complexity.

Apple has worked very hard to document the myriad APIs that make up the iPhone SDK and to provide sample code, but for some of us it's still not enough. Even Apple cannot afford to provide a chapter's worth of explanation for each sample application. Their tutorials can be quite helpful, such as the one on Core Data, but what then?

Enter *iPhone Advanced Projects*.

Ray Kiddy, who worked at Apple for 15 years in various roles, uses Apple's tutorial on Core Data as a starting point and builds from there. More than providing just an introduction, Ray shows what it's like to use Core Data in the real world.

That's the difference between documentation and a book such as this. Of course, it doesn't stop there.

Joachim Bondo, creator of the much-lauded chess application Deep Green, shares his advice and techniques for implementing correspondence gaming.

Noel Llopis, a ten-year veteran of the gaming industry, author of *C++ for Game Programmers*, and instructor of a two-day intensive class in OpenGL programming specifically for the iPhone, lends new meaning to making your application "shine" with a discussion of reflections and environment mapping in OpenGL. I found it to be a fascinating topic.

My knowledge of OpenGL is casual at best, but Ben Britten Smith provides such a clear explanation of particle systems (think smoke and fire) that this was not a hindrance at all. The chapter really was a "blast" to work through.

I've been on a private mailing list with Jonathan Saggau for several years now, and his explanations never fail to impress. Here, he discusses the difficult topic of improving interface responsiveness. (Be sure to have a copy of his sample code handy!)

And that's just the half of it! The projects also include an exploration into Core Audio, a framework for persisting data with SQLite, strategies for networking, techniques for debugging, the Apple Push Notification Service (not for the faint of heart), and intelligent in-app e-mail.

Sometimes, software is hard. With these authors as your guides, it should make your work quite a bit easier.

Organization

This book is organized roughly in order of challenge, not necessarily according to the complexity of the code as much as the total level of knowledge and effort required.

For example, the Cocoa code that is needed to support the Apple Push Notification Service (APNS) is fairly brief and straightforward, yet the discussion of APNS does not appear until near the end of the book. Why? The primary reason for this is the complexity of the surrounding infrastructure, including working with the iPhone Developer Program Portal and setting up a PHP server appropriately.

Of course, every developer has their own ideas about what is difficult or challenging and what is not, so the chapter sequence is intended only as a rough guide. Each chapter is independent of the others, so feel free to jump straight to your projects of interest.

What's in the Book

The book opens with Ben Britten Smith discussing particle systems using OpenGL. Although it's not a tutorial on OpenGL per se, Ben provides enough background and detail so that the code makes sense at a conceptual level even to those of us with only minimal experience in that area. Take your time in understanding this chapter and the sample code behind it, and the effort will be well rewarded. Besides, it's great fun!

Chapter 2 finds Joachim Bondo demonstrating how to implement correspondence gaming such as with his chess application Deep Green. You'll see the power of Python in Google App Engine, understand RESTful web services, implement a custom URL scheme (to support a URL beginning with `chess://`), and use Django's template engine to take advantage of a plist with embedded logic and variable substitution. It's a mouthful, but Joachim makes it look easy.

Audio is one of those topics that's just plain hard. Different requirements mean different APIs; it doesn't take much to become overwhelmed by the complexity. In Chapter 3, Tom Harrington shares the results of his investigation into processing audio streams, starting with the Media Player framework and moving to System Sound Services and the AV Foundation framework before settling on Core Audio. Audio is hard; take advantage of Tom's guidance.

Every iPhone developer who has written a nontrivial application has experienced a difficult-to-find bug. In Chapter 4, Owen Goss provides advice that goes well beyond using `NSLog()` and stepping through the debugger. You'll want to work through this chapter more than once to be sure you recognize which tools to use and when.

Dylan Bruzenak tackles data-driven applications in Chapter 5 with SQLite and the Active Record design pattern. Enterprise and cross-platform developers in particular will benefit from this, as will anyone who wants to keep fine-grained control over the data in their application.

Core Data is new to the iPhone with OS 3.0. It takes the task of data persistence to a seemingly magical level. (At least that's how I first experienced it on the Mac side.) In Chapter 6, Ray Kiddy guides us from Apple's tutorial on Core Data to its proper use in the real world, highlighting issues that can occur along the way and showing how to avoid them. Core Data is a big deal; you'll want to work through this chapter more than once.

In Chapter 7, Steve Finkelstein combines two open source projects with Core Data to build an intelligent offline email client. It recognizes when the network status changes and uses `NSInvocationOperation` to keep the user interface responsive while performing other operations. When sending e-mail, control stays within the application.

Peter Honeder and Florian Pflug get down to the socket level for networking in Chapter 8. In addition to discussing the ins and outs of communicating with devices on the network, they also discuss both power management and the trade-offs between using `SCNetworkReachability` for detecting a Wi-Fi network vs. rolling their own autodetection code.

An unresponsive user interface is one of the most frustrating behaviors an application can exhibit. In Chapter 9, Jonathan Saggau demonstrates techniques that can be used to address this. From `NSOperation/NSOperationQueue` to "blocks" (part of Snow Leopard but currently available on the iPhone only via Plausible Blocks) to drawing into an off-screen context and more, this chapter is very enlightening.

Joe Pezzillo provides step-by-step guidance for setting up APNS in Chapter 10. As Joe notes, the process is not particularly difficult, but it is lengthy and involved, and that's just for the creation of the distribution certificate. The Cocoa code is almost anticlimactic.

The book concludes with a fascinating chapter by Noel Llopis on environment mapping and reflections using OpenGL. You'll get more out of the chapter if you first brush off your linear algebra text, but there is still much to be learned even without it. This is the kind of polish that iPhone users love to see.

You can see that this book is packed with projects that are both relevant and interesting. Take advantage of the authors' knowledge to help your application stand above the rest!

Glenn Cole

Ben Britten Smith



Company: <http://benbritten.com>

Location: Melbourne, Australia

Former Life As a Developer: *I have been writing software in one form or another since gradeschool. Back then I wrote in BASIC and Logo. Over the intervening quarter century or so I have ranged all over the map, from writing low level assembly for embedded systems through all the major (and not so major) languages settling now and again on the big ones, like C, C++, Perl, Smalltalk, Obj C, PHP, etc.*

Somewhere along the way I got involved with a visual effects company called Spydercam, and wrote their industrial motion control system. This system is still in heavy use and is used on many feature films. Then in 2005, Spydercam's lead hardware designer, lead mechanical engineer and I were awarded an Academy Award for Technical Achievement for our efforts in 3D motion control. Some interesting trivia: the system we designed is the only one that I am aware of that runs on a mac, written entirely in native Cocoa/Obj-C.

I am also active in the Multi-touch surface open source community. I wrote an open source tracker called BBTouch and an open source OSC implementation called BBOSC.

Life as an iPhone Developer *More recently I have relocated from New York City to live in Melbourne with my wife Leonie. Here I have started offering my services as a freelance cocoa developer, and once the SDK became public, the market for iPhone work exploded. I have worked on a half dozen apps that are on the store now for various clients, titles like SnowDude, Blackout and aSleep. More recently I have begun collaborating on games of my own design, we just finished one: SnowFerno. I am currently in development on a follow-on from*

SnowDude called SkateDude, and a third as yet unnamed Dude project. After those are done I have two more collaboration projects that are in pre-production, both games and both 2D platformers.



Key Technologies: Three or four key technologies discussed:

- OpenGL
- Texture Atlases
- Particle Systems
- Cool Stuff

Particle Systems: More Fun and Easier Than You Think

When I was hired to write SnowDude, my employers, the Lycette Bros., and I set out a simple goal: we wanted a nice, clean, simple game that was easy to pick up and fun to play. There was not a big budget, so simplicity was the rule of the day.

I initially built the game using Core Animation, thinking that would be the quickest and easiest route to getting our 2D graphics onto the screen. In our early prototypes, this worked great; however, as we began adding the background elements and all the little graphic bits that made the game come alive, our performance crashed. I was forced at this point to reengineer the game model with OpenGL as the rendering API. This gave us all the performance we needed, and that micro game engine became the basis for many future projects in OpenGL on the iPhone.

SnowDude was a successful project in our eyes; it didn't break any App Store sales records, but the game was stable, clean, simple, and fun. (Go buy it!) The game was a lateral move for all the parties involved. I had built simple games in the past, but the bulk of my experience is in real-time motion control systems for feature films. The Lycette Bros. came from the world of Flash games and developing apps for other mobile platforms, so SnowDude was not just a game app but a way for everyone involved to dip their toes into a new platform.

Since then, I have gone on to develop a dozen or so apps for various clients and have released my first personal project to the app store: SnowFerno, which is a puzzle game where you take on the persona of a snowball trying to roll its way through hell.

And now, a bit less than a year after the original SnowDude was released, there is interest in a spin-off (or two), and we are starting to build the first one: SkateDude.

SnowDude was ultimately a fast-paced maze game. You are a snowboarder, and your goal is to get as far as you can down the “slope,” avoiding various obstacles along the way. You can avoid the obstacles by either jumping over them or boarding around them. If you make it to the checkpoint, you get some bonus time, and you can play for a higher score.

As far as programming complexity, SnowDude was not very. It consists of just a handful of textured quads, some clever use of the accelerometer, simple collisions, and some game logic.

When we all came to the table to start talking about SkateDude, we wanted to make it be a more active game experience. We wanted the obstacle avoidance to be only a small part of the game play. We decided to add tricks that you can do while in the air and a more robust control system. We added many more options to earn points, such as grinding along hand rails or park benches and doing multipart tricks like jumping onto a rail, grinding along it, and then jumping off and doing a trick before landing. All of these options add a sense of excitement and give the players an opportunity to feel the thrill of conquering the challenges.

One thing that we hadn’t nailed down in the early development meetings was how to visually enhance the game. We didn’t know how we would use the stunning graphics that the artist was generating to help bring the challenges alive and add a sense of accomplishment to the game play.

We started playing around with adding particle systems to the game. At first, I just added some very subtle sparks that shot out from under the skateboard when the player was grinding across something. This encouraged me to add a few more things. And then I added a few more systems and then a few more. I added a particle system to the controls so that if you hit a big jump, the button exploded in a shower of stars. I added a bunch of sparks that shot off the place where you touched the screen to do a jump. I added particles everywhere! Well, that was great and added lots of exciting elements, but I did go a bit far, and we ultimately scaled back to a few simple systems that added some fun and encouraged the players to want to grind and do tricks by rewarding them not only with points but with a fun visual system where a bubble with point values would shoot out from under the board like sparks and float up to join the score at the top of the screen.

This made the game much more visceral. Now, when you jump and grind across the various surfaces and edges in the game, you can visually see the points you are racking up, and the faster you grind or the higher your trick, the more points you get, so the particle systems that are shooting point bubbles out are exploding at the higher levels. Figure 1-1 is an early development screenshot of SkateDude; you can see the sparks coming off the skateboard trucks as well as the point indicators shooting out as you grind.



Figure 1-1. An early development screenshot from the game *SkateDude* by the Lycette Bros. This shot shows two of the particle systems I added to make the game more exciting and visceral.

Adding Life to Your Game with Particles

For the rest of the chapter, I'll go over particles and how you can use them in subtle and not-so-subtle ways to add life to your games. I'll show you how to build your own particle emitter system in OpenGL and incorporate it into your own projects.

First, what is a particle system, and why would you want to use it? A *particle system* is a large collection of small graphics (you guessed it, those are the particles) that when taken as a whole can simulate effects that would otherwise be very hard to render. Things like smoke and fire are good examples. Particles are particularly good at simulating systems that are inherently dynamic and ever-changing.

Fire is a good example. You can simulate an OK fire with an animation, but it will always have a cartoonish look. If you want a fairly decent simulation of fire, you will want to use particle systems.

SnowFerno is a good example. Given that you are a snowball in hell, we mostly use particles to simulate just fire and smoke effects (see Figures 1-2 and 1-3). But fire and smoke are not the only things you should think about simulating with particle systems.

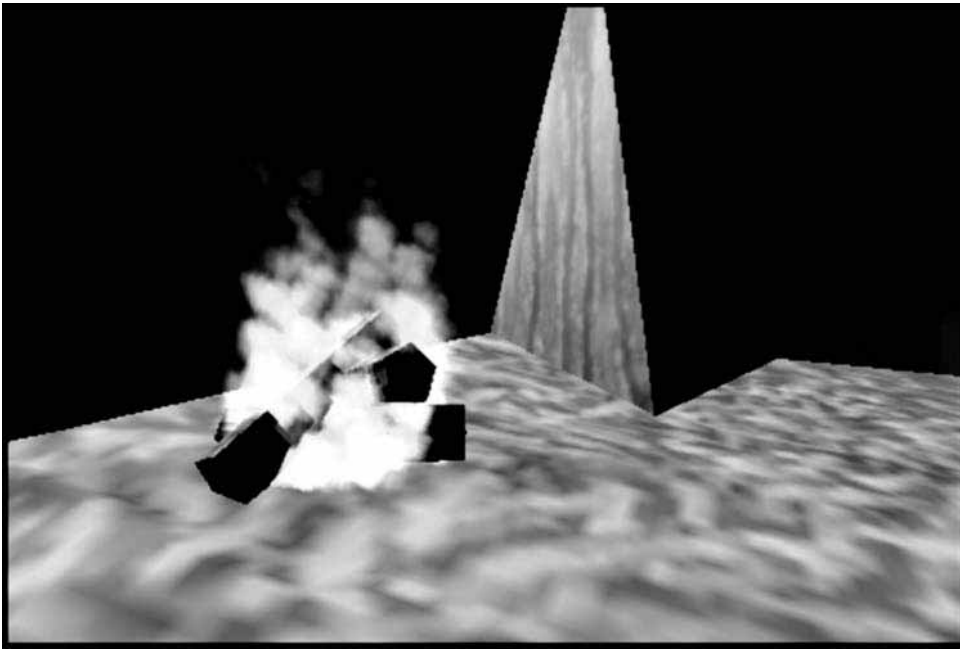


Figure 1-2. A simple fire and smoke effect using particles. This is one of the particle systems in *SnowFerno*.



Figure 1-3. *SnowFerno* was set in *Dante's Inferno*, so we had plenty of opportunities to use fire effects.

Particles are often associated with 3D games where the environments are immersive and players expect things such as realistic weather effects and smoke and fire and splattering blood and explosions. The list goes on and on. You can achieve all of these effects with particles.

However, it is also good to think about particles when designing your 2D apps as well, and not just 2D action games either. I often play some puzzle games to pass the time, such as Drop7 and AuroraFeint. Both of these use particles to add a bit of excitement and life to the game. In Figure 1-4, you can see the block-smashing effect in Aurora Feint.



Figure 1-4. *Aurora Feint uses particles to make its block smashing exciting.*

Particles do not need to be big flashy things; they don't have to be grand explosions or giant fireballs. You can add subtle fun touches to your game interface with some simple effects as well. Drop7 does this well; when you “crack” one of the unknown numbers, it breaks open with a simple particle effect. It is so subtle that you might not even notice it, but it adds that bit of life and personality that makes the game fun. When you set up a nice long chain reaction, all those little particle explosions really make it that much more satisfying.

Basic Particle Systems and You

OK, now you know where you can add particle effects to your games, so now let's talk about how to add them.

First, I will presume you have some familiarity with OpenGL. If you don't know OpenGL, that is fine; you can still do particles in Core Animation and Core Graphics, so much of

the conceptual stuff will be applicable. However, OpenGL excels at things like particle systems because it is so good at moving textures onto the screen very fast. In a Core Animation particle implementation, you might be able to get a particle system with a few dozen particles, maybe even 100 for a short while. With OpenGL, you can generate thousands of particles at once, even on the iPhone.

Overview of the Sample Code

The sample project, called *Particles*, started its life as a generic OpenGL project template from Apple. I have added a simple game harness around Apple's template code. Originally this code was written for the *Beginning Game Development for iPhone*, and the chapters I wrote in that book go into great detail about this code base. Most of the implementation details are not that important to the discussion of particle systems, but I will do a brief overview anyway.

Let's take a look at the basic design:

- **EAGLView:** This is a modified version of the EAGLView you get when you start a new Xcode OpenGL iPhone project. It is responsible for OpenGL buffer swapping as well as most of the boilerplate OpenGL initialization stuff. This is the main view for the application.
- **SceneObject:** This is the base class for anything in the game. It has the basic instance vars that most everything that needs to be rendered needs. All rendered objects inherit from this class.
- **SceneController:** This is the main controller for the game. It handles the game loop. It has a single SceneObject that is the root of all objects in the current scene. It is a singleton.
- **InputViewController:** Since the input and the main view are basically the same thing, this view controller handles the EAGLView as well as wrangling the touch events. The input controller has its own list of scene objects that get rendered last, in a heads-up display style.
- **RenderController:** This object deals with rendering all the scene objects. It performs simple culling. The render controller uses a SceneObject's mesh to render that object. The mesh is basically the collection of all the vertex data for a particular model.
- **MaterialController:** This object handles the loading of textures into OpenGL. It can handle single textures or atlases when accompanied with a .plist file describing the atlas contents.
- **GameTypes:** This is just a big collection of structs and inline functions that come in handy. The two types I use the most in the sample code are BBPoint, an xyz point struct, and BBRange, a range of floats.

The reason that I am not just showing how to build a stand-alone particles project is that I think it is important to think about how these things fit into the bigger picture. Although

the sample program does little more than show off some particle effects, it is important to think of these concepts in the context of a larger application.

The Particles sample project is not a fully realized game engine by any stretch, but it is a good place to start, and it has much of what you would need to build a simple 3D application in OpenGL. This makes it a good platform for you to explore the concepts of particle systems.

Basic Game Flow

Figure 1-5 shows the flow for the game harness. It follows the basic game design pattern that you are probably familiar with.

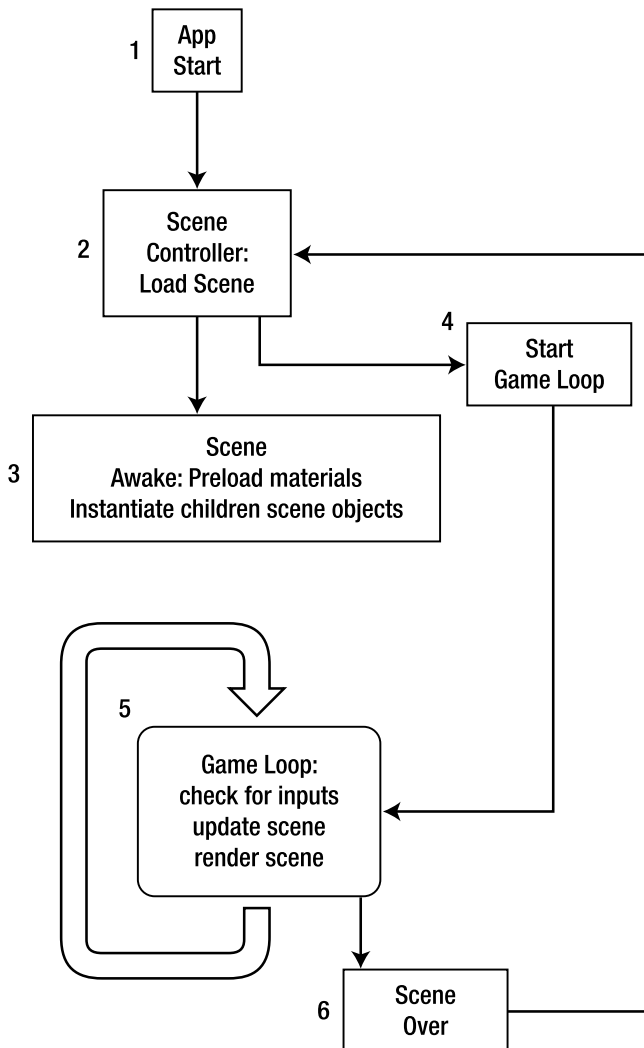


Figure 1-5. This is the basic flow for the game harness.

After the app starts up and everything is loaded from the xib files and you are ready to go, the `SceneController` is called upon to load the first scene. This scene is simply a `SceneObject` that is the parent of all the objects you want to have interact for this scene.

After the scene is “allocated,” the method `awake` is called on it, and that is where the scene will call out to the other support objects, like the material controller, to make sure that all the resources for this scene are loaded. (In this case, this will generally just be textures, but in the broader case, this might include sound files or game data of some sort.)

When everything is ready, the game loop is started.

The game loop first checks for inputs, and then it calls `update`: on the scene. The scene object will update all of its children recursively until the entire scene model has had a chance to update its state. Finally, the game loop passes the root scene object to the renderer to be rendered. Then it starts all over again.

At some point in the scene, the update portion of the loop will generate an end-of-scene notification. (Maybe your character died, you ran out of time, or you hit a button to move on to the next scene...whatever) The current scene is unloaded, and the next scene is loaded.

This is a fairly standard game engine design. The big component that’s missing here is a collision detection system. You will do some simple collision stuff with the particle systems but nothing too complicated.

The Anatomy of a Particle System

Just in case you have never come in contact with a particle system, I will start with the basics: what exactly constitutes a particle?

Particles can be any texture and are usually rendered as a textured quad (two triangles). Depending on the effect you are going for, your particle textures might be semitransparent like the simple white particle in Figure 1-6. Soft semitransparent particles will yield “fuzzy” effects quite well. This makes a nice effect because particles in a high concentration will be brighter and more intense, whereas out on the edges where there may be only a few particles, the overall effect is dimmer and “blurry.”



Figure 1-6. A simple particle texture. This is about the simplest semitransparent texture you can get. It is just a white blur, 25 X 25 pixels.

That said, you can get some great effects from fully opaque or hard-edged particles as well, such as things like marbles rolling across a floor or leaves falling.

Each particle in the system has its own state, and each particle will get its own initial conditions and then behave based on a set of rules. All of this ordered chaos—a multitude of particles that are all slightly different but similar—can create some amazing fluid, living, organic effects.

That is the particle. You also need something that generates the particles, and that is known as the *emitter*. The emitter's job is to build new particles at some predetermined rate. It has to assign each particle an initial state that meets the requirements for that particular effect. These are things such as starting position, size, life span, speed, and direction. After a particle has been created, the emitter then has to keep track of each particle, and for every rendered frame, it needs to collect all the vertex and UV and any other rendering data for each particle and build some big arrays to send off to the renderer.

In many particle effects, each particle has a life span, and once that span is over, the emitter needs to collect those particles and remove them from the scene.

So, basically the emitter itself has a mini game loop going on. Every time it gets updated, it needs to create some new particles and add them to its currently active particle list. Then it goes through all the active particles and moves them or rotates them or whatever. Then it needs to check to see whether any particles have reached the end of their life, and if so, it removes them from the active list. Finally, it needs to make the data arrays from all the particle states.

Here are a few things to keep in mind:

- The particle system needs to be able to go through thousands of particles in a single frame, so you need to find efficient ways to handle all of the particles and keep them updated.
- The emitter may need to emit a few hundred particles every frame, possibly even a few thousand, so you also need to be very efficient about creating particles. Allocating objects is a costly process, so you want to avoid it at all costs.
- Hundreds of particles can expire at the same frame, so you need to also be clever about how you clean up your particles. Memory cleanup is slow and can affect performance, so you need to be careful about releasing a zillion particles all at once.
- Dynamically mallocing vertex array memory is expensive. You want to avoid changing the size of your vertex data arrays.

How do you solve these problems?

When your particle emitter is first created, you will need to build a big reserve of preallocated particle objects. Similarly, you will malloc a big chunk of memory for your vertex data arrays, big enough to hold the maximum number of particles.

Then during the update loop, when you emit new particles, you just grab them out of the pool and assign them their initial state. This is so much faster than allocating new objects on the fly. This becomes especially important for effects such as explosions where you need to emit lots of particles all at once.

When you build your data arrays for each frame, you just use as much of the vertex data space as you need and leave the rest as reserve.

Similarly, at the end of the particle life, when you clear them out of the active list, you simply return the particle objects to the pool.

Figure 1-7 shows this life cycle. Also of note: I used a particle system to generate both the spark shower and the pool.

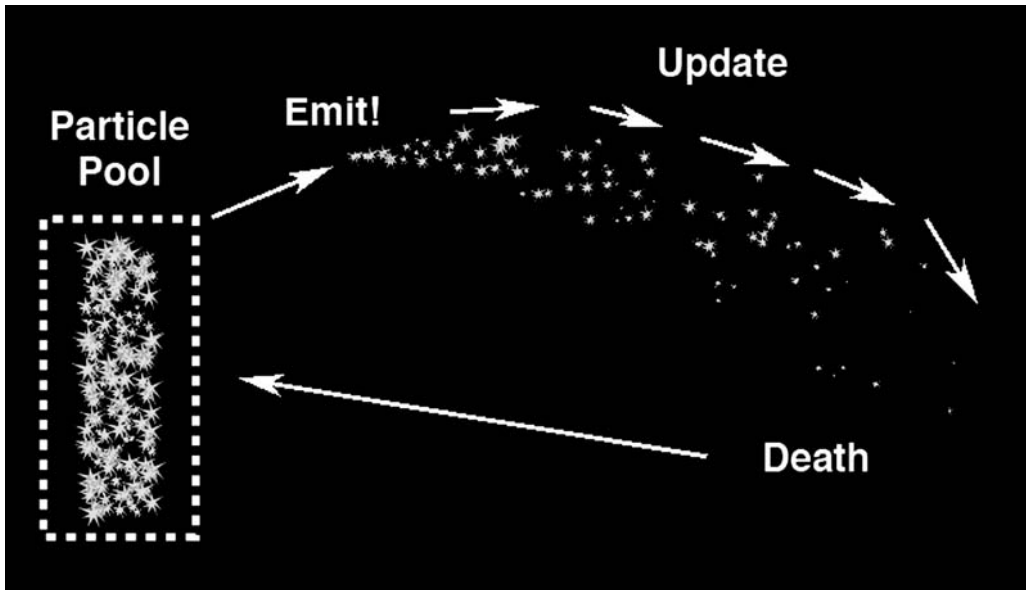


Figure 1-7. The particle life cycle. Nonactive particles start in the pool. They are pulled out of the pool and given some initial state when they are emitted. They live out their exciting particle life until they finally die. They are then collected and returned to particle limbo to await resurrection.

The downside to this method is that it can be very memory consuming, and the setup time can be significant if you have many particle systems. The secret is to tune the max particles for the type of effect you are creating. A blizzard of falling snow might require a few thousand particles, whereas a subtle foreground of falling leaves may require only a few dozen.

Code! Finally!

OK, I have rambled on for quite a few pages about the whats and whys of particles. It is time to get your hands dirty with some code.

First build a particle:

```
@interface BBParticle : NSObject {
    BBPoint position;
    BBPoint velocity;
    CGFloat life;
    CGFloat size;
    CGFloat grow;
    CGFloat decay;
}
```

```

@property (assign) BBPoint position;
@property (assign) BBPoint velocity;
@property (assign) CGFloat life;
@property (assign) CGFloat size;
@property (assign) CGFloat grow;
@property (assign) CGFloat decay;

```

This is a very basic particle. The basic state is position, life, and size. velocity, grow, and decay are the state changers. Particles can be far more complicated than this, and you will add some more stuff to your particle later, but for now let's keep it simple.

Next you look inside your particle implementation:

```

@implementation BBParticle

@synthesize position,velocity;
@synthesize life,size,grow,decay;

-(void)update:(NSTimeInterval)deltaTime
{
    position.x += velocity.x * deltaTime;
    position.y += velocity.y * deltaTime;
    position.z += velocity.z * deltaTime;

    life -= decay * deltaTime;
    size += grow * deltaTime;
    if (size < 0.0) size = 0.0;
}

```

Very simple. You have a time-based update. You take all of your state and change it by a fraction equal to the amount of time for this frame. Finally, you check your size. You don't want to go into negative size because that will just flip your particle over and make it grow.

That's it! You have a nice simple model object with a single data manipulator method.

Next, let's build a simple particle emitter object. This one is a bit more complicated than the particle:

```

@interface BBParticleSystem : BBSceneObject {
    NSMutableArray * childrenParticles;

    GLfloat * uvCoordinates;
    GLfloat * vertexes;

    NSMutableArray * unusedParticles;

    NSInteger vertexIndex;

    BOOL emit;
    CGFloat emitCounter;

    BBRange emissionRange;
    BBRange sizeRange;
    BBRange growRange;
}

```

```

    BBRange xVelocityRange;
    BBRange yVelocityRange;
    BBRange zVelocityRange;

    BBRange lifeRange;
    BBRange decayRange;

    CGFloat minU;
    CGFloat maxU;
    CGFloat minV;
    CGFloat maxV;

    CGFloat particleRemainder;
}

```

Wow, that is a fair few instance variables! One thing that you will learn quickly (or may already know if you have played with emitters before) is that a good particle emitter will be very flexible, and that requires lots of inputs to tweak to get just the right effect. Lots of inputs means lots of instance variables.

Let's get into the implementation:

```

- (id) init
{
    self = [super init];
    if (self != nil) {
        [self preload];
    }
    return self;
}

```

That was a simple init method. Basically, you just call preload, which is where you, well, preload all your particles and memory allocations:

```

-(void)preload
{
    if (childrenParticles == nil) childrenParticles = [[NSMutableArray alloc] init];
    unusedParticles = [[NSMutableArray alloc] initWithCapacity:kMaxParticles];
    NSInteger count = 0;
    for (count = 0; count < kMaxParticles; count++) {
        BBParticle * p = [[BBParticle alloc] init];
        [unusedParticles addObject:p];
        [p release];
    }
}

```

First you create your particle limbo and fill it with particles ready to be jettisoned into life to burn brightly for a few moments and then be pulled back into the land of the inactive.

```

    // remember 6 vertexes per particle + UVs
    vertexes = (CGFloat *) malloc(2 * 6 * kMaxParticles * sizeof(CGFloat));
    uvCoordinates = (CGFloat *) malloc(2 * 6 * kMaxParticles * sizeof(CGFloat));
}

```

Don't forget to malloc some room for the vertexes and UV coordinates.

I'll now go off on a tangent momentarily and talk about GL_TRIANGLES vs. GL_TRIANGLE_STRIP.

Slight Tangent About Degenerates

You are going to be drawing a whole slew of textured quads onto the screen. However, generally a quad is only four vertices. So, what is up here?

You are going to be rendering all your particles in the same draw call, and they are not connected, so you will need to figure out a good way to draw them all.

If you use `GL_TRIANGLES`, then you are basically just draw each triangle individually. Every quad is just two triangles and six vertices. This has the advantage of being very simple to program.

You could also use `GL_TRIANGLE_STRIP` and connect each quad with degenerate triangles. A *degenerate triangle* is a triangle where the three points lie on a line. You can see in Figure 1-8 how this works. A triangle with colinear points has no area, so the renderer will throw it out. The easiest way to connect two meshes with a degenerate triangle is to just duplicate the last vertex of the first mesh and the first vertex of the second mesh and then add them together. This basically inserts two colinear triangles into the strip so that the rendered effect is two separate quads. This means, on average, each quad requires six vertices, just like the `GL_TRIANGLES` method.

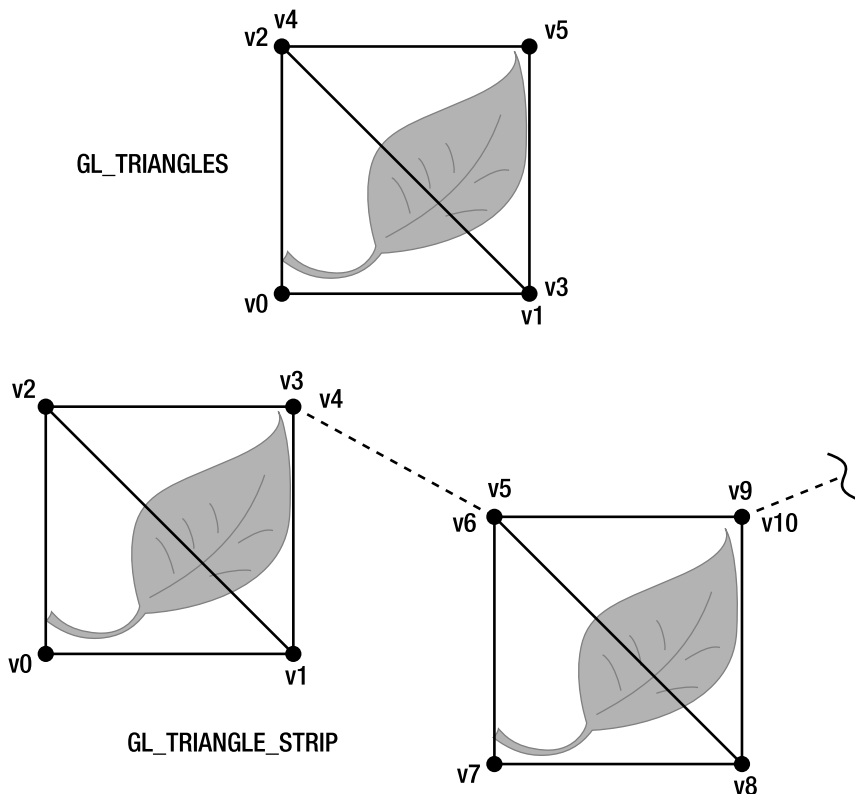


Figure 1-8. With `GL_TRIANGLES`, you have two separate polygons drawn individually. With `GL_TRIANGLE_STRIP`, all the polygons are connected, so you have to basically put two degenerate triangles in between the two separate quads.

Using degenerate triangles makes the code just ever so slightly more complex for very little practical gain. I always pick the simpler of two choices, so you are going to stay with `GL_TRIANGLES` in this chapter.

Back to the Code

You have preloaded your particles, so now you need to assign your textures:

```
-(void)setParticle:(NSString*)atlasKey
{
    BBTexturedMesh * quad = [[BBMaterialController sharedMaterialController]
quadFromAtlasKey:atlasKey];
    self.mesh = [[BBTexturedMesh alloc] init];
    [(BBTexturedMesh*)mesh setMaterialKey:quad.materialKey];
    [(BBTexturedMesh*)mesh setAtlasKey:quad.atlasKey];
}
```

You will grab a prebuilt quad from the material controller (more on this in a moment). You don't want to set your mesh to be the same as the quad's mesh because you are going to be mucking with the internal bits of our mesh. Instead, you will make a fresh one and copy over the parts you care about:

```
// need to calculate the min and max UV
CGFloat u,v;
NSInteger index;
minU = minV = 1.0;
maxU = maxV = 0.0;
CGFloat * uvs = [quad uvCoordinates];
for (index = 0; index < quad.vertexCount; index++) {
    u = uvs[index * 2];
    v = uvs[(index * 2) + 1];
    if (u < minU) minU = u;
    if (v < minV) minV = v;
    if (u > maxU) maxU = u;
    if (v > maxV) maxV = v;
}
```

To be as efficient as possible, you will be building the UV coordinate array alongside the vertex array during the update phase. To do this, you will need the min/max of your UV coordinates. You calculate those from the `texturedQuad` and store them for later:

```
mesh.vertexes = vertexes;
[(BBTexturedMesh*)mesh setUvCoordinates:uvCoordinates];

mesh.vertexStride = 2;
mesh.renderStyle = GL_TRIANGLES;
}
```

Lastly, you point the mesh vertexes and UV coordinates back at your big buffers that you have already malloced.

OK, there is something called a *mesh* and a *material controller* that I haven't really talked much about. The mesh is basically just a holder for the OpenGL vertex data arrays. The render controller uses the mesh to do the final rendering. That is why you need to give it information like the `renderStyle` and the `vertexSize`.

The material controller is a handy class that does all the heavy lifting for loading and processing texture atlases. In this case, you have a texture atlas file called `particleAtlas.png` and a texture metadata file called `particleAtlas.xml`. The XML file contains the information required to generate the UV coordinates for all the images in the atlas. The material controller loads all those textures when the scene is loaded and stores them in a string-keyed dictionary. So, to get a textured quad from the atlas, you just ask for it by name, like so:

```
BBTexturedMesh * quad = [[BBMaterialController sharedMaterialController]
quadFromAtlasKey:atlasKey];
```

In this case, the quad will be the particle texture that you want to associate with this emitter.

OK, now you want to set up the update loop in the emitter:

```
-(void)update:(NSTimeInterval)deltaTime
{
    [super update:deltaTime];

    // update active particles -> move them
    for (BBParticle * kid in childrenParticles) [kid update:deltaTime];

    // build arrays
    [self buildVertexArrays];

    // emit -> add new particles
    [self emitNewParticles:deltaTime];
}
```

It's a simple loop: update the current particles, and emit new particles. Changing the order that you call these methods will have very subtle effects on the working of the emitter, but mostly any order will work just as well as the next. For instance, I could emit new particles before I build the arrays. This means that the new particles will get rendered for one frame before ever moving. This might be what you want. I have put the emit last so that those particles will not get rendered until they have been updated at least once.

```
-(void)buildVertexArrays
{
    vertexIndex = 0;
    for (BBParticle * particle in childrenParticles) {
        // check to see if we have run out of life, or are too small to see
        // and if they are, then queue them for removal
        if ((particle.life < 0) || (particle.size < 0.3)) {
            [self removeChildParticle:particle];
            continue; // skip to the next particle, no need to add this one
        }
    }
}
```

This is the heavy lifting method of this class. This is where you do the real work of taking all your particles and making OpenGL-compatible vertex and UV arrays. You first reset `vertexIndex`, which is the instance variable that will keep track of where you are in the arrays, so it is pretty important.

Next you simply step through each child particle. First you are going to check to see whether the life has expired or whether the size is too small to bother rendering. In either case, you will queue this child for removal. In this case, you also skip to the next particle; there is no reason to add a dead particle to this rendering array.

```
// for each particle, need 6 vertexes
[self addVertex:(particle.position.x - particle.size) y:(particle.position.y -
particle.size) u:minU v:maxV];
[self addVertex:(particle.position.x + particle.size) y:(particle.position.y -
particle.size) u:maxU v:maxV];
[self addVertex:(particle.position.x - particle.size) y:(particle.position.y +
particle.size) u:minU v:minV];

[self addVertex:(particle.position.x + particle.size) y:(particle.position.y -
particle.size) u:maxU v:maxV];
[self addVertex:(particle.position.x - particle.size) y:(particle.position.y +
particle.size) u:minU v:minV];
[self addVertex:(particle.position.x + particle.size) y:(particle.position.y +
particle.size) u:maxU v:minV];
```

Next you build a vertex from the particle's state. Currently that is just the position. You are also building the UV arrays at the same time, using the stored UV max and min:

```
}
mesh.vertexCount = vertexIndex;
[BBSceneController sharedSceneController].totalVerts += vertexIndex;
}
```

You then set your vertexCount in the mesh object so that it knows how many vertexes to render. Finally, you are going to jam the particle count into a state variable in the scene controller. This is a bit of a hack, but I want to be able to display the number of particles on the screen, because I have another object that comes around later and uses this to render that number.

It is important to note the order in which you are building these vertexes. Currently, I am using front-face culling to make the 3D models slightly smaller in terms of vertexes rendered. However, the 3D models I am using require front-face culling, which means that the 3D models have clockwise (CW) windings, so I need to build these triangles in CW order as well.

Astute readers will notice that you are building what amounts to the same array of UV coordinates every time. In theory, you could just build that array once, since they are all the same. This is true, and if I didn't have a plan that involved multiple sets of UV coordinates in mind for later in the chapter, then it would be silly to build the same array over and over again.

```
-(void)addVertex:(CGFloat)x y:(CGFloat)y u:(CGFloat)u v:(CGFloat)v
{
    NSInteger pos = vertexIndex * 2.0;
    vertexes[pos] = x;
    vertexes[pos + 1] = y;
    uvCoordinates[pos] = u;
    uvCoordinates[pos + 1] = v;
    vertexIndex++;
}
```

Here is the add vertex method. It just populates the vertex and UV arrays with data and increments the vertexIndex.

Almost there! Now to emit new particles! But first, let's talk about random numbers.

Random Numbers and Initial Conditions

One of the defining characteristics of a particle system is that each particle contains its own unique state. Each new particle put in the system has its own unique initial conditions as well (and by *unique* I mean unique-ish). There is actually a pretty good chance in a particle system that you will have a few particles that are exactly the same, but I digress.

How do you make each particle unique? As you may have guessed by the title of this section, one way is with random numbers. However, that is not the only way.

You can (and many have) model your particle effects after real-world systems. You can define the various characteristics and particle behaviors with systems of equations. For instance, if you really wanted to model the way a rocket engine ejects mass to provide thrust, you might build a numeric simulation to take into account the expansion pressure of the fuel, the nozzle shape, the size of the payload, and the wind speed. You could then impart this information into your particle system and have a very realistic simulation of a rocket launching.

However, I find it much easier to just fake it.

Instead of real-world mathematic models, you can just define a range of valid values for each state variable in a particle. The more unique each particle is, the more interesting and not fake your systems will look.

This brings us to random numbers. As many know, random numbers are not really all that random, but for our purposes, semirandom will do fine. To get a nice random number from a range, you will use one of the handy inline functions that is in the `GameTypes.h` file:

```
static inline CGFloat BBRandomFloat(BBRange range)
{
    // return a random float in the range
    CGFloat randPercent = ( (CGFloat)(random() % 10001) )/10000.0;
    CGFloat offset = randPercent * range.length;
    return offset + range.start;
}
```

This just takes one of the range structures as input and returns a float value that lies somewhere in that range. Easy!

The downside to this approach is that there are lots of little things to tweak to get the exact effect you want. You will get to see this firsthand later in the chapter.

Emitting Particles

Let's get back to the particle emitter. You were just about to spawn some new particles into the world:

```
-(void)emitNewParticles:(NSTimeInterval)deltaTime
{
    if (!emit) return;
    if (emitCounter > 0) emitCounter -= deltaTime; // if emitCounter == -1, then emit
    forever
        if (emitCounter <= 0) emit = NO;
```

OK, already some strangeness. What is this emitCounter?

Often you want your particle system to simulate some short event instead of a constant flow of particles. The emitCounter is a handy way to preload an emitter with a set time before it shuts down. This is especially useful for things like explosions where you want to emit a very large number of particles in a short time. If you want your particle emitter to generate constantly for a long time, then you just need to set the emit count to some very large number, like 10000.

```
CGFloat newChance = ([self randomFloat:emissionRange] * deltaTime);
particleRemainder += newChance;

if (particleRemainder < 1.0) return;
```

Next is emissionRange. This range is the number of particles that can be emitted in a given second. Since this can be very small (maybe you are simulating a leaking faucet that drips only once every ten seconds), you need to add up all the incremental "chances" until you get one full particle. This is what the particle remainder is for; it keeps track of your incremental progress.

```
NSInteger newParticleCount = (NSInteger)particleRemainder;
particleRemainder -= newParticleCount;
```

OK, you have at least one particle! You put the fraction remains back into particleRemainder and move on to the actual emitting stage:

```
NSInteger index;
for (index = 0; index < newParticleCount; index++) {
    if ([unusedParticles count] == 0) {
        return;
    }
}
```

If you have no more particles, then you simply give up. You will have to wait until some particles die before you can emit any more. If you find yourself getting into this clause quite a bit, then you need to increase your max particles.

```
BBParticle * p = [unusedParticles lastObject];

p.position = [self newParticlePosition];
p.velocity = [self newParticleVelocity];

p.life = [self randomFloat:lifeRange];
p.size = [self randomFloat:sizeRange];
p.grow = [self randomFloat:growRange];
```

```

        p.decay =[self randomFloat:decayRange];

        [self addChildParticle:p];
        [unusedParticles removeLastObject];
    }
}

```

You grab the last particle in the pool and set the initial conditions using your fancy random float function. Then you add it to the active particles and remove it from the pool.

```

-(BBPoint)newParticlePosition
{
    return self.position;
}

-(BBPoint)newParticleVelocity
{
    return BBPointMake(
        BBRandomFloat(xVelocityRange), BBRandomFloat(yVelocityRange), BBRandomFloat(zVelocityRange)
    );
}

```

These are just some handy functions to make it easier to build the position and velocity values. Hmm...why would you need separate methods just to return the position and build a simple random point? Perhaps you will be modifying these methods later.

That is it for the simple emitter! You now have an emitter that should emit particles from a single point, each particle having a variable velocity, size, and life.

This may not seem like much, but you can simulate quite a few things with just these simple states.

Tweaking Your Particle System

Now you have the means to generate some particles, so let's get to it!

In the sample code, I have set up five scenes and a handy set of buttons to be able to load each scene. Each one of these scenes has a particle emitter in it, and they are basically set up to be particle playgrounds. The `SceneObject` will overlay the scene-changing buttons as long as you don't forget to call `[super awake]` in the subclass `awake` method.

First, let's look at `SceneZero`. This will be your first and simplest emitter. You will use the emitter code that you looked at in the past few sections, so you'll have just velocity and size and life. You will start with the Hello World of emitters: the explosion:

```

@implementation BBSceneZero
-(void)awake
{
    [super awake];
    [[BBMaterialController sharedMaterialController] loadAtlasData:@"particleAtlas"];
}

```

First you need to make sure that your materials are available, or who knows what you might get. You can call this over and over again (for instance, if you leave this scene and come back, this will get called again) because it will load the texture atlas only once.

```
particles = [[BBParticleSystem alloc] init];
particles.position = BBPointMake(0.0, 0.0, -50.0);
particles.emissionRange = BBRangeMake(2500,2500);
particles.emitCounter = 0.1;
```

Here you are setting the emission range to fall between 2,500 and 5,000 particles a second. That is a huge amount! This is why you are going to emit particles for only a tenth of a second.

```
particles.xVelocityRange = BBRangeMake(-500, 1000);
particles.yVelocityRange = BBRangeMake(-500, 1000);
particles.zVelocityRange = BBRangeMake(-500, 1000);
```

You will emit particles that are moving between 0 and 500 pixels per second in all directions.

```
particles.lifeRange = BBRangeMake(10.0,0.0);
particles.decayRange = BBRangeMake(2,0.00);
```

All the particles will have exactly 10 life, and decay at 2 life per second. This means each particle will live 20 seconds.

```
particles.sizeRange = BBRangeMake(2, 2);
particles.growRange = BBRangeMake(-1.0, 0.5);
```

The particles will start between 2 and 4 pixels wide, and they will shrink by somewhere between 0.5 and 1 pixels per second.

```
particles.emit = NO;
```

The emitter will start dormant.

```
[particles setParticle:@"whiteSubtle"];
[self addChild:particles];
}
```

Set the particle to your very translucent white blur, and add the particle system to the child array so that it will get caught by the renderer:

```
-(void)update:(NSTimeInterval)deltaTime
{
    [super update:deltaTime];
    // check our emit status
    if (particles.emit == NO) {
        particles.emitCounter = .10;
    }
}
```

In the update method, which is called every frame by the game loop, you will check to see whether the emitter has been shut down. If so, then you reset the emitCount to be ready for another explosion:

```

-(void)emitButtonDown
{
    particles.emit = YES;
}

```

Ahh, yes, the emit button. The Scene class provides a big overlay button that lays overtop the entire screen area, except where the scene-switching buttons are. It provides a method callback for that big overlay button, and it is called `emitButtonDown`. You will use this to your advantage in many scenes. In this case, you are just turning the emitter on. The emitter will run for 0.1 seconds and then shut itself off. At some point after that, your update will be called, and you will reset the `emitCount` so you can start over again.

This means that just about every time you tap the screen, you will get an explosion of white particles.

One thing I must apologize for: it is hard to take good screenshots of particle systems. The beauty of the system is in its ever-changing and fluid nature. A screen capture robs the system of its best quality: the emitter's appearance over time. So, you will need to either be very imaginative or quickly build the app for yourself and try it. Figure 1-9 is a good example, because a static shot it is just a bunch of white dots, but in motion it is so much more.



Figure 1-9. Boom! The *SceneZero* emitter after I hit it about 20 times in quick succession. Note the two numbers in the lower left. The big one is the number of particles, and the smaller one is the frame rate. I got this in the simulator. You would be hard-pressed to generate 28,000 particles and keep a 30 fps on the device.

OK, now you have seen the basic explosion, so let's tweak this particle system to look like something entirely different.

You will use the same particle, mostly just to illustrate the flexibility of particles. Let's tweak this system so it looks like the thruster exhaust from a spaceship.

We will do this in *SceneOne*. It is already set up with three particle emitters; you just need to tweak the emission parameters.

Let's take a look:

```

// thruster
particles.emissionRange = BBRangeMake(50,50);

```

We don't want to emit a kagillion particles every second like the explosion. This system will go as long as you are touching the screen, so you want to have a decent but not crazy stream.

```
particles.xVelocityRange = BBRangeMake(40, 40);
particles.yVelocityRange = BBRangeMake(-5, 10);
particles.zVelocityRange = BBRangeMake(-5, 10);
```

If you are imagining that this system is thrust exhaust from a ship or a rocket, then it will mostly be directed in a single direction, in this case, to the right. You want to give the particles some random velocity in the y and z as well; this will give you a nice cone of particles.

```
particles.growRange = BBRangeMake(-1.5, 0.5);
particles.sizeRange = BBRangeMake(2, 8);
```

Since this is exhaust, or a plasma drive or even some energy drive, you want the particles to get smaller as they go on to give them the appearance of evaporating. So, you will give them a net negative grow rate between -1.5 and -1.0 . You want the initial size to be fairly wide-ranging. Since you are shrinking about 1-pixel size per second, then after one second from the emission point, our 2-pixel particles will be a single pixel. This will give a nice effect mixed in with some bigger ones that never get that small.

```
particles.lifeRange = BBRangeMake(5.0, 0.0);
particles.decayRange = BBRangeMake(1, 1);
```

Finally, you will define the life span. Give each particle exactly 5 for the life and between 1 and 2 for the decay rate; this means that the particles will live for between 2.5 and 5 seconds. However, they can die sooner if they shrink below a visible size.

There are two more particle systems included in SceneOne. Go ahead and tweak those and see whether you can make some fun-looking thrust effects.

Figure 1-10 is what I came up with. The top system is the one I described here. (If you want to have the ships too, just uncomment the `[self addShips]` call at the end of the `awake` method.)

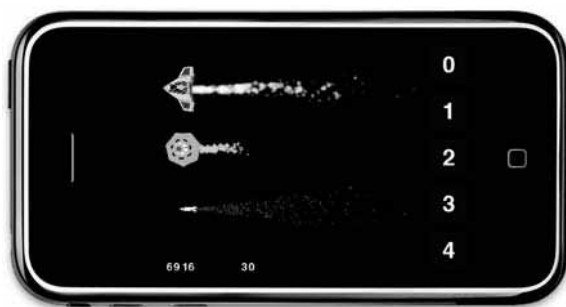


Figure 1-10. Three exhaust particle systems. I added the ships and things for inspiration.

May the Force Be with Your Particles

I haven't really even touched on the vast number of systems you can simulate with the simple particle emitter you have so far. I absolutely encourage you to play around with the particle emitters in SceneOne and see what kinds of things you can come up with.

However, it is time to advance the art of the emitter. It would be groovy if your particles could be affected by gravity or the wind or both!

It would also be quite nice if you could have the particles emit from something besides a single point. If you could emit from a larger volume, then you could create things like rain and snow, not to mention making your thruster emitters a bit nicer.

Let's start with gravity—or more generally, any force. For our purposes, force can be considered roughly equivalent to acceleration. (It is really $\text{mass} * \text{acceleration}$, but I will simplify it for our purposes; just don't tell my college physics professor.)

Just like velocity is a change in position over time, acceleration is simply a change in velocity over time. Let's look at adding a force to your particle. You will need a new instance variable:

```
@interface BBParticle : NSObject {
    .
    .
    BBPoint force;
    .
    .
}

@property (assign) BBPoint force;
.
.
```

And a simple addition to the update method:

```
-(void)update:(NSTimeInterval)deltaTime
{
    velocity.x += force.x * deltaTime;
    velocity.y += force.y * deltaTime;
    velocity.z += force.z * deltaTime;
    .
    .
}
```

Easy! Now you need to just add a force var to the emitter and update the emit method:

```
-(void)emitNewParticles:(NSTimeInterval)deltaTime
{
    .
    .
    .
    NSInteger index;
    for (index = 0; index < newParticleCount; index++) {
        .
        .

        p.force = force;
    }
}
```

```

    }
    .
}

```

Wow, that was easy. Let's see how this can affect the thruster particles:

```
particles.force = BBPointMake(0.0, -20.0, 0.0);
```

You just need to add this line to your SceneOne thrusters, and they will all get a constant negative y acceleration, which may or may not look like gravity. In Figure 1-11, I turned off the other two emitters so that I could easily see the effect of the force on the top emitter.



Figure 1-11. *The top emitter now with more gravity*

Well, that was easy, so let's go ahead and add an emission volume. This time you only need to add some stuff to the emitter object:

```

BBRange emitVolumeXRange;
BBRange emitVolumeYRange;
BBRange emitVolumeZRange;

```

You will add a few new instance vars to the ParticleSystem object, and then you just need to change the particlePosition method a wee bit:

```

-(BBPoint)newParticlePosition
{
    return
    BBPointMake(BBRandomFloat(emitVolumeXRange),BBRandomFloat(emitVolumeYRange),BBRandomFloa
t(emitVolumeZRange));
}

```

This will emit particles randomly in a squareish volume defined by the emit volume ranges. See Figure 1-12.

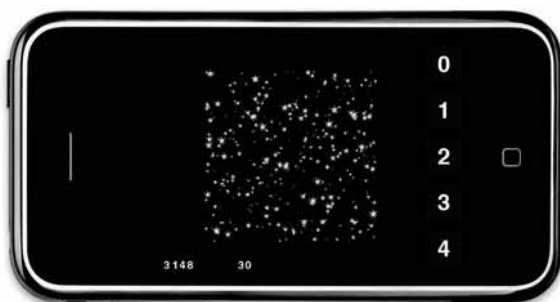


Figure 1-12. An emitter emitting stars into a 3D rectangular volume. This is in the sample code in *SceneTwo*.

Rectangular emission volumes are the easiest way to go and are often enough for what you need. However, sometimes you do not want that. It might be nicer to be able to have the particles fill a spherical volume.

```
// a random position around my position
-(BBPoint)newParticlePosition
{
    if (!sphericalEmissionVolume) return
    BBPointMake(BBRandomFloat(emitVolumeXRange),BBRandomFloat(emitVolumeYRange),BBRandomFloat(emitVolumeZRange));
```

You need to add a new property: `sphericalEmissionVolume`. I have set this to default to YES because I generally find myself wanting to use the spherical emitters.

```
BBPoint rawPos = BBPointMake([self randomFloat:zeroToOne],[self
randomFloat:zeroToOne],[self randomFloat:zeroToOne]);
    if ((rawPos.x * rawPos.x + rawPos.y * rawPos.y + rawPos.z * rawPos.z) > 1.0) rawPos
= BBPointNormalize(rawPos);
```

OK, here you are going to do some math. If you don't like math, then look away now. What you are doing is grabbing three values between 0 and 1. This should give us a point anywhere in the unit cube. Next you will check to see whether it is inside the unit radius from 0,0 by checking the length of the vector against 1 (technically 1 squared). If it falls outside the unit radius, then you normalize it.

```
rawPos.x *= [self randomFloat:emitVolumeXRange];
rawPos.y *= [self randomFloat:emitVolumeYRange];
rawPos.z *= [self randomFloat:emitVolumeZRange];
return rawPos;
}
```

Now you take your normalized vector that is guaranteed to be inside the unit radius and multiply it by your emit volume ranges. This will result in a point somewhere inside the spheroid that is bounded by the three emit volume ranges. This works for ellipsoids as well, so feel free to provide asymmetrical emit ranges. You can see in Figure 1-13 my new spherical bounding volume.

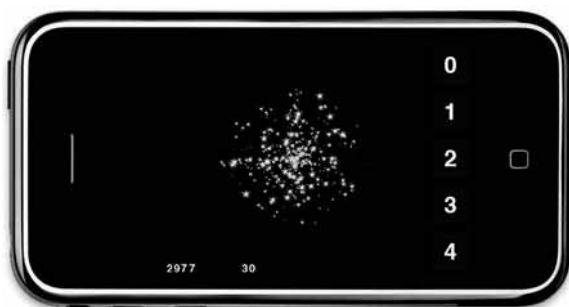


Figure 1-13. A spherical emission volume. This is in the sample code in *SceneTwo*.

A side note to the probability geeks in the crowd: this will not give you an even distribution. It is a bit of a fake to get the points in a known amount of time, though it is good enough for most things. If you really want a statistically even distribution inside the sphere (within the limitations of the random number generator), then you can't just normalize the vector. Instead, you would want to keep generating random points until you found one that lies inside the unit radius.

Amazing Technicolor Dream Particle

We are nearing the end of the chapter, and I wanted to cover the final thing that I consider a “must have” for any particle system: color animation. Color-changing particles is the final piece of the puzzle that will help bring your particle systems to life.

What do I mean by color animation?

So far, you have been using your textures quad particles without any additional color information. You have been using whatever color information the texture provided basically. However, you can just as easily enable the `GL_COLOR_ARRAY` and send in color information, tinting your textures to whatever color you want.

This is useful in two ways. First, you can now set a color for your particles, so if you don't like the white rocket exhaust for your spaceship, you can change it by just setting a color instead of using a new texture. Second, you can change the color of the particle based on its life span (or size or position or whatever you want).

You'll now learn how to add a fairly standard two-color scheme to your system, based on the life of the particle. However, there is nothing stopping you from using the same technique to animate through three or four or five colors.

Off on a Tangent: Lerp

To be able to figure out the value between two colors, you need to be able to interpolate that value. *Interpolation* is the process by which you guess the value of an unknown point based on some known values. Often it is used in curve-fitting and other data

manipulation fields. There are tons of different forms of interpolation, but you are going to look at the most simple way: linear interpolation.

Linear interpolation, also known in the graphics/game development/math world as simply *lerp*, is a very handy thing to keep in your toolbox of mathematical functions.

Lerping is really very easy, and you have probably done it once or twice before and not even realized that it had a name. So, even for the math-phobic, this section should be pretty painless.

I am bringing up lerping in the context of finding a middle color between two other colors, but it has broad-reaching uses in game development, so I wanted to at least spend a few paragraphs bringing it to your attention. Lerping is not only good at finding colors, but it is also fantastically useful for animation and tweening.

Let's say you have an enemy spaceship that needs to fly from point A to point B. You can simply lerp the position from A to B over time. Easy! Similarly, lerping is a quick and simple way to add movement to your objects and game items. When you are using Core Animation to implicitly animate your layers, whether you are moving the layer around or rotating it or whatever, Core Animation is lerping your layer from the start value to the end value. You can use it for lots of things in your games (and as I mentioned, you probably are, without knowing it).

So, let's look at a simple lerp function that you should be using for everything:

```
static inline CGFloat BBLerp(CGFloat start, CGFloat end, CGFloat amount)
{
    if (amount < 0.0) amount = 0.0;
    if (amount > 1.0) amount = 1.0;
    CGFloat spread = end - start;
    return (spread * amount) + start;
}
```

This is a very simple bounded linear interpolation over the values 0 to 1. If you send in `amount = 0`, then you will get back the starting value. If you send in `amount = 1`, then you will get the end value. If you send in 0.5, then you will get back the value that is halfway between start and end. You have probably had to do this before, and this just puts it in a nice simple form that is useful in a plethora of situations.

You can also add a handy point-to-point lerp (this is what is happening in Figure 1-14):

```
static inline BBPoint BBPointLerp(BBPoint start, BBPoint end, CGFloat amount)
{
    BBPoint lerped;
    lerped.x = BBLerp(start.x, end.x, amount);
    lerped.y = BBLerp(start.y, end.y, amount);
    lerped.z = BBLerp(start.z, end.z, amount);
    return lerped;
}
```

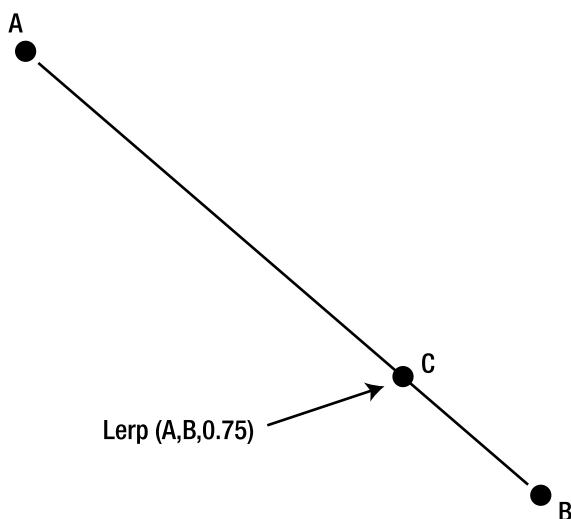


Figure 1-14. *Simpler linear interpolation. C is three quarters of the way between A and B.*

This is a book about advanced projects, and some might think that lerp-ing is a pretty basic concept. That is true, it really is, but I wanted to go over it quickly because it is so handy and simple. And if you haven't seen it before, it can be a revelation.

Anyway, back to colors.

Color-Changing Particles

To be able to lerp between two points, you will need a start point, an end point, and a value between 0 and 1 that is my position along that line. In the case of your particles, you can use the life as a positional indicator, but in order to get a life value between 0 and 1, you will need to know the starting life value.

You will also need a place to put your color values in the particle:

```
@interface BBParticle : NSObject {
    .
    .
    .
    CGFloat startingLife;
    CGFloat r;
    CGFloat g;
    CGFloat b;
    CGFloat a;
}

.
.
.

@property (assign) CGFloat r;
@property (assign) CGFloat g;
@property (assign) CGFloat b;
```

```
@property (assign) CGFloat a;
@property (assign) CGFloat startingLife;
```

In the implementation file, you only need to add the properties to be synthesized, and you will be all done with your new colored particle:

```
@implementation BBParticle
.
.
@synthesize r,g,b,a,startingLife;
.
.
```

Next you need to add a handful of new instance variables to your particle system class:

```
@interface BBParticleSystem : BBSceneObject {
.
.
.

    CGFloat startR;
    CGFloat startG;
    CGFloat startB;
    CGFloat startA;

    CGFloat endR;
    CGFloat endG;
    CGFloat endB;
    CGFloat endA;

    BOOL animateColor;
.
.
}
```

You want to make these properties, so don't forget to add the @property declarations and the @synthesize declarations in the implementation file.

The animateColor flag will tell the emitter whether you need to bother building the color arrays. If you do not plan to use the color feature, be sure to set this to NO. Don't just set the colors to white. The color array is 4 floats per vertex, and it will affect your performance to be pushing all that extra data into the renderer, so turn it off if you do not need it. I have set animateColor to default to NO in the sample project.

To use the color arrays, you need to make sure that you have a buffer malloced for that, so at the end of the preload method, add a malloc for the color array:

```
-(void)preload
{
.
.
.
    colors = (CGFloat *) malloc(4 * 6 * kMaxParticles * sizeof(CGFloat));
}
```

Also, in the setParticle: method, you need to link the mesh's color array to your new buffer:

```

-(void)setParticle:(NSString*)atlasKey
{
    .
    .
    .
    mesh.colors = colors;
    mesh.colorSize = 4;
}

```

Now you have a place to put your colors, and the mesh is all hooked up. Next you just need to set the color in the particles and generate the color array during your update loop.

Let's start at the beginning of the particle life cycle: `emitParticles`. You need to set the initial RGBA values on the particle as well as the new `startingLife` value:

```

-(void)emitNewParticles:(NSTimeInterval)deltaTime
{
    .
    .
    .
    NSInteger index;
    for (index = 0; index < newParticleCount; index++) {
        .
        .
        .
        p.r = startR; // set the colors
        p.g = startG;
        p.b = startB;
        p.a = startA;
        p.life = BBRandomFloat(lifeRange);
        p.startingLife = p.life; // set so you can do color animation
        .
        .
        .
    }
}

```

Now your newly minted particles will all have the right initial conditions. Moving to the next stage of the particle life: the update. This is the update method in the particle emitter. You don't actually need to change the individual particle update method.

```

-(void)update:(NSTimeInterval)deltaTime
{
    // update active particles -> move them
    [super update:deltaTime];
    for (BBParticle * kid in childrenParticles) {
        [kid update:deltaTime];
        if (animateColor) {
            kid.r = BBLerp(startR, endR, (kid.startingLife -
            kid.life)/kid.startingLife);
            kid.g = BBLerp(startG, endG, (kid.startingLife -
            kid.life)/kid.startingLife);
            kid.b = BBLerp(startB, endB, (kid.startingLife -
            kid.life)/kid.startingLife);
            kid.a = BBLerp(startA, endA, (kid.startingLife -
            kid.life)/kid.startingLife);
        }
    }
}

```

```

    }
}
// emit -> add new particles
// build arrays
[self buildVertexArrays];
[self emitNewParticles:deltaTime];
if (animateColor) [(BBTexturedQuad*)[self mesh] setUseColors:YES];
}

```

There are two new things going on here. First, as you loop through all the child particles, you will lerp the new color based on how long that particle has lived in relation to its total life. Then, at the end of the method, you make sure that your mesh is set to use the color arrays.

Finally, you look at the array construction:

```

-(void)buildVertexArrays
{
    vertexIndex = 0;
    for (BBParticle * particle in childrenParticles) {
        .
        .
        if (animateColor) {
            [self addColorsR:particle.r g:particle.g b:particle.b a:particle.a
vertexes:6];
        }
        // for each particle, need 2 triangles, so 6 verts
        // first triangle of the quad. Need to load them in clockwise
        // order since our models are in that order
        [self addVertex:(particle.position.x - particle.size) y:(particle.position.y +
particle.size) u:minU v:minV];
        .
        .
        .
    }
}

```

It is fairly important that you build the color array before you start adding vertexes. Since you are setting the same color to each vertex, you can do it all at once, but the `addVertex` method increments the `vertexIndex`, so if you do not do the colors first, you will lose your place.

As for the actual add colors method, it is pretty straightforward:

```

-(void)addColorsR:(CGFloat)r g:(CGFloat)g b:(CGFloat)b a:(CGFloat)a
vertexes:(NSInteger)verts
{
    NSInteger index;
    for (index = vertexIndex; index < (vertexIndex + verts); index++){
        NSInteger pos = index * 4.0;
        colors[pos] = r;
        colors[pos + 1] = g;
        colors[pos + 2] = b;
        colors[pos + 3] = a;
    }
}

```

Just add the same color to the color array for each vertex. Simple!

You can now try your new color-changing emitter! Open SceneThree, and let's take a look at an animated Technicolor dream emitter. Now that you can do color-changing particles, you can simulate a fairly decent fire.

```
particles.position = BBPointMake(0.0, -80.0, -50.0);
```

You will put the fire at the bottom of the screen, so you have room to burn:

```
particles.emissionRange = BBRangeMake(40,50);
particles.xVelocityRange = BBRangeMake(0, 0);
particles.yVelocityRange = BBRangeMake(1, 10);
particles.zVelocityRange = BBRangeMake(0, 0);
```

This is fire, so mostly it will just be going up, so you will set our x and z velocities to 0:

```
particles.emitVolumeXRange = BBRangeMake(-30, 60);
particles.emitVolumeYRange = BBRangeMake(-5, 10);
particles.emitVolumeZRange = BBRangeMake(-30, 60);
```

You will have it emit from a flattened spheroid:

```
particles.force = BBPointMake(0.0, 10.0, 0.0);
particles.growRange = BBRangeMake(-1.5, 1.5);
```

And give it a nice upward force, since the particles should be lighter than air:

```
particles.sizeRange = BBRangeMake(6, 6);
particles.lifeRange = BBRangeMake(2.5, 0.0);
particles.decayRange = BBRangeMake(0.5, 0.1);
```

```
// start with a nice pure yellow
particles.startR = 1.0;
particles.startG = 1.0;
particles.startB = 0.0;
particles.startA = 1.0;
// end with a dark red
particles.endR = 0.5;
particles.endG = 0.0;
particles.endB = 0.0;
particles.endA = 1.0;
```

```
particles.animateColor = YES;
```

And finally, you set the start color to a pure yellow and your end color to a dark red. Don't forget to set animateColor to YES:

```
// this will make it a rectangular emission volume
//particles.sphericalEmissionVolume = NO;

particles.emit = YES;
[particles setParticle:@"whiteBlur"];
```

Finally, you set your particle to the good old standby: whiteBlur (which is the brighter cousin of whiteSubtle).

Figure 1-15 shows the result of the color-changing efforts. The color change really adds that bit of life that really makes the effect jump out and look great.

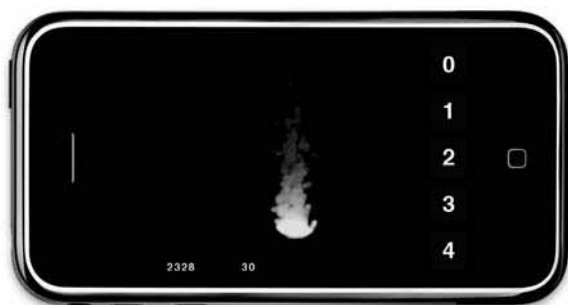


Figure 1-15. *A pretty decent fire effect, all things considered*

Summary

You explored the world of particle generation in this chapter, and I covered the basics: life, growth, speed, acceleration, and color. These five things are the basis of all particle systems. If I had more time and space, I could talk about per-particle rotation, moving particle emitters, particle collisions, and multitexture systems. And those are just a few of the many permutations that you can add to your particle systems.

I encourage you to go out and experiment—try to add particle rotations and multicolor animations. Have your system randomly select a set of UVs from a list of textures and make a multitexture system. Add a collision detection system, and apply it to the particles so you can simulate realistic effects.

The other thing that I wanted to touch upon was actual particle artwork. For the most part, I have been using a very simple white blur for all the effects in this chapter (with a small digression with some poorly drawn stars and thruster options). This was mostly on purpose to show that the versatility of the particle system lies not in the individual textures of the particles but in the infinite flexibility of the system and all of the configuration variables.

That said, the next step is to play with various particle textures to try to achieve the effect you need. Each texture will give a very different look and feel to the same emitter settings, so play around. In Figure 1-16 I built four very different effects by just changing the color and the particle texture of the fire effect.

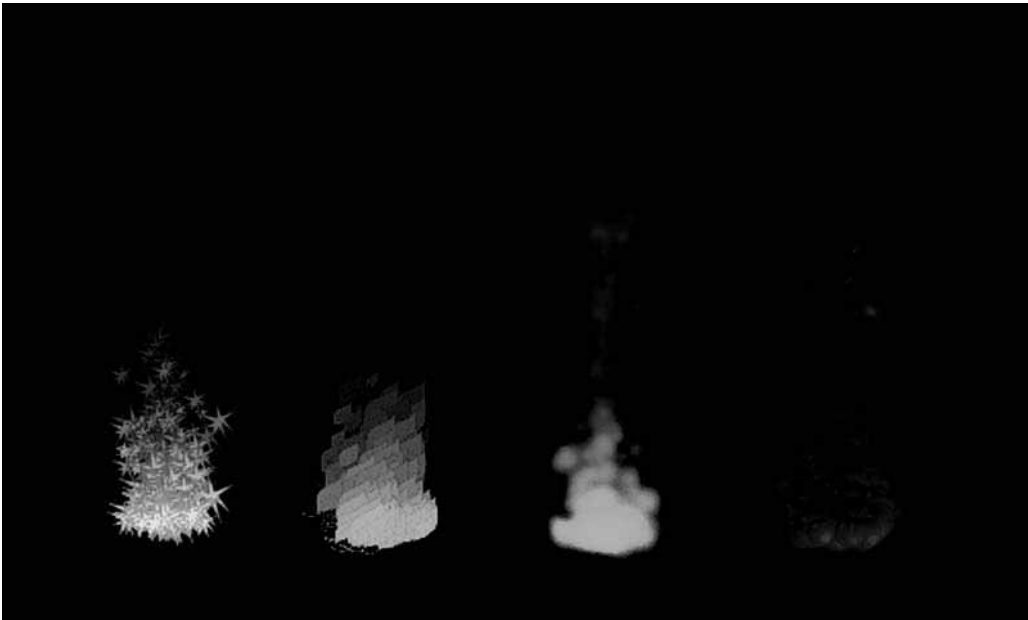
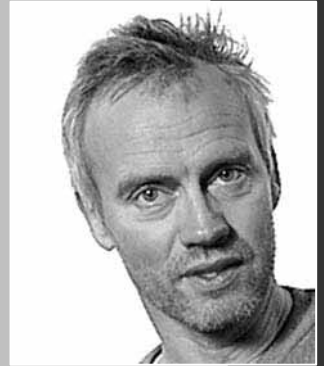


Figure 1-16. *Playing with the fire effect. Just a few permutations of the color and particle texture have huge effects on the look of the system.*

In the sample code, I left the fourth scene empty. It is your playground. Go crazy and experiment. Particle systems are very fun to play with, and I spent most of my time while writing this chapter just tweaking the various effects. Not only did I want to get them to look good for the chapter, but it was just so much fun to see what the fire would look like if the yellow were a bit more orange. Or if I could make it look like some evil magic energies if I changed the colors to go from green to purple (answer: yes!).

Joachim Bondo



Company: Cocoa Stuff (one man shop)

Location: Copenhagen, Denmark, Europe

Former Life As a Developer: 27 years of experience in starting up and running smaller software development companies and developing software using a wide range of programming languages such as: BASIC, COMAL 80, Pascal, C, C++, Objective-C, SQL, NewtonScript, PHP, JavaScript, Bash

...in environments such as: THINK C and TCL (Think Class Library), MPW (Macintosh Programmer's Workshop), Metrowerks CodeWarrior and PowerPlant, 4th DIMENSION, NTK (Newton Toolkit), Sybase, MySQL, TextMate, Xcode, Cocoa, Cocoa Touch

...on platforms such as: Mac OS 3–8, Newton OS, Palm OS, UNIX (FreeBSD, Mac OS X), Mac OS X Panther–Leopard, iPhone OS

Life as an iPhone Developer: Deep Green, chess game, using the official iPhone SDK from Apple since the day it was released.



What's in This Chapter: Deep Green, or: How I Achieved the Goal of Simplicity

With the focus on creating a beautiful, elegant, and powerful user interface, and in a non-code language, I'm going through key areas of what have made Deep Green a successful application on the App Store, featured by Apple in several sections such as What's Hot and Staff Favorites.

Key Technologies:

- *User Interface Design*
- *Simplicity*
- *Product Statement*

Chess on the 'Net: Correspondence Gaming with Deep Green

As I'm writing this, version 2.0 of my popular chess application, Deep Green, is under development. One of the big new features is correspondence chess. In other words, users will be able to play chess with their friends independent of time and place. This is in contrast to over-the-board chess where you sit at the chessboard at the same time.

So, all from within Deep Green, you'll be able to invite a friend to a game of chess and then each make your moves in turn as you normally do. Your moves will be stored in a database on a central server, and after each move, the system will push a remote notification to the opponent. The time interval between moves can be anything from seconds to weeks.

In this chapter, I'll show you how to code the support for sending an invitation, accepting the invitation, sending moves back and forth, and storing it all on the server in a database. I'll go through what's needed on the client (that is, the iPhone or iPod touch device), as well as what's needed on the server, including the choice of platform, database, and programming language. I'll show the mechanics so that you'll be able to implement similar functionality for your own applications.

As it turns out, you won't see a whole lot of code, which is a good thing because it shows that it's fairly easy to do some relatively powerful stuff using the chosen technologies.

But let me first make you a little bit familiar with the application.

Deep Green, an Already Awesome Application

Deep Green 1.0 was released in December 2008. It created a lot of buzz for its beautiful and intuitive user interface (see Figure 2-1).



Figure 2-1. *Deep Green running on the iPhone*

I originally released Deep Green for Apple's Newton platform in 1998 (see Figure 2-2). But only ten days later, Steve Jobs closed the whole Newton division down. Although it put somewhat of a damper on my development efforts, users were still enthusiastic.



Figure 2-2. *Deep Green running on the Newton*

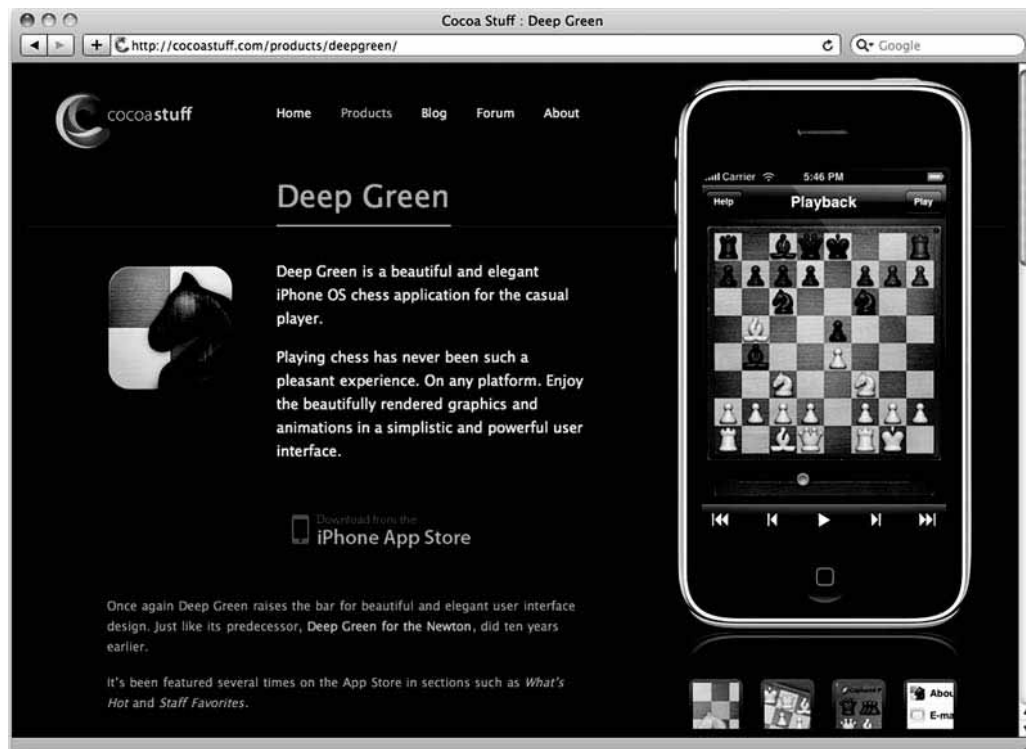
Today, ten years later, there are still Newton users out there who claim they have yet to see a better handheld device. Despite their claims, it was evident I had to carry Deep Green over to iPhone OS.

And that was exactly what I decided to do the minute I saw Steve Jobs unveil the iPhone at Macworld in January 2007. I remember seeing the slide that mentioned Cocoa as one of the many technologies iPhone was built on. So, later that year, I started coding Deep Green's model layer in Cocoa for Mac OS X 10.5. When the SDK was announced and released in March 2008, I could finally start coding natively for the platform.

Ever since the Newton version, I've maintained an extensive list of features I wanted to implement, and in the long period since then, I've added many new ones to the list. As we've become increasingly connected, correspondence chess became an obvious one. So, even though Deep Green 1.0 for iPhone OS didn't offer this, it was developed with this in mind—and much more, of course.

Now that both versions 1.0 and 1.1 are out and pretty much match the Newton version's feature set, I'm ready to start implementing some of these long-planned features.

If you want to learn more about Deep Green, and even see it in action, please visit the home page at <http://cocoastuff.com/products/deepgreen/> (see Figure 2-3).

**Figure 2-3.** *Deep Green's home page*

And while I'm in the department of shameless self-promoting plugs...if you want to read about the meticulous user interface design of Deep Green, including what \$30,000 can get you in graphics design, treat yourself to a copy of *iPhone Games Projects*, also from Apress. Visit <http://apress.com/book/view/1430219688> for more information.

The Tasks at Hand

So, what tasks are involved in making correspondence chess (or pretty much any time-shifted, turn-based game)? One thing is for sure—you need a central server that stores all the user data and that each user interacts with.

Although I had planned this feature all along, I hadn't thought about how to implement it or what components and technologies to use. So, to start, I had to define what tasks needed to be supported, which can be boiled down to the following list:

- Inviting a friend to a game
- Accepting the invitation
- Making a move
- Getting notified about new moves

I'll deal with the three first items in this chapter and will keep them relatively simple in order to focus on the main aspects of implementing this solution. So, for example, when inviting a friend, Deep Green will allow you to pick a person from your built-in Contacts application as well as just entering a username or e-mail address. Instead of going into details about implementing a people picker, I'll simply assume you have the e-mail address at hand.

Also, in a real-life solution, you'd have to handle all sorts of security issues and error scenarios. What happens if there isn't a usable network connection available? What happens if the user quits the application before it was able to send the request to the server? What happens if your friend never receives the invitation or she changed her e-mail address? What happens if she declines the invitation? How can the server know you're you, and not somebody who just knows your e-mail address?

Deep Green handles all such situations. In fact, a large portion of the time and energy that goes into designing a solution like this is spent on thinking about these odd scenarios and finding a good solution to them.

But I won't talk too much about it here, because I'd have to end the chapter before I'd even get started on the more interesting parts. I will, however, touch on where and how I've added support for these situations, where applicable.

I'll now explain a little bit more about what each task involves before diving into the code.

Inviting a Friend to a Game

Given an e-mail address of your friend, your user credentials, and a few pieces of information about the game you're inviting them to, you'll send a request to a web service that you're going to establish.

The web service will create the game, store it in a database, and send out an e-mail to your friend, carbon-copied to yourself so you know your request was made and an invitation was sent out.

The e-mail will contain standard text, explaining what it's all about, as well as links for accepting and declining the invitation.

Accepting the Invitation

Your friend will probably accept your challenge by tapping the appropriate link in the invitation e-mail.

In Deep Green, I implemented a custom URL scheme, `deepgreen://`, instead of just using `http://`. The reason for this is that I wanted Deep Green to make the request to the web service, not whatever web browser the user happens to use. By doing it this way, I can supply extra information about the user, available only on the client, and thereby eliminate a manual registration process.

I'll show how to implement a custom URL scheme on the iPhone.

Assuming the obvious, that both you and your friend have some variant of Deep Green installed, the application will launch and will take the appropriate action.

The web service that we'll develop in this chapter will save the information to the database under the game and send out an e-mail to both players letting you know the game is on.

Making a Move

When you make your move, Deep Green sends it, along with game and state information, to the web service on the server. The server records the move under the game.

It was an important design goal of mine that the server should not have to know anything about chess except that the players move in turn. But I didn't want to have to code and maintain the chess rules in several places.

Getting Notified

When you make your move, the opponent should know about it. And that's exactly what Deep Green does. Since iPhone OS doesn't allow third-party processes to run in the background, developers are left to using the Apple Push Notification Service (APNS) in

situations like this, where you want to tell the user about an event, even when your app isn't running on the user's device.

I'm not going to cover APNS in this chapter, but I'll briefly explain what it is because it's a very useful feature to implement.

Each iPhone OS device maintains a persistent connection with Apple's Push Notification server (cloud). When APNS receives a notification request from a developer's application server, it pushes it to the given device that can then display a message, play a sound, update the app icon badge, or do any combination of these depending on the user's preferences.

The tricky part, from your perspective as a developer, is the communication between your server and APNS. Not only does it require a persistent, raw socket connection, but it also requires properly issued and applied certificates (and if you've been developing iPhone apps for a while, you know what a circus that can all be).

This can all be done, of course, and I wouldn't be surprised if third-party services were available by the time you read this.

If you want to roll your own solution, however, I suggest you read *iPhone SDK 3 Projects* from Apress. There's a whole chapter devoted to this topic.

The Tools of the Trade

Now, how do you tie these things together? How do you establish our web service, and where? Which database should you be using? How do you communicate with the server?

Before I knew the answers to these questions, the only thing I knew was that I didn't want to host the solution myself. I've been doing a lot of LAMP (Linux, Apache, MySQL, PHP) over the years, but I treasure my sleep (with two little kids, even more so) and didn't want to have to worry if my server was up and running. And what about load balancing? What if users started playing a lot, which I sure hoped they would? Would the solution scale?

I had no idea until a friend of mine pointed me to Google App Engine. It was a no-brainer: build your web services using the feature-rich and elegant Python scripting language, have your objects stored in a high-performance object datastore, deploy once, and become hosted on Google's own infrastructure with thousands of servers worldwide, maintained and monitored for you. Sounds expensive, right? Actually, it's free. What's not to love?

The service is free up to certain quotas, which, at least for a chess game, seem very generous. Check with the current Google App Engine Terms of Service at <http://code.google.com/appengine/terms.html>. Over a certain limit of traffic, you'll start being charged. I guess you could say it bears the price of success: more users, more money.

Google App Engine (GAE) lets you install and deploy websites and web applications and will take care of load balancing and replicating your datastores. You can even serve your applications securely using `https://`.

Google also offers a Java SDK, if that's a better fit for you. I hadn't seriously used either Python or Java when I started coding the web service, but I felt Python was a better choice because of its increased popularity and momentum over the last years, while Java seemed to have had its days of glory.

Even if you don't know Python, you'll quickly realize from my examples that it's very easy to pick up, especially if you've already used another scripting language such as PHP. It's not totally unlike Objective-C either. The official website, <http://python.org>, offers a wealth of information but may not necessarily be the easiest place to start. I ended up buying Dr. Drew McCormack's e-book for the iPhone, *Scientific Scripting with Python*, available from the Mental Faculty at http://www.mentalfaculty.com/python_scripting. I'm not a scientist, not even a mad one, but with the e-book, I was up to speed after a few hours of reading.

Stop Talking, Start Coding!

OK, let's get our hands dirty. We'll start by implementing the web service on GAE and then the client code on the iPhone for interacting with the web service.

I'll do some basic stuff to begin with, so if you want to follow along, take a moment and sign up with GAE at <http://appengine.google.com> if you haven't done so already. As I said, it's free. You need to download and install the SDK. I hadn't used GAE before, so I'll quickly take you through the same easy steps I went through—mostly to illustrate how simple it is.

Installing the Tools

Go to <http://code.google.com/appengine/downloads.html>, and download the current version of the SDK. I'm assuming you're on Mac OS X, because that is where you're doing your iPhone development in the first place. If you use another platform for this part of the development, choose the appropriate download, and follow the installation instructions on the GAE home page. On the Mac, it's a few easy steps:

1. Open the downloaded disk image (as shown in Figure 2-4), and drag the GoogleAppEngineLauncher icon to your hard drive.

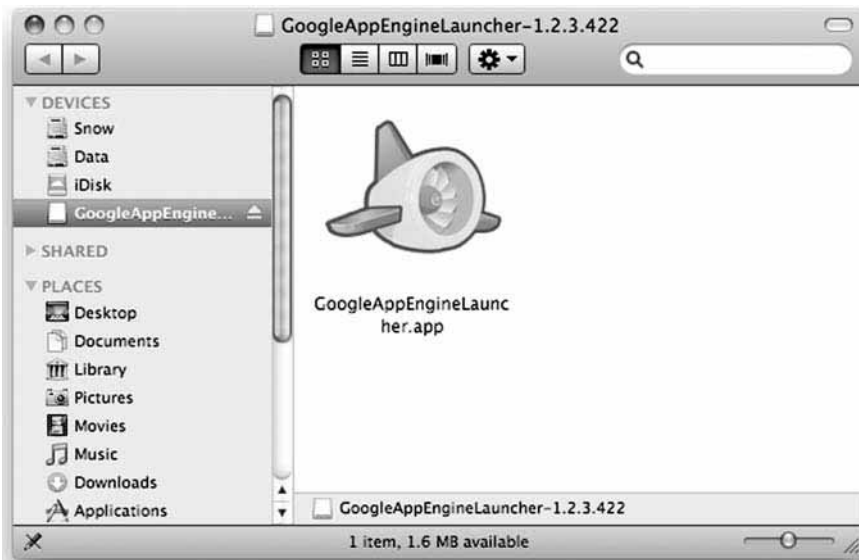


Figure 2-4. Copy the Launcher application to your local hard drive.

2. Open the Launcher and choose to make command symlinks, as shown in Figure 2-5. You'll have to do this only once.



Figure 2-5. Create the command symlinks.

3. You guessed it; there is no step 3.

The application basically sits there and mimics the whole server runtime environment. This is a huge time-saver because you don't have to upload your incremental changes to a server and because you don't have to set up a test environment on the server while developing. It also allows you to develop without a connection to the Internet.

When you're ready to deploy, you just click the Deploy button in the toolbar, as shown in Figure 2-6. I won't go through the deployment process in this chapter.

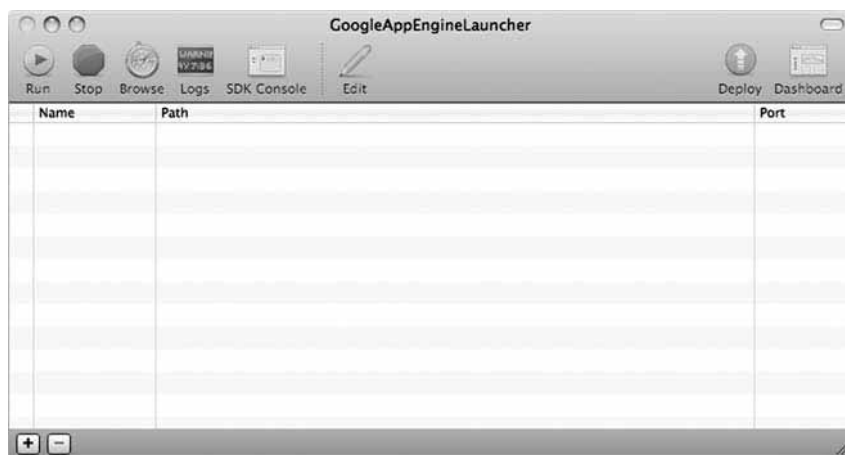


Figure 2-6. *GoogleAppEngineLauncher ready to serve you*

We're ready. Let's code....

Coding the Web Service

From here on, you'll code your web service as it will be on the server. All examples are run on the local machine. As I said, you don't have to deploy anything on Google's servers until you're ready to release your code.

Start by clicking the + button to create a new web application. You'll be prompted for a name and location, as shown in Figure 2-7.

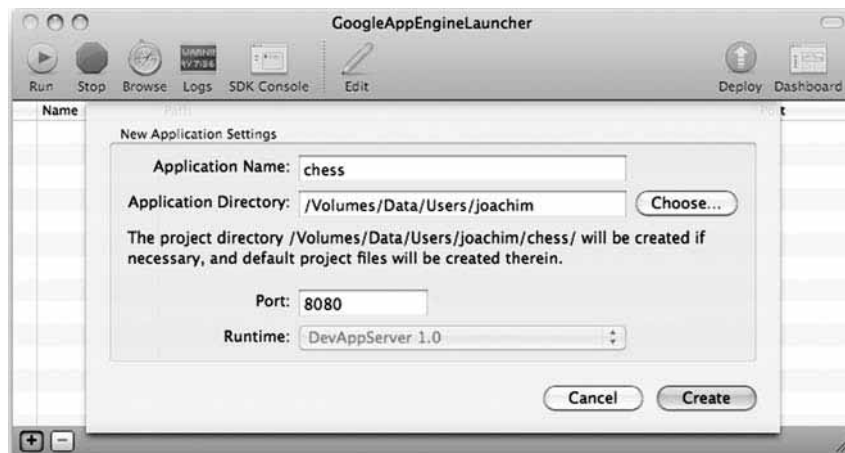


Figure 2-7. *Creating your new web application*

When you click the Create button, GoogleAppEngineLauncher will create the web application folder containing three files:

`app.yaml`: Your application configuration file

`main.py`: Your web application's main code file

`index.yaml`: An automatically generated file that you can ignore for now

If you have a text editor that can open directories, you can simply click the Edit button in the toolbar (see Figure 2-8) and have all files ready for you to read and edit. In TextMate, it looks like Figure 2-9.

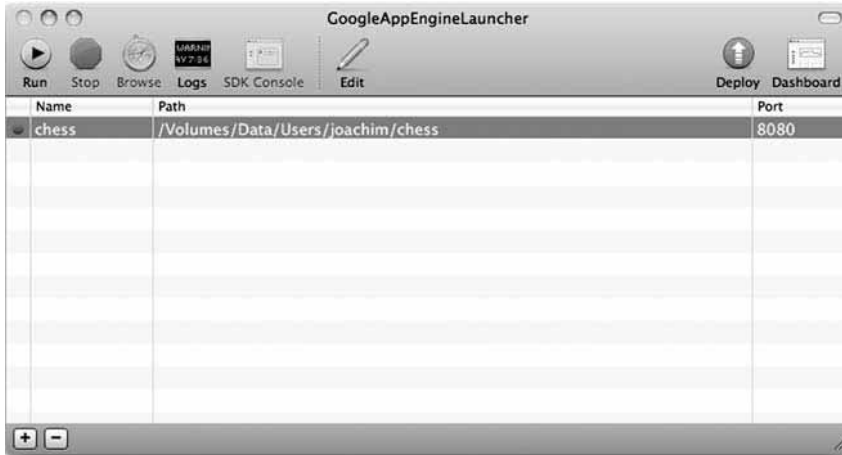


Figure 2-8. *GoogleAppEngineLauncher showing the new application*



Figure 2-9. *The web application directory in the TextMate editor*

Using your favorite text editor, open `app.yaml`. It should look like this:

```
application: chess
version: 1
runtime: python
api_version: 1

handlers:
- url: .*
  script: main.py
```

This tells the GAE runtime environment the name of your application and what script file to execute for all requests matching the regular expression `.*`.

Let's create a simple script to test whether it works. Replace the contents of `main.py` with the following:

```
print 'Content-Type: text/plain'
print ''
print 'Checkmate!'
```

Back in `GoogleAppEngineLauncher`, start the application by clicking the Run button in the toolbar (as shown earlier in Figure 2-8). Then click the Browse icon and admire the result, as shown in Figure 2-10.

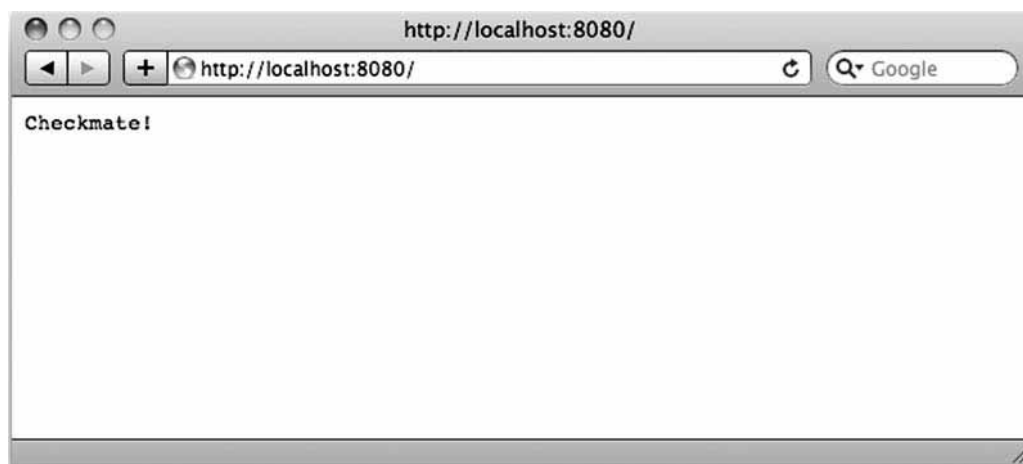


Figure 2-10. *The output from your very first web service application*

Hey, you just created a GAE web service! Perhaps it's not the most useful, although who would have thought you could checkmate in ten lines of code?

OK, it's time for some real code. You'll be implementing three *handlers*, one for each of your requests:

- Receive invitation request
- Receive invitation accept request
- Receive move request

In `main.py`, now put the mechanisms in place to deal with the requests by making it look like this:

```
from google.appengine.ext import webapp
from google.appengine.ext.webapp.util import run_wsgi_app

class GameController (webapp.RequestHandler):
    def get (self):
        opponent = self.request.get ('opponent')
        self.response.headers ['Content-Type'] = 'text/plain'
        self.response.out.write ('Looks like you want to invite ' + opponent)

application = webapp.WSGIApplication (
    [('/', GameController)],
    debug=True)

def main():
    run_wsgi_app (application)

if __name__ == "__main__":
    main()
```

You're now taking advantage of some of the Python modules available to you under GAE in order to make the coding a lot simpler. You've created a `GameController` class, which for now is being instantiated when you receive a request. It's the entry point of your web service that basically provides the methods you need in order to handle the various requests around the game play. In this case, you've implemented only a `get()` method.

This is very convenient when testing from the web browser, because it always sends GET requests when entering a URL in the address field. As you start to make these requests from the iPhone client, you'll embrace a more RESTful approach, which means that you'll use GET requests for getting data, POST for creating data, PUT for updating, and DELETE for deleting data. This will make you a good citizen of the modern Web 2.0 world.

To test the previous code, type `http://localhost:8080/?opponent=Garry%20Kasparov` in the address field of your browser. The result should look like Figure 2-11.

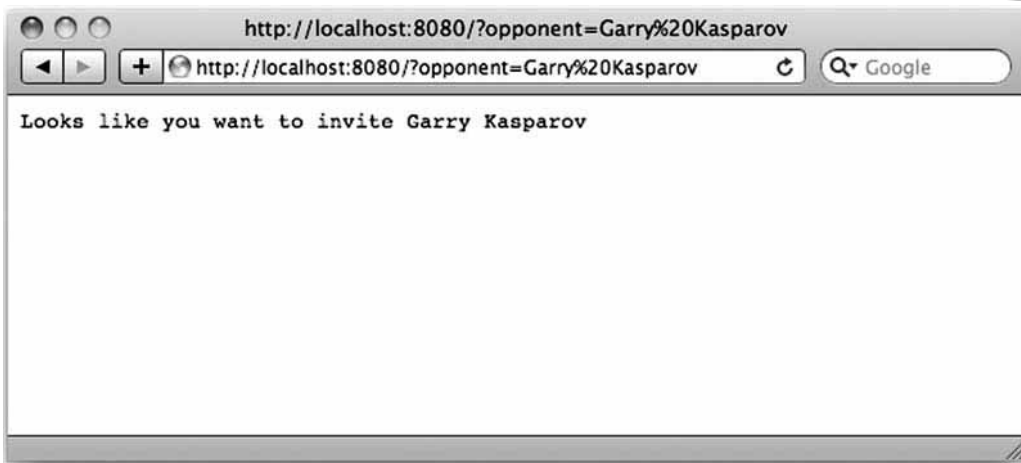


Figure 2-11. Browser output when inviting an opponent

No harm done yet. Even though we know Garry very, very well, he has no clue about our intentions. But you can now pass parameters to the service, and you're going to use some of the built-in modules to make the code simple and clean.

Let's finish the invitation implementation. Once again, you'll take advantage of a couple of the prebuilt modules in the GAE framework. There's a lot of new stuff, all of which I'll explain after the code:

```
from google.appengine.ext import webapp
from google.appengine.ext.webapp.util import run_wsgi_app
from google.appengine.ext import db
from google.appengine.api import mail

class Game (db.Model):
    inviter = db.StringProperty ()
    invitee = db.StringProperty ()
    created = db.DateTimeProperty (auto_now_add = True)
    moves = db.StringListProperty ()

class GameController (webapp.RequestHandler):
    def post (self):
        game = Game ()
        game.inviter = self.request.get ('inviter')
        game.invitee = self.request.get ('invitee')
        game.put ()
        mail.send_mail (sender = game.inviter,
                        to = game.invitee,
                        subject = "Hey, let's play chess!",
                        body = "Click here: chess://domain.com/game?action=accept&email=%s&game=%s" %
                              (game.invitee, str (game.key ()))
                        )
        self.redirect ('/games')
    def get (self):
        self.response.out.write ('<html><body>')
        self.response.out.write ('<p>Your games:</p><ul>')
        games = db.GqlQuery ("SELECT * FROM Game ORDER BY created DESC LIMIT 10")
```

```

    for game in games:
        self.response.out.write ('<li>%s vs. %s (%s)</li>' %
                                (game.inviter, game.invitee, str (game.key ())))
    self.response.out.write ('</ul></body></html>')

class EntryForm (webapp.RequestHandler):
    def get (self):
        self.response.out.write ("""<html><body>
<form action="/game" method="post">
    <div>Opponent: <input type="text" name="invitee"/></div>
    <div>You: <input type="text" name="inviter"/></div>
    <div><input type="submit" value="Invite!"/></div>
</form>
</body></html>""")

application = webapp.WSGIApplication ([
    ('/', EntryForm),
    ('/games', GameController),
    ('/game', GameController),
], debug=True)

def main ():
    run_wsgi_app (application)

if __name__ == "__main__":
    main ()

```

You're importing two new modules: `db` and `mail` for interfacing with the database and for sending e-mails, respectively.

There are two new classes: the `Game` model class, which you're storing to the database (and later will be retrieving), and `EntryForm`, which is just a temporary class for checking in the browser that the code works.

Notice how the application object gets initialized with a list of URLs and corresponding classes. The application now supports three URLs: `/`, `/games`, and `/game`.

When a client requests the root, an `EntryForm` object is being instantiated, serving an HTML form that allows you to enter your friend's and your own e-mail addresses, as shown in Figure 2-12.

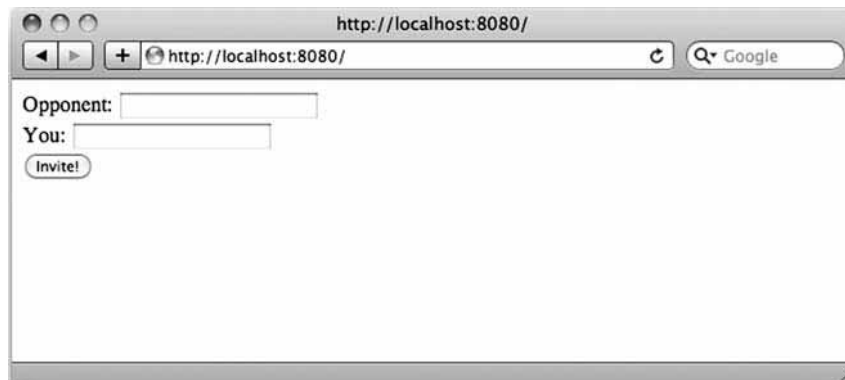


Figure 2-12. The output generated by the `EntryForm` class

Submitting the form sends a POST request to the /game URL with the two e-mail addresses as parameters. This is just a temporary mechanism to let you test the API. Eventually, the client application on the device will send the request.

The POST request is being handled by GameController's `post()` method, which then creates the Game in the data store. That's the first four lines of the method. That's pretty powerful. Notice how you don't have to make an explicit connection with the database or verbose `INSERT INTO table VALUES ()` SQL stuff. You're just creating the Game object, setting its property values, and then using `put()` to put it in the datastore. The properties of the game are defined in its class declaration.

After having created the game in the datastore, you're sending an invitation to the opponent per e-mail. The Game's key is its unique identifier, which you're using for the link so that it can be passed around as a parameter when the user taps the link in the e-mail. This way, the game key will end up as a parameter in your application on the iPhone.

Finally, the web service redirects to /games, which causes the `get()` method of the GameController to be called, and the ten most recent games get listed in reverse chronological order, as shown in Figure 2-13.



Figure 2-13. The output generated by GameController's `get()` method

The web application now illustrates the mechanics, but in a real-world scenario this wouldn't be sufficient. As mentioned earlier, you'd have to check for errors (such as invalid e-mail addresses) and apply some level of security. From the game key, for example, it would be very easy to guess keys of other ongoing games in the datastore to which one could add moves, and so on, by sending the "right" URL requests.

In addition, you'd also need more information in the Game class, such as who plays which color, any time limit per move, any nonstandard start position, and so on. These are fairly trivial exercises, so I won't waste space on this here.

Accepting the Challenge on the Device

There are still things left to code on the server, such as receiving invitation accepts and moves, but let's change the scene a bit to see how to deal with the invitation on the iPhone.

As I said earlier, the invitation URL uses a custom scheme. In this case, you're using `chess://`, but because this solution is very application specific, using the game key, for instance, you really should pick a more application-specific URL scheme instead of the very generic `chess://` URL. But again, I'm just illustrating the mechanics rather than providing a shrink-wrapped product here.

On the device, you want the application to launch when the user taps the `chess://` link in the e-mail. To make that happen, you need to define the scheme, which you just did, and implement a URL handler in your application. The latter consists of two parts: letting the iPhone OS know about your capabilities and implementing the code in your application.

Registering URL Scheme Support with iPhone OS

All you have to do to let the iPhone OS know you can handle a certain URL scheme is to provide the information in the application's `Info.plist` file:

```
<key>CFBundleURLTypes</key>
<array>
  <dict>
    <key>CFBundleURLName</key>
    <string>com.yourcompany.chessscheme</string>
    <key>CFBundleURLSchemes</key>
    <array>
      <string>chess</string>
    </array>
  </dict>
</array>
```

By adding these lines, you're telling iPhone OS that you can handle the `chess://` URL scheme. The `CFBundleURLName` string is a key that will be used by the system to look up the localized name for the scheme in your `InfoPlist.strings` file. This allows you to provide localized names of the scheme that may be displayed to the user.

Handling the URL Request

When the user taps the `chess://` link in the e-mail that the web service sent, `Mail.app` will recognize that a URL was tapped and will execute the `openURL:` method of the shared application object, `[[UIApplication sharedApplication] openURL:url]`. iPhone OS looks up what applications support the URL scheme, picks one of them, and calls its delegate's `-application:handleOpenURL:` method.

Here's an example of how this method could be implemented:

```
- (BOOL)application:(UIApplication *)application handleOpenURL:(NSURL *)url
{
    if (![url scheme] isEqualToString:@"chess"])
        return NO;

    NSString *userID;
    NSString *serverURLStr;
    NSURL *serverURL;
    NSDictionary *game;

    userID = [[NSUserDefaults standardUserDefaults] stringForKey:@"userID"];
    serverURLStr = [NSString stringWithFormat:@"http://%@%?%&userid=%@",
        [url host], [url path], [url query], userID ? userID : @""];
    serverURL = [NSURL URLWithString:serverURLStr];
    game = [NSDictionary dictionaryWithContentsOfURL:serverURL];
    if (!game) {
        // Deal with the error
    }
    return YES;
}
```

You're passed the URL and check that it's the right scheme. Depending on the path and query, you can do what's needed in the given situation. In this case, you're simply swapping the URL scheme and, for illustration purposes, getting the user identification from the user defaults and appending it to the server request. Since the application is running at this point, you'd also want to load the game and display it to the user, although that's not shown here.

The original

chess://example.com/game?action=accept&email=king@example.com&game=key link becomes an http://example.com/game?action=accept&email=king@example.com&game=key&userid=123 GET request. When testing this locally during development, you can use localhost:8080 as the host.

Make sure your application can handle any kind of URL string gracefully, because anybody can invoke this application by entering some chess://weird/string/here-type URL in Mobile Safari and have your application execute code.

In the previous code, you're invoking the request on the server by calling `+dictionaryWithContentsOfURL:`. You wouldn't normally want to do that, because you'd want to make the server call asynchronous, and you need more error information than just failure/success. But I wanted to show this cool feature of `NSDictionary`.

The `+dictionaryWithContentsOfURL:` call is a convenient way of sending a server request and getting the result in a handy `NSDictionary` object structure—in just one line of code. The call assumes the server returns a string representation of a property list whose root object is a dictionary.

This brings me to a point I've been itching to make about separating code and data representation on the server.

Separating Data and Representation on the Server

This web service focuses on handling the data in its own, proprietary format and shouldn't have to care about how a client might want the data represented. It certainly shouldn't serve hard-coded HTML or other markup inside its methods.

One nice way of making this separation is by using templates. Google App Engine's webapp module includes the Django's template engine, which provides for a very elegant separation. Take another look at the earlier `get()` method with its embedded HTML code, and compare it with this:

```
def get (self):
    game = Game.get (self.request.get ('key'))
    template_values = {'game': game}
    path = os.path.join (os.path.dirname (__file__), 'game.plist')
    self.response.out.write (template.render (path, template_values))
```

You'll have to import the template module from `google.appengine.ext.webapp` and the `os` module:

```
import os
from google.appengine.ext.webapp import template
```

The `template.render()` call merges the values in the `template_values` dictionary with the `game.plist` template file, which is a slightly modified XML property list file, located in the application directory on the server:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
"http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
  <key>inviter</key>
  <string>{{ game.inviter }}</string>
  <key>invitee</key>
  <string>{{ game.invitee }}</string>
  <key>created</key>
  <date>{{ game.created|date:"Y-m-d\Th:i:s\Z" }}</date>
  <key>moves</key>
  <array>
    {% for move in game.moves %}
      <string>{{ move }}</string>
    {% endfor %}
  </array>
</dict>
</plist>
```

Note how you're accessing the properties of the game object in the `{{ }}` pairs and how you're iterating over the moves with the `{% for move in game.moves %}`, `{% endfor %}` pair.

The output becomes a perfectly valid `.plist` file that you can use to effortlessly create `NSDictionary` objects in your client code, as shown in the `-application:handleOpenURL:` method earlier. This saves you from having to manually parse XML files. I'll show you another example of this soon.

If you want to support other types of output, the client could add a parameter to its request telling the web service which format it wanted the response in. You'd then add a template on the server for each supported format.

In the same spirit, since this is a Cocoa-specific template, it makes perfect sense to make the `{{ game.created|date:"Y-m-d\Th:i:s\Z" }}` date conversion in the template and not in the generic server code. The converted date value becomes a valid date string representation that can be turned into an `NSDate` object at runtime on the client.

Even though the `+dictionaryWithContentsOfURL:` is very handy, I'll show how you can make the server communication from the client a little bit more robust when adding moves to the games.

Making a Move

When you've made your move, it should be sent from your device to the web service and appended to the game. Instead of waiting for the request to be sent and the server to respond, thereby leaving the application unresponsive to the user, we'll handle the request asynchronously. It's a bit more code than the one-liner you used before, but the user will be happy not having to sit with an application that's locked up.

NOTE: As an alternative to asynchronous calls, you could use synchronous calls and encapsulate them in an `NSOperation`, which is executed on its own thread.

In Deep Green I've encapsulated all server communication in a singleton class that manages a request queue (an `NSOperationQueue`) and makes sure they're all dealt with successfully before being removed from the queue. It notifies if something goes wrong, giving the implicated controllers a chance to report any error to the user. It persists the queue to the file system so that it can be re-created in case the user quits the application before the request has been received by the server. And the server code handles duplicate requests gracefully to deal with the situations where the application didn't get the response from the server and therefore thinks it needs to send the request again.

On the Device

In the following code example, I'm doing none of the above but simply providing the skeleton needed to communicate the move from the device to the server. Since this is being done asynchronously, the entry method doesn't return anything. No `BOOL`, no `NSError`:

```
- (void)sendMove:(NSString *)move forKey:(NSString *)key
{
    NSString      *urlStr;
    NSURLRequest  *request;
    NSURLConnection *connection;
```

```

urlStr = [NSString stringWithFormat:
    @"http://example.com/game?action=move&move=%@&key=%@", move, key];
request = [NSURLRequest
    requestWithURL:[NSURL URLWithString:urlStr];
    cachePolicy:NSURLRequestReloadIgnoringLocalAndRemoteCacheData
    timeoutInterval:60.0];
[request setHTTPMethod:@"PUT"];
connection = [[NSURLConnection alloc] initWithRequest:request delegate:self];

if (!connection) {
    // Deal with the error
} else
    receivedData = [[NSMutableData alloc] initWithCapacity:500];
}

```

The move is just a string here, and it still doesn't use any form of security. But the code shows how to initiate a request.

You're composing the URL string containing the move and the game key. The server request is created, ignoring any previously cached data to make sure the request gets sent to the server. You make it a PUT request so that the proper method, `put()`, gets called in the web service code. Being an asynchronous call, `initWithRequest:delegate:` returns immediately, and if the connection was made, you allocate an `NSMutableData` instance variable to receive the confirmation data from the server.

After this method exits, the program execution resumes to the main event loop, and the user will be able to use the application again.

But you're not done yet. You just started the request. You need to prepare for receiving a number of callbacks during the download process. That's the price for "asynchronosity." You'll implement the minimum set of required callback methods: -`connection:didReceiveData:`, -`connectionDidFinishLoading:`, -`connection:didReceiveResponse:`, and -`connection:didFailWithError::`

```

- (void)connection:(NSURLConnection *)connection didReceiveData:(NSData *)data
{
    [receivedData appendData:data];
}

- (void)connectionDidFinishLoading:(NSURLConnection *)connection
{
    [connection release];

    NSDictionary *response;
    NSString *errorString = nil;

    response = [NSPropertyListSerialization
        propertyListFromData:receivedData
        mutabilityOption:NSPropertyListImmutable
        format:NULL
        errorDescription:&errorString];
    [receivedData release];

    if (!response) {
        // Handle the error
        [errorString release];
    }
}

```

```

        return;
    }
    // Do something with response
}

- (void)connection:(NSURLConnection *)connection
didReceiveResponse:(NSURLResponse *)response
{
    [receivedData setLength:0];
}

- (void)connection:(NSURLConnection *)connection didFailWithError:(NSError *)error
{
    [connection release];
    [receivedData release];
    // Notify about the error
}

```

In the `-connection:didReceiveData:` method, you're being passed the most recently downloaded data, which you simply append to whatever is already in the `receivedData` instance variable.

After all data has been downloaded, you receive the `-connectionDidFinishLoading:` call, which gives you a chance to check the server response and clean up. You're copying the `receivedData` into the response variable as an `NSDictionary` using the `NSPropertyListSerialization` class method. And you're releasing the connection and `receivedData` objects. That was the other example I wanted to give on how to very easily tuck a server response into a native Cocoa object structure. It saves you from parsing with `NSScanner`, `NSXMLParser`, or the like.

The `-connection:didReceiveResponse:` call can be received multiple times during a connection and most often occurs at server redirects. Each time this happens, any previously received data should be discarded.

Lastly, `-connection:didFailWithError:` will be called if an error occurs during the connection. You should release the memory related to the connection and notify the relevant controllers, for instance, by using key/value observation (KVO), notifications, or callbacks.

This is a typical pattern for using `NSURLConnection`, and even though it's more code than the `+dictionaryWithContentsOfURL:` one-liner, it's still a modest amount, and it gives you a clean encapsulation of the various outcomes of the connection.

Next I'll show you how to deal with the request on the server.

On the Server

When the move is being sent from the device to the server as just illustrated, you'll want to update the game in the datastore by appending the move to it. You therefore use the HTTP PUT method for the request. In the `main.py` file, a PUT request gets passed to the `put()` method:

```
def put (self):
    game = Game.get (self.request.get ('key'))
    game.moves.append (self.request.get ('move'))
    game.put ()
    template_values = {'game': game, 'success': True}
    path = os.path.join (os.path.dirname (__file__), 'move.plist')
    self.response.out.write (template.render (path, template_values))
```

In this method, you append the move to the game's list of moves and return another property list to the client. This is the list that ends up as an NSDictionary in the response variable in the previous listing using the `+propertyListFromData:mutabilityOption:format:errorDescription:` call.

For the sake of completeness, the `move.plist` template file on the server looks like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
"http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
  <key>key</key>
  <string>{{ game.key }}</string>
  <key>success</key>
  {% if success %}
  <true/>
  {% else %}
  <false/>
  {% endif %}
</dict>
</plist>
```

Note how you come from the native Python success boolean to the property list `<true/>` or `<false/>` version. As per the data/representation separation, the `put()` method shouldn't have to deal with native Cocoa data types, because the next request could possibly come from a non-Cocoa-aware client. The Cocoa-specific Django templates deal with the conversion from model to client. In Model/View/Controller lingo, you can regard the template as a controller.

I have to repeat myself by saying that you shouldn't just pass a game key to the server like in the previous examples. It would be way too easy for a hacker to guess other players' game keys and add arbitrary moves to them. As a minimum, you should provide your e-mail address as well and have the server code look up the game on both fields, assuming it would be quite hard for somebody to guess both the game key and the player in turn's e-mail address (except for your opponent, that is). But even better, you should authenticate the user and communicate via a secure, encrypted connection. I'll leave that as an exercise for you, once you have all the other pieces in place.

Summary

In this chapter, you took the step from a clean slate to a fully functional client/server solution ready to being deployed on Google's industry-leading platform. Admittedly, it's a very simplified solution, but it contains the components and design principles you'd use in a polished App Store application that's available to millions of iPhone OS users.

What was most surprising to me when I first got the core functionality up and running is the small amount of code that's needed to achieve even large and complex tasks. This is mainly because of the richness of the Python scripting language and the way Google has embraced it in its App Engine service. Coupled with the solid Cocoa framework, this chapter dealt with some of the most exciting and powerful technologies available.

It sure is an exciting time to be an iPhone OS user—let alone an iPhone OS developer.

Tom Harrington



Company: Atomic Bird, LLC

Location: Colorado Springs, CO

Former Life As a Developer: Before switching to iPhone development I spent several years as an independent Mac OS X software developer, which followed previous experience developing Linux and embedded system software. My Mac experience goes back to the earliest days of Mac OS X, before Xcode was called Xcode and before PowerBooks started being made of metal. My experience leads me to work “close to the metal”, and even though I was writing Objective-C and using Cocoa on Mac OS X, my applications all ended up being various background services and utilities. Working on the iPhone I’ve expanded my repertoire to cover user interface design as well.

Life as an iPhone Developer: I’ve mainly worked as an iPhone contractor, developing applications for clients. The following are currently in the web store or awaiting approval:

- **AirMe**, a camera app that uploads to numerous photo-sharing sites.



- **The iPhone app for MSN’s photoWALL web site.**
- **Mocapay**, which takes the place of store gift cards, enabling users to make purchases with their phone instead of with a separate card.



On my own I wrote an iPhone app for KRCC, a public radio station in Colorado Springs, CO. This app plays the station's audio streams and includes an auto-updating program schedule, and is the basis for the code I present in this chapter.



***What's in This Chapter:** The chapter describes my ultimately successful quest to write an iPhone app that could play streaming internet audio with a custom user interface. Along the way I review various approaches to audio playback, including playing system sounds, using AVAudioPlayer and Audio Queue Services. Finally I settle on Audio File Stream Services as the best solution. Sample code is included for each technique. Along the way are diversions into useful related details like how to download data from a web site.*

Key Technologies:

- **Core Audio**
- **Audio File Stream Services**
- **Audio Queue Services**
- **NSURLConnection**

Audio Streaming: An Exploration into Core Audio

I've been a Mac OS X developer for several years. But I'm always looking for something new and interesting, so when the iPhone SDK was announced in March 2008, I jumped at the opportunity.

And so did everyone else, or so it seemed.

I was also on the lookout for a new way of doing business. For the previous five years I had worked as an independent software developer, writing Mac software that I sold directly to end users. Back in March 2008 I wasn't sure how this would play out in the iPhone world, so when I saw a local company looking for an iPhone contractor, I contacted them immediately. With the similarities between iPhone and Mac development, who could be better placed than an experienced Mac developer? Moving on Internet time, I was on the job within a couple of days.

And that's when I really started to think that everyone else in the world had also jumped at the opportunity. To deploy software to an iPhone—even your own—you need to have a paid-up membership in Apple's iPhone developer program. Apple appeared to be overwhelmed with applicants. After a few weeks, my membership had still not come through, and my client decided there wasn't much point continuing until we could test their app on a real phone.

I wanted to continue exploring the platform, though, so I started looking for an interesting project I could pursue while waiting for Apple's gears to turn.

Hey, I Could Write an App to Play Music

Simultaneously, my music library continued its growth.

I'm the kind of person whose iTunes library is so big that it requires its own external hard drive. I rely on sophisticated schemes involving multiple iTunes smart playlists just to get a reasonable subset of iTunes music onto my iPhone. I use Last.fm, Pandora, and any other interesting online music site I can find.

I'm that guy, the one who won prizes several years in a row at Apple's Worldwide Developer Conference for recognizing songs played during the "Stump the Experts" event.

And so it was that my attention turned to a music search web site known as SeeqPod. It had a search engine that would find playable tracks available on the Internet and play them in a web browser.

It also had a simple REST-based API that third-party apps could use to make queries that returned URLs of tracks to play. I had found my project.

Talking to SeeqPod from an application would involve sending requests to its server, parsing the XML response, presenting these results to the user, and playing audio files found at URLs in the response.

I had covered all that in previous projects—except for the part about actually playing the audio.

The URLs I would be dealing with mostly pointed to MP3s. Though I had more than my share of MP3s in my iTunes library and though I can read music, the technical aspects of audio encoding were mostly a black box to me. I'd have to figure that out to make this idea work.

MPMoviePlayerController: Hey, This Is Easy! Right?

It seemed that this capability must be included in the extensive APIs that made up the iPhone SDK. I had heard talk of something called Core Audio, but to my untrained eyes, it gave the impression of being a dark, complex system, probably overkill for my fairly simple needs. I just wanted to play audio files; it's not like I was planning to write the next GarageBand, so surely there must be a simpler way.

And there is, sort of. Looking through the documentation, I came across the `MPMoviePlayerController` class. It's designed to be a very simple class to play movies at a given URL. It's extremely simple to use.

To use `MPMoviePlayer` controller, all you need to do is give it a URL and tell it to start playing. It sends out notifications at times while playing, and one, `MPMoviePlayerPlaybackDidFinishNotification`, is useful to avoid leaking the `MPMoviePlayerController` object. This is almost the entire API, leaving out only minor

details such as setting the background color. Listing 3-1 shows a simple example (the only kind for this class, really).

Listing 3-1. Using *MPMoviePlayerController*

```
- (IBAction)playMovie:(id)sender
{
    NSURL *movieURL = ...;
    // Create the movie player object
    MPMoviePlayerController *theMovie = [[MPMoviePlayerController alloc] ➤
        initWithContentURL:movieURL];
    // Listen for notifications that the player has finished
    [[NSNotificationCenter defaultCenter] addObserver:self ➤
        selector:@selector(movieFinished:) ➤
        name:MPMoviePlayerPlaybackDidFinishNotification ➤
        object:theMovie];
    // Start playing
    [theMovie play];
}

- (void)movieFinished:(NSNotification *)note
{
    MPMoviePlayerController *theMovie = [note object];
    // Remove self from future notifications
    [[NSNotificationCenter defaultCenter] removeObserver:self ➤
        name:MPMoviePlayerPlaybackDidFinishNotification object:theMovie];
    // Release the player to avoid leaking memory
    [theMovie release];
}
```

The *MPMoviePlayerContoller* takes care of putting itself on the screen, downloading the movie, buffering it as needed, and removing itself from the screen once playback finishes. If you've ever used the iPhone's YouTube application, *MPMoviePlayerController* provides the same user experience when playing a video, including the playback and volume controls.

Movies generally include sound, so I wondered if it would play a music-only file. And it does. It plays them in a simple but rather dull full-screen view, as shown in Figure 3-1, which is not exactly ideal. But it's so simple to use that it's tempting to make compromises with it in order to get off easy with the audio playback.



Figure 3-1. *MPMoviePlayerController's* user interface when playing an audio-only file.

At this point, I felt like I was on easy street, because I had knocked out the only part of the app that looked like it might be difficult and the day wasn't even half over. I proceeded to implement a basic search UI and a back end to send search requests and parse the results. And it was good. For a little while, anyway.

Before long, the app was crashing, badly enough that I needed to force-quit the iPhone Simulator (because I was still not part of the iPhone developer program). I started my detective work to find out why. I soon found that certain URLs would reliably crash the application when `MPMoviePlayer` attempted to play them. In most cases, I'd expect programmer error, so I carefully reviewed what I was doing. I found nothing that I could be sure was wrong. I tried the URLs in Safari and at the command line with `curl`, but that seemed OK. In fact, if I downloaded the audio file to my Mac and started `MPMoviePlayerController` with a `file://` URL, everything seemed fine.

I looked in the phone's system console to see whether there were any clues. It was pretty unequivocal:

```
Apr 11 15:24:21 atomicbird SimpleMediaPlayerAudio[15153]: -[AVController ➡  
failPlayback:reason:notifyClient:]: item with path [omitted] failed to ➡  
open with err -12784  
Apr 11 15:24:21 atomicbird SimpleMediaPlayer[15153]: ERROR!!! Please file a Radar!!!
```

I'm always hesitant to blame my app crashes on the underlying frameworks, because they're almost always my fault. But if the framework specifically asks me to file a bug against it, who am I to argue? So, I filed a bug and waited. In the meantime, Apple apparently found my application behind a file cabinet somewhere and admitted me to the iPhone developer program.

Several weeks and a couple of iPhone SDK betas later, the bug was reported as fixed. I rejoiced and went back to my audio experimentation. I soon found that, although `MPMoviePlayer` seemed more reliable than in the past, it still exhibited the same symptoms at times—not always, and not even most of the time, but it still was way too often for me to keep using it. And besides, I wasn't thrilled with the UI.

Finding a Better Approach

Clearly something else was needed. I had initially latched on to `MPMoviePlayerController` because it was so easy to use that I hardly needed to think. Other options existed, and in order to evaluate them I had to clarify my needs:

- I needed a solution that would play audio files as they downloaded. I didn't want to make the user sit and wait while I downloaded an entire MP3 to a file before I started playing it. `MPMoviePlayerController` would start playing as soon as it had enough bytes buffered, and I wouldn't consider anything with unnecessary extra delays.
- The solution had to handle a variety of audio formats, including those likely to be found by SeeqPod searches.

- Ideally it should be easy to use, because that's how it's supposed to be with Cocoa Touch, right? This was more of a goal than a hard requirement, because I wasn't sure where the search might lead.

Browsing the documentation, I found several likely candidates.

The System-Sound Way

The first candidate was `AudioServicesPlaySystemSound()`, which I came across in Apple's *Audio & Video Coding How-To's* documentation. It said right there that this function is intended for only sounds of 30 seconds or less, but it gave me the possibly mistaken impression this was not a hard limit. It's almost as easy to use as `MPMoviePlayerController`, as Listing 3-2 shows.

Listing 3-2. Using `AudioServicesPlaySystemSound`

```
// Set up a system sound object
NSString *url = ... // URL pointing to an audio file
SystemSoundID mySoundID;
AudioServicesCreateSystemSoundID((CFURLRef)url, &mySoundID);

// Play the system sound object. This function call will return immediately.
AudioServicesPlayAlertSound(mySoundID);

// Clean up the system sound object (do this later, for example in -dealloc).
AudioServicesDisposeSystemSoundID(mySoundID);
```

Aside from its simplicity, though, it didn't serve my needs. It's certainly simple, but it's limited to loading sounds from file URLs, which meant that I'd have to download a full track before playing it. I also found that it was limited to playing only WAV, AIFF, and CAF files, and it wouldn't decode MP3s or other compressed formats. I didn't bother finding out whether the 30-second limit was real or just a suggestion, because I was already looking for an alternative.

AVAudioPlayer: The Not-Available-in-Beta Way

If you're looking through the iPhone SDK documentation today, the `AVAudioPlayer` class looks like a potential candidate for my needs. It drops the 30-second limit of system sounds, and it'll play a much wider variety of formats.

I didn't investigate `AVAudioPlayer`, though, not because of any technical limitations but because of a notice you might miss at the top of its class documentation—the part that says “Available in iPhone OS 2.2 and later.” By this time I was working on a late beta of iPhone OS 2.0. Developers working on jail-broken phones had access to `AVAudioPlayer` and much more from APIs that were present but undocumented and unsupported. I had ambitions of getting into the App Store eventually, though, and Apple wouldn't have been very likely to let me get away with it. Apps have been rejected from the store for much less.

In the interest of completeness, though, Listing 3-3 demonstrates a simple use of `AVAudioPlayer`. In this case, there's a lot more to the API than the listing shows, but the basics of setting up the player and getting it going are quite similar to previous examples. If `AVAudioPlayer` meets your needs, you'll find it offers a rich API for control and monitoring of audio playback.

Listing 3-3. Simple `AVAudioPlayer` Example

```
- (IBAction)play:(id)sender
{
    NSURL *url = ... // Valid file:// URL
    NSError *error = nil;
    // Create the player and set its delegate to self
    audioPlayer = [[AVAudioPlayer alloc] initWithContentsOfURL:url error:&error];
    audioPlayer.delegate = self;
    // Check that the player was created before playing
    if ((audioPlayer == nil) || (error != nil)) {
        [audioPlayer release];
    } else {
        [audioPlayer play];
    }
}

// AVAudioPlayerDelegate method
- (void)audioPlayerDidFinishPlaying:(AVAudioPlayer *)player successfully:(BOOL)flag
{
    // Don't leak memory
    [audioPlayer release];
}
```

Just to be sure I hadn't missed an easy fix, I tried using the code from Listing 3-3 with an HTTP URL. It turns out they're not kidding about using only file URLs. The `initWithContentsOfURL:error:` method returned `nil`, and the error parameter indicated initialization had failed with a "file not found" error.

As a result, `AVAudioPlayer` didn't meet my needs even if it had been available when I started work on this project. It gets tantalizingly close, covering the audio formats I needed and offering detailed control over audio playback. The lack of any way to play downloaded data before the data was complete makes it unsuitable for my situation.

As an inveterate hacker, I tried to trick `AVAudioPlayer` into doing what I needed. In addition to initialization from a URL, `AVAudioPlayer` can be initialized from an `NSData` object. I decided to try the following experiment:

1. Start with a URL pointing to an MP3, and begin downloading it.
2. As bytes arrive, append them to an `NSMutableData` object (which is a subclass of `NSData` and should therefore be acceptable to `AVAudioPlayer`).

3. When enough bytes have been downloaded, create an `AVAudioPlayer` and start it playing. If “enough bytes” means I’ve buffered enough data, the player should be able to continue playing bytes that were available even as I was simultaneously adding new bytes at the end of the `NSMutableData` object.

The drawback to this scheme would be that the entire audio file would end up being stored in memory by the time downloading was complete. But my gut feel was that the kind of songs I was likely to be playing would not be so large as to make this a problem. I’d end up having an `NSMutableData` object containing a few megabytes of data by the time I was done, but when I was done, I’d just release it and move on.

Listing 3-4 outlines the scheme.

Listing 3-4. Attempting to Fool `AVAudioPlayer`

```
// In the class's interface
NSURLConnection *audioConnection;
NSMutableData *audioData;
AVAudioPlayer *audioPlayer;

// In the class's implementation
// See if we can feed NSURLConnection data into an AVAudioPlayer
- (IBAction)playData:(id)sender
{
    NSURL *url = ... // an HTTP URL pointing to an MP3
    audioData = [[NSMutableData alloc] init];
    audioConnection = [[NSURLConnection alloc]
        initWithRequest:[NSURLRequest requestWithURL:url]
        delegate:self];
}

- (void)connection:(NSURLConnection *)connection didReceiveData:(NSData *)data
{
    // Add the new data to the audioData object
    [audioData appendData:data];
    if (audioPlayer == nil) {
        NSError *error = nil;
        // Attempt to create the player with the available data
        audioPlayer = [[AVAudioPlayer alloc] initWithData:audioData
            error:&error];
        if ((audioPlayer == nil) || (error != nil)) {
            NSLog(@"Error creating player: %@", error);
            [audioPlayer release];
        } else {
            [audioPlayer play];
        }
    }
}
```

It didn't work out as expected, though. The network connection received 1,440 bytes at a time, so initially the audio player couldn't be created because there weren't enough bytes in the data object. Buffering wouldn't make the situation any better though, because `AVAudioPlayer` reads audio file length from the audio file header and then crashes if it can't read to the end. The end result is that the only buffer that's big enough is one that's at least as big as the file being played, which effectively means that playing during download is impossible.

DOWNLOADING DATA WITH `NSURLConnection`

I've written several iPhone applications to date, and every one of them has needed to connect to a web site at some point, either to upload data or to download data.

There's more than one way to download data from a URL on the iPhone, but I've found `NSURLConnection` to hit the sweet spot between ease of use and power. It offers two basic schemes, synchronous and asynchronous downloading.

The synchronous approach is easy but, being synchronous, blocks execution until the connection is finished. That makes it suitable only for use on a background thread, because blocking the main thread means locking up your user interface. That's not a good idea even on a fast connection. If the iPhone is in an area with EDGE coverage, there's a good chance of the connection taking so long that users will think the app has crashed.

The asynchronous approach solves this by dealing with the network in the background and notifying you of progress through delegate methods. In effect, it gives you the advantage of using a background thread but does so transparently. You start it up and then go about whatever other business needs taking care of, and it'll call you back when anything interesting happens. Listing 3-5 shows how to begin the process.

Listing 3-5. *Starting a Download with `NSURLConnection`*

```
// In the class interface
NSMutableData *receivedData;
NSURLConnection *myConnection;

// In the class implementation
- (void)startDownload
{
    NSURL *url = ...; // Valid URL
    NSURLRequest *request = [NSURLRequest requestWithURL:url];
    // Create a data object to hold data returned by the connection
    receivedData = [[NSMutableData alloc] init];
    // Create the connection, which will start loading immediately
    myConnection= [[NSURLConnection alloc] ➡
        initWithRequest:request ➡
        delegate:self];
}
```

Upload data, if any, would be attached to the `NSURLRequest` object, which is a lot more flexible than I'm demonstrating here. It has a mutable subclass, aptly named `NSMutableURLRequest`, which has methods for setting the HTTP method, request body, and header fields. By using `NSMutableURLRequest`, you can extend this sample code to work with a wide variety of web services.

The minimal set of delegate methods covers those that receive data from the connection and those that are called when the connection has finished or failed. Listing 3-6 shows a simple implementation of these methods. See the `NSURLConnection` class documentation for information on others.

Listing 3-6. Delegate Methods for `NSURLConnection`

```
// NSURLConnection delegate method
- (void)connection:(NSURLConnection *)connection didReceiveData:(NSData *)data
{
    // This method may be called repeatedly
    [receivedData appendData:data];
}

// NSURLConnection delegate method
- (void)connectionDidFinishLoading:(NSURLConnection *)connection
{
    // This method will be called once, if no errors occurred.
    // All data has been received and the connection is closed.
    // Call any methods that process the received data (e.g. XML parsing) here.
    [receivedData release];
    receivedData = nil;
    [myConnection release];
    myConnection = nil;
}

// NSURLConnection delegate method
- (void)connection:(NSURLConnection *)connection
    didFailWithError:(NSError *)error
{
    // This method is called once, if an error occurred.
    // The "error" argument contains information about the error.
    [receivedData release];
    receivedData = nil;
    [myConnection release];
    myConnection = nil;
}
```

In addition, Listing 3-7 shows an additional delegate method that can be used to retrieve the HTTP status code, if the URL scheme was HTTP. The response argument is declared as an `NSURLResponse` object, but if the protocol is HTTP, it will actually be a subclass, `NSHTTPURLResponse`, which contains the HTTP status. You might prefer to allocate the `NSMutableData` object in this method instead of when starting the connection and do so only if the status code indicates HTTP success (200). An empty `NSMutableData` object takes up so little space that it's more a matter of style than anything else.

Listing 3-7. Getting the HTTP Status

```
// NSURLConnection delegate method
- (void)connection:(NSURLConnection *)connection ➡
    didReceiveResponse:(NSURLResponse *)response
{
    [receivedData setLength:0];
    // See if it's an NSHTTPURLResponse and typecast it if it is.
    if ([response isKindOfClass:[NSHTTPURLResponse class]]) {
        NSHTTPURLResponse *httpResponse = (NSHTTPURLResponse *)response;
```

```
        NSLog(@"HTTP Status: %d", [httpResponse statusCode]);  
    }  
}
```

Doing It the Cowboy Way with Core Audio

Using `MPMoviePlayer` or one of the other APIs I had tried would have been the easy way. But as the legendary Ranger Doug would say, “It wouldn’t be the cowboy way.” Clearly, it was time for Core Audio. All along it had been lurking in the SDK, an API so powerful I dreaded I might hurt myself trying to use it.

This was going to be a learning experience, and I hoped it wouldn’t be the kind where you learn not to do something again after inciting a disaster. Not really knowing where to begin, I opened up Xcode’s documentation viewer and typed **Core Audio** into the search field.

The results were kind of overwhelming at first. I tried to start with the simple stuff, but before I had finished the first page of “Core Audio Overview,” it was already talking about setting up a Core Audio–based recording studio in a diagram containing about 15 different blocks, none of which I had the first clue about. After spending some time exploring the documentation, skipping stuff I didn’t understand or that seemed not relevant to my project, I decided that the most likely option was something called Audio Queue Services. The documentation for this API described a scheme for playing audio files via Core Audio, so I started following along, implementing my own code to parallel the documented scheme.

Getting Halfway There: Audio Queue Services

Reading through the documentation and the sample code, I gradually realized that the Audio Queue Services approach was designed around reading audio from a file, as with previous methods I’d tried. I pressed on anyway. The sample code didn’t seem to grab the entire audio file at once—instead, it used a collection of buffers that would be successively filled with part of an audio file and then played. By cycling through the buffers, the entire file would gradually be played. That sounded like a promising approach, since I expected that once I knew what I was doing, I could modify the code to take data from the network instead of from a file.

The general approach is a repeating cycle in which you read a chunk of audio data from a file into a buffer and add it to an audio queue. The audio queue plays the buffer and then calls a callback function, which repeats the process until no more file data remains. Figure 3-2 illustrates this cycle.

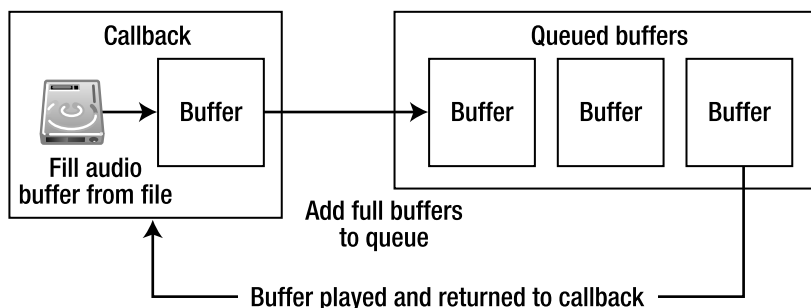


Figure 3-2. *Audio Queue Services playback cycle*

Apple provides a sample application that illustrates this approach, which is called *SpeakHere*.

A WORD ON PROGRAMMING LANGUAGES

In the course of discussions with other iPhone developers, I've met a surprising number of people who don't simply prefer to stick to Objective-C for iPhone development but who regard pure C APIs with something approaching fear and dread. If that's not you, fantastic, but it's an attitude that I've encountered much more than I would have expected.

For some people, this feeling goes so far that they'd prefer not to do something if using C is the only way. I'm not sure what the source of this feeling is, but if you're going to use Core Audio, you'll need to recognize one very important detail: Core Audio is entirely a C-based API. No Objective-C is involved.

Of course, Objective-C is a proper superset of C, so you already know C at least as well as you know Objective-C. You will, of course, have to use C-style arrays instead of `NSArray`, and you'll encounter `malloc/free`-style memory management. Not to mention potentially using pointers in new and unfamiliar ways. The callback function shown in Figure 3-2 is a C function that you would implement. But you can and should use Objective-C for most other aspects of your application, and Core Audio code will integrate well.

If this sounds dangerous, error prone, or just distasteful, relax. You can do this. There is no monster hiding under this bed.

BEWARE THE SIMULATOR

If you've done any significant amount of iPhone development work, you've probably run across cases where the iPhone Simulator differs from working on a real iPhone or iPod touch. Often it's something obvious like the lack of a camera or the fact that the simulator's Core Location data always shows the current location to be at Apple's headquarters.

With Core Audio, it's less obvious but no less important. Developing Core Audio code on the iPhone Simulator is a recipe for frustration and confusion, because although code often fails to work as expected, it doesn't fail in a manner that's immediately obvious as being a simulator issue. You may find, for example, that a key Core Audio function call never returns, for no readily apparent reason, and that while

it's busy doing who-knows-what it also draws 90 percent or more of your Mac's CPU. Application bugs are often your own fault, so it's natural to try to analyze your code to see where you went wrong.

But then you try the same code on a phone, and it just works.

I've filed bugs with Apple about this, but as of this writing, they're unresolved in any released version of the iPhone SDK. With any luck the situation will improve in a future release. Until then, if you choose to work in the simulator and you find that your Core Audio code is not working quite as expected, try working on a real device first before spending too much time trying to find the bug.

Listing 3-8 shows a class declaration for a class that uses Audio Queue Services to play audio files. A number of instance variables are necessary to track audio format and maintain the audio queue. Property declarations are a matter of style, but some kind of setter and getter methods will be useful later.

Listing 3-8. Class Declaration for a Class Using Audio Queue Services

```
@interface SimpleAQPlayViewController : UIViewController {
    AudioStreamBasicDescription    mDataFormat;
    AudioQueueRef                  mQueue;
    AudioQueueBufferRef            mBuffers[kNumberBuffers];
    AudioFileID                    mAudioFile;
    UInt32                         bufferSize;
    SInt64                         mCurrentPacket;
    UInt32                         mNumPacketsToRead;
    AudioStreamPacketDescription   *mPacketDescs;
    BOOL                           mIsRunning;
}

@property (readwrite) UInt32          mNumPacketsToRead;
@property (readwrite) AudioFileID     mAudioFile;
@property (readwrite) AudioStreamPacketDescription *mPacketDescs;
@property (readwrite) SInt64          mCurrentPacket;
@property (readwrite) AudioQueueRef    mQueue;
@property (readwrite) BOOL             mIsRunning;

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

The play method to start audio playback is designed to be the target of a button or other user interface item. Listing 3-9 shows its implementation. There are a lot of steps, but it's not as complicated as it might look. The following are the key details to be aware of:

- Core Audio has its own API for opening, reading, and closing files. This starts with `AudioFileOpenURL`. The third argument to this call is a hint about the file type. Usually, `AudioFileOpenURL` can work out the file type for itself, so you pass 0 to indicate you're not giving it any hints. If the code was intended to use a specific file type, you could pass a format-specific value here, such as `kAudioFileMP3Type` or `kAudioFileAIFType`. Other Core Audio file management calls will appear later.

- `AudioQueueNewOutput()` creates the playback queue. The queue doesn't read the file directly. Instead, the second argument, `AQOutputCallback`, is the callback function shown in Figure 3-2. It will read data from the file into buffers and add those buffers to the queue.
- The third argument to `AudioQueueNewOutput()` is an argument to the callback function, which can be any data you find useful in that function. In this case, I'm passing `self` so that the callback function will be able to make Objective-C method calls on my audio playback class.
- You need to set the buffer size used when reading data from the file. Choosing a buffer size is a balancing act. Buffers that are too small will mean the callback function gets more calls, which means more time spent filling buffers. In an extreme case, the callback could be called so frequently that audio playback would stutter. At the other extreme, larger buffers require more memory, and memory is always at a premium on the iPhone. In this case, I used a utility method, `DeriveBufferSize()`, to find the right buffer size for a certain amount of time, as determined by the audio format. It's also reasonable to just use a fixed buffer size, if you expect to deal only with certain encodings. Deriving sizes on the fly allows more flexibility, but greater flexibility is not always necessary.
- Audio data needs to be handled slightly differently depending on whether it is encoded with a constant bit rate (CBR) or a variable bit rate (VBR). VBR packets may vary in size, so Core Audio uses packet descriptors to track the size and other information about individual packets. CBR packets are all the same, so this is not necessary. It's important to note that Core Audio will treat compressed formats such as MP3 as VBR data even if they are encoded at a constant bit rate. Uncompressed formats like AIFF will be treated as CBR data.
- The `play` method calls the callback function directly before starting playback to prime the queue with its initial set of buffers.
- Playback begins with the call to `AudioQueueStart()`. It continues only so long as the application's run loop is executing. If you play audio on the main thread of an iPhone app, this happens automatically. Background threads don't automatically have a run loop, though, so if you use a background thread, you'd need to create your own run loop for playback to continue beyond the initial priming of the queue.

Listing 3-9. Starting Audio Playback Using Audio Queue Services

```
- (IBAction)play:(id)sender
{
    OSStatus result;

    // Open the audio file from an existing NSString path
    NSURL *sndFileURL = [NSURL fileURLWithPath:path];
```

```

AudioFileOpenURL((CFURLRef)sndFileURL, kAudioFileReadPermission, 0, &mAudioFile);

// Get the audio format
UInt32 dataFormatSize = sizeof(mDataFormat);
AudioFileGetProperty(mAudioFile, kAudioFilePropertyDataFormat,
    &dataFormatSize, &mDataFormat);

// Create the playback queue
AudioQueueNewOutput(&mDataFormat, AQOutputCallback, self,
    CFRunLoopGetCurrent(), kCFRunLoopCommonModes, 0, &mQueue);

// Get buffer size, number of packets to read
UInt32 maxPacketSize;
UInt32 propertySize = sizeof (maxPacketSize);
// Get the theoretical max packet size without scanning the entire file
AudioFileGetProperty(mAudioFile, kAudioFilePropertyPacketSizeUpperBound,
    &propertySize, &maxPacketSize);
// Get sizes for up to 0.5 seconds of audio
DeriveBufferSize(mDataFormat, maxPacketSize, 0.5, &bufferByteSize,
    &mNumPacketsToRead);

// Allocate packet descriptions array
bool isFormatVBR = (mDataFormat.mBytesPerPacket == 0 || ➡
    mDataFormat.mFramesPerPacket == 0);
if (isFormatVBR) {
    mPacketDescs = (AudioStreamPacketDescription*) ➡
        malloc (mNumPacketsToRead * sizeof (AudioStreamPacketDescription));
} else {
    mPacketDescs = NULL;
}

// Get magic cookie (for compressed formats like MPEG 4 AAC)
UInt32 cookieSize = sizeof(UInt32);
OSStatuscouldNotGetProperty = AudioFileGetPropertyInfo(mAudioFile,
    kAudioFilePropertyMagicCookieData, &cookieSize, NULL);
if ((couldNotGetProperty == noErr)&& cookieSize) {
    char* magicCookie = (char *) malloc (cookieSize);
    AudioFileGetProperty(mAudioFile, kAudioFilePropertyMagicCookieData,
        &cookieSize, magicCookie);
    AudioQueueSetProperty(mQueue, kAudioQueueProperty_MagicCookie,
        magicCookie, cookieSize);
    free(magicCookie);
}

// Allocate and prime audio queue buffers
mCurrentPacket = 0;
for (int i = 0; i < kNumberBuffers; ++i) {
    AudioQueueAllocateBuffer(mQueue, bufferByteSize, &mBuffers[i]);
    AQOutputCallback(self, mQueue, mBuffers[i]);
}

// Start and run queue
mIsRunning = true;

AudioQueueStart(mQueue, NULL);
}

```

The function in Listing 3-10 is the callback function shown in Figure 3-2 and attached to the audio queue in Listing 3-9. The purpose of this function is to read data from the audio file into a buffer and then add it to the audio queue. Or if no more data is available (as indicated by the value returned by `AudioFileReadPackets` in its `numPackets` argument), arrange to stop playback. Data is read using the second part of Core Audio's file API, `AudioFileReadPackets()`.

Listing 3-10. Audio Queue Services Callback Function

```
void AQOutputCallback(void *userData, AudioQueueRef inAQ,
    AudioQueueBufferRef inBuffer) {

    SimpleAQPlayViewController *self = (SimpleAQPlayViewController *)userData;
    UInt32 numBytesReadFromFile;
    UInt32 numPackets = self.mNumPacketsToRead;

    // Read up to numPackets packets from the file.
    AudioFileReadPackets (self.mAudioFile, false, &numBytesReadFromFile,
        self.mPacketDescs, self.mCurrentPacket,
        &numPackets, inBuffer->mAudioData);

    if (numPackets > 0) {
        // Set the byte count to the number of bytes actually read from the file.
        inBuffer->mAudioDataByteSize = numBytesReadFromFile;
        // Add the buffer to the audio queue.
        AudioQueueEnqueueBuffer(self.mQueue, inBuffer,
            (self.mPacketDescs ? numPackets : 0), self.mPacketDescs);
        self.mCurrentPacket += numPackets;
    } else {
        // If no packets were read, stop the queue.
        [self stopPlaying];
    }
}
```

The callback function also illustrates a way of bridging the gap back to the audio playback class. Since you previously called `AudioQueueNewOutput` with `self` as the callback function argument, a reference to the object is passed to the callback as `userData`. By typecasting this to a pointer to the audio playback class, it's possible to make Objective-C method calls to that object from the C callback function. This is why it was important to have getter and setter methods (even synthesized ones) for some of the instance variables declared in Listing 3-8. In this callback, you need to be able to access these variables from outside the scope of the class interface.

Listing 3-11 shows the utility function used to determine buffer size in the `play` method.

Listing 3-11. Determining an Optimum Buffer Size

```
void DeriveBufferSize(AudioStreamBasicDescription ASBDesc,
    UInt32 maxPacketSize,
    Float64 seconds,
    UInt32 *outBufferSize,
    UInt32 *outNumPacketsToRead) {

    // Set limits on buffer size. Max size = 128kB min size = 16kB
    static const int maxBufferSize = 0x20000;
    static const int minBufferSize = 0x4000;
```

```

    if (ASBDesc.mFramesPerPacket != 0) {
        Float64 numPacketsForTime = ASBDesc.mSampleRate / ➡
            ASBDesc.mFramesPerPacket * seconds;
        *outBufferSize = numPacketsForTime * maxPacketSize;
    } else {
        *outBufferSize = MAX(maxBufferSize, maxPacketSize);
    }

    if (*outBufferSize > maxBufferSize && *outBufferSize > maxPacketSize)
        *outBufferSize = maxBufferSize;
    else {
        if (*outBufferSize < minBufferSize) {
            *outBufferSize = minBufferSize;
        }
    }
    *outNumPacketsToRead = *outBufferSize / maxPacketSize;
}

```

Finally, Listing 3-12 shows the stop function called by the callback function once all audio data has been read. This method gets called from the callback function as soon as `AudioFileReadPackets` indicates that no more audio data is available. When that happens, there may still be unplayed buffers in the queue, so you don't want to stop playback immediately. Fortunately, `AudioQueueStop()` deals with this—passing `false` as the second argument indicates that playback should not stop until all buffers have been processed.

You do know that you've read all the available audio data, though, so you can call `AudioFileClose()`, the last part of Core Audio's file API, to close the file.

Listing 3-12. Stopping Audio Queue Services Playback

```

- (void)stopPlaying
{
    AudioQueueStop(self.mQueue, false);
    self.mIsRunning = false;

    // All data has been read from the file, so close it
    AudioFileClose (mAudioFile);
}

```

This was great. I now had code that would play any audio file supported by the iPhone with not a lot of code. It wouldn't take over the user interface like `MPMoviePlayerController`. And it looked like I was on the right track for streaming. True, I was still tied to reading audio data from files, but I was only doing that as a way to get audio buffers that I could feed into a queue. If I could arrange to get those buffers from a network connection, I'd have achieved my goals for the project.

Getting the Rest of the Way There: Audio File Stream Services

Taking raw data from a network connection and getting it into something playable by an audio queue took me back to the iPhone SDK documentation. However, the documentation was somewhat opaque, at least to me. I'm sure it's great if you know a thing or two about digital audio encoding, but if you've been following along from the start, you know that I was not such a person. Audio File Stream Services is a Core Audio API that's designed for the case where you want to play audio but don't have the entire audio file available. I'd need to learn to use it.

I had what looked like a better option than the documentation, though. I attended WWDC 2007, and I had the session videos. Looking through them, I found that there had been a session covering exactly this topic—session 404, “Queueing, Streaming, and Extending Core Audio.” I eagerly watched it and found that the API actually started to make sense.

Some details were left out, though, with the expectation that attendees would refer to the session's sample code to fill in the details. I didn't have the sample code, and it was not available on Apple's web site any more. To overcome this, I started a two-pronged approach. First, from the session video, I knew which parts of the API I needed to puzzle out into working code, so I went back to the documentation with renewed focus.

Second, I started a campaign of asking every developer I knew if they had the sample code and would they please send me a copy if they did. Between e-mail, IRC, Twitter, and face-to-face discussions, I eventually got my request passed along to a friend of a friend who had the code. I now had everything I needed to complete the application.

Streaming audio is somewhat more complicated than playing a complete audio file, because the lack of file data translates into a lack of information about what you're trying to play. You can't create an audio queue until you know the exact audio format, and you can't know the audio format until you have enough of the file to read its properties. But how do you know when that happens? Downloading an arbitrary number of bytes and hoping for the best is not a good solution.

The solution involves two callback functions instead of the one you saw earlier. One of them is called the *property listener callback*, because the audio stream calls it whenever new property information is available. Properties can include things such as the audio data format and the packet size, but the most useful one for streaming is the “ready to produce packets” property, a flag that indicates all metadata has been read and that audio data is available.

That's when the second callback—the audio data callback—comes into play. The audio stream passes audio packets to this callback, which bundles them up into buffers and adds them to the audio queue.

The `AudioFileStreamParseBytes()` function is responsible for taking incoming raw data and making sense of it, calling the two callback functions as necessary. The general flow

then is to read data, pass it to the parser, and receive audio data in your callback functions. The callback functions will almost always be called multiple times as data is received. Figure 3-3 illustrates how this operates once playback has started.

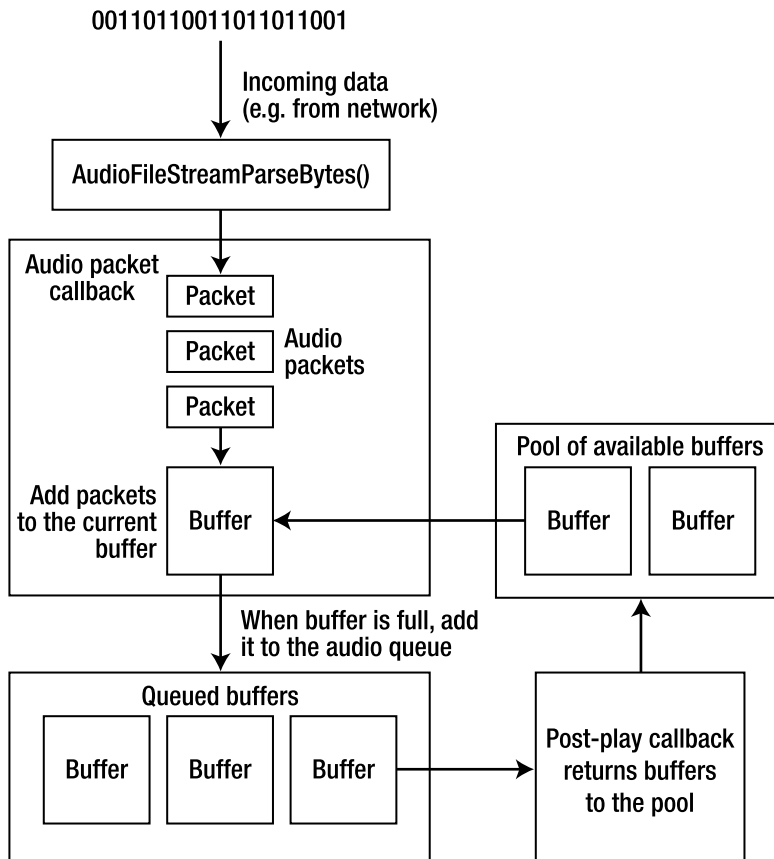


Figure 3-3. Playing streaming audio with Audio File Stream Services

Figure 3-3 also shows yet another callback, here called the *post-play callback*. As with Audio Queue Services, you're creating buffers of audio data that you then add to the audio queue. Once a buffer has been played, though, it can't be passed back to the audio data callback, because it's expecting audio packets from the parser. So, instead, the audio queue passes the buffer to another callback function so that you can take care of cleaning it up. You could choose to allocate new buffers on the fly and then release the memory in this callback. This might tend to use a lot of memory, though, unless you strictly limited the number of buffers that were permitted to exist at any one time. A simple way to accomplish this is to preallocate a pool of buffers. The audio data callback takes an existing buffer from the pool or blocks if none is available. When you add a buffer to the audio queue, you mark it as being in use, and after it has been played, you clear the "in-use" setting and return it to the pool.

An important consideration with this new callback is that Audio Queue Services will invoke the callback on a separate thread, which is created to handle the actual playback. Once playback completes, the post-play callback is called on this same thread. Any code you include in the callback needs to be aware of the fact that it's running on a different thread than the rest of the code. You'll use an `NSCondition` object to synchronize threads and to safely handle the audio buffer pool.

Listing 3-13 shows the interface for a simple audio streaming class. It starts with declarations for the number and size of buffers. In this case I'm using fixed values instead of using the utility function from earlier to try to optimize the sizes. This means that I don't know exactly how much playback time will be contained in each buffer, but it also means that I'll know exactly how much memory I'm using for buffers. I could just rely on the `DeriveBufferSize()` function from earlier imposing its maximum size limits. In the constrained iPhone environment, I prefer to make the memory requirements as predictable as possible, and fixed sizes make that possible. If you'd like the same predictability but aren't sure what makes for a good size, you might use the `DeriveBufferSizes()` during development to get a feel for how big the buffers need to be and then set your fixed values to something comparable to its results.

The class interface defines a structure called `PlayQueueData`, which is used by the class to manage audio buffers. It contains the buffer itself (the `AudioQueueBufferRef`) as well as several other associated items that need to go along with it. The `NSCondition` object, for example, is used when marking buffers as in use and to block the code from enqueueing a new buffer when all existing buffers are in use. Instances of the streaming class will create an array of `PlayQueueData` structs.

As the interface suggests, this class is initialized with a URL pointing to an audio source and will rely on an `NSURLConnection` to download the data from that source.

Listing 3-13. Audio Streaming Class Interface

```
// Number of audio queue buffers we allocate
#define kNumAQBufs 3
// Number of packet descriptions in our array
#define kAQMaxPacketDescs 512
// Use a hard-coded buffer size.
#define kAQBufSize 1048576 /* 1 MB, or 2**20 */

// Data structure containing an audio queue buffer as well as its associated data.
typedef struct PlayQueueData {
    AudioQueueBufferRef buffer;
    NSCondition *queuedCondition;
    UInt32 packetCount;
    AudioStreamPacketDescription packetDescriptors[kAQMaxPacketDescs];
    size_t bytesFilled;
    BOOL inUse;
} PlayQueueData_t;

@interface SimpleStreamer : NSObject {
    NSURL *url;
    NSURLConnection *networkConnection;

    AudioFileStreamID myAudioStream;
```

```

    AudioQueueRef playQueue;

    BOOL queueStarted;
    BOOL queueRunning;

    PlayQueueData_t *playQueueDataRecs;
    unsigned int currentBufferIndex;
}

@property (readonly) NSURL *url;
@property (readwrite) AudioQueueRef playQueue;
@property (readwrite) BOOL queueRunning;
@property (readwrite) BOOL queueStarted;
@property (readwrite) unsigned int currentBufferIndex;
@property (readwrite) PlayQueueData_t *playQueueDataRecs;

- (id)initWithURL:(NSURL *)url;
- (void)play;
- (void)stop;

@end

```

BUFFERING ON MOBILE DEVICES

It's important to make sure that you buffer enough audio data to handle changing network conditions. Because the iPhone is a mobile device, apps need to be designed to handle the possibility that the device is in motion while the app is running. That might mean unexpected transitions between 3G and EDGE networks, for example, or just a weak signal and lower data rates at times. This is much more important on the iPhone than with desktop computers and even laptops, which are merely portable but less likely to be used in motion.

The iPhone is surprisingly robust in these situations. When the network changes from 3G to EDGE or back, network connections usually will not drop. Instead, they'll stall briefly before resuming. In a streaming app, this means you'll stop receiving incoming bytes for a while but that your network connection should eventually pick up and start sending data again. It's during this stall time that your audio buffers are most useful. There's no guarantee of how long the stall will last, but in my testing I've found that small buffers won't be up to the task. Planning for one to two seconds of downtime is nowhere near sufficient to avoid audio dropouts. Make the buffers as large as you think you can afford.

A side effect of buffering is that, in the case of a live audio stream, your playback may lag compared to the original audio. That's a necessary consequence of planning to keep playback going in changing network situations.

Listing 3-14 shows the initializer for the streaming class. This is where the array of `PlayQueueData` structures is allocated, although the audio buffers they contain are still `NULL` at this point. They can't be allocated until you know something about the audio format, so you'll leave that until you have that information.

Listing 3-14. Initializing the Audio Streaming Object

```

- (id)initWithURL:(NSURL *)audioUrl
{
    if (self = [super init]) {
        url = [audioUrl retain];
        playQueueDataRecs = (PlayQueueData_t *)malloc(sizeof(PlayQueueData_t) *
            kNumAQBufs);
    }
    return self;
}

```

Listing 3-15 shows the starting point for audio playback. This is a very short method, especially in comparison to the one used earlier in the Audio Queue Services example. The reason is that with streaming audio you don't have an audio file at first, so you can't start looking at its audio format or other characteristics yet. Instead, you just create the `AudioFileStreamID myAudioStream` and then start the network connection. The second and third arguments to `AudioFileStreamOpen()` are the property listener and audio data callback functions, which will be called as soon as the stream has enough data to start making sense of the incoming data.

The fourth argument to `AudioFileStreamOpen()` is a hint about the audio format, which you saw earlier in the Audio Queue Services code. In this case I'm passing 0, which implies that the stream should attempt to determine the format.

Listing 3-15. Starting the Stream

```

- (void)play
{
    // Create audio stream using callback functions.
    // Third argument is an optional hint to file type.
    AudioFileStreamOpen(self, propertyListenerCallback, audioDataCallback, 0,
        &myAudioStream);

    // Create the network connection
    NSURLRequest *networkRequest = [NSURLRequest requestWithURL:self.url];
    networkConnection = [[NSURLConnection alloc] initWithRequest:networkRequest
        delegate:self];
}

```

Note that I've used the trick of passing `self` again here, so that the callback functions will be able to make method calls on the streaming object.

The `NSURLConnection` created in Listing 3-15 will connect and begin receiving data as soon as it has been created. Since I've set `self` as the connection's delegate, the connection will supply this data in the `connection:didReceiveData:` method on the streaming object. Recall that this method may be called many times, often with as little as 1KB to 2KB of data. Listing 3-16 shows this method. Whenever this happens, I pass the data to the parsing function. The first argument, `myAudioStream`, is the stream object I created earlier. The parsing function will use this argument to find the callbacks that I registered for it.

Listing 3-16. Receiving Data from the Network and Parsing It

```
// NSURLConnection delegate method
- (void)connection:(NSURLConnection *)connection didReceiveData:(NSData *)data
{
    // Pass incoming bytes to audio stream parser.
    AudioFileStreamParseBytes(myAudioStream, [data length], [data bytes], 0);
}
```

When enough data has been received, it will be possible to start determining properties of the audio. The property callback will be called for each one. The property information continues to be available after the callback, though, so although the property listener will be called as soon as the data format is known, it's still possible to look up this property on the audio stream later. It's not necessary to store this information in your own variables. As a result, the property listener function will do nothing until the stream is ready to produce audio packets, at which point it can look up all the information you need. This property indicates that all metadata has been received and that playback can begin. Listing 3-17 shows the property listener callback.

Listing 3-17. Property Listener Callback for Audio File Stream Services

```
void propertyListenerCallback (void *inClientData,
AudioFileStreamID inAudioFileStream,
AudioFileStreamPropertyID inPropertyID,
UInt32 *ioFlags)
{
    SimpleStreamer *self = (SimpleStreamer *)inClientData;
    OSStatus err = noErr;
    UInt32 propertySize;

    if (inPropertyID == kAudioFileStreamProperty_ReadyToProducePackets) {
        // The stream is ready to produce audio packets

        // Get the audio format
        AudioStreamBasicDescription dataFormat;
        propertySize = sizeof(dataFormat);
        err = AudioFileStreamGetProperty(inAudioFileStream,
            kAudioFileStreamProperty_DataFormat, &propertySize, &dataFormat);

        // Create the play queue
        AudioQueueRef playQueue;
        err = AudioQueueNewOutput(&dataFormat, postPlayCallback,
            self, NULL, kCFRunLoopCommonModes, 0, &playQueue);
        [self setPlayQueue:playQueue];

        // Set up audio buffer structures
        for (int i=0; i<kNumAQBufs; i++) {
            self.playQueueDataRecs[i].queuedCondition =
                [[NSCondition alloc] init];
            err = AudioQueueAllocateBuffer(playQueue,
                kAQBufSize, &(self.playQueueDataRecs[i].buffer));
        }

        self.currentBufferIndex = 0;

        // Lock the initial buffer, which is where we'll start writing data.
        NSCondition *queuedCondition =
            (NSCondition *)self.playQueueDataRecs[0].queuedCondition;
```

```

[queuedCondition lock];
self.playQueueDataRecs[0].inUse = YES;
[queuedCondition unlock];

// Get the magic cookie (for compressed formats like MPEG 4 AAC)
// from the file stream and set it on the audio queue
err = AudioFileStreamGetPropertyInfo(inAudioFileStream,
    kAudioFileStreamProperty_MagicCookieData, &propertySize, NULL);
void *magicCookie = calloc(1, propertySize);
err = AudioFileStreamGetProperty(inAudioFileStream,
    kAudioFileStreamProperty_MagicCookieData,
    &propertySize,
    magicCookie);
if (err == noErr) {
    err = AudioQueueSetProperty(playQueue,
        kAudioFileStreamProperty_MagicCookieData,
        magicCookie,
        propertySize);
}
free(magicCookie);
}
}

```

The property listener callback starts by checking its `inPropertyID` argument to see what new property information is available. If it's not `kAudioFileStreamProperty_ReadyToProducePackets`, it does nothing. When the stream is ready to produce packets, it finishes the setup that wasn't possible earlier when the stream was created. First it looks up the audio format property and creates the play queue. This makes use of the `self` variable both to set the play queue on the streaming object and to pass `self` along in `AudioQueueNewOutput` so that it will be available in the `postPlayCallback()` function.

Next the callback creates the audio buffers, since you now have enough information to do so. You'll create the buffers once and reuse them as many times as needed. It sets `self`'s `currentBufferIndex` to 0 so that the first buffer in the array will be current and marks that buffer as being in use.

Now, the playback queue has been created, and you're ready to start playing audio. As new data comes in, you'll continue to pass it to `AudioFileStreamParseBytes()`. This will lead to calls to the audio data callback, the aptly named `audioDataCallback()`. Listing 3-18 shows this function.

Listing 3-18. Audio Data Callback

```

void audioDataCallback (void *inClientData,
    UInt32 inNumberBytes,
    UInt32 inNumberPackets,
    const void *inInputData,
    AudioStreamPacketDescription *inPacketDescriptions)
{
    SimpleStreamer *self = (SimpleStreamer *)inClientData;

    // Run through the incoming packets.
    for (int i=0; i<inNumberPackets; i++) {
        @synchronized(self) {

```

```

        if (self.queueStarted && (!self.queueRunning)) {
            // Stop if the queue is not running.
            return;
        }

        // Get size and offset of the current packet's data
        SInt64 packetOffset = inPacketDescriptions[i].mStartOffset;
        SInt64 packetSize = inPacketDescriptions[i].mDataByteSize;

        // See if there's enough byte space left in the current buffer.
        size_t bufSpaceRemaining = kAQBufSize -
            self.playQueueDataRecs[self.currentBufferIndex].bytesFilled;
        if (bufSpaceRemaining < packetSize) {
            // Not enough space in the current buffer, so enqueue it and
            // go to the next buffer.
            enqueueCurrentBuffer(self);
        }

        // Copy data to the audio queue buffer
        AudioQueueBufferRef fillBuf = elf.playQueueDataRecs[
            self.currentBufferIndex].buffer;
        memcpy((char*)fillBuf->mAudioData +
            self.playQueueDataRecs[self.currentBufferIndex].bytesFilled,
            (const char *)inInputData + packetOffset, packetSize);
        // Fill out packet description
        self.playQueueDataRecs[self.currentBufferIndex].packetDescriptors[self.playQueueDataRecs
            [self.currentBufferIndex].packetCount] = inPacketDescriptions[i];

        self.playQueueDataRecs[self.currentBufferIndex].packetDescriptors[self.playQueueDataRecs
            [self.currentBufferIndex].packetCount].mStartOffset =
            self.playQueueDataRecs[self.currentBufferIndex].bytesFilled;
        // Keep track of bytes and packets filled in the current buffer
        self.playQueueDataRecs[self.currentBufferIndex].bytesFilled +=
            packetSize;
        self.playQueueDataRecs[self.currentBufferIndex].packetCount += 1;

        // See if we've run out of packet space
        size_t packetDescriptorsRemaining = kAQMaxPacketDescs -
            self.playQueueDataRecs[self.currentBufferIndex].packetCount;
        if (packetDescriptorsRemaining == 0) {
            // No more packet descriptors in the current buffer,
            // so add it to the queue.
            enqueueCurrentBuffer(self);
        }
    }
}

```

The audio data callback's purpose is to receive parsed audio packets and accumulate them in the current buffer. When the buffer fills, it adds it to the audio queue via a utility function called `enqueueCurrentBuffer()`.

Incoming data in this callback can consist of an arbitrary number of audio packets, with the actual number depending on how much data has been received from the network. The body of the function loops through the packets one at a time. It starts by checking to see whether the current buffer has enough space to hold the packet's data. If not, it

calls `enqueueCurrentBuffer`, which adds the current buffer to the queue and moves on to the next buffer.

Once that's done, you know that the current buffer can hold the current packet—whether or not it's the same buffer that was current before you checked the remaining buffer capacity. To copy the audio data into the current buffer, you get a reference to the `AudioQueueBufferRef` field of the current `PlayQueueData` struct. The call to `memcpy()` copies bytes from the incoming packet to the buffer. You then add details describing the current packet and update the current count of packets and bytes in the current buffer.

Finally, there's a second check on the current buffer—this time to see whether there's room for any more packets. As with the previous check, if the current buffer is full, you drop into `enqueueCurrentBuffer` to add the buffer to the queue and move on to the next one.

Adding data to the queue is done in a utility function called from `audioDataCallback()`, because you need to call this code in a couple of different places, and it's long enough that just duplicating it would be ugly. Listing 3-19 shows the `enqueueCurrentBuffer()` function.

Listing 3-19. Enqueueing the Current Buffer

```
void enqueueCurrentBuffer(SimpleStreamer *self)
{
    OSStatus err = noErr;

    @synchronized(self) {
        if ((self.queueStarted == YES) && (self.queueRunning == NO)) {
            // If the queue has stopped, don't enqueue any more data.
            return;
        }

        // Mark the current buffer as "in use".
        self.playQueueDataRecs[self.currentBufferIndex].inUse = YES;
        // Set the data size of the buffer.
        AudioQueueBufferRef fillBuf = ➡
            self.playQueueDataRecs[self.currentBufferIndex].buffer;
        fillBuf->mAudioDataByteSize = ➡
            self.playQueueDataRecs[self.currentBufferIndex].bytesFilled;
        // Add the buffer to the queue
        err = AudioQueueEnqueueBuffer([self playQueue],
            fillBuf,
            self.playQueueDataRecs[self.currentBufferIndex].packetCount,
            self.playQueueDataRecs[self.currentBufferIndex].packetDescriptors);
        if (err) {
            // Could not enqueue buffer
            return;
        }

        // Start the playback queue, if it's not running.
        [self startQueue];

        // Go to the next buffer
    }
}
```

```

self.currentBufferIndex++;
if (self.currentBufferIndex >= kNumAQBufs) {
    self.currentBufferIndex = 0;
}

// If the new current buffer is in use, wait for it to be returned to the pool.
NSCondition *queuedCondition = (NSCondition *) ➡
    self.playQueueDataRecs[self.currentBufferIndex].queuedCondition;
[queuedCondition lock];
@synchronized(self) {
    if (self.queueStarted && (!self.queueRunning)) {
        // Don't wait on the buffer if the queue has stopped.
        return;
    }
}
while (self.playQueueDataRecs[self.currentBufferIndex].inUse) {
    [queuedCondition wait];
}
[queuedCondition unlock];
}

```

The main purpose of `enqueueCurrentBuffer` is to take the current buffer and pass it to `AudioQueueEnqueueBuffer()` so that it can be played. The rest is housekeeping and maintenance details that need to be handled to keep the queue running smoothly.

First, you make sure the current buffer is marked as being in use and add the data to the queue. Immediately after that, you call a method named `-startQueue` to make sure audio is actually playing. Up until this point, you've been managing data packets and buffers, but this is where the sound starts coming out of the speakers. Listing 3-20 shows the `startQueue` method.

Now that you've enqueued the current buffer, it's time to move on to the next one. You do this by incrementing `currentBufferIndex`, taking care not to let it get larger than the number of existing buffers. If you're less familiar with C, this is one place the difference can be apparent. If you had been using an `NSArray` and you went beyond the array bounds, you'd immediately get a runtime error. C-style arrays follow the C approach of assuming that you know what you're doing, so going past the end of an array might go unnoticed at first. You'd end up accessing whatever data happened to be in memory just after the array and potentially changing it. This usually leads to a crash, but it's not always immediate—it means you have some bogus data, somewhere, that will be a problem if you try to use it. Avoiding the problem is easy enough, though, so in this case you just make sure that `currentBufferIndex` is always less than or equal to `kNumAQBufs`, the number of buffers that you created back in `-init`.

If you have a fast network connection, it's possible that data is coming over the network connection faster than you're playing it. In that case, the buffers start filling up, with each being added to the queue. Eventually you could reach a state where all existing buffers are in use. If you moved on to the next one at that point, you'd overwrite some of the audio that's already in the queue. The best that can be said about that situation is that it would sound really bad. This possibility is handled at the end of `enqueueCurrentBuffer()`. Each buffer is marked as "in use" when it's added to the queue, and this setting is cleared once the buffer has been played. Once the

current buffer index has been incremented, `enqueueCurrentBuffer()` checks to see whether the next buffer is still in use. If so, it means you need to wait until it has been played.

This is handled with an `NSCondition` object, which is part of the `PlayQueueData` structure you've been using. If you call `-wait` on this object, execution will block until someone calls `-signal` on the same object. That will happen later, in the post-play callback function. Since `enqueueCurrentBuffer()` is called from the audio data callback, this also means that no packets will be added to the buffer until the condition is signaled. If you worked your way back up the call stack, it turns out that this also means that `AudioFileStreamParseBytes()` won't return until the current buffer has been played. The entire chain of reading data and getting it played is put on hold until the current buffer has been played.

The `startQueue` method serves to call `AudioQueueStart` if and only if the queue has not been started yet. It's called from `enqueueCurrentBuffer()` because you want to start the queue as soon as you've added a buffer to the queue.

Listing 3-20. Starting the Playback Queue

```
// Start the audio queue, if it's not already playing
- (void)startQueue
{
    if (!queueStarted) {
        AudioQueueStart(playQueue, NULL);
        @synchronized(self) {
            queueStarted = queueRunning = YES;
        }
    }
}
```

Now that you're getting audio from the network, parsing it, and playing it, the only thing that remains is to make sure you don't run out of buffers. This is the job of the post-play callback, which you registered in `propertyListenerCallback()` when you created the audio queue. Listing 3-21 shows this.

Listing 3-21. Post-Play Callback Function

```
void postPlayCallback (void *aqData,
AudioQueueRef inAQ,
AudioQueueBufferRef inBuffer)
{
    SimpleStreamer *self = (SimpleStreamer *)aqData;
    PlayQueueData_t *currentBufferData = NULL;

    // Find the playQueueDataRecs entry corresponding to inBuffer.
    for (int i=0; i<kNumAQBufs; i++) {
        if (self->playQueueDataRecs[i].buffer == inBuffer) {
            currentBufferData = &(self->playQueueDataRecs[i]);
            break;
        }
    }

    if (currentBufferData != NULL) {
        // Mark the buffer as being available, so it'll
```

```

        // be available for new audio data.
        NSCondition *queuedCondition =
            (NSCondition *)currentBufferData->queuedCondition;
        [queuedCondition lock];
        currentBufferData->inUse = NO;
        // Reset the packet and byte count on the buffer.
        currentBufferData->packetCount =
            currentBufferData->bytesFilled = 0;
        // Signal the condition in case enqueueCurrentBuffer
        // is waiting on it.
        [queuedCondition signal];
        [queuedCondition unlock];
    }
}

```

The first thing you need to do is locate the structure containing the audio buffer. The `inBuffer` argument gives you the actual audio buffer, but you need the full `PlayQueueData` structure that contains it. It might seem that you could just use the `currentBufferIndex` value to look it up, but if you're still getting audio data over the network, then it's almost certainly been incremented by the time this function is called. So, you use a loop, running through each structure until you find the right one. You have only three of them, so this will be quick.

Once you've found the right structure, you make sure it's ready for use for new audio data. You set the "in use" flag to `NO` and reset the count of packets and bytes to zero. You don't need to release any memory here, because the buffers were allocated once back in `propertyListenerCallback()` and can be reused until you don't need them anymore. Resetting the packet and byte counts gets us back to the beginning of the buffer, ready to copy new data in.

Finally, you get the `NSCondition` object and call its `-signal` method. This will handle the case described previously where `enqueueCurrentBuffer` was blocked because of a lack of available buffers. Calling `-signal` here will unblock `enqueueCurrentBuffer` and allow it to continue.

THE AUDIO SESSION

Besides dealing with the audio stream, it's important to be aware of the Audio Session API. Audio sessions are the iPhone's system for specifying how your application works with regard to the audio hardware and other audio applications. For example, the Audio Session API allows you to specify whether your application should continue playing music when the user turns off the iPhone's screen. It also lets you control whether your application will interrupt audio being played by the iPod application and lets your app respond to audio interruptions such as incoming phone calls.

In this example, I'm leaving these details at the system defaults. It's still necessary to initialize the audio session though, or the app won't get access to the audio hardware. You initialize the session only once—there's no corresponding deinitialize method—so I decided to do this in the streaming classes' `+initialize` method (Listing 3-22). This method will be automatically called by the system as soon as the class is loaded by the application.

Listing 3-22. *Initializing the Audio Session*

```
+ (void)initialize
{
    AudioSessionInitialize(NULL, NULL, NULL, NULL);
}
```

Passing NULL for all arguments gets access to the audio hardware without specifying any custom configuration.

Putting It All into an App

Remember Alice? This is a song about Alice.

Arlo Guthrie, “Alice’s Restaurant”

I didn’t finish the SeeqPod application immediately. Soon my membership in the iPhone developer program came through, and I got busier than I could remember being working on iPhone contracts. The SeeqPod idea got moved to the back burner. And then, when I needed the back burner, the idea got put in the fridge to finish later. Time passed, as it is wont to do.

When I was able to return to the project, I made an awful discovery: SeeqPod had shut down its audio search system. Its web site was not clear about when or if it might return.

I like learning new stuff and writing interesting new code that stretches my abilities. But I’m also practical, and I hate doing all that and having no use for the result.

Fortunately, audio streaming code is useful enough that it’s not a solution that has to look hard for a problem to solve. While chatting in IRC one day, someone mentioned the idea of writing a custom iPhone app for a radio station. And the light bulb went on again. I’ve been a member of KRCC, a local public radio station, for something like 17 years. Its broadcasts were available online. And I had met Delaney Utterback, the station manager, and was pretty sure he’d like the idea. I contacted him, he was all for it, so I went to work applying my streaming code to their broadcasts.

One More Thing

For some reason, it didn’t work. I’d start downloading data and passing it to the parser, but the parser never indicated that it was ready to produce packets. In fact, it never even notified me that it had worked out the audio format of the stream.

The reason turned out to be a simple but significant difference between my initial goal and my current one. SeeqPod had provided URLs to complete audio files. But KRCC’s broadcast stream—as with most live streams—was a continuous sequence of audio data. The key difference is that an audio file contains a header segment that includes information about the audio encoding, while a continuous stream doesn’t have this

information. Maybe it was provided once, when the stream started, but it wasn't available to me. Figure 3-4 illustrates the difference.

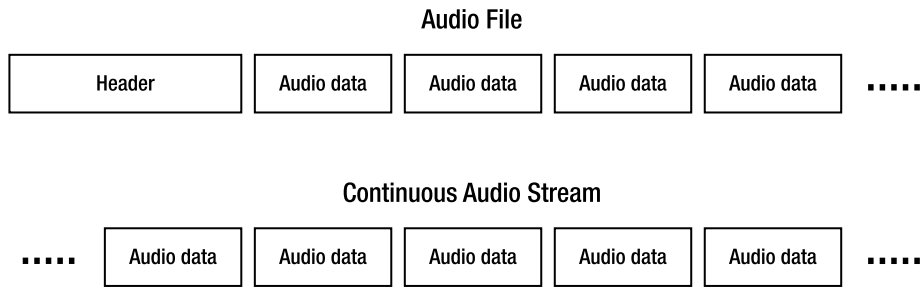


Figure 3-4. Audio file layout compared with audio stream data layout

It turns out that this is one of the cases where you need to give audio file stream services a hint about the file type. KRCC's stream is basically a never-ending MP3 file, but without the header information, Core Audio couldn't figure that out. Previously I didn't bother with this, because it wasn't necessary when dealing with complete files.

To get audio playing, I gave my audio streaming class a new instance variable that could be used to provide a hint about the audio format. This changed the class interface, as shown in Listing 3-23.

Listing 3-23. Modified Class Interface with Hinting Property

```
@interface SimpleStreamer : NSObject {
    NSURL *url;
    NSURLConnection *networkConnection;

    AudioFileStreamID myAudioStream;
    AudioQueueRef playQueue;

    AudioFileTypeID fileTypeHint;

    BOOL queueStarted;
    BOOL queueRunning;

    PlayQueueData_t *playQueueDataRecs;
    unsigned int currentBufferIndex;
}

@property (readonly) NSURL *url;
@property (readwrite) AudioQueueRef playQueue;

@property (readwrite) AudioFileTypeID fileTypeHint;

@property (readwrite) BOOL queueRunning;
@property (readwrite) BOOL queueStarted;
@property (readwrite) unsigned int currentBufferIndex;
@property (readwrite) PlayQueueData_t *playQueueDataRecs;
```

Instance variables have a default value of 0, so if I didn't provide a hint, the code would fall back on the previous behavior of trying to work out the format from the available

data. I made one change to the play method to specify the hint when creating the stream, as shown in Listing 36-24.

Listing 3-24. Modified play Method with Hinting

```
- (void)play
{
    // Create audio stream using callback functions.
    // Third argument is an optional hint to file type.
    AudioFileStreamOpen(self, propertyListenerCallback,
        audioDataCallback,

        self.fileTypeHint,
        &myAudioStream);

    // Create the network connection
    NSURLRequest *networkRequest = [NSURLRequest requestWithURL:self.url];
    networkConnection = [[NSURLConnection alloc] initWithRequest:networkRequest
        delegate:self];
}
```

To provide a hint, I'd set the fileTypeHint property after creating the streaming object but before starting playback, as Listing 3-25 shows.

Listing 3-25. Providing a Hint for Audio Streaming

```
- (IBAction)play:(id)sender
{
    streamer = [[SimpleStreamer alloc] initWithURL:url];
    streamer.fileTypeHint = kAudioFileMP3Type;
    [streamer play];
}
```

That was all it took. With this simple change I was able to play KRCC's Internet stream (and incidentally any stream the iPhone is capable of playing).

In some cases hinting may not be sufficient, though. I've seen some streams where, despite hinting, audio file stream services misidentified the stream format. Since it was trying to interpret one format as some other format, it was unable to play the audio. There are a couple of approaches to this, both of which involve modifying the property listener function in Listing 3-17.

The first approach would be to specify the format yourself. The code in Listing 3-17 makes use of the hint but works out the details for itself. Although I hint that the stream is MP3, I let Core Audio work out the sample rate, whether it's stereo or mono, and so on. But I don't have to do that. I could fill in these details myself, by setting up the values in an `AudioStreamBasicDescription` on my own instead of by looking them up using the `kAudioFileStreamProperty_DataFormat` key. If I know what stream I'm working with, I would presumably know all the necessary encoding details.

But if you've been following along, you know that I probably don't want to do that. I'm not an expert at audio encoding, and there's a reasonable chance I'd get something wrong. In addition, it's a possibility that the audio stream I'm playing might change at some point, say to a higher bit rate. If I've specified one rate and the stream changes to another one, my app will unexpectedly break.

An alternative approach, possibly cruder but more reliable in the field, is to look up the format as you've been doing and then compare that with the hint you provided. The `AudioStreamBasicDescription` you look up in Listing 3-17 is a C struct, and one of its fields is called `mFormatID`. That tells you what format Core Audio thinks the stream is. By comparing that to your hint, you can work out whether Core Audio has at least found the correct encoding. If it did, great, and if it didn't, close the network connection and start over. Core Audio gets the encoding right most of the time, so although restarting like this is not elegant, it's effective.

Launch It!

SeeqPod had made me somewhat wary, but I was pretty sure KRCC would be around for a while. It had survived several decades already. So, now I had an app I could actually launch. And so I did (Figure 3-5). And the people at the radio station loved it. On the day the app made it into Apple's store I dropped by the station, and they put me on the air to talk about it. Cool!



Figure 3-5. KRCC application showing available audio streams

iPhone 3.0 and Further Work

I originally did the work I've described with various versions of iPhone OS 2.1. Since then, iPhone OS 3.0 has been released. All of the code I've described works just as well with 3.0 as with earlier versions. The only difference that may be of interest is the addition of the `AVAudioSession` class.

`AVAudioSession` provides an Objective-C API for dealing with audio sessions. In Listing 3-22 you initialized the audio session for the app but left all session options at their

default values. `AVAudioSession` implicitly initializes the audio session the first time it's asked to do anything, so Listing 3-22 could be modified to just ask for the system's shared `AVAudioSession` object, as in Listing 3-26.

Listing 3-26. *Initializing the Audio Session via the shared `AVAudioSession` Object*

```
+ (void)initialize
{
    AudioSessionInitialize(NULL, NULL, NULL, NULL);
}
```

Where `AVAudioSession` may come in handy is if you want to customize the behavior of the audio session and if you prefer to work in Objective-C rather than C. As I mentioned previously, the audio session lets you customize how your app's audio playback interacts with other apps and with the hardware. `AVAudioSession` doesn't add any new capabilities in this area, but it'll let you use Objective-C instead of C for these aspects of your app.

Summary

Audio playback can be easy or hard, depending on your needs. I found that I needed the hard way, but I figured it out, and I hope this helps you do the same.

In many cases, it's possible to use one of the easier approaches, and in those cases there's not much reason to bother with the Core Audio approach I've described. The streaming code will work just as well for locally stored files (via `file://` URLs) as it does for remote files. But `AVAudioPlayer` is an excellent and simpler choice for playing audio files included in your application. If that's what you're doing, then there's no reason to complicate matters.

The Core Audio approach is still best if you're playing music downloaded from the Internet—whether it's a live broadcast or an audio file available at a web site. For the live broadcast it's the only option, and for the static file this approach makes it possible to begin playback without waiting while you download the entire file first.

I also talked about the `AVAudioSession` class in light of the iPhone OS 3.0 release.

Good luck with your audio projects. And don't be afraid of C!

Owen Goss



Company: Streaming Colour Studios

Location: Toronto, Ontario, Canada

Former Life As a Developer: Lead User Interface Programmer at Electronic Arts Canada on five PSP games. Gameplay Programmer at Electronic Arts Canada on one PSP game. Lead User Interface Programmer at Propaganda Games on one Xbox 360 and PS3 game. Senior Gameplay Programmer at Propaganda Games partially on one Xbox 360 and PS3 game.

Life as an iPhone Developer: Creator of Dapple (Games: Puzzle, Family)



Dapple is entirely OpenGL-ES. Dapple is approximately 80% C++ and 20% Objective C.

What's in This Chapter: Chapter Title: "You Go Squish, Now! Debugging on the iPhone"

- Custom Debugging Macros
- Using Crash Logs
- Reproducing Rare Crashes
- Memory Stomps
- malloc_error_break
- NSZombieEnabled
- Enable Guard Malloc
- Watching Variables
- Link Map Files
- Conclusions

Key Technologies:

- ***iPhone Crash Logs***
- ***XCode Debugger***
- ***iPhone Simulator & Device***

You Go Squish Now!¹

Debugging on the iPhone

It all started with an idea. You downloaded the SDK, you taught yourself Objective-C, you built your app, and now you've found a horrible crash. Debugging tricky crashes can be challenging and can try your patience. However, it can also be a lot of fun when approached in the right way.

In this chapter, I'll walk you through some advanced debugging concepts and techniques for the iPhone.

Before coming to the iPhone, I worked for about five years in the console games industry building PlayStation Portable, Xbox 360, and PlayStation 3 games. While working in that industry, I spent much of my time as a lead user interface programmer. User interface programming is an interesting job because it touches almost all aspects of the game in some way. This meant I got to do a lot of debugging in a lot of different areas of the game. This proved to be an invaluable skill as I progressed in my career. In addition to my role as a UI programmer, I also was a senior game-play programmer dealing with physics and math systems, as well as low-level optimizations and debugging on PlayStation 3's multiprocessor system.

Last year I set out on my own and formed my own company, Streaming Colour Studios. We released our first game, Dapple, in February 2009 for the iPhone. Dapple is a color-matching game where players have to mix paint colors to make matches (see Figure 4-1). It's a challenging twist on the match-three genre of game.

¹ "You go squish now!" is a line from *The Simpsons* episode "Treehouse of Horror V" directed by Jim Reardon and written by Greg Daniels, Dan McGrath, David Cohen, and Bob Kushell. Original airdate in North America: October 30, 1994.



Figure 4-1. *Dapple for iPhone and iPod touch*

I started working with the iPhone because I saw it as an opportunity to develop the kinds of games that I wanted to make, with little overhead, and publish them on a powerful gaming machine. When I started working with the iPhone, I quickly realized that much of what I had learned about debugging on large console games could be applied to iPhone development. Nasty crash bugs caused by memory stomps are just as likely to happen in an iPhone app as in any major console title. In discovering this, the major challenge was learning how to apply the tricks and techniques I had previously learned to Xcode and the iPhone hardware.

I'm pleased to be able to share some of the knowledge I have picked up over the years. My first tip is perhaps the most important: remember that debugging can be fun! I know, I must be crazy to say something like that, but it's true. Entering into debugging with the right frame of mind can help you find your bugs much more quickly and painlessly. Instead of looking at debugging as a chore, imagine yourself as a code detective, trying to get to the bottom of the Mysterious Case of the Random Crash!

Got your pipe and Sherlock Holmes hat on? Good. Let's dig in!

Assumed Knowledge

Before I go any further, I want to go over some of the things I'm going to assume you already understand. These are things I'm not going to cover in this chapter, but they are all concepts and tools that you should already be using to debug your apps. If you are unsure of how to do any of these things, I'll wait right here while you read up on them:

- You know how to run your app in the debugger, both in the simulator and on your iPhone or iPod touch device.
- You understand how to read a call stack and know what it's telling you.
- You know how to set a breakpoint on a line of code in Xcode.
- You know how to examine the value in a variable in the debugger.

TIP You can find information on the previous techniques in Apple's "Xcode Debugging Guide" (<http://developer.apple.com/documentation/DeveloperTools/Conceptual/XcodeDebugging/>).

Though not required for this chapter, two more techniques are extremely helpful to understand:

- You know where to find the Expressions window and how to use it.
- You know where to find the Memory Browser and how to use it.

TIP: You can find information on both the Expressions window and the Memory Browser in the "Viewing Variables and Memory" section of the "Xcode Debugging Guide" (http://developer.apple.com/documentation/DeveloperTools/Conceptual/XcodeDebugging/600-Viewing_Variables_and_Memory/variables_and_memory.html). Figure 4-2 shows an example of both.

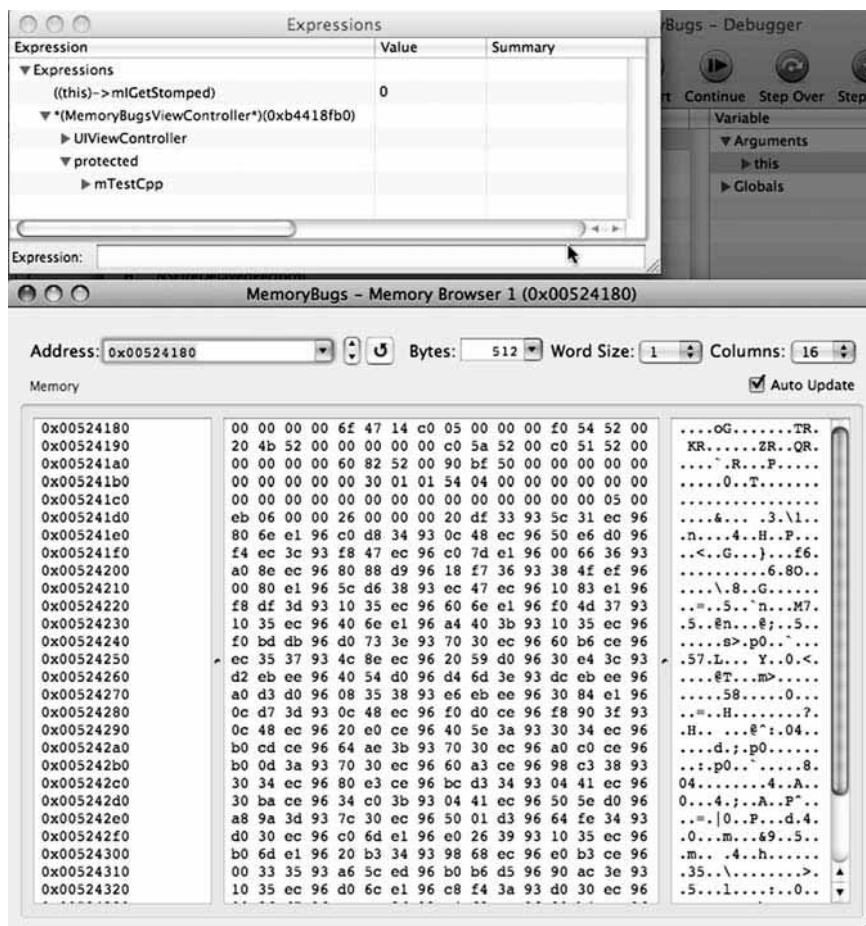


Figure 4-2. An example of the Expressions window and Memory Browser when stopped at a breakpoint

In essence, I'm assuming that you already know how to track down and fix simple bugs and that you already understand the basics of the debugger. I'm assuming that what you're looking for now is information on how to track down those really nasty bugs—the ones that keep you up at night. If you're clear on all this, then let's get started!

Objective-C vs. C and C++

iPhone development is very exciting. One of the reasons that I was so excited by my early forays into iPhone development was the ability to mix Objective-C and C/C++. This allows you, as a developer, to use whatever language you feel most comfortable with.

For example, my game, Dapple, uses Objective-C for only the most top-level classes that operate at the view and view controller levels. Everything at a lower level is done in C++. I chose this approach because I knew C++ much better than Objective-C when I

started. It also allowed me to more closely manage my memory usage. However, there are a few issues to consider when looking at your language choices:

- Objective-C provides a lot of memory protection for you. Built-in classes such as `NSArray` and `NSMutableArray` make sure that you don't overrun buffers. However, these classes come at a cost of a slight increase in memory and potential performance overhead when using them.
- C++ allows you to manage memory more directly, but it is much easier to introduce memory bugs with it.

Some of the topics I'll cover in this chapter will apply only to Objective-C code or only to C/C++ code, because there are certain techniques that work with only one or the other. However, much of what I'll be discussing applies to either language. I will point out where something applies to only one language.

While You're Writing That Code

I first want to cover the tasks that you can do while you're writing your code. It's a good idea to build up a library of debugging code that is at your disposal. You'll want to do this early on so that you can use this code throughout development. Doing so will help you catch bugs sooner and make tracking them down a lot easier.

Custom Asserts

Asserts are checks that you can put in your app that cause the application to halt if it fails a condition. C++ has a built-in call that you can use, as does Objective-C. For example, in C++, the following assert checks to see whether the condition is true:

```
assert(myVariable == someOtherVariable);
```

If it is true, then everything continues. If it's false, then the program will halt in the debugger (usually by forcing a crash).

NOTE: Objective-C provides two assert methods: `NSAssert` and `NSCAssert`. `NSAssert` should be used only in Objective-C methods, while `NSCAssert` can be used only in C methods. Both can be removed from code by defining `NS_BLOCK_ASSERTIONS` as a preprocessor macro.

The problem with using the built-in C++ assert call is that ideally you want your asserts to be active only while you're debugging. You want a way to disable all of your asserts when you go to ship your app. Asserts are invaluable in helping you track down variables that are out of bounds and other common problems, but in your shipped product, they're just taking up valuable CPU cycles, because your app should be bug-free by that point. Right?

The best way to deal with this is to create custom assert code. In this section, you'll create a custom assert macro that can be disabled via a build-time preprocessor define.

First, create a header file that's going to hold the macros you're about to write. I'm going to call mine `MyDebug.h` so that I know these are my debugging functions, but you can call it whatever you'd like. Then you can include this header wherever you need it.

Add the following code to the header file, and then I'll talk about what it's doing:

```
#if defined(APP_STORE_FINAL)
    #define MY_ASSERT(STATEMENT) do { (void)sizeof(STATEMENT); } while(0)
#else
    #define MY_ASSERT(STATEMENT) do { assert(STATEMENT); } while(0)
#endif
```

Look at the first line of code:

```
#if defined(APP_STORE_FINAL)
```

This line is checking to see whether something called `APP_STORE_FINAL` is defined. The reason you don't see this defined anywhere in the code is that you'll add this as a compile-time define, *but only to your build that's ready for distribution or submission to the App Store*. If you don't already have a build target for distribution/submission, create that now. Open the build target properties for your distribution build, and find the Preprocessor Macros Not Used in Precompiled Headers entry. Enter **`APP_STORE_FINAL`** into the field (see Figure 4-3).

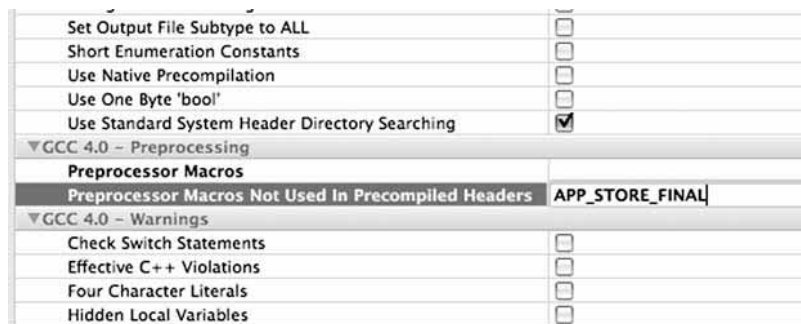


Figure 4-3. Setting `APP_STORE_FINAL` in the built target settings

The next line of code defines a macro that will be used when `APP_STORE_FINAL` is defined. You might be asking, “Hey, why not just define it to be nothing at all?” Good question! The problem with that approach is that you might have a variable used in your assert that is used *only* in the assert. If you're compiling with Treat Warnings as Errors, you could get a whole bunch of errors when you turn off your custom asserts. The following line of code generates a very small amount of assembly while still avoiding those build warnings:

```
#define MY_ASSERT(STATEMENT) do { (void)sizeof(STATEMENT); } while(0)
```

The `do...while` loop that exists around the macro body is there to avoid nasty scoping errors that can come up if you were to use your custom assert inside an `if` statement without braces, for example.

Finally, you can clearly see that when you're building your debug build, your custom assert just uses the standard assert function (with the do...while wrapped around it for safekeeping):

```
#define MY_ASSERT(STATEMENT) do { assert(STATEMENT); } while(0)
```

NOTE: This code is based on code from Charles Nicholson's blog. You can read the full article (with even more suggestions for custom asserts) at <http://cnicholson.net/2009/02/stupid-c-tricks-adventures-in-assert/>.

Fantastic! Now you have a custom assert that can be compiled out in your final build! Very handy! You can use it like this:

```
MY_ASSERT(result == true && "Result returned false, but we need it to be true!");
```

In this example, if the variable `result` is ever false, then the assert will fail, and program execution will halt in the debugger. The text string is just there so that you can remember why this particular assert is there.

The other advantage of writing your own custom assert code is that you can perform other actions in debug builds when an assert fires. Do you always want to print out the values in your game's state machine prior to an assert firing? You can do that and know that it will be compiled out before releasing.

Custom Logging

You're now going to use that handy `APP_STORE_FINAL` define you have set up to create a custom logging function that can be compiled out. It's basically the same as the earlier custom assert, so I'll just show the code here so you can put it into your debugging header file:

```
#if defined(APP_STORE_FINAL)
    #define MY_LOG(format, ...)
#else
    #define MY_LOG(format, ...) CFShow([NSString stringWithFormat:[NSString \
                                                                    stringWithUTF8String:format], ## __VA_ARGS__]);
#endif
```

NOTE The `__VA_ARGS__` identifier was introduced in the C99 standard. As such, all versions of Xcode used to build iPhone apps should support it. However, if you're attempting to use this with much older compilers for other projects, it may fail to compile.

There are a couple of things to note here. First, I've chosen to use `CFShow` instead of `NSLog` here because `CFShow` doesn't print out a lot of extra date and time information to the console, allowing for faster output of lots of debug text. You can change it to `NSLog` if you prefer to have the date and time print with each message.

Second, you can see that I'm assuming that MY_LOG will be passed a cstring as a parameter. I chose this because I have a lot of cstrings in my C++ code. You can replace it with the following code, if you want to always pass it an NSString instead:

```
#define MY_LOG(format, ...) CFShow([NSString stringWithFormat: format, \
                                ## __VA_ARGS__]);
```

You'll notice that MY_LOG compiles to nothing when APP_STORE_FINAL is defined. This is done because it's unlikely that you would declare a variable just for the sake of logging, so the odds of this generating a warning are slim. If it does cause a problem, you can use a technique similar to the assert macro to fix the errors.

Using #define

The use of #define in this section isn't code that will go into your debugging header, but rather it's something to keep in mind. If you end up writing big blocks of debugging code (for example, when logging complex data structures to track down a problem), don't just delete that code when you're done! If you need it again in the future, you'll regret having deleted it. Instead, try using #define to block it out.

I make extensive use of this in my code for logging complex data structures or debug rendering. It's quite handy, especially if you need to turn on debugging code in several places in your code at once. If you have a class where you have debug code written in several places, create a #define at the top of your class. Here's an example:

```
#define ENABLE_SOME_NAME_DEBUG 1
```

Then, wherever you need to run debug code on your data structure, wrap it with this:

```
#if (ENABLE_SOME_NAME_DEBUG)
    // Do my debug code here
#endif
```

Now you can turn that debugging code off and on just by changing ENABLE_DATA_STRUCTURE_DEBUG_LOGS to 0 or 1, respectively. You can set up multiple kinds of debugging code and wrap sections with differently named #defines, allowing you to turn on/off specific debug functionality as you go. For example, in my game Dapple, I have the following at the top of my game logic class (the one that searches the board for matches):

```
#define ENABLE_ALL_FUNCTION_TRACE 0
#define ENABLE_SEARCH_RESULTS_VISUAL_DEBUG 0
#define ENABLE_ENTIRE_SEARCH_PATH_VISUAL_DEBUG 0
#define ENABLE_CELL_POSITION_VISUAL_DEBUG 0
#define ENABLE_TEST_BOARD_STARTUP 0
#define ENABLE_END_GAME_IMMEDIATELY 0
#define ENABLE_AI_DECISION_OUTPUT 0
```

Each of those defines allows me to turn on specific debugging code without having to remember the four places in the code where I need to enable the debugging functionality. Hooray! For example, turning on ENABLE_CELL_POSITION_VISUAL_DEBUG allows me to output the positions of all my sprites every frame for animation debugging.

Figure 4-4 shows the output. Note that I'm using the SC_LOG macro that was mentioned earlier for the output.



Figure 4-4. Dapple running in the simulator with one of the debug `#define` values turned on

Crash!

No matter how hard you might try to write bug-free code, sooner or later you'll run into problems. Whether you experience a problem while running your app locally or one of your beta testers has a crash, you need to know how to track it down and fix it.

Getting a Crash Log from Your Testers

Chances are that one of your testers will find a crash you've never seen before. After all, that's why you have other people testing your app in the first place. The first thing you'll want to do is get them to send you a crash log from their device. Ask them to plug their iPhone or iPod touch into their computer; this will transfer the crash log files to their computer. Where those files are depends on their operating system (according to the Apple Developer Connection):

- *Mac OS X:* ~/Library/Logs/CrashReporter/MobileDevice/<Device_Name>
- *Windows XP:* C:\Documents and Settings\<Username>\Application Data\Apple computer\Logs\CrashReporter\<Device_Name>
- *Windows Vista:* C:\Users\<Username>\AppData\Roaming\Apple computer\Logs\CrashReporter\MobileDevice\<Device_Name>

Have them find the crash log for your app (the one that ends in .crash) with a date and time that closely matches the time of the crash (if they can remember). If they can't remember, you can always have them send you all the crash logs with your app's name.

TIP: When you have a tester send you a crash log, make sure they let you know what build number they were running. Chances are you have sent out several builds, and not every tester is going to update to the latest build. Knowing which build crashed can save you a lot of headache.

You Have Been Saving Your dSYM Files, Right?

When you're running your app in the debugger, the reason you get nice text names for the functions in your call stack (instead of hex memory addresses) is that you have debug symbols included in the app. The iPhone does this in a great way by building your debug symbols into a file with a .dSYM extension every time you compile. You'll find it in the same directory that your .app file was generated when you built.

TIP: Every time you create a build to send to testers, archive the .dSYM file along with the app!

That tip is so important I'm going to say it again. Every time you create a build to send to testers, archive the .dSYM file along with the app!

Why is this so important? It's important because the dSYM file is what will allow you to get a readable call stack from a crash log instead of just a bunch of hex memory addresses. Chances are, when a user gets a crash, the crash log they send you will contain just the memory addresses of the functions in the call stack. However, if you have the dSYM file that matches the build they were running, you can "symbolicate" the crash and get a human-readable call stack out of it!

Symbolicating a Crash Log

Lucky, there exists a script that will help you to symbolicate a crash! It's called symbolicatecrash. You can find it here:

/Developer/Platforms/iPhoneOS.platform/Developer/Library/Xcode/Plug-ins/ ➡

iPhoneRemoteDevice.xcodeplugin/Contents/Resources/symbolicatecrash

NOTE: In iPhone OS 3.0, symbolicatecrash has been moved to a new location: /Developer/Platforms/iPhoneOS.platform/Developer/Library/PrivateFrameworks/Darwin.framework/Versions/A/Resources/.

However, you might want to copy it into a location that's part of your path so that you can just execute it by typing **symbolicatecrash** into a terminal instead of the whole path.

CAUTION: There are known bugs in symbolicatecrash for iPhone OS 2.x. Bryan Henry has posted a fixed version of the script at <http://openradar.appspot.com/6438643>.

You run symbolicatecrash from the command line. Open a terminal window, and pass it a crash file as a parameter. If you need to symbolicate it against a specific dSYM file, you can pass that in as an optional parameter:

```
> symbolicatecrash MyApp.crash Build1234.dSYM
```

That should dump out a call stack that's in a human-readable format, allowing you to see where your app crashed. It should turn something like this:

```
Thread 0 Crashed:
0  OpenAL    0x33abdbb8 0x33aac000 + 72632
1  OpenAL    0x33ab77c8 0x33aac000 + 47048
2  Dapple    0x000046ce 0x1000 + 14030
```

into something like this:

```
Thread 0 Crashed:
0  OpenAL    0x33abdbb8 0ALSource::Play() + 76
1  OpenAL    0x33ab77c8 alSourcePlay + 224
2  Dapple    0x000046ce 0x1000 + 14030 SoundEngineEffect::Start() line 1047
```

That top of a call stack is from the nastiest crash I encountered during the development of Dapple. I'll talk more about this particular crash in the "Reproducing Rare Crashes" section.

Using atos

If you're having problems with symbolicatecrash or if you want to go at it old-school, you can try your hand with atos. This is the command that symbolicatecrash uses at its heart. atos lets you find a symbol name for a given memory address.

To use atos, you need to put an app and its corresponding dSYM in the same directory. You'll want to copy the app and dSYM file that you archived when you sent the build to your testers into the same directory as your crash log file. Copy just the memory addresses from the crash log into a new text file, and place that file in the same

directory. On a terminal, navigate to the directory where you have everything, and issue a command like this:

```
atos -o MyApp.app/MyApp -arch armv6 -f stackAddresses.txt
```

NOTE: You need to run `atos` on the actual app binary, so you need to pass it the path to the binary inside the `.app` file. The `.app` file is actually a directory (in other words, a bundle) that contains a bunch of files. The binary is found inside the bundle.

NOTE: The architecture I passed in was `armv6`. If you're working with an iPhone 3GS, you may need to use `armv7`.

The result should be a dump of all the function calls that happened inside your app. `atos`, in this case, won't pull up the symbols for any of the framework methods that were called. However, it should give you enough information to move forward.

Reproducing Rare Crashes

What if the crash your testers are reporting is extremely rare and you've never seen it yourself? Fixing a bug can really be done only once you know how the bug happens in the first place. The first thing you should do is attempt to reproduce the crash.

Sometimes when you get a crash log from a tester, they will have sent you very detailed reproduction steps to make the crash happen, and therefore you'll be able to reproduce it easily. However, often you'll get an email that says, "I was doing something, and it crashed. I think it was after I hit this button." Sometimes you might get nothing at all. If this is the only person who has ever seen the crash and it happened only once, you might have trouble reproducing (*reproing*) the bug. If you have a symbolicated crash log, it's the only thing you have to go on, so use it.

If you have a crash log from a rarely occurring crash, here are some things you want to think about while you're examining the crash log:

- Which thread is crashing
- Which app system the crash occurred in
- Race conditions

Thread

The first thing you want to do is look at which thread is crashing. This will give you important clues about what might be causing the problem. Is it the main thread that

crashed? Is it the audio thread that crashed? Is it one of your other spawned threads that crashed?

Knowing which thread died is important, not only because you'll want to be looking at that thread's call stack but also because it will give you an idea of what might have gone wrong.

System

Now that you know which thread crashed, look at the system that the crash occurred in. Did it die in the rendering system? Audio system? Animation system? Some other system? This will help you narrow your scope when you're trying to reproduce the crash.

Race Conditions

Race conditions is almost a dirty word, but it needs to be mentioned. Look at all the threads that are running, and look at their states. Are two threads in the same system at the same time? Are two threads both trying to do something that could cause a race condition? Always be aware of thread interactions if you have an app that's using multiple threads of execution.

The Scientific Method of Debugging

You've had a chance to examine the crash log, and you've thought about things. I like to use a simplified version of the scientific method to track down these kinds of bugs. Yes, debugging is just like high-school science class:

1. Form a hypothesis.
2. Create a test for your hypothesis.
3. Prove or disprove your hypothesis.

By following these three steps, your goal is to find a way to increase the probability of the crash and therefore determine the cause more quickly.

Forming a Hypothesis

The first step is to form a hypothesis based on all the available data. Based on the crash log information and what your tester has told you, come up with an idea of what might be going wrong. It should be something specific.

To help illustrate this, I'm going to use an example that occurred late in the development of Dapple. Just after the alpha phase, I received a crash log from one of my testers. He said that he was playing the game normally, he scored a big combo, and the game crashed. He was nice enough to provide me with a crash log. The main thread had crashed trying to play a sound effect. The top of the call stack looked like this:

```
Exception Type: EXC_BAD_ACCESS (SIGSEGV)
Exception Codes: KERN_INVALID_ADDRESS at 0xc0000003
Crashed Thread: 0
```

Thread 0 Crashed:

```
0  OpenAL    0x33abdbb8  OALSource::Play() + 76
1  OpenAL    0x33ab77c8  alSourcePlay + 224
2  Dapple    0x000046ce  0x1000 + 14030  SoundEngineEffect::Start() line 1047
```

What was curious to me was that the crash happened inside OpenAL and not inside my own sound code. Given that this happened after a large combo, my hypothesis was this: “The crash happened because OpenAL was given too many sound effects to play at once.”

Creating a Test for Your Hypothesis

Once you have a hypothesis, you need to design a test for it. If you know the repro steps for the crash, and you can get it to happen, great. However, with rare crashes, only one person might have seen it happen once. Or perhaps it has happened only a handful of times. In these cases, I often find it helpful to create specific tests to hammer only the hypothesis.

Returning to the Dapple example, I had suspected that the crash was being caused by too many sound effects being played at the same time. I had 20 people testing the game, and they had been playing it for weeks. The crash had happened twice. I knew that I wouldn’t be able to rely on being lucky enough to happen to catch it in the debugger. So, I created some test code. The pseudo-code for the test was this:

```
Loop 1000 times
{
    PlaySound(1)
    PlaySound(2)
    PlaySound(3)
    PlaySound(4)
    PlaySound(5)
}
```

I made sure that the loop was called every single frame of execution. The result would be 1,000 PlaySound calls being made every frame for each of 5 different sound effects, totaling 5,000 sound calls per frame.

TIP: This is one of those places where those debug #defines I mentioned earlier come in handy. In this case, I wrapped the code with an #if (ENABLE_AUDIO_CRASH_TEST) so that I could enable the test any time I wanted. This also allowed me to disable it for shipping without losing the test code.

Proving or Disproving Your Hypothesis

Once you have written your test code, let it run! If you think the crash is timing or memory related, you may need to let it run a while. If your test code reproduces the crash, then great! You now have a way to repro the crash quickly, which also means that you have a way to test any potential fix you put into your code.

I built my test code and let it run. After two minutes, the program crashed with the same crash log my tester had sent me! Success! I now had a way to reproduce the bug, and it seemed to prove my hypothesis. I put some code into place to stop the audio system from playing the same sound more than once per frame. When I reran the test with the new fix in place, I let it run for 30 minutes without a crash. I called it fixed.

Increasing the Probability of the Crash

The important lesson here is that, in these rare crash cases, if you can increase the probability of the bug happening in your test case, you can more quickly determine the cause and more quickly test a solution.

It is possible that with a rare crash like this you may never feel completely confident that the bug has been fixed. The best you can do is prove to yourself, if your test case is sufficiently rigorous, that you have fixed the bug with a high enough probability.

So, You Have a Call Stack

So, you've symbolicated the crash file your tester sent you, or you managed to catch the crash in the debugger. You have a handy call stack that shows you where the app crashed. Great! You've looked at all the obvious answers, but none of them yields any results. Now what?

What I'm going to do now is walk you through several techniques for debugging some of the nastier, more obscure memory bugs you might encounter. These kinds of bugs are the ones that I find the most challenging, so it's good to have as many tools as possible at your disposal for recognizing them and then tracking them down.

To start, you'll write some basic code that I'll have you add to as we go. Each time you add some code, it will cause a very specific kind of bug, and I'll show you a new technique for finding the bug.

Starting Code

In Xcode, create a new iPhone view-based project, and call it MemoryBugs. First, add a new NSObject class, and call it TestClass (have it create the source and header files for you).

TestClass.h

```
#import <Foundation/Foundation.h>
@interface TestClass : NSObject {
    NSString* myString;
}

@property (nonatomic, retain) NSString* myString;

- (void)doNothing;

@end
```

TestClass.m

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

@synthesize myString;

- (void)doNothing
{
    // Do nothing
}

- (void)dealloc
{
    [myString release];
    [super dealloc];
}

@end
```

The TestClass you just created is just a simple class I'll use to demonstrate some memory problems you can run into with an Objective-C class.

Now create a C++ class. The easiest way to do this is add a new file based on NSObject. When asked for the class name, enter **TestCPPClass.mm** (note the *mm* extension) and have it generate the corresponding header. Replace the entire contents of the generated files (removing all the Objective-C that Xcode generates automatically for you) with the following code.

TestCPPClass.h

```
class TestCPPClass
{
public:
    // Constructor
    TestCPPClass();

    // Destructor
    ~TestCPPClass();

    void DoSomething();

    void ForceBufferOverrun();
private:
```

```

    int mSomeNum;
    int mOverrunMe[16];
    int mIGetStomped;
};

```

NOTE: If you're not used to C++, make sure you put that semicolon (;) at the end of the class declaration, after the closing brace!

TestCPPClass.mm

```

#import "TestCPPClass.h"

TestCPPClass::TestCPPClass()
: mSomeNum(0)
, mIGetStomped(0)
{
    mIGetStomped = -1;
}

TestCPPClass::~~TestCPPClass()
{
}

void TestCPPClass::DoSomething()
{
    ++mSomeNum;
}

void TestCPPClass::ForceBufferOverrun()
{
    // Write one too many ints into the array
    for (int i = 0; i < 17; i++)
    {
        mOverrunMe[i] = i;
    }

    // The loop above will have written the value "16" into mIGetStomped,
    // since it lies directly after the array in memory.
    NSLog(@"mIGetStomped = %d", mIGetStomped);
}

```

This class will be used to demonstrate how to track some bugs you can run into in C++ code.

NOTE: For your ViewController class to be able to use this C++ class, you need to rename `MemoryBugsViewController.m` to `MemoryBugsViewController.mm`. This tells the compiler to treat it as an Objective-C++ class. Alternatively, in Project view, you can click `MemoryBugsViewController.m` and select Get Info. On the General tab, change the File Type drop-down from `sourcecode.c.objc` to `sourcecode.cpp.objcpp`.

Find your `MemoryBugsViewController` class, and add a private member variable for the C++ class you just created in the header.

MemoryBugsViewController.h

```
...
@interface MemoryTestViewController : UIViewController {
    struct TestCPPClass* mTestCpp;
}
...
```

Note the use of the keyword `struct`. This is your C++ class. To use it within an Objective-C class, it needs to be declared with the `struct` keyword. If you leave the `struct` out of the declaration, the compiler will generate an error.

At the top of your `MemoryBugsViewController` class implementation, include the following two classes.

MemoryBugsViewController.mm

```
...
#import "MemoryBugsViewController.h"
#import "TestClass.h"
#import "TestCPPClass.h"
...
```

Finally, inside your `MemoryBugsViewController` class implementation, find the `viewDidLoad` method, and uncomment it.

MemoryBugsViewController.mm

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

You will be adding code to `viewDidLoad` each time you look at a new kind of bug.

What Is a Memory Stomp?

If you look at the `TestCPPClass` code, you'll notice that I mention that one of the values will get "stomped." The term *memory stomp* is used when something changes a value in memory that it wasn't supposed to change.

Mem stomps (as I'll refer to them) occur in a lot of different ways, but in my experience these are the three most common ways you'll encounter a mem stomp:

- Buffer overruns
- Calling a method on an object that has been deleted
- Returning from a callback into an object that has been deleted

Buffer Overruns

This is one of the most common ways to cause a mem stomp in C/C++. However, you'll recall that at the beginning of the chapter I mentioned that NSArray and NSMutableArray make this hard to do. This is because once you've create an NSArray, you can't change the contents, and the NSArray will generate errors if you try to write out of bounds of the array. This is extremely useful. The NSMutableArray, on the other hand, will grow in a safe way if you try to write beyond the bounds of the existing array, making things safer for you (at a potential performance cost).

However, in C/C++, overrunning a buffer is as simple as iterating one too many times in a loop. For example:

```
int myInts[8];
for (int i = 0; i <= 8; i++)
{
    myInts[i] = i;
}
```

If you're not thinking about what you're doing, that code might look OK at first glance, but I just wrote nine values into an array that can hold only eight values. Whatever 4 bytes live in memory right after my array just got the integer value 8 written into them (see Figure 4-5).

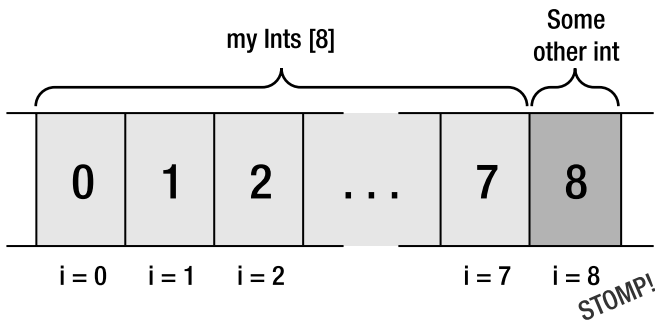


Figure 4-5. An example of a buffer overrun

Buffer overruns are tricky because oftentimes there might not be anything allocated right after the buffer in memory and so it won't have any noticeable effect. Or you might end up writing a value into memory that is within an acceptable range for the variable you just stomped.

I'll go into more detail on tracking these bugs down later.

Calling a Deleted Method

This kind of mem stomp is less common, but I've seen it happen a fair bit. The usual sequence of events is as follows:

1. An object is created.
2. The object is used.
3. The object is deleted.
4. Something that didn't know the object was deleted calls a method on that object.

Sometimes your app will crash immediately when this happens. However, sometimes the object will have been deleted, but the memory on the heap hasn't been allocated to something else yet. So, the result is that the variable still points to what looks like an object in memory, and the function call "works." In doing so, if that method starts changing member variables, it can now be changing memory on the heap that it doesn't own anymore, resulting in a stomp of other newly allocated objects.

For an example of one of these situations, see Figure 4-6.

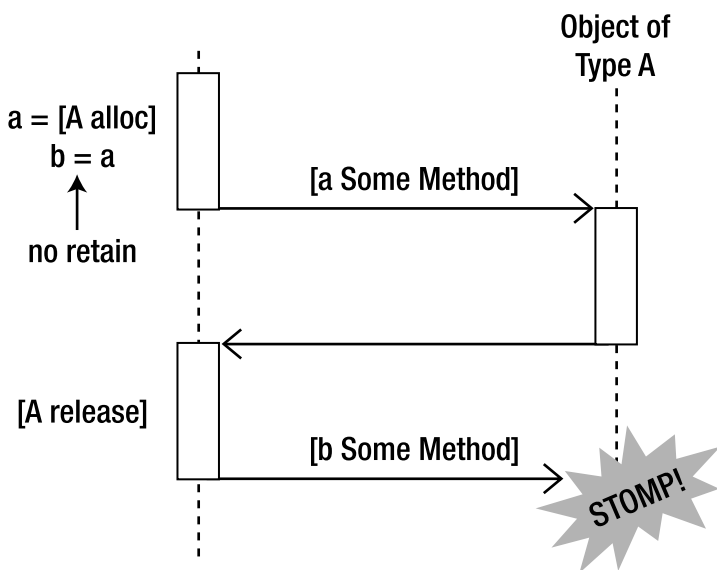


Figure 4-6. An example of calling a method on a deleted object

Like I said, this one isn't as common, but it's important to be aware that it can happen so that you at least look for it once you've exhausted other options.

Returning to a Deleted Object

This kind of mem stomp is a lot harder to have happen in Objective-C than C/C++ because of the reference counting of pointers that takes place. Using retain and release means that objects that are still needed don't get deleted too soon. However, it's quite easy to get this to happen in C/C++ when you're dealing with callbacks and function pointers, and you can get it to happen in Objective-C if you're not managing your memory properly.

The most common place I've run into this kind of mem stomp is with animation systems. The usual sequence of events is something like this:

1. Object A creates an animation instance.
2. Object A sets a callback to itself so that it is notified when the animation completes.
3. Object A triggers the animation.
4. The animation runs.
5. The animation completes and calls Object A's callback.
6. Inside the callback, Object A causes an app state change that requires that the animation gets deleted.
7. The animation object gets deleted.
8. The callback completes and returns into the animation object (which has now been deleted).
9. The animation object does some cleanup code, which involves modifying some member variables.
10. What it's actually doing is changing memory that has been allocated to something else now.

I have typically run into these kinds of problems when transitioning between front-end menu screens and in-game state (and vice versa) in games. It's not uncommon for an animation to play in response to user input and at the end of the animation trigger a state change. For a simplified example, see Figure 4-7. Of course, this kind of mem stomp can occur in other ways, so just be aware of it.

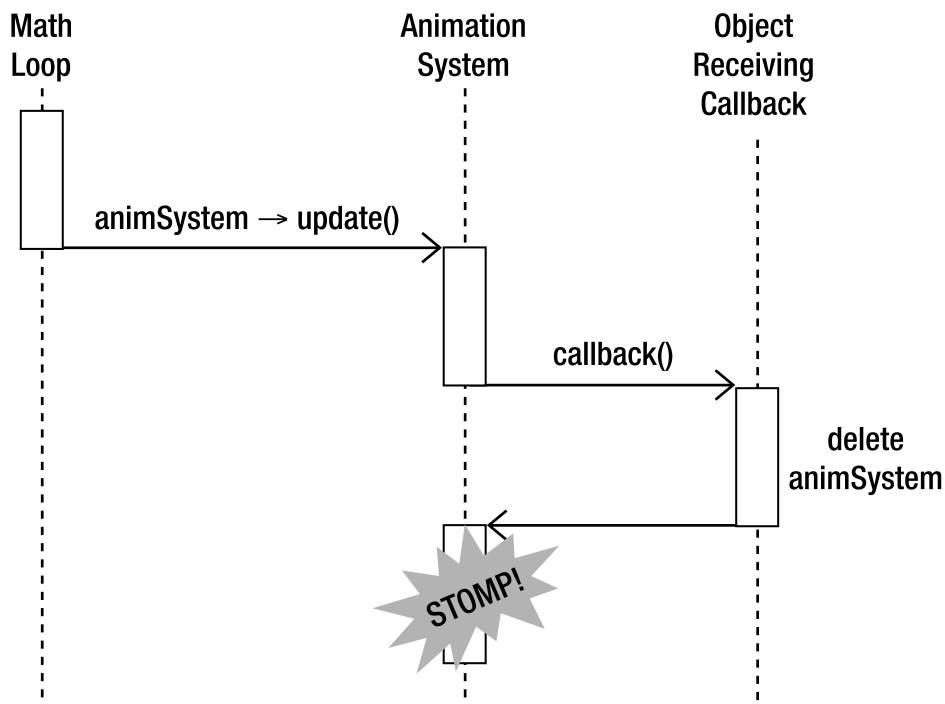


Figure 4-7. An example of returning from a callback into a deleted object

Identifying a Mem Stomp

Mem stomp bugs are fairly rare, compared to other kinds of bugs, but they can be the most difficult to track down and fix. One of the important first steps is learning how to recognize when something might be a mem stomp. However, it is also important not to leap to the mem stomp conclusion too soon. Look for the simple explanations first. Then if you can't find a simple explanation, start looking at the harder ones.

Here are a few things to look for that *might* indicate a mem stomp:

- The app suddenly crashes, but the crash happens at a different point each time you run the app. (This could also indicate a timing/threading problem or could indicate multiple different crashes.)
- The app behaves in a random, incorrect manner after consistent events. (For example, your game starts with a random, nonzero score every time you load into the game from the menus.)
- Unexpected values show up in variables after unrelated events. (For example, game state data changes after you load an image into memory.)

- You see the following message show up in your console: `malloc: *** error for object 0XXXXXX: Non-aligned pointer being freed` (see the “Enable Guard Malloc” section for more information).

CAUTION: A memory stomp won’t always cause a crash. Oftentimes it will just cause strange behavior. If the stomp writes a valid value into some other variable, the program might not crash; it might just behave erratically.

During the development of Dapple, I ran into a strange problem: upon entering the game, playing for a short time, and then quitting back to the main menu, sometimes one of the menu items would show up in white, as shown in Figure 4-8. It turned out to be a stomp being caused by a callback returning to the animation system after the state change from in-game to the front end had occurred, after the animation system had been deleted. The animation system was stomping over an image object in memory.



Figure 4-8. An example from Dapple of the strange behavior that can occur as a result of a memory stomp

Tools to Detect Memory Problems

Now I’ll walk you through several tools at your disposal for tracking down nasty memory-related bugs.

`malloc_error_break`

`malloc_error_break` is a symbol that you can, and *should*, set a breakpoint on. This handy method will print out warning messages to your console when certain memory

“weirdness” occurs. The log message will advise you to set a breakpoint on the function and rerun the app. I recommend that you *always* have a breakpoint set on this symbol. It will help you track down several kinds of memory problems as soon as they occur!

TIP: Always have a breakpoint set on `malloc_error_break` to catch certain memory problems as soon as they occur.

`malloc_error_break` will tell you when a few important things happen:

- You have double-released an Objective-C object.
- You try to release memory that may have been stomped.

At this point, crack open `MemoryBugsViewController.mm`, and add the following code to `viewDidLoad`:

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

    // Test case for double release - malloc_error_break
    NSDate* date = [NSDate date];
    [date release];
    [date release];
}
```

In this example, you can clearly see that `date` is being released twice (three times if you count the autorelease that would have happened at the end of the method). However, in your own, more complicated app, the fact that something is being released twice might not be so obvious.

Compile and run the code, and you should see that the app actually crashes. However, there are instances where your app will double-free (that is, attempt to release or free the same memory twice) and continue merrily on its way, even though you’ve done something potentially dangerous. If you open the console window (Shift-Command-R by default in Xcode), you’ll see that you should have something like this printed after running:

```
MemoryBugs(5745,0xa04cc720) malloc: *** error for object 0x525250: double free
*** set a breakpoint in malloc_error_break to debug
```

The app is telling you to set a breakpoint in `malloc_error_debug`, so do that now. To set a breakpoint on a symbol for which you don’t have source code, you’ll use the Breakpoints window. Open the Breakpoints window (Option-Command-B by default). Scroll to the bottom of the window, and you should see a blue box with the text “Double-Click for Symbol” next to it, as shown in Figure 4-9.

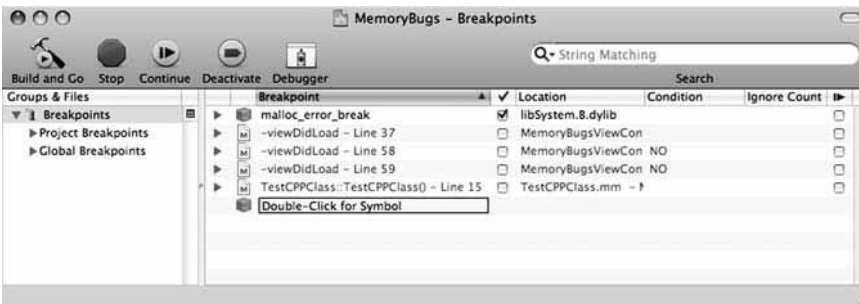


Figure 4-9. Adding a symbol breakpoint

Double-click in that box, and you'll be prompted to enter a symbol name. Enter **malloc_error_break**, and press Enter. The Breakpoints window should now show a new breakpoint set on that function.

Run the app again (via **Run ► Debug**), and this time, you should hit the breakpoint. If you look at the call stack in the debugger, you should see something like Figure 4-10.

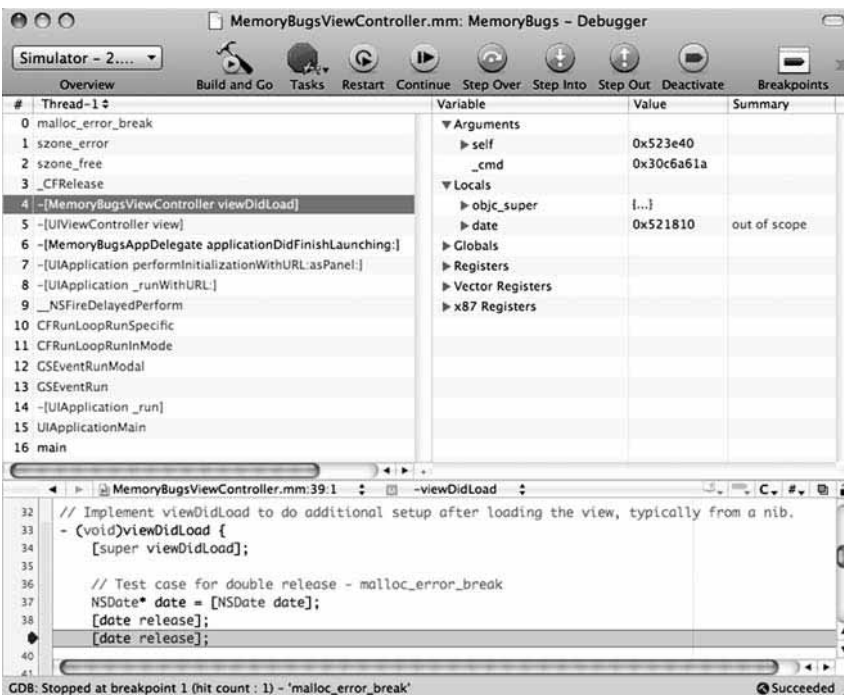


Figure 4-10. Debugger halting on second release because `malloc_error_break` was triggered

If you click the call stack at `[MemoryBugsViewController viewDidLoad]`, you'll see that the breakpoint was hit on the second release of the date object. This tells you exactly where the problem occurred. Now you can figure out why it was released twice and fix your bug!

NSZombieEnabled

NSZombieEnabled is a fantastic tool for tracking down tricky memory problems with Objective-C objects.

NOTE: NSZombieEnabled can be used when debugging both in the simulator and on a device.

NSZombieEnabled is an environment variable that you set up for your app. What it does is tell the app to never actually release memory when you call `release`. Instead, objects that get released have their types changed to `_NSZombie`. The result is that if something tries to act on that object after it has been freed, the debugger will break to the line that caused the error, instead of potentially crashing somewhere completely different.

CAUTION: Never, *ever*, leave this turned on when you don't need it. It means that your memory allocations aren't freed properly and could result in your app using a significant amount of memory. I recommend that you set up the argument and leave it disabled, except when you want to test with it enabled.

First, add the following code to the `MemoryBugsViewController` class's `viewDidLoad` method, and comment out the test from the previous section:

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

    // Test case for double release - malloc_error_break

    /*
        NSDate* date = [NSDate date];
        [date release];
        [date release];
    */

    // Test case for NSZombie
    TestClass* testInstance = [TestClass alloc];
    [testInstance release];
    NSLog(@"Test Class's myString = %@", testInstance.myString);
}
```

The example code is clearly doing something stupid: it's trying to get the value of a property for an object that has already been released. If you build and run this code in the debugger, it should crash. In this case, the call stack should be inside `viewDidLoad`, and it's easy to see why it crashed. However, this kind of bug won't always crash at the

point where the problem occurred. Sometimes it will crash later or in a different area of code. This can make tracking down the line that caused the problem quite difficult.

However, NSZombieEnabled comes to the rescue! To enable this handy tool, in the project viewer find your MemoryBugs executable in the Groups & Files pane, and double-click it to open up the executable info window (see Figure 4-11).

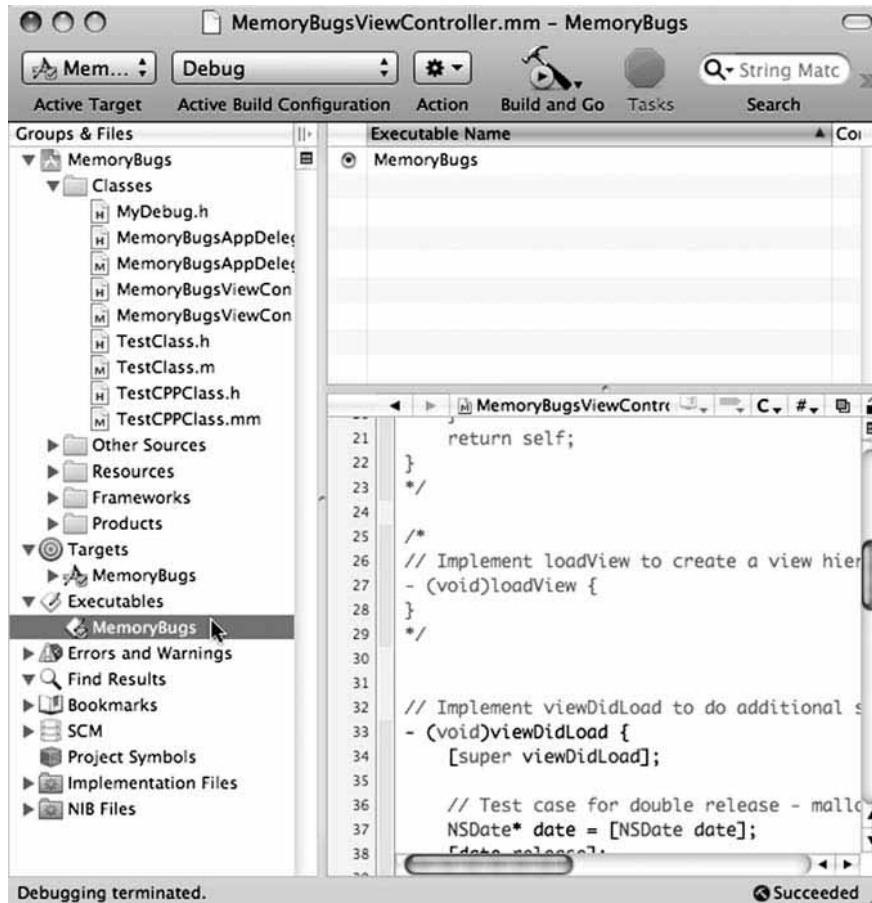


Figure 4-11. Double-click the MemoryBugs executable to bring up the executable info window.

When the executable info window opens, click the Arguments tab at the top of the screen. The bottom pane of this window should be labeled “Variables to be set in the environment.” Click the + at the bottom to add a new variable. Name it NSZombieEnabled, and set its value to YES (see Figure 4-12).

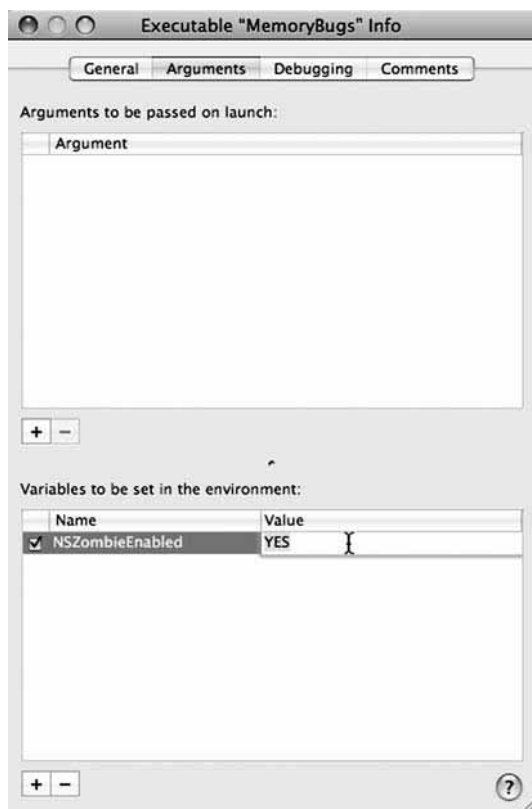


Figure 4-12. Setting *NSZombieEnabled*

Now build and run the app again. You should see the following print out in the console window:

```
2009-05-04 10:26:53.905 MemoryBugs[19558:20b] *** -[TestClass myString]: message  
sent to deallocated instance 0x536560
```

and program execution should halt. If you look at the call stack, you'll see that it halted at the line where the program tried to access the released object.

Although this is a very simple example, *NSZombieEnabled* can help you track down much more complicated memory violations. It's a great place to start if you're seeing what you suspect is a memory stomp.

However, sometimes this won't turn anything up. Perhaps your bug is being caused by a C/C++ class instance. If this is the case, then *NSZombieEnabled* won't be able to help you, because it tracks only Objective-C allocations and releases. If the problem has to do with an object that has been newed/deleted, then you need a different tool.

Enable Guard Malloc

Enable Guard Malloc to the rescue! Maybe! Enable Guard Malloc is a similar tool to `NSZombieEnabled`, but it tracks problems with `new` and `delete` or with `malloc` and `free`. This can be used to track down memory violations in your C++ classes.

Enable Guard Malloc puts memory guards around memory every time it is allocated or freed. The net effect of this is that it can detect when something tries to use memory that has been freed/deleted. This is very useful for tracking down those callback bugs mentioned earlier.

TIP: Have you ever see the following error message print out in your console: `malloc: *** error for object 0XXXXXX: Non-aligned pointer being freed? Then you most likely have a memory stomp. By default, the iPhone allocates new memory to aligned boundaries, so if it tries to free memory that's not aligned, there's a good chance that a memory stomp has occurred. If you see that error message, turn on Enable Guard Malloc and rerun the app in the debugger. It may help you find the problem.`

Add the following to your `viewDidLoad` code, and comment out the last code you added:

```
// Implement viewDidLoad to do additional setup after loading the view, typically
// from a nib.
- (void)viewDidLoad {
    ...

    // Test case for NSZombie

    /*

    TestClass* testInstance = [TestClass alloc];
    [testInstance release];
    NSLog(@"Test Class's myString = %@", testInstance.myString);

    */

    // Test case for Enable Guard Malloc - delete and use
    mTestCpp = new TestCppClass();
    delete mTestCpp;
    mTestCpp->DoSomething();
}
```

Again, this is clearly a simple example; an object is being used after it has been deleted. However, this kind of thing can crop up in code quite easily without it being so obvious. This can be especially problematic if two threads act on the same object in a non-thread-safe way. This is also quite easy to do in complex callback mechanisms.

Make sure that `NSZombieEnabled` is still set to `YES`, and then build and run the code. Did you see what happened? Absolutely nothing. The program ran fine. `NSZombieEnabled` wasn't able to catch this because you used `new` to instantiate your C++ class. Also

notice that the program didn't crash. This is where memory problems can be awful to track down. Depending on what `DoSomething()` is actually doing and when it was called after `mTestCpp` has been deleted, it could have stomped over memory that had been allocated to a different object.

First, disable `NSZombieEnabled` (refer to the previous section to see how to find it) by deselecting the check box next to it. Now turn `Enable Guard Malloc`.

NOTE: Unfortunately, `Enable Guard Malloc` can be used only with the simulator. It cannot be used to debug on your device.

Make sure you set your target to build for the simulator. Then go to the `Run` menu and turn on `Enable Guard Malloc`, right at the bottom of the menu, as shown in Figure 4-13.

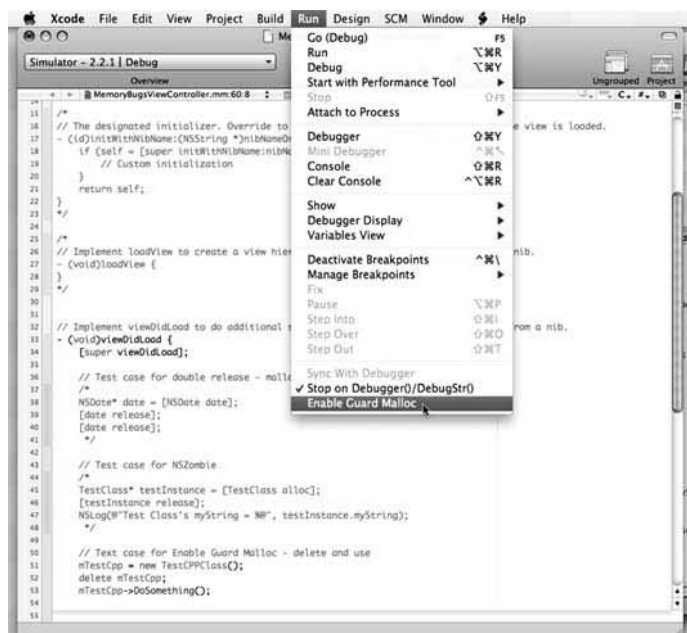


Figure 4-13. Activating `Enable Guard Malloc` for the simulator

TIP: If `Enable Guard Malloc` is grayed out in the `Run` menu, it is probably because the build target is still set to `Device` instead of `Simulator`. Make sure the target is set to `Simulator`, and try again.

Build and run the app again. This time the app should halt in the debugger. If everything went according to plan, then the debugger should have halted at this line:

```
void TestCppClass::DoSomething()
{
```

```
++mSomeNum; // Debugger should have halted here  
}
```

The debugger is halting here this time because it's at this point that Enable Guard Malloc has caught something trying to write to memory that has been freed. In this function, `mSomeNum` is a member variable of an object that has been deleted. By trying to increment it, what's actually happening is that some 4-byte chunk of memory that no longer belongs to the `mTestCpp` object is being incremented.

CAUTION: Running with Enable Guard Malloc turned on will most likely cause any remotely complex app to run *extremely* slowly, because it does extra processing for every memory allocation and free. Turn it on only when you need it to track down a problem.

In this example, because `DoSomething()` is being called immediately after the object was deleted, chances are this isn't going to do anything dangerous. However, if the call to `DoSomething()` was made after other memory had been allocated on the heap, this code might now be incrementing memory that belongs to some other object. Memory stomp!

Luckily, Enable Guard Malloc caught the problem as soon as something tried to access memory it didn't own anymore. However, it's up to you to figure out why this object was deleted before you thought you were finished with it. The best way I know to do this is to put a breakpoint in the object's destructor and watch where it gets hit. From there you can usually track it back to problem.

CAUTION: Enable Guard Malloc won't find all of your memory stomps. It will find instances only where memory that has been marked as freed is changed.

Watching Variables

I've covered a few tools that are available to you for tracking down some memory bugs. However, sometimes a stomp will happen without causing any of the memory violations that the previous tools will detect. In this case, the bug can be extremely tricky to track down. The first step in the process is determining which variable (or variables) is being stomped. That is left as an exercise for you, because that's just good, old-fashioned debugging.

If you know that a particular variable is being stomped (or even just changed and you don't know why), one of the most useful tools for debugging this is a variable watch. Xcode allows you to put a watch on any given variable. Setting a watch on a variable halts execution any time something changes the value stored by the variable.

Watches can be incredibly useful if something unexpected is changing the value of one of your variables in memory. It can also be useful if something is stomping memory and you know what memory is being stomped, but you don't know from where.

CAUTION: If the variable that is being changed gets changed multiple times per frame of execution in a valid way, then setting a watch probably won't tell you much, because execution will halt so frequently that you can't tell what's going on. If this is the case, the first thing to do is look at the class declaration for clues about what sits next to it in memory. If you're dealing with a global object, or code memory, the "Map Files" section might help.

Open `MemoryBugsViewController.mm`, and add some more new code to the `viewDidLoad` method (and comment out the old code again):

```
// Implement viewDidLoad to do additional setup after loading the view, typically
// from a nib.
- (void)viewDidLoad {
    ...

    // Text case for Enable Guard Malloc - delete and use
    /*

    mTestCpp = new TestCPPClass();
    delete mTestCpp;
    mTestCpp->DoSomething();
    */

    // Test case for buffer overrun
    mTestCpp = new TestCPPClass();
    mTestCpp->ForceBufferOverrun();
    delete mTestCpp;
}
```

If you look at the code in `ForceBufferOverrun()`, you'll see that the function writes 17 ints into an array that's only of size 16. This stomps the contents of the member variable directly after the array in the class. If you look at the class header, you'll see that `mIGetStomped` sits directly after the array in memory, so it's what gets stomped.

If you build and run this code, you'll see that everything runs totally fine. Look at the console output, though, and you should see this:

```
mIGetStomped = 16
```

Nowhere in the code do you explicitly set the value of `mIGetStomped` to 16, but the buffer overrun does that. If you run this code with `NSZombieEnabled` turned on or `Enable Guard Malloc` turned on, it will still run fine. This is because you're not trying to access freed memory. The method stomped only the memory that belongs to the same class, so the previous tools I covered don't do anything.

However, all is not lost! This is one of those situations where watching a variable can pinpoint exactly what's going on. I will assume that you've found out that it's the contents of `mIGetStomped` that are being stomped (which is why you added that handy

NSLog into the function—how thoughtful!). To figure out what is doing the stomping, set a watch on `mIGetStomped`. To do this, first set a breakpoint in the constructor for `TestCPPClass` so that the program will halt execution somewhere before the stomp happens. Build and run in the debugger.

The app should halt when the `TestCPPClass` object is instantiated for the first time, in the constructor, where you put your breakpoint. Open the debugger window (Command-Shift-Y by default in Xcode), and find the `mIGetStomped` variable in the Variable pane. Right-click (or Ctrl-click) `mIGetStomped`, and select `Watch Variable` from the menu (see Figure 4-14).

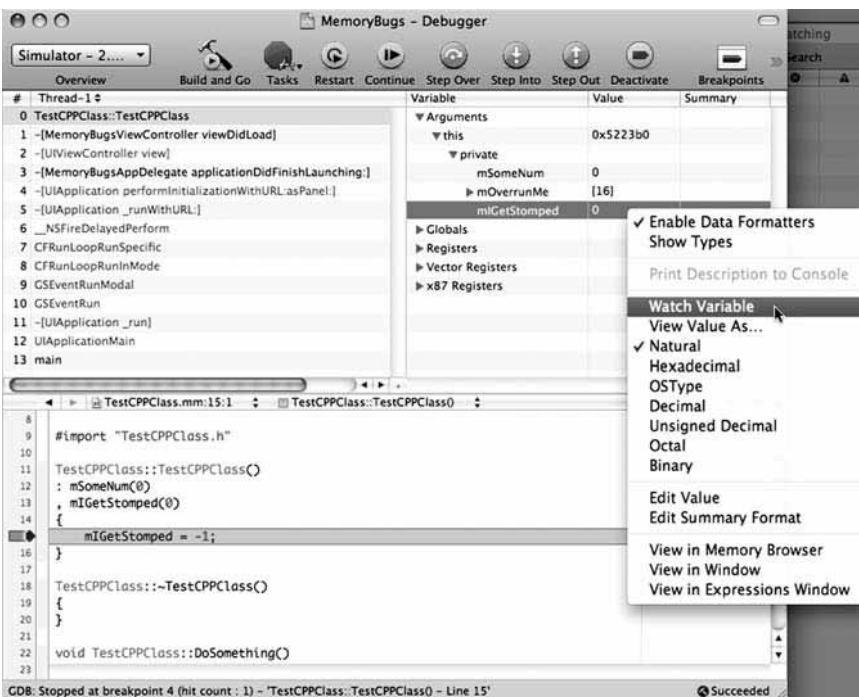


Figure 4-14. Setting a watch on a variable

Once the watch has been set, a small magnifying glass icon will appear next to the variable in the debugger (see Figure 4-15).

Variable	Value	Summary
▼ Arguments		
▼ this	0xb482ffb0	
▼ private		
mSomeNum	0	
▶ mOverrunMe	[16]	
🔍 mIGetStomped	0	
▶ Globals		

Figure 4-15. The magnifying glass next to `mIGetStomped` tells you that it's being watched.

Now that you have set the watch, continue execution of the program. If you set your first breakpoint on the line in the constructor where `mIGetStomped` is assigned the value `-1`, then you will see the program halt as soon as `mIGetStomped` becomes `-1`. This is expected, so click OK, and then continue execution of the app.

The program execution should halt a second time with a message that looks like what you see in Figure 4-16.

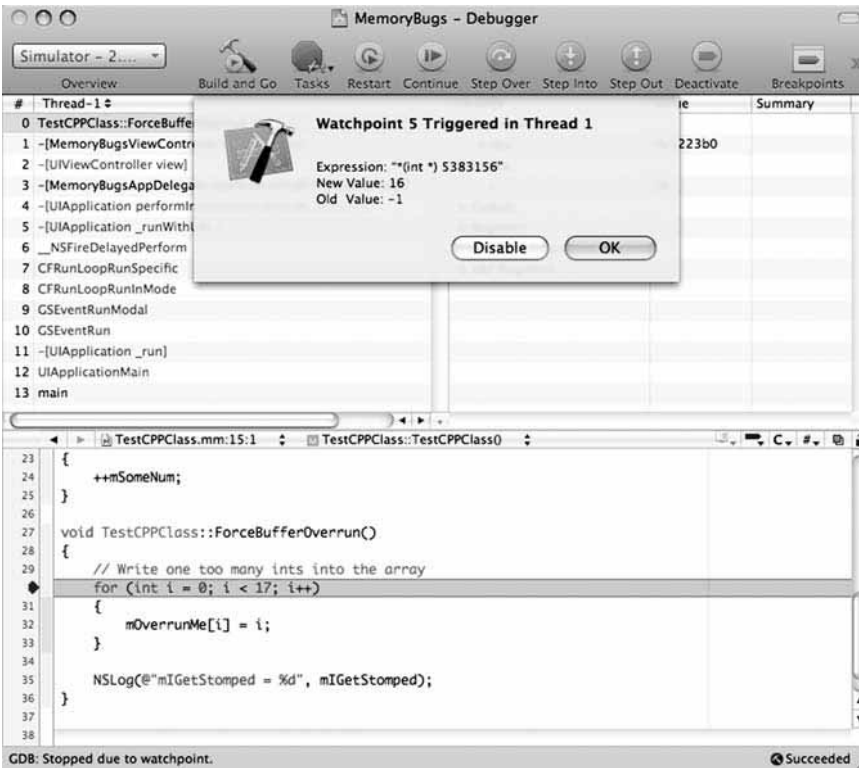


Figure 4-16. The watched variable detecting a bad change in value

Clearly, `mIGetStomped` should never be assigned the value 16, so this is where the stomp occurred. Click OK, and look at where the program counter is. It will have halted at the next instruction after the line that caused the value to change. Look at the value of `i` in the debugger, and you'll see that it's 16. This means that the previous instruction was when `i` was 16 and the program executed:

```
mOverrunMe[i] = i;
```

If you look at the declaration of `mOverrunMe`, you'll see that it's declared as an `int` array of size 16. This means that only indices of 0–15 are valid, so when `i = 16`, the program stomps the next 4 bytes in memory, or `mIGetStomped`.

Now that you know which line of code caused the stomp, it's just a matter of fixing the loop so that it doesn't overrun the buffer.

Again, this is a very simple example, but the method can be applied to find much more complicated memory stomps.

NOTE: The default behavior, if you put a watch on a pointer, is to just watch the value of the pointer. That means that the debugger will halt only if the address that the pointer is pointing to changes. This is often not what you want. You can instead tell the debugger to watch the entire contents of the object to which the pointer points. To do this, while debugging and at a breakpoint, find the memory address for the object you want to watch. Open the Expressions window, and use an expression in this format: `*(<class_name>*)(<mem_address>)`. For example, enter `*(TestCppClass*)(0xb482ffb0)`. Then right-click the expression, and choose Watch Variable. Now when any member of the object changes, the program will halt. Be careful, though, because this can slow the debugger significantly!

Link Map Files

But, Owen, you say. What if the variable that's being stomped gets changed all over the place in a legitimate way? I can't have the program halt every time something changes the variable!

To that I say, fine, you're right. If you're in this situation, you're in a tight spot, and tracking the stomp down is going to be hard work; I won't lie. One last tool I want to share with you is the link map file. This can sometimes point you to a problem if you've exhausted all the earlier techniques and you still don't know what's causing the problem. This is kind of a last-resort tool that I use. It's rare that I use it, and it's rare that it helps. However, it did once help me find a memory stomp I had been tracking for three days straight, so I won't discredit it.

A *link map file* is a file that you can optionally build at link time that dumps out a memory map of the symbols in your binary. The file contains a list of all the symbols in the binary along with their memory addresses, showing you how the binary will be arranged in memory when the executable is loaded.

For the previous example, the stomp happened within the class, so looking at the class declaration in the header should have given you a good idea of what was causing the stomp. If you look at the header, you'll see that the `mOverrunMe` array sits directly in front of `mIGetStomped`, so there was a good chance that a buffer overrun was causing the stomp. The variable watch confirmed that.

However, there will be some memory stomps that happen because of a global variable or code. The link map file can give you hints about what might be causing those problems.

To use a link map file, you first need to set up your build target to create the file. Open the target for your Debug build of MemoryBugs, and select the Build tab. Type **link** into

the search field, and you will get a reduced listing of build options. There are two that you should pay attention to (see Figure 4-17).

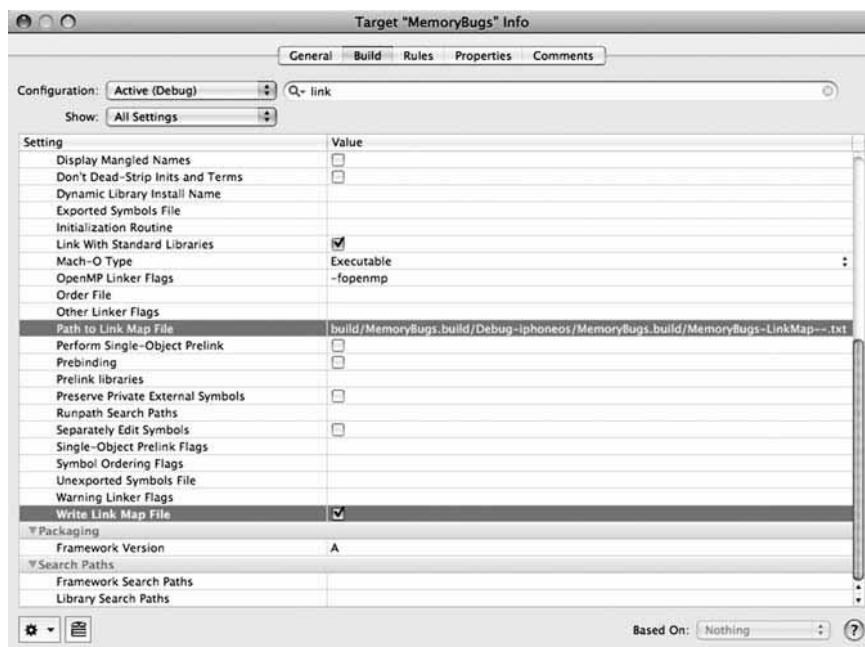


Figure 4-17. Settings to pay attention to for building link map files

The first to look at is the Path to Link Map File field. Look at the path where it will place the link map file, because you will need to find it after you build it. The second is the check box Write Link Map File, which is probably deselected. Select it now, and close the target info window.

NOTE: The link map file will be different based on the architecture you're building for. If you're tracking the bug on your device, build the link map file for the device. If you're debugging on the simulator, build the link map file for the simulator. The text file that is generated will have the architecture that it was built for in the file name (i386 for Intel Macs, armv6 for iPhone/iPod touch).

Build the app again, and then navigate in the browser to the directory that was specified in the Path to Link Map File field. Open the text file that was generated, and you'll see the map of the binary. It should look something like this (this is a link map file generated for device, which is why the memory addresses are so low):

```
...
# Address      Size          File Name
0x00002000    0x0000004C   [ 1] start
0x0000204C    0x00000020   [ 1] dyld_stub_binding_helper
0x0000206C    0x00000084   [ 2] _main
```

```

0x000020F0    0x00000090    [ 3] -[MemoryBugsAppDelegate ➡
  applicationDidFinishLaunching:]
0x00002180    0x0000008C    [ 3] -[MemoryBugsAppDelegate dealloc]
0x0000220C    0x00000028    [ 3] -[MemoryBugsAppDelegate viewController]
0x00002234    0x00000048    [ 3] -[MemoryBugsAppDelegate setViewController:]
0x0000227C    0x00000028    [ 3] -[MemoryBugsAppDelegate window]
0x000022A4    0x00000048    [ 3] -[MemoryBugsAppDelegate setWindow:]
...

```

In the file you can see the layout of all the classes and functions as well as the layout of any global data that exists.

I'll be honest: it's not often that I end up digging into a link map file to help fix a bug. However, there have been times where the link map file provided the only clue as to what was happening. For the odd time that you need that clue about what lies next to something else in memory, you'll be thankful that you know it's there.

Summary

Thanks for sticking with me through the chapter. I know that, to many people, debugging isn't the most exciting or glamorous topic in the world. However, debugging is a skill that must be learned, just like any other. It can take years of practice, and even then you'll still run into bugs you've never seen before. The programmers I've worked with who were the best debuggers are the ones who have done it the most. They're also the ones who enjoy it the most. It comes back to what I said at the beginning: if you go into it with the right frame of mind, it can make things a lot easier on you.

Don't be frustrated if the concepts I've covered aren't immediately obvious or if you're not quite sure when to use one tool over another. Just try things, and you'll gradually learn what the best approach is for a certain kind of problem.

As I wrap things up, I want to offer a couple of final thoughts. Look for patterns. Often a bug will follow a similar pattern to other bugs you've seen before. But be adaptable. Sometimes a bug will look like nothing you have seen before. Just do your best, take a deep breath, and dive in.

Good hunting!

Dylan Bruzenak



Company: *Idea Swarm*

Location: *Minneapolis, MN*

Former Life As a Developer: *IdeaSwarm, Inc. 1 year. Owner*

- *Developer of the WhatNext task management iPhone application and the AppViz iPhone Sales tracking application for the Mac.*
- *Tech: Objective-c, Core Data, SQLite*
- *Adobe Systems, Computer Scientist (consultant), 1 year, 1 month*
- *Developed task management tools, build tools, and did installer work for the Adobe Photoshop Lightroom project.*
- *Tech: Java, Ruby on Rails, SQL, InstallShield, Build Forge, Javascript, HTML/CSS, Perl*
- *Adobe Systems, Whitebox QE (consultant), 4 months*
- *Whitebox QE for the LiveCycle Java platform.*
- *Tech: Java, JBoss, IBM WebSphere, BEA WebLogic, Apache Web Server, load balancing, JMS,*
- *United Health Group, Developer (consultant), 4 months*
- *Developed the <http://www.urnparentsteps.com/> application with a small team.*
- *Tech: Java, Spring, Hibernate, Javascript, HTML/CSS*
- *Fidelity National Financial, consultant, 10 months*
- *Worked on the Touchpoint Sales and Service application for banks.*
- *Tech: XML, Java, Javascript, HTML/CSS*
- *Value Vision Media, Java Programmer, 1 year 3 months*

- *Worked on various internal applications including reporting and shipping tracking applications.*
- *Tech: Java, Swing, Spring, Hibernate, Javascript, HTML/CSS, Ruby, XML, XSLT*

Life as an iPhone Developer:

- *WhatNext – Task and List Manager*



***What's in This Chapter:** This chapter includes a walk through of developing an Active Record style database wrapper around the SQLite APIs included in the iPhone SDK. The user should leave with an understanding of SQL handling on the iPhone and the code necessary to easily incorporate this storage method into their applications.*

Key Technologies:

- **SQL**
- **SQLite**
- **Active Record**

Stick Around: Building Data-Driven Applications with SQLite

Welcome. I'll be your guide on a wondrous journey through the depths of SQLite support on the iPhone. My intent with this chapter is to demystify the C API and wrap it in some more lovable Objective-C. I will then show you how to add a deliciously simple Active Record mapping layer on top to facilitate communication with your object model.

I'm assuming that you have some knowledge of SQL going into this chapter, but you don't need anything too advanced and certainly nothing specific to SQLite. But first...

A Short Road Off a High Cliff (How I Got Here)

I entered the Objective-C ecosystem the same way many new iPhone developers do: I took a leap of faith.

By early 2008 I had entrenched myself in the Java and sometimes Ruby consulting communities, doing some interesting work interspersed between standard internal business projects. A year previously I had been lucky enough to get a contract with Adobe Systems doing tools work for the Adobe Photoshop Lightroom project in the Arden Hills, Minnesota, office. I loved it there, but unfortunately my contract was coming to an end, and I needed to look for something else to keep the spring rain from making my life very uncomfortable. I was faced with a difficult choice: return to consulting, take a full-time job offered by a friend of mine working on some exciting new stuff for a large company, or strike out on my own again as I had longed dreamed of doing. After a long couple of weeks of oscillating, staring at my bank statements, and counting and recounting my meager savings, I decided to go my own way. The siren song of developing on a platform that I loved and building software for myself again lured me

into a land of ridiculous hours and high potential. Eventually I had decided that the opportunity of the platform was too great to risk letting it pass me by.

And so, I set off to build my first iPhone application.

Ready! Set! Wait, What? (Why I Decided to Write a To-Do Application)

The first few weeks found me drifting. I was having trouble deciding on one idea to pursue. My hard drive is littered with cast-off source—skeletons of ideas that never quite made it far enough to capture my attention. I was having trouble getting things done with the call of spring outside. I was having trouble scheduling my work without the pressure of external deadlines, which I had come to rely on over the years. Consulting had made me strong in some ways but weak in others, and I was running headlong into a confrontation with those weaknesses.

When I confront a problem, the first thing that I generally do is take a stab at it myself. I try to think of a solution without a lot of external help; I think that this allows a bit more creativity and understanding of the problem space before getting locked into seeing things through the results and ideas of others. It also lets me exercise my reckless streak. This wasn't working so well for me here, so I turned to my second step: research.

I bought a number of popular books on scheduling, time and task management, and personal motivation. I hopped from system to system, never quite finding one with the correct balance of time put in to work coming out. I also needed something that worked on my phone, something that I could carry with me everywhere. WhatNext was born.

My initial design was complex and had pieces from the various systems that I had tried. After working with it for a while and assessing what I knew some of my potential competitors were working on, and with the date for the store launch hurtling toward me, I decided to go simpler. I pared WhatNext down to just my essential needs. It left an application that is simple to use and does exactly what I need it to do. Figure 5-1 shows the main view for WhatNext.

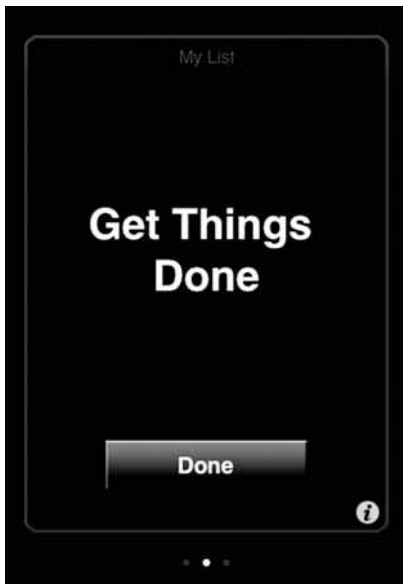


Figure 5-1. *WhatNext—a super simple task manager*

Data-Driven Applications on the iPhone

WhatNext is a data-driven application. This type of application revolves around collecting data from various sources and displaying it to the user. The source could be anything, but some typical sources are the user themselves (think of list applications), some form of web service (weather stats, RSS feeds, and so on), or a built-in data set of some kind that you might ship with your application (geological surveys, molecule data, and so).

One of the first questions that I come to when designing any data-driven application is, how do I store and retrieve the various bits of data that I'll be accumulating and displaying? Thankfully, the iPhone SDK has several good ways of storing data. One of the most versatile and robust ways is to use a SQL database.

You may already be familiar with SQL, as I was, especially if you've come from the web world or a number of other industries where it is the de facto standard for data storage. It was built from the ground up to handle the specific task of working with data, with the end result that you can often take whole paragraphs of code and reduce them to a single statement in SQL. It is also fast, well documented, and dependable.

The iPhone SDK includes a SQL database called SQLite. It is small, durable, and extremely fast, and including it is as simple as including the associated library into your Xcode project.

Active Record: A Simple Way of Accessing Data

One of my early experiments in the iPhone ecosystem involved adapting the SQLite Books example code from Apple's site to work with my own data model. The example uses raw SQL and the SQLite C APIs and was relatively straightforward to adapt, but I found myself constantly having to drop out of Objective-C and my higher-level domain model thinking to think about SQL and C. That kind of context switching was costing me development brain cycles, so I set out to solve the problem.

I ended up creating a higher-level framework that takes care of most of the raw SQL work by mapping to and from simple objects in my domain model. The framework makes it easy to create, find, save, and delete these objects while minimizing the amount of SQL that has to be written and maintained. It is based on the Active Record design pattern.

In the Active Record pattern, each database table is represented by a class in your application. The individual instances of this class represent rows in the table. There are class methods for retrieving instances of these objects, and the instances themselves contain the methods responsible for deleting, saving, and updating themselves. This provides a simple and natural API for handling most data access needs and also allows for dropping down to custom SQL if you need to do something more complex. Here's an example of working with an Active Record object:

```
GroceryItem *bread = [[[GroceryItem alloc] init] autorelease];
bread.name = @"Bread";
bread.number = [NSNumber numberWithInt: 2];
[bread save];
```

To find all grocery items, you would use this:

```
NSArray *items = [GroceryItem findAll];
```

Deleting, updating, and finding a specific item is just as easy.

In the rest of this chapter, I'll be walking you through implementing an Active Record framework. However, before you can get to this implementation, you first need to simplify working with the C APIs. I'll cover that in the next section.

Writing a Database Wrapper Around the C API: ISDatabase

Writing an Active Record implementation, even a simple one, can be fairly complex. It isn't helped by having to drop down to C every time you have to run a SQL statement. To make things a bit easier, you will first be creating a wrapper around the C APIs for SQLite to take care of some common "housekeeping" issues:

- Managing the opening and closing of the database connection
- Handling transactions (groups of SQL statements that should be run together and fail together if any single statement fails)
- Processing SQL statements and returning wrapped results
- Handling parameters
- Managing memory

The wrapper will do all of these tasks, allowing you to display a data-driven interface such as the one shown in Figure 5-2.

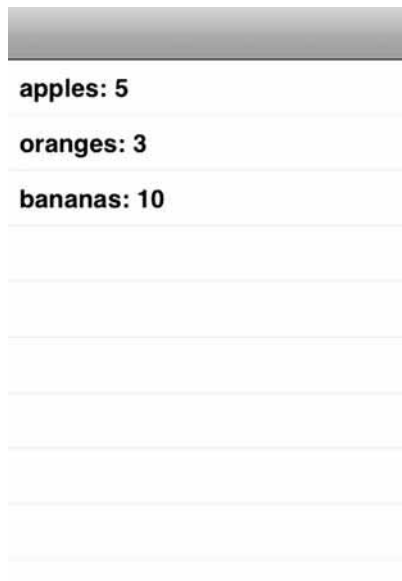


Figure 5-2. *Where you're going*

Setting Up the Example Project

First you'll need a project for the code to live in. Create a new navigation-based application using **File ► New Project** in Xcode. Name the project **GroceryList**.

Right-click the target for your application, and choose **Get Info**. Switch to the **General** tab. Click the plus button at the bottom to add a **Linked Library**, and choose **libsqlite3.dylib**. This will link in the SQLite framework. Close the info window, and drag the **libsqlite3.0.8.6.dylib** entry from the tree view in your main window to the **Frameworks** folder. Your project tree should look like the one shown in Figure 5-3.

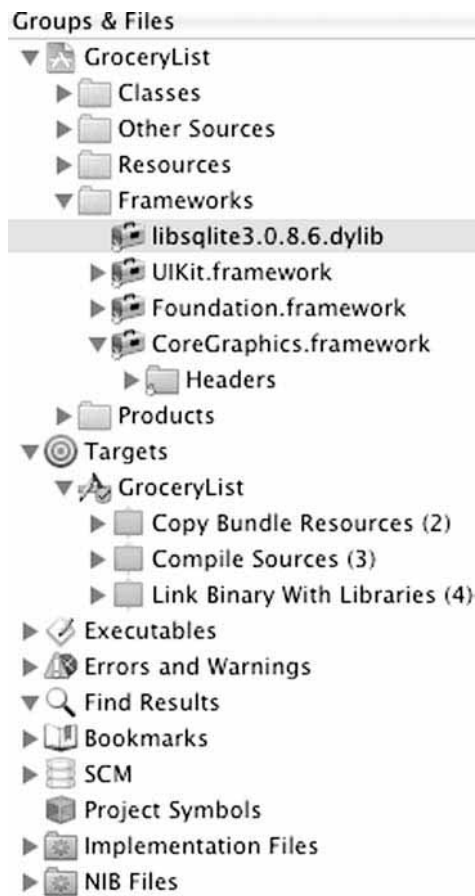


Figure 5-3. *The GroceryList project*

Next open `RootViewController.h`, and add an `NSArray` property called `results`. This property will contain the list to be displayed in the `UITableView` that was created by default by the project template as the main view. `RootViewController.h` should look like this:

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

@interface RootViewController : UITableViewController {
    NSArray *results;
}

@property (nonatomic, retain) NSArray *results;
@end
```

Make sure to @synthesize this property in the RootViewController.m file and release it in the dealloc method. Find and replace the implementations of the following functions in the .m file as well:

```
- (void)viewDidLoad
{
    [super viewDidLoad];
    self.results = [NSArray arrayWithObjects: @"Apple", @"Banana", nil];
}

- (NSInteger)tableView:(UITableView *)tableView numberOfRowsInSectionSection: _
(NSInteger)section
{
    return [results count];
}

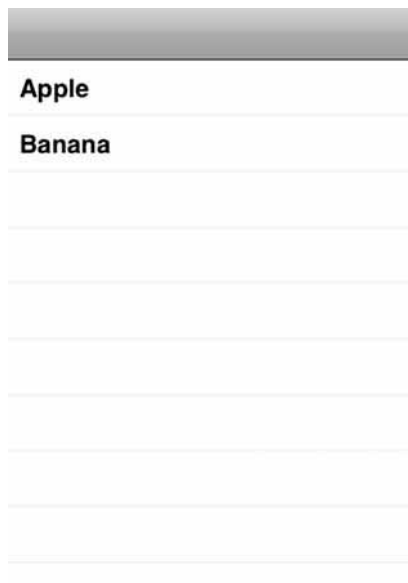
- (UITableViewCell *)tableView:(UITableView *)tableView cellForRowAtIndexPath: _
(NSIndexPath *)indexPath
{
    static NSString *CellIdentifier = @"GroceryCell";

    UITableViewCell *cell = [tableView
        dequeueReusableCellWithIdentifier:CellIdentifier];
    if (cell == nil)
    {
        #if __IPHONE_OS_VERSION_MIN_REQUIRED >= 30000
            cell = [[[UITableViewCell alloc] initWithStyle:UITableViewCellStyleDefault
                reuseIdentifier:CellIdentifier] autorelease];
        #else
            cell = [[[UITableViewCell alloc] initWithFrame:CGRectZero _
                reuseIdentifier:CellIdentifier] autorelease];
        #endif
    }

    #if __IPHONE_OS_VERSION_MIN_REQUIRED >= 30000
        cell.textLabel.text = [results objectAtIndex:indexPath.row];
    #else
        cell.text = [results objectAtIndex:indexPath.row];
    #endif
    return cell;
}
```

The iPhone version check here lets you use the recommended methods for the iPhone 3.0 SDK while remaining compatible with older versions.

Building and running should show you a simple list of grocery items, as shown in Figure 5-4.



Apple
Banana

Figure 5-4. *A simple grocery list*

Creating and Initializing the Database

Next you'll create the file that contains the code that wraps the database methods. Create a new NSObject subclass. I used my company name prefix and kept it simple, naming the class ISDatabase.

The following functions will use three properties that must be declared in the header. Your header should start out looking like this:

```
#import <sqlite3.h>

@interface ISDatabase : NSObject {
    NSString *pathToDatabase;

    BOOL logging;

    sqlite3 *database;
}

@property (nonatomic, retain) NSString *pathToDatabase;
@property (nonatomic) BOOL logging;

- (id) initWithPath: (NSString *) filePath;
- (id) initWithFileName: (NSString *) fileName;
@end
```

Remember to @synthesize these in the .m file. Next, you'll create some simple init functions to aid in creating a database in the resources directories for the application:

```
- (id) initWithPath: (NSString *) filePath
{
```

```

    if(self = [super init])
    {
        self.pathToDatabase = filePath;

        [self open];
    }

    return self;
}

- (id) initWithFileName: (NSString *) fileName
{
    NSArray *paths = NSSearchPathForDirectoriesInDomains(NSDocumentDirectory,
                                                         NSUserDomainMask, YES);
    NSString *documentsDirectory = [paths objectAtIndex:0];

    return [self initWithPath: [documentsDirectory
                               stringByAppendingPathComponent:fileName]];
}

```

The first `init` function creates and opens a connection to a database at the given path; the second is a convenience function that creates or opens a database in the application documents directory with the given file name.

Opening a Database Connection

All data in a SQLite database is stored in a single cross-platform file on disk. To work with a SQLite database, you first have to open a connection and specify the database file. When done working with the database, you should close that connection. You'll add two functions to do this:

```

- (void) close
{
    if(sqlite3_close(database) != SQLITE_OK)
    {
        [self raiseSQLiteException:@"failed to close database with message '%S'."];
    }
}

- (void) open
{
    //opens database, creating the file if it does not already exist
    if(sqlite3_open([self.pathToDatabase UTF8String], &database) != SQLITE_OK)
    {
        sqlite3_close(database);
        [self raiseSQLiteException:@"Failed to open database with message '%S'."];
    }
}

```

These are pretty straightforward. `open` opens the connection and stores the database handle in the database property. They report errors by calling the `raiseSQLiteException` function:

```

- (void) raiseSQLiteException: (NSString *) errorMessage

```

```
{
    [NSEException raise:@"ISDatabaseSQLiteException" format:errorMessage,
                      sqlite3_errmsg16(database)];
}
```

This calls `sqlite3_errmsg16`, which takes the database handle and returns the error message in plain English. This is then wrapped in an `NSEException` and raised.

You then clean up the database connection in the `dealloc` (along with `pathToDatabase`):

```
- (void) dealloc
{
    [self close];
    [pathToDatabase release];

    [super dealloc];
}
```

If you compile now, you will see two warnings appear. The first warning is in the `initWithPath:` method, as shown in Figure 5-5.

```
- (id) initWithPath: (NSString *) filePath
{
    if(self = [super init])
    {
        self.pathToDatabase = filePath;

        [self open];
    }

    return self;
}
```


 warning: 'ISDatabase' may not respond to '-open'
(Messages without a matching method signature will be assumed to return 'id' and accept '...' as arguments.)

Figure 5-5. *You've been warned.*

The compiler needs to be informed about methods that are declared later in the class that are used earlier, such as the `open` method here. The other warning is similar, informing you that the compiler cannot find the `raiseSQLiteException:` method. Instead of rearranging the methods to fit the demands of the machine rather than readability, add a private category to contain these method declarations. Add the following to the top of the `ISDatabase.m` file:

```
@interface ISDatabase(PrivateMethods)
- (void) open;
- (void) raiseSQLiteException: (NSString *) errorMessage;
@end
```

Recompile, and the warnings are no more.

Now that you've completed these methods, you can see them in action. Open `GroceryListAppDelegate.h`, and add `ISDatabase` as a forward class and a property called `database` to hold the database instance. `GroceryListAppDelegate.h` should look like this:

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

@class ISDatabase;

@interface GroceryListAppDelegate : NSObject <UIApplicationDelegate> {
    UIWindow *window;
    UINavigationController *navigationController;

    ISDatabase *database;
}

@property (nonatomic, retain) IBOutlet UIWindow *window;
@property (nonatomic, retain) IBOutlet UINavigationController *navigationController;
@property (nonatomic, retain) ISDatabase *database;

@end
```

Import `ISDatabase.h` in `GroceryListAppDelegate.m`, and also add `@synthesize database`. Remember to release the database property in the `dealloc` function as well. Replace the following method:

```
- (void)applicationDidFinishLaunching:(UIApplication *)application {
    database = [[[ISDatabase alloc] initWithFileName:@"TestDB.sqlite"] autorelease];
    NSLog(@"The database opened properly!");

    // Configure and show the window
    [window addSubview:[navigationController view]];
    [window makeKeyAndVisible];
}
```

Running the project now will display the console output similar to Figure 5-6.

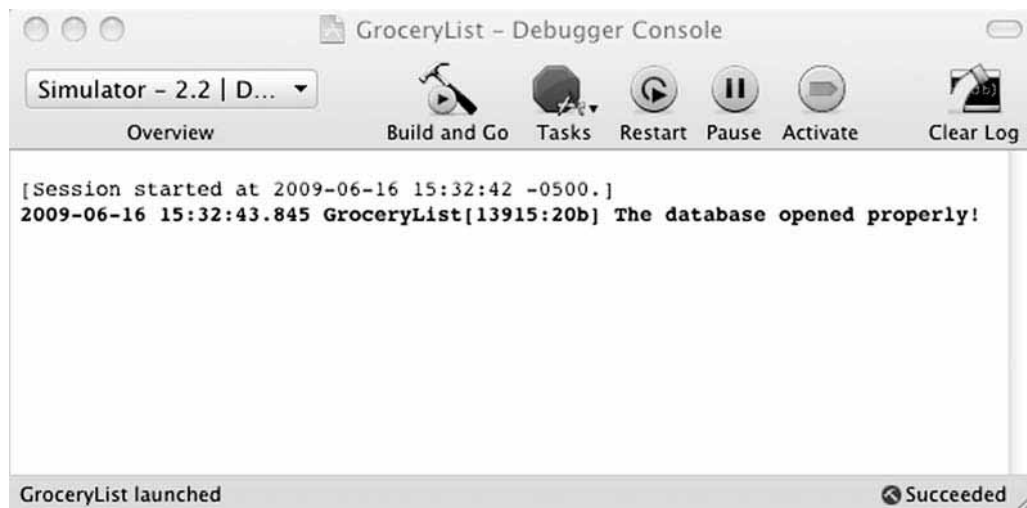


Figure 5-6. *The database is open for business.*

If the database file fails to open, it will raise an exception, so reaching the log statement indicates that everything worked properly.

Making Simple Requests

After opening the database, you'll want to make some requests in SQL. These range from creating the initial tables to data access operations. This is considerably more involved and takes up the majority of the lines of code for this class.

The simplest statement has no parameters: `Select * from GroceryItem`. You can create a function in `ISDatabase.m` to handle these statements:

```
- (NSArray *) executeSql: (NSString *) sql
{
```

This is the function declaration; it takes a SQL string and returns an `NSArray` of `NSDictionary` objects. Each dictionary object represents one result row. If there are no results, then an empty array will be returned.

```
NSMutableDictionary *queryInfo = [NSMutableDictionary dictionary];
[queryInfo setObject:sql forKey:@"sql"];
```

Store the SQL in a dictionary for use in error reporting.

```
NSMutableArray *rows = [NSMutableArray array];
```

Declare the array that will contain the result rows:

```
if(logging)
{
    NSLog(@"SQL: %@ \n", sql);
}
```

If the logging parameter is set to YES, log the SQL message here. This can be useful for tracking down bugs and performance optimization.

```
sqlite3_stmt *statement = NULL;
if(sqlite3_prepare_v2(database, [sql UTF8String], -1, &statement, NULL) == _
    SQLITE_OK)
{
```

This creates the `sqlite3_stmt` variable and “prepares” it. The preparation step compiles the SQL into a bytecode program that SQLite can understand. It initializes the statement variable with a pointer to the prepared statement that can be used to step through the results. It returns either `SQLITE_OK` or an error code. We won't handle error codes explicitly here, instead relying on the exception handling to report these errors. There are certain other cases that you may want to handle; you can find them in the SQLite documentation at http://www.sqlite.org/c3ref/c_abort.html.

```
BOOL needsToFetchColumnTypesAndNames = YES;
NSArray *columnTypes = nil;
NSArray *columnNames = nil;
```

These variables are used to cache the column names and types for the statement. While processing a result set from the database, each column for each row will need to be mapped to an Objective-C type for return and stored in an `NSDictionary` with the column name as the key. Looking up the column name for the key and the type for mapping the value can be costly for larger result sets, so you look them up for the first row and remember them for each subsequent row:

```
while (sqlite3_step(statement) == SQLITE_ROW)
{
```

You then step through each result using `sqlite3_step`, which returns `SQLITE_ROW` if a row has been returned or `SQLITE_DONE` if the statement is finished executing:

```
if(needsToFetchColumnTypesAndNames)
{
    columnTypes = [self columnTypesForStatement: statement];
    columnNames = [self columnNamesForStatement: statement];
    needsToFetchColumnTypesAndNames = NO;
}
```

Get the column types and the column names on the first time through this loop:

```
NSMutableDictionary *row = [[NSMutableDictionary alloc] init];
[self copyValuesFromStatement: statement toRow: row queryInfo: queryInfo _
    columnTypes: columnTypes columnNames: columnNames];
```

Create the row dictionary, and copy the results from the statement into the row:

```
[rows addObject:row];
[row release];
}
```

Add the row to the results, and release the row. This won't clean up the memory since the rows still retain a reference to the row dictionary, but it is slightly faster than waiting for the autorelease pool to iterate over the row objects to release them:

```
}else{
    sqlite3_finalize(statement);
    [self raiseSQLiteException: [[NSString stringWithFormat:@"failed to execute _
        statement: '%" with message: ", sql] stringByAppendingString:@"%S"]];-
}
```

If there is an error, delete the prepared statement, releasing any associated memory, and raise an exception:

```
sqlite3_finalize(statement);
return rows;
}
```

This deletes the prepared statement and returns the row results. Make sure to add the `executeSql:` method declaration to `ISDatabase.h`.

To retrieve the column names for the statement, you iterate from 0 to the number of columns (retrieved using `sqlite3_column_count`) and call `sqlite3_column_name` for each column. You then take the C string returned and wrap it in an `NSString`.

```
- (NSArray *) columnNamesForStatement: (sqlite3_stmt *) statement
{
    int columnCount = sqlite3_column_count(statement);

    NSMutableArray *columnNames = [NSMutableArray array];
    for(int i = 0; i < columnCount; i++)
    {
        [columnNames addObject:[NSString _
            stringWithUTF8String:sqlite3_column_name(statement, i)]];
    }
}
```

```
    return columnNames;
}
```

It's important to note that for SELECT statements like `SELECT SUM(number) FROM GroceryItem`, the column name will actually be `SUM(number)`. This also picks up on SQL aliases.

Similarly, you iterate over the columns to get the types:

```
- (NSArray *) columnTypesForStatement: (sqlite3_stmt *) statement
{
    int columnCount = sqlite3_column_count(statement);

    NSMutableArray *columnTypes = [NSMutableArray array];
    for(int i = 0; i < columnCount; i++)
    {
        [columnTypes addObject:[NSNumber numberWithInt:[self _
            typeForStatement:statement column:i]]];
    }

    return columnTypes;
}
```

To get the actual type, you use `typeForStatement:column:`, like so:

```
- (int) typeForStatement: (sqlite3_stmt *) statement column: (int) column
{
    const char * columnType = sqlite3_column_decltype(statement, column);

    if(columnType != NULL)
    {
        return [self columnTypeToInt: [[NSString stringWithUTF8String: columnType] _
            uppercaseString]];
    }

    return sqlite3_column_type(statement, column);
}
```

`typeForStatement:column:` returns an integer defining the column type. First it checks the declared type of the column using `sqlite3_column_decltype`, which returns the string associated with the column by the database schema. This is the type you declare for the column when creating the table. Then this type is converted to a standard type using `columnTypeToInt:`. If there is no declared type for the column (this occurs when using some calculated fields), use the type of the value instead, returned by `sqlite3_column_type`.

`columnTypeToInt` is a simple mapping, defined like so:

```
- (int) columnTypeToInt: (NSString *) columnType
{
    if([columnType isEqualToString:@"INTEGER"])
    {
        return SQLITE_INTEGER;
    } else if([columnType isEqualToString:@"REAL"])
    {
        return SQLITE_FLOAT;
    }
}
```

```

    }else if([columnType isEqualToString:@"TEXT"])
    {
        return SQLITE_TEXT;
    }else if ([columnType isEqualToString:@"BLOB"])
    {
        return SQLITE_BLOB;
    }else if ([columnType isEqualToString:@"NULL"])
    {
        return SQLITE_NULL;
    }

    return SQLITE_TEXT;
}

```

You default to text if none of the other types work. SQLite is a bit unique among databases in that it stores types dynamically; column types do not actually restrict the type of data stored in a column. This can make mapping from SQLite tables to Objective-C types difficult; for the purposes of ISDatabase, it is assumed that every column will be marked with an appropriate type from the previous list. These are added as part of the SQL CREATE statement for the given table.

`executeSql:` calls `copyValuesFromStatement:toRow:columnTypes:columnNames:` to map the results from the prepared statement to a specific row in the dictionary:

```

- (void) copyValuesFromStatement: (sqlite3_stmt *) statement toRow:
(NSMutableDictionary *) row queryInfo: (NSDictionary *) queryInfo columnTypes: _
(NSArray *) columnTypes columnNames: (NSArray *) columnNames
{
    int columnCount = sqlite3_column_count(statement);

    for(int i = 0; i < columnCount; i++)
    {
        id value = [self valueFromStatement:statement column:i queryInfo: queryInfo_
                                                                    columnTypes: columnTypes];

        if(value != nil)
        {
            [row setValue: value forKey: [columnNames objectAtIndex:i]];
        }
    }
}

```

This function steps through each column in the statement and calls `valueFromStatement:column:queryInfo:columnTypes:` for each column. The results are stored in the row `NSMutableDictionary` with the column name as the key. Getting and wrapping the value is a little more complex:

```

- (id) valueFromStatement: (sqlite3_stmt *) statement column: (int) _
column queryInfo: (NSDictionary *) queryInfo columnTypes: (NSArray *) columnTypes
{
    int columnType = [[columnTypes objectAtIndex:column] intValue];

    //force conversion to the declared type using sql conversions; this saves some
    //problems with NSNull being assigned to non-object values
    if(columnType == SQLITE_INTEGER)
    {

```

```

        return [NSNumber numberWithInt:sqlite3_column_int(statement, column)];
    }else if(columnType == SQLITE_FLOAT)
    {
        return [NSNumber numberWithDouble: sqlite3_column_double(statement, _
            column)];
    }else if(columnType == SQLITE_TEXT)
    {
        const char *text = (const char *) sqlite3_column_text(statement, column);
        if(text != nil){
            return [NSString stringWithUTF8String: text];
        }else{
            return nil;
        }
    }else if (columnType == SQLITE_BLOB)
    {
        //create an NSData object with the same size as the blob
        return [NSData dataWithBytes:sqlite3_column_blob(statement, column) _
            length:sqlite3_column_bytes(statement, column)];
    }else if (columnType == SQLITE_NULL)
    {
        return nil;
    }
    }

    NSLog(@"Unrecognized SQL column type: %i for sql: %@", columnType, [queryInfo _
        objectForKey:@"sql"]);

    return nil;
}

```

For each type, you use a specific SQLite function to retrieve the value and then convert that value as necessary before wrapping it in an Objective-C type and returning it. If the type is not recognized, you log the error and return nil. This skips the column for this result row.

If you compile now, a bunch of warnings will pop up letting you know that the methods are out of the order that the compiler expects. Update the PrivateMethod category at the top of this class to look like the following to get rid of these warnings:

```

@interface ISDatabase(PrivateMethods)
- (void) open;
- (void) raiseSqliteException: (NSString *) errorMessage;
- (NSArray *) columnNamesForStatement: (sqlite3_stmt *) statement;
- (NSArray *) columnTypesForStatement: (sqlite3_stmt *) statement;
- (int) typeForStatement: (sqlite3_stmt *) statement column: (int) column;
- (int) columnTypeToInt: (NSString *) columnType;
- (void) copyValuesFromStatement: (sqlite3_stmt *) statement toRow:
(NSMutableDictionary *) row queryInfo: (NSDictionary *) queryInfo columnTypes: __
(NSArray *) columnTypes columnNames: (NSArray *) columnNames;
- (id) valueFromStatement: (sqlite3_stmt *) statement column: (int) column_
queryInfo: (NSDictionary *) queryInfo columnTypes: (NSArray *) columnTypes;
@end

```

You are now ready to execute some SQL statements. Replace the `applicationDidFinishLaunching:` in `GroceryListAppDelegate.m` with the following:

```

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

```

```

self.database = [[[ISDatabase alloc] initWithFileName:@"TestDB.sqlite"] autorelease];
[database executeSql:@"create table GroceryItem(primaryKey integer primary key _
autoincrement, name text NOT NULL, number integer NOT NULL)"];
[database executeSql:@"insert into GroceryItem (name, number) _
values('apples', 5)"];
[database executeSql:@"insert into GroceryItem (name, number) _
values('oranges', 3)"];

[window addSubview:[navigationController view]];
[window makeKeyAndVisible];
}

```

This code creates a simple database and populates it with a row.

Next you need to update `RootViewController.m` to load the list of items from the database. Add `ISDatabase.h` to the imports, and replace the following function:

```

- (void)viewDidLoad {
    [super viewDidLoad];
    GroceryListAppDelegate *appDelegate = (GroceryListAppDelegate *)[[UIApplication _
sharedApplication] delegate];
    self.results = [appDelegate.database executeSql:@"SELECT * from GroceryItem"];
}

```

This will be returning an array of `NSDictionary` objects, so you need to update `tableView:cellForRowAtIndexPath:` to get the value stored under the name key. Change the following line:

```
cell.textLabel.text = [results objectAtIndex:indexPath.row];
```

to the following:

```
cell.textLabel.text = [[results objectAtIndex:indexPath.row] objectForKey:
@"name"];
```

and change this:

```
cell.text = [results objectAtIndex:indexPath.row];
```

to the following:

```
cell.text = [[results objectAtIndex:indexPath.row] objectForKey: @"name"];
```

Running this code now (but only once; more on this shortly) will produce the view shown in Figure 5-7.

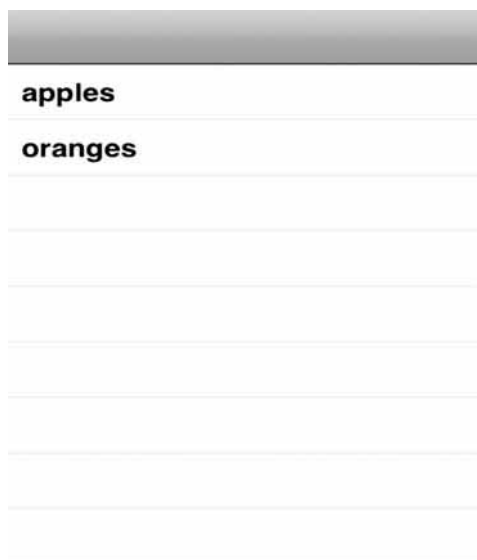


Figure 5-7. A data-backed view

More Advanced SQL

Now that you have basic SQL statements running, you can move on to processing more complex statements. This section will cover making the code less brittle, handling parameters, adding some nice convenience methods to make your life easier, and finally grouping statements into transactions.

Preventing Duplicate Create Statements

Try running the code from the previous section again. You'll be unpleasantly surprised by the error message in Figure 5-8.



Figure 5-8. Running again presents this nice exception.

You see this exception because the setup code in the previous section is brittle. If you run it more than once with the same database, it will throw an error on an attempt to re-create the table `GroceryItem`, which already exists. The simplest fix is to alter the create statement to be `create table IF NOT EXISTS GroceryItem (primaryKey integer primary key autoincrement, name text NOT NULL, number INTEGER NOT NULL)`. This prevents the table from being re-created and gets rid of the exception, but you'll still get duplicate data from the inserts.

A better fix involves checking to see whether the table already exists before running the schema creation code. Add the following functions to `ISDatabase`:

```
- (NSArray *) tables
{
    return [self executeSql:@"select * from sqlite_master where type = 'table'"];
}

- (NSArray *) tableNames
{
    return [[self tables] valueForKey:@"name"];
}
```

This queries the `sqlite_master` table that is automatically created by SQLite and used to manage the metadata for the database. This uses a neat feature of the `NSArray` class. If you call `valueForKey:` on an `NSArray`, it will in turn call `valueForKey:` for each of its member objects, returning a new `NSArray` containing the results. In this case, those results are the names of the tables.

Add `tableNames` to `ISDatabase.h`.

Now change `applicationDidFinishLaunching:` to the following:

```
- (void)applicationDidFinishLaunching:(UIApplication *)application
{
    self.database = [[[ISDatabase alloc] initWithFileName:@"TestDB.sqlite"] autorelease];

    if(![[database tableNames] containsObject:@"GroceryItem"])
    {
        [database executeSql:@"create table GroceryItem(primaryKey integer primary
            key autoincrement, name text NOT NULL, number INTEGER NOT NULL)"];
        [database executeSql:@"insert into GroceryItem (name, number) _
            values('apples', 5)"];
        [database executeSql:@"insert into GroceryItem (name, number) _
            values('oranges', 3)"];
    }

    [window addSubview:[navigationController view]];
    [window makeKeyAndVisible];
}
```

The check for the table name inserted around the schema creation code prevents the tables from being re-created and the rows from being reinserted. Reruns should now show the same view every time without crashing.

Handling Parameters

The next step to SQL dominance is adding the ability to handle parameters. The most obvious way is to use `stringWithFormat:` to create a SQL string and insert the parameters right into the string. Unfortunately, this is brittle and tends to be painful because you have to guard against and add escapes for a number of custom cases that may not be immediately obvious. Fortunately, SQLite allows you to “bind” parameters to a SQL string and takes care of all the necessary checking and escaping for you.

To allow this, you’ll add a function that takes a SQL statement and an array of parameters. Add the following function before the previous `executeSql:` function:

```
- (NSArray *) executeSql: (NSString *) sql withParameters: (NSArray *) parameters _
{
    NSMutableDictionary *queryInfo = [NSMutableDictionary dictionary];
    [queryInfo setObject:sql forKey:@"sql"];

    if(parameters == nil)
    {
        parameters = [NSArray array];
    }

    //we now add the parameters to queryInfo
    [queryInfo setObject:parameters forKey:@"parameters"];

    NSMutableArray *rows = [NSMutableArray array];

    if(logging)
    {
        //log the parameters
        NSLog(@"SQL: %@ \n parameters: %@", sql, parameters);
    }

    sqlite3_stmt *statement = nil;
    if(sqlite3_prepare_v2(database, [sql UTF8String], -1, &statement, NULL) _
        == SQLITE_OK)
    {
        [self bindArguments: parameters toStatement: statement queryInfo: _
            queryInfo];

        BOOL needsToFetchColumnTypesAndNames = YES;
        NSArray *columnTypes = nil;
        NSArray *columnNames = nil;

        while (sqlite3_step(statement) == SQLITE_ROW)
        {
            if(needsToFetchColumnTypesAndNames)
            {
                columnTypes = [self columnTypesForStatement: statement];
                columnNames = [self columnNamesForStatement: statement];
                needsToFetchColumnTypesAndNames = NO;
            }

            id row = [[NSMutableDictionary alloc] init];
            [self copyValuesFromStatement: statement toRow: row queryInfo: _
                queryInfo columnTypes: columnTypes columnNames: columnNames];
        }
    }
}
```

```

        [rows addObject:row];
        [row release];
    }
} else {
    sqlite3_finalize(statement);
    [self raiseSQLiteException: [[NSString stringWithFormat:@"failed to _
        execute statement: '%@', parameters: '%@' with message: ", sql, _
        parameters] stringByAppendingString:@"%S"]];
}

sqlite3_finalize(statement);
return rows;
}

```

Add this function to `ISDatabase.h` as well.

Besides a little additional logging, the big difference here is the call to `bindArguments:parameters:toStatement:` after `sqlite3_prepare_v2`. As described, this function takes the array of parameters and binds them to the prepared statement:

```

- (void) bindArguments: (NSArray *) arguments toStatement: _
(sqlite3_stmt *) statement queryInfo: (NSDictionary *) queryInfo
{
    int expectedArguments = sqlite3_bind_parameter_count(statement);

    NSLog(@"Number of bound parameters _
        does not match for sql: %@ parameters: '%@'",
        [queryInfo objectForKey:@"sql"], [queryInfo objectForKey:@"parameters"]);

    for(int i = 1; i <= expectedArguments; i++)
    {
        id argument = [arguments objectAtIndex:i - 1];
        if([argument isKindOfClass:[NSString class]])
            sqlite3_bind_text(statement, i, [argument UTF8String], -1, _
                SQLITE_TRANSIENT);
        else if([argument isKindOfClass:[NSData class]])
            sqlite3_bind_blob(statement, i, [argument bytes], [argument length], _
                SQLITE_TRANSIENT);
        else if([argument isKindOfClass:[NSDate class]])
            sqlite3_bind_double(statement, i, [argument timeIntervalSince1970]);
        else if([argument isKindOfClass:[NSNumber class]])
            sqlite3_bind_double(statement, i, [argument doubleValue]);
        else if([argument isKindOfClass:[NSNull class]])
            sqlite3_bind_null(statement, i);
        else
        {
            sqlite3_finalize(statement);
            [NSException raise:@"Unrecognized object type" format:@"Active record _
                doesn't know how to handle object: '%@' bound to _
                sql: %@ position: %i", argument, [queryInfo _
                objectForKey:@"sql"], i];
        }
    }
}

```

You first check to make sure that the number of parameters that the statement expects matches the number of parameters passed into the method. The assert will raise an exception if this is not the case.

Next you cycle over the number of expected arguments and bind the parameters one by one. You determine the class of the argument and execute a specific SQLite function for each type. If the class of the argument is not one that you support, you release the statement and raise an exception.

Add the `bindArguments:parameters:toStatement` method to the `PrivateMethods` category.

Refactoring and Cleanup

Now that you have this function, you can do a bit of cleanup.

`executeSql:` is really a simpler case of `executeSql:withParameters:`, so let's change it to reflect this:

```
- (NSArray *) executeSql: (NSString *) sql
{
    return [self executeSql: sql withParameters: nil];
}
```

Next we'll add a convenience function to allow parameters to be specified using variable arguments similar to the `NSArray arrayWithObjects:` method.

```
- (NSArray *) executeSqlWithParameters: (NSString *) sql, ...
{
    va_list argumentList;
    va_start(argumentList, sql);
    NSMutableArray *arguments = [NSMutableArray array];
    id argument;

    while(argument = va_arg(argumentList, id))
    {
        [arguments addObject: argument];
    }

    va_end(argumentList);

    return [self executeSql:sql withParameters: arguments];
}
```

Add this function to `ISDatabase.h`.

Note that this list should take only object arguments and always be nil terminated, just like `NSArray arrayWithObjects:`. You can try this by modifying `viewDidLoad` in `RootViewController.m`. Change the `executeSQL` line to the following:

```
self.results = [appDelegate.database executeSqlWithParameters:@"SELECT * from
GroceryItem _
                        where number < ?", [NSNumber numberWithInt:5], nil];
```

Build and run. Figure 5-9 shows the resulting view.

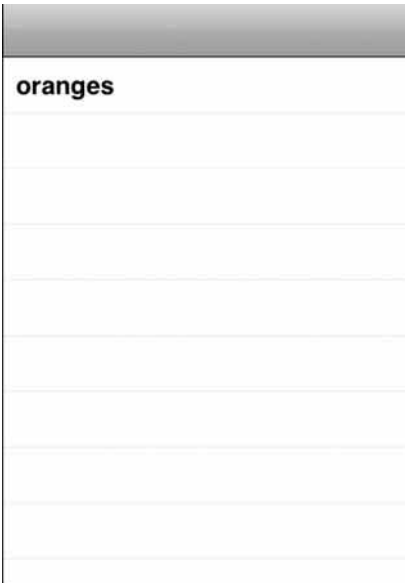


Figure 5-9. Another riveting data-backed view

Grouping Statements into Transactions

No database implementation is complete without support for transactions. Transactions allow multiple SQL statements to be grouped so that they can be submitted or reverted as a group, preventing partial commits and corrupted data structures in the case that a statement fails. Transaction support is easy in SQLite. Add these three functions to `ISDatabase.m`:

```
- (void) beginTransaction
{
    [self executeSql:@"BEGIN IMMEDIATE TRANSACTION;"];
}

- (void) commit
{
    [self executeSql:@"COMMIT TRANSACTION;"];
}

- (void) rollback
{
    [self executeSql:@"ROLLBACK TRANSACTION;"];
}
```

Add these three methods to `ISDatabase.h`.

You use the `IMMEDIATE` transaction type here. This is generally sufficient for most needs. There are other transaction types that may be useful in certain situations. These are summarized in Table 2-1.

Table 2-1. *SQLite Transaction Types*

Transaction Type	Description
IMMEDIATE	Database locks are acquired when the <code>BEGIN</code> statement is issued. This prevents other threads or processes from writing to the database until the transaction is committed or rolled back.
DEFERRED	This is the default transaction type. No locks are acquired until the database is written to or read from. Reading creates a shared lock that allows other threads to also read; writing creates a reserved lock that prevents other threads from writing to the database until the transaction is committed or rolled back.
EXCLUSIVE	This type of locks prevents other threads or processes from reading or writing to the database until the transaction is committed or rolled back.

Now you'll add transactions to the schema creation code in `GroceryListAppDelegate.m` `applicationDidFinishLaunching`:

```
if(![[database tableNames] containsObject:@"GroceryItem"])
{
    [database beginTransaction];

    [database executeSql:@"create table GroceryItem (primaryKey integer
        primary key autoincrement, name text NOT NULL, number INTEGER NOT NULL)"];
    [database executeSql:@"insert into GroceryItem (name, number) _
        values('apples', 5)"];
    [database executeSql:@"insert into GroceryItem (name, number) _
        values('oranges', 3)"];

    [database commit];
}
```

Writing a Simple Active Record Layer: ISModel

Easy access to raw SQL result sets is only half the battle; the other half is mapping the returned results into your object model. In times of yore, developers created manual mappings between the `NSDictionary/HashMap/What-Have-You` and each model object. Every time an object or table changed, the mapping code had to change.

Objective-C's message-driven nature and key/value coding make this significantly easier. In this section, I'll walk you through creating a flexible mapping layer that handles this automatically. This requires extension from a base class; to continue with my naming convention, I called this class `ISModel`. The header for `ISModel` with the properties starts out looking like this:

```
@class ISDatabase;

@interface ISModel : NSObject {
    NSUInteger primaryKey;
    BOOL savedInDatabase;
}

@property (nonatomic) NSUInteger primaryKey;
@property (nonatomic) BOOL savedInDatabase;

@end
```

primaryKey is the unique ID of the object; savedInDatabase keeps track of whether this object has already been saved or not. Add ISDatabase.h to the list of imports and the following synthesize statements in the ISModel.m file:

```
@synthesize primaryKey;
@synthesize savedInDatabase;
```

Maintaining the Database Connection

The Active Record design pattern involves using “finder” class methods on the model classes to retrieve the model objects from the database. To do this, they will need a reference to the database connection. It has to be either passed in to the method or stored somewhere that is available to all the model classes that require it. I use a static variable in the ISModel class to contain this (add this before the @implementation line):

```
static ISDatabase *database = nil;
```

Since this connection is global to all ISModel classes in the implementation, you need to add some class-level setters and getters for this property:

```
+ (void) setDatabase: (ISDatabase *) newDatabase
{
    [database autorelease];
    database = [newDatabase retain];
}

+ (ISDatabase *) database
{
    return database;
}
```

These will need to be called with the database before any of the ISModel SQL methods are called. Add the declarations to ISModel.h so that you can call them from outside the class.

The Model Object: Grocery Item

Next you create a simple subclass of ISModel with two properties:

```
#import "ISModel.h"
```

```
@interface GroceryItem : ISModel {
    NSString *name;
    NSNumber *number;
}

@property (nonatomic, retain) NSString *name;
@property (nonatomic, retain) NSNumber *number;

@end
```

And do this in the implementation:

```
@synthesize name, number;

- (void) dealloc
{
    [name release];
    [number release];
    [super dealloc];
}
```

This is all the code you'll need to create a model that can work with the database. Everything else is handled by the `ISModel` superclass.

How Groceries Are Mapped

Before you can get to the various SQL operations for `ISModel`, you'll need to add some basic methods to get the metadata for the class. This will tell you how to map instances of the class to rows and columns in a database table. Unless otherwise stated, the following methods should be added to the `ISModel.m` file. First up is retrieving the name of the associated table:

```
+ (NSString *) tableName
{
    return NSStringFromClass([self class]);
}
```

Here you're relying on the convention that the table name will be the same as the class name. This method could be altered to allow for other naming conventions such as custom table prefixes, but you'll keep it simple here.

To do the mapping, you need a list of column names:

```
- (NSArray *) columns
{
    if(tableCache == nil)
    {
        tableCache = [[NSMutableDictionary dictionary] retain];
    }

    NSString *tableName = [[self class] tableName];
    NSArray *columns = [tableCache objectForKey:tableName];

    if(columns == nil)
    {
        columns = [database columnsForTableName: tableName];
    }
}
```

```

        [tableCache setObject: columns forKey: tableName];
    }

    return columns;
}

```

This relies on a static variable you add to the top of ISModel:

```
static NSMutableDictionary *tableCache = nil;
```

You cache the column names so that you don't have to retrieve them from the database each time you run a SQL statement. Column names are retrieved from ISDatabase using a new function defined in ISDatabase.m:

```

- (NSArray *) columnsForTableName: (NSString *) tableName
{
    NSArray *results = [self executeSql: [NSString stringWithFormat:
        @"pragma table_info(%@)", tableName]];

    return [results valueForKey:@"name"];
}

```

This function uses the `table_info` SQLite pragma command to return the list of column names.

Add `columnsForTableName:` to ISDatabase.h.

For many methods, SQLite handles the primary key column for you, so it is useful to have a convenience method to get the list of columns without the key. Add the following to ISModel.m:

```

- (NSArray *) columnsWithoutPrimaryKey
{
    NSMutableArray *columns = [NSMutableArray arrayWithArray: [self columns]];
    [columns removeObjectAtIndex:0];

    return columns;
}

```

To persist the properties, you need their values in the same order as the columns:

```

- (NSArray *) propertyValues
{
    NSMutableArray *values = [NSMutableArray array];
    for(NSString *columnName in [self columnsWithoutPrimaryKey])
    {
        id value = [self valueForKey: columnName];

        if(value != nil)
        {
            [values addObject: value];
        }else{
            [values addObject:[NSNull null]];
        }
    }
    return values;
}

```

Let's add some code to test that everything is working as expected. Add the following method:

```
- (void) testProperties
{
    NSLog(@"column names: %@", [self columns]);
    NSLog(@"column names without primary key: %@", [self _
        columnsWithoutPrimaryKey]);
    NSLog(@"propertyValues: %@", [self propertyValues]);
}
```

First you need to tell ISModel which database to use. Import ISModel.h in GroceryListAppDelegate.m, and add the following line before the addSubview: call:

```
[ISModel setDatabase:database];
```

Then switch to RootViewController.m. Import GroceryItem.h, and add the following lines to the bottom of viewDidLoad:

```
GroceryItem *item = [[[GroceryItem alloc] init] autorelease];
item.name = @"Kiwi";
item.number = [NSNumber numberWithInt: 20];
[item testProperties];
```

Compile and run (ignore the warning about testProperties for now), and you should see something similar to Figure 5-10.

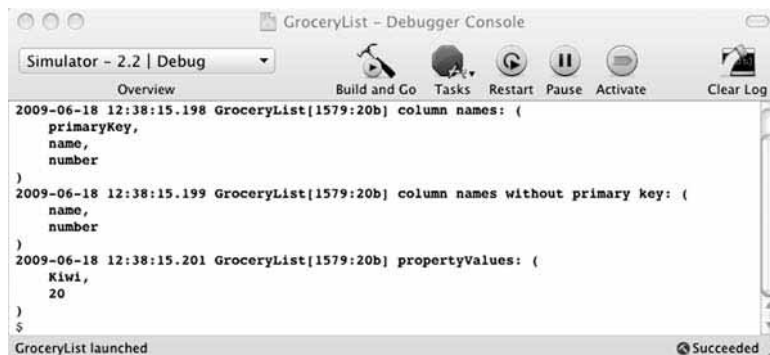


Figure 5-10. Console output showing the model information

After verifying that this works, delete the testProperties function in ISModel.m and the test code you just added in RootViewController.m viewDidLoad.

Saving

The first operation you'll look at is saving an instance of this object to the database. You'll start from the save method and work your way through each required piece. This is in ISModel.m. The following is the code for saving:

```
- (void) beforeSave
{
}
}
```

```

- (void) save
{
    [[self class] assertDatabaseExists];

    [self beforeSave];

    if(!savedInDatabase)
    {
        [self insert];
    }else{
        [self update];
    }
}

```

Add save to the header. This function first checks to see whether the database exists using the following:

```

+ (void) assertDatabaseExists
{
    NSAssert1(database, @"Database not set. Set the database using [ISModel _
        setDatabase] before using ActiveRecord.", @"");
}

```

This raises an exception if the database has not been set before this code is run. Add this near the top of the file to prevent warnings from being generated every time you call it.

Next the beforeSave callback is called; this allows subclasses to add custom behavior for each object when it is saved.

The code then checks to see whether the object has already been saved. If it has not, insert is called, creating a new row in the database for this object. Otherwise, update is called, saving the object state to the existing database row. You insert a row by delegating to ISDatabase:

```

- (void) insert
{
    NSMutableArray *parameterList = [NSMutableArray array];

    NSArray *columnsWithoutPrimaryKey = [self columnsWithoutPrimaryKey];

    for(int i = 0; i < [columnsWithoutPrimaryKey count]; i++)
    {
        [parameterList addObject: @"?"];
    }

    NSString *sql = [NSString stringWithFormat:@"insert into %@ (%@) values(%@)", _
        [[self class] tableName], [columnsWithoutPrimaryKey componentsJoinedByString: _
        @"", ","], [parameterList componentsJoinedByString:@"", ","]];

    [database executeSql: sql withParameters: [self propertyValues]];
    savedInDatabase = YES;
    primaryKey = [database lastInsertRowId];
}

```

Create a new PrivateMethods category at the top of the ISModel file similar to what you did for ISDatabase. Add this method to it.

insert gets a list of the columns, excluding the primary key (which is handled automatically by SQLite), and steps through them, adding a ? placeholder for each. You then construct the SQL statement using the column names and the placeholders. `executeSql:` takes the statement and the list of property values and does the actual save to the database. The object is then marked as saved, and the primary key is retrieved and stored in the `primaryKey` property using a new method that you need to define in `ISDatabase.m` and expose in `ISDatabase.h`:

```
- (NSInteger) lastInsertRowId
{
    return (NSInteger) sqlite3_last_insert_rowid(database);
}
```

This returns the primary key of the last inserted row.

Updating

If you've already saved the object, you need to update the existing row instead of inserting a new one:

```
- (void) update
{
    NSString *setValues = [[[self columnsWithoutPrimaryKey]
                           componentsJoinedByString:@" = ?, "]] stringByAppendingString:@" = ?";
    NSString *sql = [NSString stringWithFormat:@"update %@ set %@ where primaryKey = ?",
                                                [[self class] tableName], setValues];
    NSArray *parameters = [[self propertyValues] arrayByAddingObject: [NSNumber _
                                                                    numberWithUnsignedInt:primaryKey]];

    [database executeSql: sql withParameters: parameters];
    savedInDatabase = YES;
}
```

This runs a simple SQL update statement. Add this method to the `PrivateMethods` category.

Deleting

Deleting an object is equally simple:

```
- (void) beforeDelete
{
}

- (void) delete
{
    [[self class] assertDatabaseExists];
    if(!savedInDatabase)
    {
        return;
    }

    [self beforeDelete];
}
```

```

NSString *sql = [NSString stringWithFormat:
    @"delete from %@ where primaryKey = ?", [[self class] tableName]];
[database executeSqlWithParameters: sql,
    [NSNumber numberWithInt:primaryKey], nil];
savedInDatabase = NO;
primaryKey = 0;
}

```

You mark the object as not saved and clear out the primary key. You also call another stub callback function, `beforeDelete`.

Add delete to `ISModel.h`.

Let's add some test code to make sure that you can create, update, and delete objects. Switch to `RootViewController.m`, and replace `viewDidLoad` with the following:

```

- (void)viewDidLoad {
    [super viewDidLoad];
    GroceryListAppDelegate *appDelegate = (GroceryListAppDelegate *)[[UIApplication
        sharedApplication] delegate];

    NSString *sql = @"SELECT * from GroceryItem";
    GroceryItem *kiwi = [[[GroceryItem alloc] init] autorelease];
    kiwi.number = [NSNumber numberWithInt:5];
    kiwi.name = @"Kiwi";
    NSLog(@"items before save: %@", [[appDelegate database] executeSql:sql]);
    [kiwi save];
    NSLog(@"items after save: %@", [[appDelegate database] executeSql:sql]);
    kiwi.name = @"Kiwifruit";
    [kiwi save];
    NSLog(@"items after update: %@", [[appDelegate database] executeSql:sql]);
    [kiwi delete];
    NSLog(@"items after delete: %@", [[appDelegate database] executeSql:sql]);

    self.results = [appDelegate.database executeSqlWithParameters:@"SELECT *
        from GroceryItem where number < ?", [NSNumber numberWithInt:5], nil];
}

```

Compile and run. You should see a list of items for each change printing out in the console. The kiwi item is added, updated, and then deleted. Once you've run it, undo these changes.

Finding Grocery Items

You can now create, update, and delete your grocery items, but all of that is pretty useless if you can't get them back out of the database. In this section, you'll add methods to look up your model objects. In Active Record, these are usually referred to as *finder methods*, or just *finders*.

All the finders should be added to the `ISModel.m` class and its header file. You'll start with the most specific, on which the others are based:

```
+ (NSArray *) findWithSql: (NSString *) sql withParameters: (NSArray *) parameters
{
    [self assertDatabaseExists];

    NSArray *results = [database executeSql:sql withParameters: parameters
                                   withClassForRow: [self class]];

    [results setValue:[NSNumber numberWithInt:YES] forKey:@"savedInDatabase"];

    return results;
}
```

This takes a simple SQL statement that may contain placeholders and a list of parameters to bind to the statement and calls a new function in `ISDatabase`: `executeSql:withParameters:withClassForRow`. `ISDatabase` will return an `NSArray` of results that are instances of the provided class. Their properties will be automatically set.

In `ISDatabase`, change the `executeSql:withParameters:` method signature to the following:

```
- (NSArray *) executeSql: (NSString *) sql withParameters: (NSArray *) _
    parameters withClassForRow: (Class) rowClass
```

Add this as a new signature to the header, rather than replacing the existing signature. We'll be adding a new `executeSql:withParameters` method shortly.

Next find the following line in this method:

```
id row = [[NSMutableDictionary alloc] init];
```

Change this to the following:

```
id row = [[rowClass alloc] init];
```

Values are now set on instances of the passed-in class rather than always being set on `NSMutableDictionary` objects. Setting values into an instance of the model object class directly using key-value coding saves you a copy step from the dictionary to the final class. This results in a significant performance boost for larger data sets that justifies the added complexity.

The `copyValuesFromStatement:toRow:queryInfo:columnTypes:columnNames:` function needs its signature changed as well, changing the `toRow` parameter type from `NSMutableDictionary` to `id`. Remember to update the `PrivateMethods` category at the top of the file as well.

```
- (void) copyValuesFromStatement: (sqlite3_stmt *) statement toRow: _
    (id) row queryInfo: (NSDictionary *) queryInfo columnTypes: _
    (NSArray *) columnTypes columnNames: (NSArray *) columnNames
```

Finally, add the following convenience method, replacing the old method with the same signature:

```
- (NSArray *) executeSql: (NSString *) sql withParameters: (NSArray *) parameters
```

```

{
    return [self executeSql:sql withParameters:parameters withClassForRow: _
            [NSMutableDictionary class]];
}

```

Now that you have the base finder method, you can add a few more that make certain types of lookups easier. Add the following to `ISModel.m`:

```

+ (NSArray *) findWithSqlWithParameters: (NSString *) sql, ...
{
    va_list argumentList;
    va_start(argumentList, sql);

    NSMutableArray *arguments = [NSMutableArray array];
    id argument;
    while(argument = va_arg(argumentList, id))
    {
        [arguments addObject: argument];
    }

    va_end(argumentList);

    return [self findWithSql:sql withParameters: arguments];
}

+ (NSArray *) findWithSql: (NSString *) sql
{
    return [self findWithSqlWithParameters:sql, nil];
}

+ (NSArray *) findByColumn: (NSString *) column value: (id) value
{
    return [self findWithSqlWithParameters:[NSString stringWithFormat:@"select *
        from %@ where %@ = ?", [self tableName], column], value, nil];
}

+ (NSArray *) findByColumn: (NSString *) column unsignedIntegerValue:_
(NSUInteger) value
{
    return [self findByColumn:column value: [NSNumber numberWithInt:_
        value]];
}

+ (NSArray *) findByColumn: (NSString *) column integerValue: (NSInteger) value
{
    return [self findByColumn:column value: [NSNumber numberWithInt:value]];
}

+ (NSArray *) findByColumn: (NSString *) column doubleValue: (double) value
{
    return [self findByColumn:column value: [NSNumber numberWithDouble:value]];
}

+ (id) find: (NSUInteger) primaryKey
{
    NSArray *results = [self findByColumn: @"primaryKey"
        unsignedIntegerValue: primaryKey];
}

```

```

        if([results count] < 1)
        {
            return nil;
        }
        return [results objectAtIndex:0];
    }

+ (NSArray *) findAll
{
    return [self findWithSql: [NSString stringWithFormat:@"select * from %@", _
        [self tableName]]];
}

```

Add all of these to the header file. You now have the methods you need to do any type of lookup you want, including custom SQL lookups when you need them. I try to put most SQL statements used with `findWithSql:` into the subclass as additional finder methods. This keeps the SQL from leaking out into my controller layer.

Putting It All Together

Now that you have the create, read, update, and delete (CRUD) operations complete, you are ready to get rid of the raw SQL in the example and start working directly with the model objects. Add `GroceryItem.h` to the import statements for `GroceryListAppDelegate.m`. Replace `applicationDidFinishLaunching:`

```

(void)applicationDidFinishLaunching:(UIApplication *)application
{
    database = [[[ISDatabase alloc] initWithFileName:@"TestDB.sqlite"] autorelease];

    if(![[database tableNames] containsObject:@"GroceryItem"])
    {
        [database beginTransaction];
        [database executeSql:@"create table GroceryItem(primaryKey integer primary
            key autoincrement, name text NOT NULL, number integer NOT NULL)"];
        [database executeSql:@"insert into GroceryItem (name, number)_
            values('apples', 5)"];
        [database executeSql:@"insert into GroceryItem (name, number)_
            values('oranges', 3)"];
        [database commit];
    }

    [ISModel setDatabase:database];

    NSArray *results = [GroceryItem findByColumn:@"name" value:@"Bananas"];

    if([results count] < 1)
    {
        GroceryItem *bananas = [[[GroceryItem alloc] init] autorelease];
        bananas.name = @"Bananas";
        bananas.number = [NSNumber numberWithInt: 10];
        [bananas save];
    }

    // Configure and show the window
    [window addSubview:[navigationController view]];
}

```

```
    [window makeKeyAndVisible];
}
```

In `RootViewController.m`, remove the `ISDatabase.h` import statement, and change the `viewDidLoad` function to the following, which is simpler:

```
- (void)viewDidLoad {
    [super viewDidLoad];
    self.results = [GroceryItem findAll];
}
```

Add the following line to the top of `tableView:cellForRowAtIndexPath:`

```
GroceryItem *result = [results objectAtIndex:indexPath.row];
```

Further down in the function, change the following:

```
cell.textLabel.text = [[results objectAtIndex:indexPath.row] objectForKey:@"name"];
```

to this:

```
cell.textLabel.text = [NSString stringWithFormat:@"%@@: %@", result.name,
                                                                result.number];
```

Also change the following:

```
cell.text = [[results objectAtIndex:indexPath.row] objectForKey:@"name"];
```

to this:

```
cell.text = [NSString stringWithFormat:@"%@@: %@", result.name, result.number];
```

You're now displaying the number of items as well. Running this should produce the view in Figure 5-11.

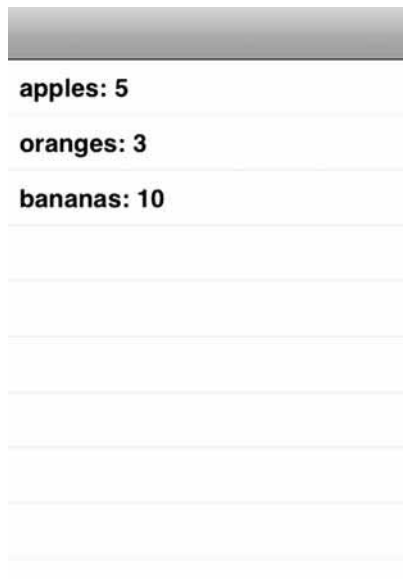


Figure 5-11. A grocery list backed by Active Record. Now with 100 percent more bananas!

Simple Migration Handling

I don't think any persistence implementation is complete without some discussion of migrating from one version of a database schema to another. I try to keep migrations simple. I rely on a new table, which I usually call `ApplicationProperties`, that stores the current version of the schema and any other properties I want to store for the entire application. I then check that version number on launch, and if an old version is detected, I run the necessary SQL to do the migrations. I stay away from using the actual model classes during the migration, because that can cause compatibility issues if you ever delete or rename a model class that would prevent the migrations from running.

First, create a subclass of `ISDatabase` called `ExampleDatabase`. This will contain all the application-specific database handling and migration code.

Add `ISModel` to the import statements for `ExampleDatabase.m`, and add the following `init` method:

```
- (id) initWithMigrations
{
    if(self = [super initWithFileName:@"Example.sqlite"])
    {
        [self runMigrations];
        [ISModel setDatabase:self];
    }

    return self;
}
```

Add this to the header file as well.

This creates a database file in your app's documents directory with the name `Example.sqlite` and sets the `ISModel` database so that Active Record will work properly. Next add the method to actually run the migrations:

```
- (void) runMigrations
{
    [self beginTransaction];

    NSArray *tableNames = [self tableNames];

    if(![tableNames containsObject: @"ApplicationProperties"])
    {
        [self createApplicationPropertiesTable];
        [self createGroceryItemTable];
        //add any other version 1 schema creation code here
    }

    [self commit];
}
```

You wrap the entire migration in a transaction. If any step fails, all the changes are rolled back. This prevents some steps from failing and later steps from succeeding that would put the database in an inconsistent state that is very difficult to recover from.

The first check is just for the existence of the `ApplicationProperties` table. Subsequent migration steps will check the version number to determine whether a migration should be run. Put all your initial schema creation code in this block.

Individual tables and changes are broken out into their own functions to make it easier to see the general flow of the migration separate from the sometimes-complex implementation details. Add the following implementation functions:

```
- (void) createApplicationPropertiesTable
{
    [self executeSql:@"create table ApplicationProperties (primaryKey integer _
                    primary key autoincrement, name text, value integer)"];
    [self executeSql:@"insert into ApplicationProperties (name, value) _
                    values('databaseVersion', 1)"];
}

- (void) createGroceryItemTable
{
    [self executeSql:@"create table GroceryItem (primaryKey integer primary key
                    autoincrement, name text NOT NULL, number INTEGER NOT NULL)"];
    [self executeSql:@"insert into GroceryItem (name, number) values('apples', 5)"];
    [self executeSql:@"insert into GroceryItem (name, number) _
                    values('oranges', 3)"];
}
```

At the top of the file, add a new `PrivateMethods` category:

```
@interface ExampleDatabase(PrivateMethods)
- (void) runMigrations;
- (void) createApplicationPropertiesTable;
- (void) createGroceryItemTable;
@end
```

Now change the application delegate `applicationDidFinishLaunching:` to the following:

```
- (void)applicationDidFinishLaunching:(UIApplication *)application {
    self.database = [[[ExampleDatabase alloc] initWithMigrations] autorelease];

    NSArray *results = [GroceryItem findByColumn:@"name" value:@"Bananas"];

    if([results count] < 1)
    {
        GroceryItem *bananas = [[[GroceryItem alloc] init] autorelease];
        bananas.name = @"Bananas";
        bananas.number = [NSNumber numberWithInt: 10];
        [bananas save];
    }

    [window addSubview:[navigationController view]];
    [window makeKeyAndVisible];
}
```

Remove the `ISDatabase` and `ISModel` import statements, and add `ExampleDatabase.h`.

Compile and run. A new database file will be created and migrated to version 1, creating the initial tables and populating them. Subsequent runs will see that `ApplicationProperties` already exists and skip that migration step.

Next you will make a small change to the schema, adding a new column to GroceryItem called price. Add the following to runMigrations in ExampleDatabase.m, before the commit:

```
if([self databaseVersion] < 2)
{
    [self setDatabaseVersion:2];
    [self addPriceToGroceryItemTable];
}
```

Add the function addPriceToGroceryItemTable:

```
- (void) addPriceToGroceryItemTable
{
    [self executeSql: @"alter table GroceryItem add price real"];
}
```

Add the following convenience functions:

```
- (void) updateApplicationProperty: (NSString *) propertyName value: (id) value
{
    [self executeSqlWithParameters: @"update ApplicationProperties set value = ? _
                                   where name = ?", value, propertyName, nil];
}

- (id) getApplicationProperty: (NSString *) propertyName
{
    NSArray *rows = [self executeSqlWithParameters: @"select value from
ApplicationProperties where name = ?", propertyName, nil];

    if([rows count] == 0)
    {
        return nil;
    }

    id object = [[rows lastObject] objectForKey:@"value"];
    if([object isKindOfClass: [NSString class]])
    {
        object = [NSNumber numberWithInt:[(NSString *)object integerValue]];
    }
    return object;
}

- (void) setDatabaseVersion: (NSUInteger) newVersionNumber
{
    return [self updateApplicationProperty:@"databaseVersion" value:[NSNumber _
numberWithUnsignedInteger: newVersionNumber]];
}

- (NSUInteger) databaseVersion
{
    return [[self getApplicationProperty:@"databaseVersion"] unsignedIntegerValue];
}
```

Add these methods to the PrivateMethods category:

```
@interface ExampleDatabase(PrivateMethods)
- (void) runMigrations;
- (void) createApplicationPropertiesTable;
```

```
- (void) createGroceryItemTable;
- (void) addPriceToGroceryItemTable;
- (void) updateApplicationProperty: (NSString *) propertyName value: (id) value;
- (id) getApplicationProperty: (NSString *) propertyName;
- (void) setDatabaseVersion: (NSUInteger) newVersionNumber;
- (NSUInteger) databaseVersion;
@end
```

These functions allow you to easily set and get any database property with helper functions for the database version property. Running the code now will migrate the database to version 2 and throw an error; we forgot to add the price property to the `GroceryItem` model class. Add it as an `NSNumber` property. All columns in the database must have corresponding properties on the model objects. The inverse is not true; the model can have “transient” properties that are not stored in the database.

Running the application again will work as expected. That’s it! Adding new migrations is as simple as adding a little bit of SQL. All previous versions will upgrade seamlessly, and you are free to migrate data and update other resources inside the migrations as well.

CAUTION: You should note that although supported by SQLite, access from multiple threads is not recommended. This Active Record code makes no attempt to handle multiple threads. In particular, access to the same database handle from multiple threads will result in errors being thrown. I recommend that you have one thread that handles all your SQL access.

Alternative Implementations

There are several alternatives to this implementation that are worth considering.

For very simple data storage needs, you may be able to get by with serializing your data to a plist file using the `NSCoder` APIs. This has the advantage of being as simple as a call to `writeToFile:atomically:`. The disadvantage is that the entire data set has to be loaded into memory each time, and there is no built-in way to search through the objects and bring back just the necessary set. This isn’t recommended for storing more than a handful of records.

If your application has higher data demands, there are other higher-end alternatives as well. It seems like every time I check, there are a few more floating around to handle this. The two I’m familiar with are FMDB and Apple’s implementation of an object graph persistence framework: Core Data.

FMDB is roughly equivalent to the `ISDatabase` class, with some code to handle more automatic retries, the option to store prepared statements for a speed boost, and handling for more SQLite error states. It’s a solid implementation, and you wouldn’t be remiss in using this as the basis for an Active Record variant of your own.

Core Data is Apple’s persistence framework. It has built-in Xcode support for creating schemas and migrations visually. It is fast, robust, and available as of iPhone SDK 3.0.

Being used to SQL, I find myself more comfortable in that world. If you're less familiar with SQL, then this should be the first persistence framework that you try if you are looking for something to handle the heavy lifting for you. I suggest using the SQLite persistence store option; it is generally better at handling larger data sets efficiently and has no real drawbacks.

Summary

In this chapter, you went through building a simple Active Record implementation from the ground up. You started with the SQLite C APIs and built a wrapper around them to make data access using SQL less of a chore. From there, you used a simple class as the basis for saving and retrieving your model objects from the database. Then you took a quick foray into the world of migrations, making sure that you can build on earlier versions of your database when moving to the next version of your application. Finally, I discussed a few alternatives to this implementation.

I hope you now have a good working knowledge of the SQLite APIs available on the iPhone and a good basis for building a data-driven application. I look forward to hearing about what you do with it!

Ray Kiddy



Company: Consultant, Ganymede Resources

Location: Sunnyvale, CA

Former Life As a Developer: Data Application and UI development, 15 years in Apple, 7 in the WebObjects team in Developer Tools. Managed servers when gopher was still cool. Developed Cocoa applications, Xcode plugins, Server Admin UI, deployment tools for web applications. Developer for extensions and custom web UI in Firefox, worked at Mozilla after Apple. Studying mathematics at SJSU.

Life as an iPhone Developer: ClickAccuracy (Developer Utility), Collectionator, iPhoneSmart Analytics framework/infrastructure, MindOverMatter game suite



What's in This Chapter:

- **Modeling with CoreData**
- **Dealing with Schema Migration**
- **A “Flexible Schema” Object Class**
- **Connecting to Remote Databases**
- **Testing Data iPhone Apps**

Key Technologies:

- **CoreData**
- **Xcode**
- **MigrationManager**
- **NSKeyValueCoding**

Core Data and Hard-Core Design

The iPhone is an amazing device for getting information at a moment's notice. Whether I am waiting for a train, waiting in line for coffee, or waiting on hold for someone to pick up the phone, thoughts come to me, and I may need to answer a question quickly in order to do something useful with the thought. If I wait until later, when I have more time, then I have often forgotten the idea or question.

The iPhone has a lot of data applications, suited to whatever particular task you have. And if you have that niche market idea or that odd situation where people want to organize information in some particular way, then you can create your own data application for them. The iPhone has tools that make it surprisingly easy to design these apps so that you can let users use your interface and reuse their data in interesting ways.

The key to designing a good data application for the iPhone is simplicity. For most data people, this cuts across the grain. Every time I create a data application, I want the application to know everything, do everything, hold the users in its hands, and make them feel comfortable because absolutely everything is taken care of by the application. The iPhone interface is constrained, though. You absolutely cannot do everything, no matter how much you want to do so. You must do just what your users need at that moment and nothing more.

Making your application responsive and flexible is more important than handling every situation. So, how can the tools that we have available help us create the right kind of application? It turns out that if you want to focus on the user experience and not on the database and if you want to think about the user workflow and not tables and columns and joins, then Core Data will be your friend.

Where Did Core Data Come From?

It turns out that Core Data did not just leap, fully grown, out of someone's head in Cupertino. Core Data is not even new with Mac OS X 10.4, where it first appeared within Apple. Core Data has a long interesting history, and, really, you never know when you will need an extra acronym or two to put on your resume.

Core Data was first added to the iPhone SDK with version 3.0. Core Data appeared in Mac OS X in Tiger (Mac OS X 10.4), but it actually has a much longer history than that. It was first developed at NeXT Computer as the DBKit framework in 1992, which then became the Enterprise Object Framework (EOF) in 1994. EOF was used to develop flexible applications on NeXT's operating systems, NeXTStep and OpenStep. You could build these applications for NeXT systems, Windows NT, or Solaris. Realizing that objects can be displayed via HTML as easily as via a custom application interface, the engineers at NeXT added WebObjects as a display layer for EOF.

EOF is an object-relational mapping (ORM) framework. It is well designed, is robust, encourages Model View Controller (MVC) design patterns, and was the conceptual forerunner of much later work in the industry. When I was at Apple, I heard that Sun attempted to purchase EOF and, when it could not, decided to create the JDBC library. And this may even be true. Additionally, Microsoft has been trying to create tools that match what is provided in WebObjects and EOF and has hired many of the software designers who worked at NeXT.

EOF and WebObjects were ported to Java in 1997 so that applications could run on any platform on which a Java VM was available. This got Apple out of the job of supporting, for example, an HP/UX version of the framework. It was very successful, it generated sales of Mac OS X Server machines, and there were active discussion groups among users of EOF and WebObjects that were more active than with almost any other Apple technology. Apple has never been comfortable with the enterprise market, though, and WebObjects has the stigma of being an "enterprise technology." WebObjects and EOF entered a long period of...quiet. The online store still used it, so Apple depended on it for money but was not sure about how to market it to other businesses. Personally, I believe that Steve Jobs will be comfortable talking about "the enterprise" when Pixar is making a *Star Trek* movie and not one day before.

The Client Is King

Core Data is, in a way, the step backward that was needed to make a step forward. From when I was at Apple, I know the company has always been about making shiny, pretty applications. It wants the "Wow!" in everything it does. It was never clear how EOF fit into that. It organizes your data for you? Really? That's nice. *Yawn....*

EOF had a history, with the EOF Application of the OpenStep days, as a client-access technology, so performance had been tuned for that usage. But it was ported to Java and then was used for very large applications at Apple and in companies such as Bell South and Deutsche Bank. The Apple Store still uses it, and the iTunes Store is a WebObjects application modified to push out XML instead of HTML. So, it was also

tuned for long-running processes, large data sets, and a long mean time between failures (MTBF).

Tuning for both quick and agile client access and for long-running server processes is a contradiction. This contradiction was never been resolved in WebObjects/EOF. With Core Data, though, Apple ported EOF back to Objective-C. Objective-C is a very flexible dynamic language and runtime, and this helped bring back the old efficiencies. It was now clear where the goal of the performance tuning should be. Core Data is only about the client. Many Mac OS X applications do not track MTBF in their testing, for example, and it is even less of a concern on the iPhone.

Core Data provides very deep classes for very little effort. You do not need to think about tables but rather entities, which are just “things.” You do not need to think about columns or joins but rather attributes and relationships. And these attributes and relationships are abstractly defined, in much the same way you see attributes defined in Eiffel. A join between two tables in a database has to follow some rules and be defined in a certain way, but a relationship in a Core Data entity may merely be a method that returns a collection of multiple...things. These things can be described in different ways. Key-Value Coding (KVC) and Key-Value Observing (KVO) are protocols for tracking or retrieving data from an object. As long as an object responds to the protocols, it can do almost anything it wants.

A Very First Core Data App

When I first got an iPhone and looked at the database apps, I knew there was work to do there. Well, at first I could not find any database apps. Then I realized that finding things in the usual way on the App Store is a joke, so after plowing through a bunch of stuff, I realized there was still a lot to do. What could I do first? First, I wanted to do my usual “keep track of everything” application. I have created this application on different platforms with different languages and tools, and, somehow, it is never finished. Hm. Perhaps I should try something simpler, with a slightly more manageable scope. A to-do application? Another one? How about something else?

An application for the iPhone can be finely tuned to some task. It is harder to make an iPhone app that does something big. What about event planning? Event planners need to be able to keep track of events they are planning, keep track of what they need to have for an event, and export this information to something outside the phone. What will I need to track in order to create successful events? I have absolutely no idea! But, it is important that I know that I do not know this.

It is easy, sometimes far too easy, to take the fact that I understand software and use that to convince myself that I understand something else, such as event planning. I can make an application that is better than just a “take a note” application. How much better? For \$1.99 on the App Store, how much better does it have to be? I will just make sure that whatever data the user puts in, they can get to it somewhere else. Letting the user reuse their own data is never, ever a bad thing. I decided to make a simple application to track events and their locations. But I can create a schema that is flexible enough to be modified for other uses as well.

First, Steal Code (Not Music!)

The first rule of being an effective software developer is that you need to know how to steal code from people who are smarter than you are. In that spirit, I made my job easier by starting with a project that Apple has provided in the iPhone SDK. I always look for tutorials and try them so I can learn by doing and not just reading. You can find the “Core Data Tutorial for iPhone OS” tutorial in the iPhone Developer Connection at <http://developer.apple.com>, which is a very good place to start. I am using the GM version of the iPhone SDK v 3.0, and this is what I will be referring to when I say “the iPhone SDK.”

The tutorial that Apple provides steps you through the process of building the Locations project. This project, as Apple has developed it, programmatically creates objects, stores the objects in a database, and displays a list of those objects. Apple has written some very good documentation here, and we should take advantage of that. But you cannot assume that the documentation is very complete. The path that Apple has laid out in that project is very clear, but there are several small steps you can take from there that will lead you into very deep weeds. I will be trying to show you how to get out of those weeds. I got covered in thorns here, and perhaps you will not need to do the same. Go to the tutorial and step through the process until you can build and run the application without crashes. There were a few problems with the tutorial, but they have been corrected in the GM version of iPhone SDK, so everything in the tutorial works as advertised.

One of the first things I wanted to do after the Locations project was built was rename the `eventsArray` ivar. I ended up not doing this. If you do decide to rename any of the ivars that you are using for properties, be careful. Make backup copies of your project before you begin. I experimented with renaming `eventsArray` with the GM version of the iPhone SDK, and it worked. However, when I was using one of the beta versions of the SDK, I managed to crash Xcode as I was doing this. I am not sure whether I forgot something as I was renaming things in the code or whether there was a bug in Xcode. Using the `@property` and `@synthesize` tokens gives you some autogenerated code. For example, you will see in Apple’s code that it calls the `setEventsArray:` method. This is a method created for you because of the `@synthesize` token in the `RootViewController` implementation. This is explained in the “Introduction to the Objective-C 2.0 Programming Language” document provided with the iPhone SDK. These method names are created so that they follow the rules of the KVC protocol. KVC is powerful, but KVC problems can be obnoxiously hard to diagnose. KVC lets you define an interface using strings, and those strings are opaque to the compiler, but the strings must match up with something at runtime. Errors may appear only at runtime, and the paths by which they occur can seem...obfuscated. With the power of the abstractions, you pay a price.

The tutorial application is very simple, and it initializes the database and sets up a lot of things like magic. In particular, the `UINavigationController` does a lot for you, and the tutorial relies quite a bit on the default behavior of this object. As you add to the application, you will have to go back and understand some of what has been given to

you and fill in some details. But you can see, from Figure 6-1, that the tutorial has given you a start.



Figure 6-1. *The Locations project, before customization, with default table view and navigation controller behaviors*

But this does not do very much. You can add a data record and delete records, but you cannot edit anything. You cannot drill down into any object. But it does store data. (Yawn....) Let's go get some coffee, shall we? (*Queue intermission music.*)

All right, we're back! One obvious way to extend the app is to let the user drill down into a particular object instance. You could have a new type of view for this, or you could just use another `UITableViewController` subclass. Of course, this will need an array, not just one object. But I am going to treat the object as an array of attributes. I am going to use KVC to examine the object. This will work with any `NSManagedObject` subclass and not just this particular object.

A View to an Object, Any Object

You can make this controller more useful by making it more general. Right now, it is displaying the event object in a very specific way. You may also notice that the “smarts” about the way the event is being displayed is in the `RootViewController` class. However, turning the numbers and dates in the object into displayable strings is not really a job for the view. It would be more in keeping with the principles of the MVC design if the `Event` class itself knew how to display an event. You can thus simplify the `tableView:(UITableView *)tableView cellForRowAtIndexPathIndexPath:` method by pushing that logic to the `Event` class.

Now the `RootViewController` method will look like this:

```

- (UITableViewCell *)tableView:(UITableView *)tableView
cellForRowAtIndexPath:(NSIndexPath *)indexPath {

    static NSString *CellIdentifier = @"Cell";
    // Dequeue or create a new cell    UITableViewCell *cell = [tableView
dequeueReusableCellWithIdentifier:CellIdentifier];
    if (cell == nil) cell = [[[UITableViewCell alloc]
initWithStyle:UITableViewCellStyleSubtitle reuseIdentifier:CellIdentifier] autorelease];

    cell.textLabel.text = [[eventsArray objectAtIndex:indexPath.row] description];
    cell.detailTextLabel.text = [[eventsArray objectAtIndex:indexPath.row]
subdescription];

    return cell;
}

```

The Event class now contains the following:

```

#import "Event.h"

@implementation Event

@dynamic longitude;
@dynamic latitude;
@dynamic creationDate;

- (NSString *)description {

    static NSDateFormatter *dateFormatter = nil;

    if (dateFormatter == nil) {
        dateFormatter = [[NSDateFormatter alloc] init];
        [dateFormatter setTimeStyle:NSDateFormatterMediumStyle];
        [dateFormatter setDateStyle:NSDateFormatterMediumStyle];
    }

    return [dateFormatter stringFromDate:[self creationDate]];
}

- (NSString *)subdescription {

    static NSNumberFormatter *numberFormatter = nil;

    if (numberFormatter == nil) {
        numberFormatter = [[NSNumberFormatter alloc] init];
        [numberFormatter setNumberStyle:NSNumberFormatterDecimalStyle];
        [numberFormatter setMaximumFractionDigits:3];
    }

    return [NSString stringWithFormat:@"%s, %s",
        numberFormatter stringFromNumber:self.latitude,
        numberFormatter stringFromNumber:self.longitude]];
}

@end

```

The table given in the Locations tutorial does not respond when you click a row, but you want something to happen. The UINavigationController gives you an easy way to manage different kinds of views that are related to a hierarchy of objects. It allows you to maintain a stack of view controllers. When your UI is constrained, as it is on an iPhone, you need to make views that are simple, are obvious, and do just what you need them to do. Since a view can do only so much, you need a lot of views, and it turns out that managing the views on a stack is amazingly useful. If you respond to a click in a table row by creating another UIViewController subclass, you can push the new view controller onto the stack. The UINavigationController will take care of setting up a “back” button in the navigation bar. When the user clicks this button, the new view controller will be automatically popped, and you will be back to the top level of your object tree.

```
- (void)tableView:(UITableView *)tableView didSelectRowAtIndexPath:(NSIndexPath
*)indexPath {
    [tableView deselectRowAtIndexPath:indexPath animated:NO];

    EventViewController * eventController = [[EventViewController alloc] init];
    [eventController setEvent:[eventsArray objectAtIndex:indexPath.row]];
    [[LocationsAppDelegate delegate].navigationController
pushViewController:eventController animated:YES];
    [eventController release];
}
```

You can create the EventViewController class now. It needs to be a subclass of UIViewController. It is useful to test the behavior at this point, before you add anything to your new class. If you test your application at this point, you will see the list of objects on the left when you launch (or a similar list), and then you will see the blank view on the right when you click a row. But you can verify that you can pop into this view and then hit the Locations button in the navigation bar, and you will pop back to the list of objects. As you can see in Figure 6-2, you can click any of the rows and come back out of the view and then click another object and do it again.



Figure 6-2. The *Locations* project with an empty *EventViewController*

Let's change the *EventViewController* so that it is a subclass of the *UITableViewController* class and add a few properties. Now the interface file will be thus:

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

@interface EventViewController : UITableViewController {
    NSArray * attributeNames;
    NSManagedObject * event;
}

@property (nonatomic,retain) NSArray * attributeNames;
@property (nonatomic,retain) NSManagedObject * event;

@end
```

You may be looking at the earlier declaration for the event and be wondering why you are using an *NSManagedObject* here instead of the *Event* class itself. At first, I did use the *Event* class, and then I had to import the header in the implementation files and use the class tag for it in the interface files, but I realized I was not doing anything with *Event* that an *NSManagedObject* could not do. This makes the *EventViewController* a very reusable class. You can use the table view in such a way that it can display any object at all. I could be talking about an *Event* object or a *Book* object or a *BottleOfWhiskey* object. It does not matter. Put them into your array, and this view controller will display each attribute of the object, each in its own row. OK, this may not seem cool the first time you are doing this, but the more code you have written, the more you get excited about truly reusable view controllers.

The following are the UITableViewDataSource methods that you need to put in the EventViewController implementation file. You also need a viewDidLoad method to do some setup:

```
- (NSInteger)tableView:(UITableView *)tableView numberOfRowsInSection:(NSInteger)section
{
    return [[[event entity] attributesByName] count];
}

- (UITableViewCell *)tableView:(UITableView *)tableView
cellForRowAtIndex:(NSIndexPath *)indexPath {

    static NSString * CellIdentifier = @"Cell";

    // Dequeue or create a new cell
    //
    UITableViewCell *cell = [tableView
        dequeueReusableCellWithIdentifier:CellIdentifier];
    if (cell == nil) {
        cell = [[[UITableViewCell alloc] initWithStyle:UITableViewCellStyleSubtitle
            reuseIdentifier:CellIdentifier] autorelease];
    }

    cell.textLabel.text = [[event valueForKey:[self.attributeNames
        objectAtIndex:indexPath.row]] description];
    cell.detailTextLabel.text = [self.attributeNames objectAtIndex:indexPath.row];

    return cell;
}

- (void)viewDidLoad {

    [super viewDidLoad];

    self.title = [[event entity] name];

    self.attributeNames = [[[event entity] attributesByName] allKeys];
}
```

With no more work than this, you can see in Figure 6-3 that you now have a much more interesting object view.



Figure 6-3. Now you see attribute names and values from any *NSManagedObject*.

You can use the `UINavigationController` here again and do something elegant. You created an `EventViewController`, a subclass of a `UITableViewController`, and pushed it in on top of the `RootViewController`. You can push other kinds of `UIViewController` objects onto this stack as well. So, you can create a view and controller that takes an attribute of an `NSManagedObject` and allows you to edit it. Most significantly, you can look at the type of data you have in the attribute and push on a `UIViewController` subclass configured to allow you to edit an attribute of that type.

For now, I have done this in the simplest manner possible. I created a `KeyValueViewController` class and a `KeyValueView` class. Make sure you drag both the header and implementation files for both of these classes from the downloaded code to your project. I override the `layoutSubviews` method of the `KeyValueView` and programmatically add the UI elements I need for editing the attributes. I am just displaying the values as strings by calling the `description` method on them, and then, when the value is changed in the text field, I have to use the edited string and create a new object of the appropriate type for the attribute. Every data type has a string serialization and some way to get that object from a string serialization, so this should always be possible.

There are more interesting views for particular data types that you could create. An obvious one would be a map that you could click to set the latitude and longitude attributes or a date picker. The point is that you can look at the type of the attribute for the data in the selected row and, for each type, come up with the appropriate view controller. Dynamically create that view controller and push it onto the navigation controller's stack, and everything works wonderfully.

Our Very First Crash, or Perhaps Not!

The `NSManagedObject` has many tools that come with it that allow you a lot of flexibility. Most people start out thinking of an entity as a wrapper for a table and attributes as wrappers for columns. You can stop there and still do powerful applications. But a lot more is possible. Be careful, though, because there are also some hidden traps. Every framework has them. You know this. The designers document these things, but are they proud of them? Do they make the documentation as easy to find or as obvious as they can? Perhaps not.

Before you do anything else here, you should make a small fix to your application that will prevent a crash and spare you the confusion it causes. Replace the `persistentStoreCoordinator` static method in the `LocationsAppDelegate` class with this:

```
- (NSPersistentStoreCoordinator *)persistentStoreCoordinator {
    if (persistentStoreCoordinator != nil) { return persistentStoreCoordinator; }

    NSURL *storeUrl = [NSURL fileURLWithPath: [[self applicationDocumentsDirectory]
stringByAppendingPathComponent: @"Locations.sqlite"]];

    NSError *error;
    NSDictionary * options = NSDictionary dictionaryWithObjectsAndKeys:
    [NSNumber numberWithInt:YES], NSMigratePersistentStoresAutomaticallyOption,
    [NSNumber numberWithInt:YES], NSInferMappingModelAutomaticallyOption,
    nil];
    persistentStoreCoordinator = [[NSPersistentStoreCoordinator alloc]
initWithManagedObjectModel: [self managedObjectModel]];
    if (![persistentStoreCoordinator addPersistentStoreWithType:NSSQLiteStoreType
configuration:nil
                                URL:storeUrl
                                options:options
                                error:&error]) {
        // Handle error
    }

    return persistentStoreCoordinator;
}
```

Then click the `xcdatamodel` file in your project, and select the Design ► Data Model ► Add Model Version menu option. Do not worry, for now, about what it does. It will make sense soon. The tutorial from Apple had you pass `nil` for the `options:` parameter of the `addPersistentStoreWithType:configuration:URL:options:error` method and did not talk about versions of the data model. But now that you have done these two things, you have gained some breathing space.

When I was working on this, before I knew how to deal with it, making changes to my schema caused me a major upset. At this point, I wanted to modify the Event entity that I had created. I added an attribute to my entity, which quickly led to a crash.

It seemed simple enough to do what I wanted. I innocently thought, “Hey, I want to add this other thing to my Event entity....” Just dive into that shiny, bright graphical editor,

and a few clicks later you created the whatever attribute, built the app, ran it, and ... oops.

```
Locations[20398:20b]*** Terminating app due to uncaught exception
'NSInternalInconsistencyException', reason: 'This
NSPersistentStoreCoordinator has no persistent stores. It cannot perform a
save operation.'
```

Hm. What is “no persistent stores”? But they were there just a minute ago, weren’t they? This does not seem very persistent. What happened? And what am I now reading at the end of the tutorial?

NOTE: One important item to remember is that, if you change the schema in your managed object model, your application typically won’t be able to open files you created with an earlier version.

CoreData Tutorial for iPhone OS: Managing Model Migrations

The problem is that I modified the schema in the project, but the database as it was last used by my application, and how it appears in the iPhone’s database files, does not match the new schema. But I cannot save objects into my new schema until I run the application. But I cannot run the application, because the database as it was last used by my application does not match the new schema. But I cannot save objects into my new schema until I run the application. And around and around we go.

The implications of this issue for the applications you ship to your users are profound, and I think it is worthwhile to understand what you can do about this issue and understand what it will cost you if you do not plan for it. You may think, right now, that you can define the schema for your application, and there will be no need to change it in the future. Not only is that probably not true, but if you do not do the right thing when there is a schema change, your application will crash. The user will have only one option. They will have to delete your application, getting rid of all of its data, and redownload the application again, without the data they had entered. Well, we hope they will redownload the application again. They may not.

The Easy Migrations Are Easy

Migrating schemas in any database systems is complicated, but automatic migrations can make it relatively painless.

I actually think it was a bit irresponsible for Apple to not include this option in the code in its tutorial. That first tutorial leads developers down a set path, a simple path, a clear path, but it is a path that leads directly to a cliff. The developers will try to change their schema, and their app will crash. Are the steps to take at this point documented? Yes,

but they are not very easy to find, given the severity of the error. The error says “no persistent stores,” and this does not lead one to immediately look to the “Core Data Model Versioning and Data Migration Programming Guide,” which Apple has published to lead you away from the cliff...or to lead you off the rocks at the bottom of the cliff. Apple engineers have told me that this “automatic” schema migration will handle most of what users will want to do. I believe them. They are justifiably proud of what it can do. But I do not know why they then sought to hide this functionality under a basket.

But let’s look at what you have been given. Using this option’s dictionary means that if the changes you make to your model are relatively simple, then the application’s `PersistentStoreCoordinator` instance will be able to figure out what to do. Or not. The problem here is that Apple knows that, really, migrating databases is incredibly hard. Or rather, it is one of those problems where handling the first 90 percent is deceptively easy, handling the next 8 percent gets obnoxious surprisingly quickly, and dealing with some part of the last 2 percent will take more years than the age of the universe to figure out. So, Apple has given you a black box. It is probably a very smart black box. After all, the people who used to make NeXT workstations know how to make black boxes. Let’s look at your project now. You had an `xcdatamodel` file, and now it has become an `xcdatamodeld` directory, with two `xcdatamodel` files in it. If you click the disclosure triangle next to the `Locations.xcdatamodeld`, you will see that one of the files is “current” and one is not. You can see that the file that is “current” is marked with a green check mark (see Figure 6-4).



Figure 6-4. Now you have two `xcdatamodel` files, one current and one not so current.

The first thing I always do when Xcode creates files with names like this is give the files better names. I renamed the `Locations.xcdatamodel` file to `Locations01.xcdatamodel` and renamed `Locations 2.xcdatamodel` to `Locations02.xcdatamodel`. You know, over the years, if you count up all the bugs in Xcode that were triggered by a space in a path or a resource name, there are just too many to count. Really, life is too short to take risks like that.

After you rename these files, take a moment to build and run the application. It should run without an error. But remember, you still have not made any changes to the schema. Before you do make any changes, select the `Locations02.xcdatamodel` file in Xcode, and select **Design** ► **Data Model** ► **Set Current Version**. The green check mark moves to the second file, and *now* you can make changes to this second file. For example, you can add a whatever attribute of type `String`, and then you would see the whatever attribute in the application and could even assign a value to it.

Well, I'm sure glad that's over! Actually, let's not get too excited. Make sure you do not run your application yet, because you are not finished making changes. You might want to take this opportunity to see what kinds of changes you can make to your schema and what kinds of things will cause problems. While you are developing your application and running it to debug something and then making changes and running it again, there are definitely ways to cause yourself a problem.

If you experiment with this now, you will probably be able to avoid more problems later. For example, if you create a second schema, set that schema to current, make changes to the second schema and run, you are good. If you switch the "current" marker back to the first schema, delete the second schema file, and build and run, you will crash. Why? The reason this crashes is that it could migrate your schema back from the second version to the first, but you deleted the second version. So, no migration is possible. Stop and think about this to make sure it makes sense to you. You need to get rid of the second schema version, but do not delete it until you set the "current" marker back to the first version and run the app. After you run the app successfully, *then* you can delete the second version of the schema. For the automatic migration to work, both the "from" and the "to" models have to exist in the project. And if you have run with the second version of the schema, do not just make other changes to it. Otherwise, you will back in the same place I was with my first change.

If you are iteratively making changes to the second version of the schema, you need to do things in a particular order. You can create your second version, make a change, and build and run the application. If you want to make another change to your second version, you need to switch back to the first version, build and run the application again, and only then you will be able to switch the "current" marker back to the second version and make any other change to that second version without causing a crash.

Adding a New Entity

Adding an entity is one of the changes that the system can handle for you, so let's at least do this much.

Select the latest (and presumably current) version of your data model, and add another entity to the data model. It is a `PartyFavor` entity, and I want to create attributes in it for a name (a `String`), a quantity (an `Integer 16`), and a price (a `Decimal`). Now I want to create my relationships. To make this clearer, do these steps in this order:

1. Click the `Event` entity and create a `partyFavors` relationship. Do not worry about making any changes to it.
2. Click the `PartyFavor` entity, and create an `event` relationship.
3. Click the `partyFavors` relationship in the `Event` entity. Select the `To Many Relationship` check box.
4. Set the `Destination Entity` drop-down menu to `PartyFavor`.
5. Then set the `Inverse Relationship` drop-down menu to `event`.

This last step actually finishes the job for you. It is very useful to use this “inverse relationship” feature in Core Data. In EOF, relationships had only an implicit inverse relationship. Developers usually had an inverse relationship for any relationship, but the two relationships were separate, and historically there were many problems when people created a relationship and an inverse that was misconfigured in some way. It is much harder to make this mistake in Core Data.

You can now create the new sources for the `PartyFavor` class and create new sources for the `Event` class. There is an odd thing that Xcode does here that you will see in a moment. When you select the `File ► New File` menu item, Xcode does not always allow you to pick the `Managed Object Class` template to create the file. One of the ways to make sure Xcode does allow you to pick that template is to click the `xcdatamodel` file. So, click the current `xcdatamodel` file. Then select the `File ► New File` menu item, and the multipane wizard launches. In the first pane, you are asked to select the target, but there is only one target in this project. In the second pane of the wizard, select both the `Event` and `PartyFavor` entities. Then it will create the source files for you. Oddly, Xcode will then put the sources in the `Resources` group of your project. Actually, they are inside the `xcdatamodeld` directory. But if you think about it, you will realize that you had the `xcdatamodel` file selected. So, Xcode only followed your suggestion, right? Of course, if you had not selected the `xcdatamodel` file, then Xcode (currently) would not offer you the use of the `Managed Object Class` template. It's strange looking, but in this case, what Xcode is doing is OK. Remember that, a while back, you added two methods to the `Event` class. You want to move the `description` and `subdescription` methods that you added to `Event` to the new `Event` sources, the one inside the `xcdatamodeld`. Remember to copy over the method declarations in the interface file as well. Now, your project should look like Figure 6-5.

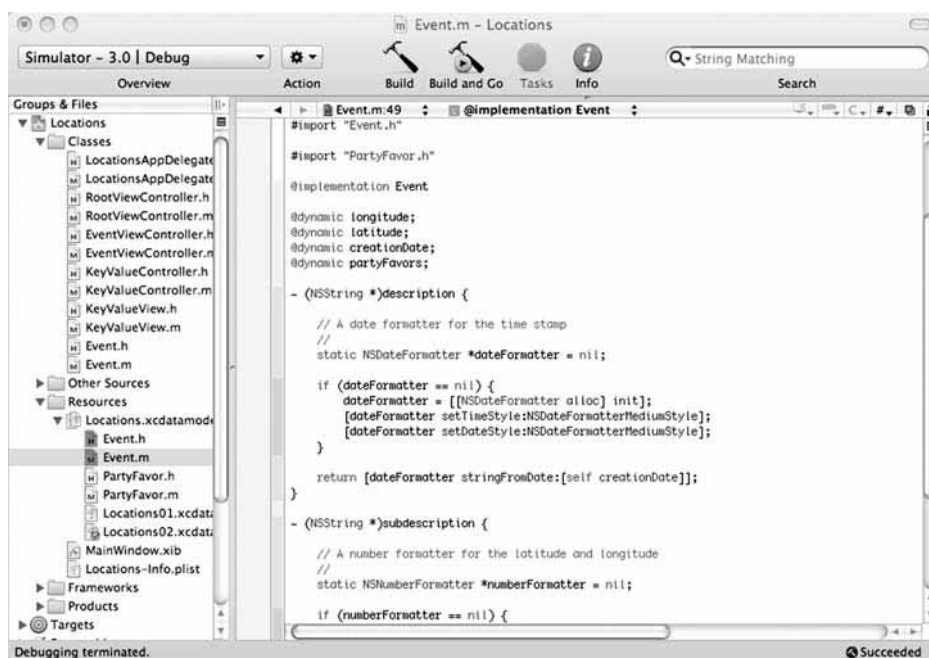


Figure 6-5. You have added the new sources. It looks strange, but Xcode in the beta versions of the iPhone SDK did something worse, so really, it is not so bad.

After editing the new copy of Event.h and Event.m, be sure to delete the old copies and move the files up to the Classes group in your project, as shown in Figure 6-6.

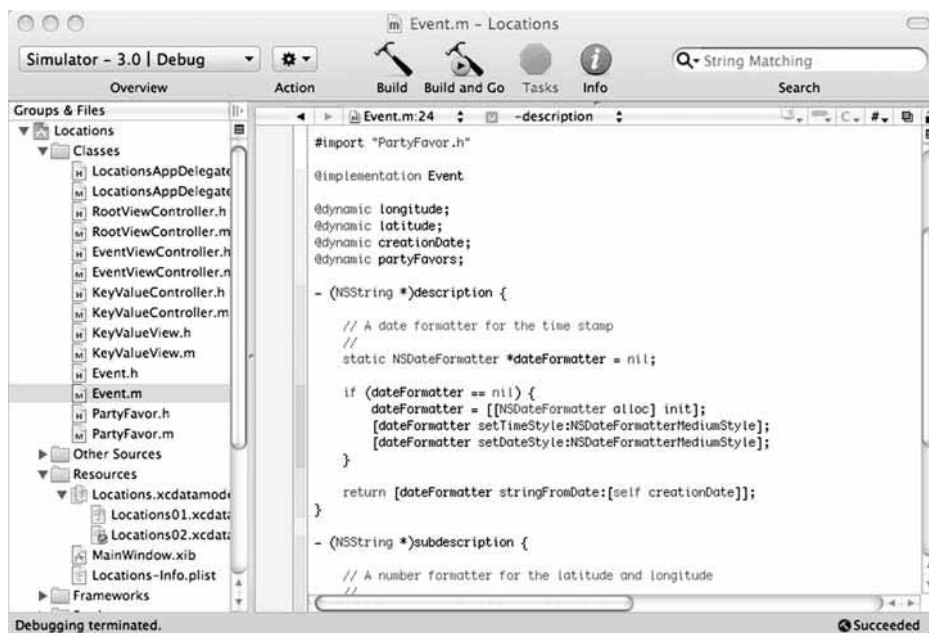


Figure 6-6. The project is looking quite a bit less strange now.

Using Key-Value Coding to Create a Reusable Object

Instead of going into more detail about what you can do to be sure that you can always migrate an object, and there would be a lot of detail, let's ask whether there is a way you can avoid having to change a schema. Surely not! You can write your application now and try to imagine everything you need to keep track of, but once people start using it, they are going to have suggestions. One of the downsides of putting an application out into the world is that you then have to deal with users, and users have suggestions and report bugs and behave in other inconvenient ways. You have to deal with users who have entered data into your application and who do not want you to upgrade your app in such a way that their data gets deleted.

When I was working on EOF and WebObjects for Apple, I created a class that uses KVC in an unusual way. I am going to use this trick again to make it so that the Event entity will not have to be changed, even when you want to see new attributes in the UI. To do this, add another entity to your schema. It will be helpful to create a third version of your schema at this point and add the entity in that version. Call the new entity EventExtra. Add an attribute called name and an attribute called value. These are both of type String. After you are done, your new Locations03.xcdatamodel file should look like Figure 6-7.

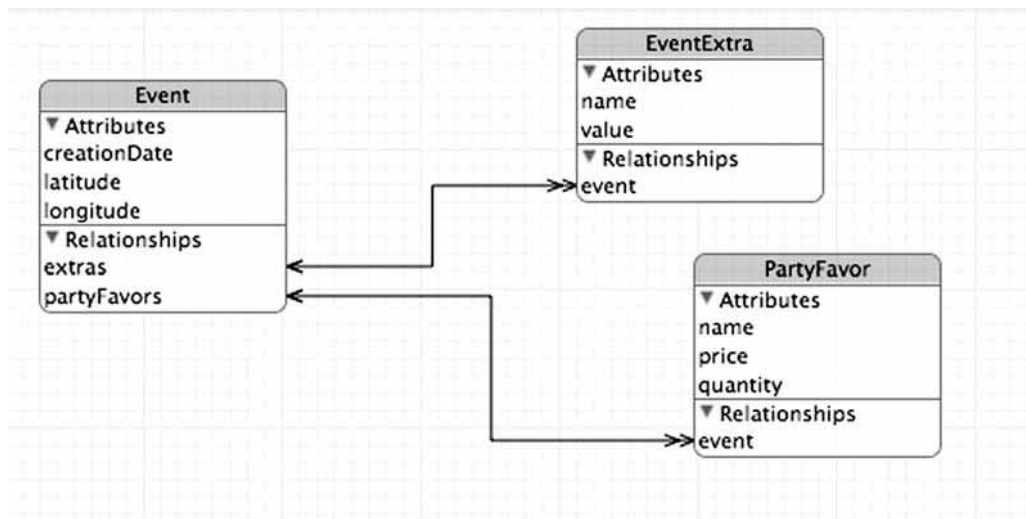


Figure 6-7. You're going to use the EventExtra entity in an unusual manner.

You are going to rely on something that the `NSManagedObject` gives you that most people do not take full advantage of. It is the fact that the KVC protocol has two methods for recovering from the use of nonexistent keys when reading from or writing to an object. Most developers do not override these methods, and the default behavior of these methods is to throw an exception. But these methods can be overridden, and they can be used to dynamically modify the attributes of an entity.

First, save your schema, and verify that your application works as it did before. You will see that the object has the same attributes. Adding a new relationship did not change that. Here is what you need to add to your `Event.m` file:

```

- (id)valueForUndefinedKey:(NSString *)name {
    NSArray * extra = [[self extras] allObjects];
    for (int idx = 0; idx < [extra count]; idx++) {
        if ([name isEqualToString:[extra objectAtIndex:idx] valueForKey:@"name"]]) {
            return [[extra objectAtIndex:idx] valueForKey:@"value"];
        }
    }
    return nil;
}

- (void)setValue:(id)value forUndefinedKey:(NSString *)name {
    // Look for existing object for this name. If one exists, replace its value.
    NSArray * extra = [[self extras] allObjects];
    for (int idx = 0; idx < [extra count]; idx++) {
        if ([name isEqualToString:[extra objectAtIndex:idx] valueForKey:@"name"]]) {
            [[extra objectAtIndex:idx] setValue:value forKey:@"value"];
        }
    }
    return;
}
// If an object for this name does not exist, create one.
NSManagedObject * eventExtra =
    [NSEntityDescription insertNewObjectForEntityForName:@"EventExtra"
inManagedObjectContext:[self managedObjectContext]];
[eventExtra setValue:name forKey:@"name"];
[eventExtra setValue:value forKey:@"value"];
[self addExtrasObject:eventExtra];
}

```

Think for a moment about what you can do with the Event entity now. Right now, its attributes are `creationDate`, `latitude`, and `longitude`. But suppose that one event is a child's birthday party and another is for Oktoberfest. In the first case, you can use code like this:

```
[kidsEvent setObject:@"Bruno" forKey:@"clownName"];
```

For the second case, you can use code like this:

```
[oktoberEvent setObject:[NSNumber numberWithInt:2]
forKey:@"dancingBearsCount"];
```

So, in the first case, the entity seems to have the attributes `creationDate`, `latitude`, `longitude`, and `clownName`, and, in the second case, the entity seems to have the attributes `creationDate`, `latitude`, `longitude`, and `dancingBearsCount`. Does this seem a little silly? Of course! But the point is that you can dynamically choose some new attribute for your Event entity and just stick the data in there, and it will work.

You do, though, have a detail to consider. You have to override the method that you use to get your list of available attributes for an entity. You can start with the list of attributes for your entity that is supplied to you by Core Data, but that no longer gives you the entire story. Now you have to look at the contents of the EventExtra entity. You have to return the distinct list of attributes that exist, so you have to get all the name entries from

all of those objects, create a nonduplicative list of names from this, and add that to what you have gotten from Core Data. You can add this method to your `Event.m`:

```
- (
    NSArray *)attributeNames {

    NSEntityDescription * extrasEntity = [[[self entity] managedObjectModel]
entitiesByName] objectForKey:@"EventExtras"];

    NSFetchRequest * request = [[NSFetchRequest alloc] init];

    [request setEntity:extrasEntity];

    NSArray * fetchResults = [[self managedObjectContext] executeFetchRequest:request
error:nil];

    NSMutableSet * extraAttributes =
[[NSMutableSet alloc] initWithSet:[NSSet setWithArray:[[[self entity] attributesByName]
allKeys]]];

    NSArray * foundNames = [fetchResults valueForKey:@"name"];
    for (int idx = 0; idx < [foundNames count]; idx++) {
        if ([foundNames objectAtIndex:idx] != [NSNull null])
            [extraAttributes addObject:[foundNames objectAtIndex:idx]];
    }

    return [extraAttributes allObjects];
}
```

Now, you need to change the methods in `EventViewController.m` that use the list of attributes:

```
- (NSInteger)tableView:(UITableView *)tableView numberOfRowsInSection:(NSInteger)section
{
    return [[event attributeNames] count];
}

- (UITableViewCell *)tableView:(UITableView *)tableView
cellForRowAtIndex:(NSIndexPath *)indexPath {
    static NSString * CellIdentifier = @"Cell";

    // Dequeue or create a new cell
    UITableViewCell *cell = [tableView
dequeueReusableCellWithIdentifier:CellIdentifier];
    if (cell == nil) {
        cell = [[[UITableViewCell alloc] initWithStyle:UITableViewCellStyleSubtitle
reuseIdentifier:CellIdentifier] autorelease];
    }

    cell.textLabel.text = [[event valueForKey:[event attributeNames]
objectAtIndex:indexPath.row] description];
    cell.detailTextLabel.text = [[event attributeNames] objectAtIndex:indexPath.row];

    return cell;
}
```

Now you have an object into which you can create new attribute values, but you have no UI to do this in your application. It is easy enough, though, to add this. When you are looking at a single object and seeing the list of its attributes and values, you want a way to add an attribute. If you look back at `RootViewController.m`, you will see how this can be done. You can add a + button, something the `UINavigationController` makes it easy to do. Add this code to the `viewDidLoad` method of the `EventViewController.m`:

```
addButton = [[UIBarButtonItem alloc]
initWithBarButtonSystemItem:UIBarButtonSystemItemAdd target:self
action:@selector(addAttribute)];
addButton.enabled = YES;
self.navigationItem.rightBarButtonItem = addButton;
```

That code is calling the `addAttribute` method, and you still need to add that to `EventViewController.m` as well.

```
- (void)addAttribute {
    KeyValueController * eventController = [[KeyValueController alloc] init];

    [eventController setAttributeName:@"extra"];
    [eventController setEvent:self.event];

    [[LocationsAppDelegate delegate].navigationController
pushViewController:eventController animated:YES];
    [eventController release];
}
```

Since the `KeyValueView` uses a `UITextField` for the attribute's name as well as the attribute's value, this is all you need to do to get a working application. Now you may never change the `Event` entity again. If you have code in your interface that creates data for a new key and that calls the object with a new key to retrieve the data, the `Event` object will automatically appear to have an attribute of that name. In Figure 6-8, I have added an `extraName` attribute with the value `extraValue`.

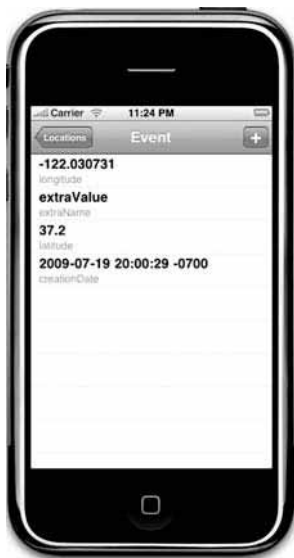


Figure 6-8. *Event with the dynamically created, potentially more interesting, attribute and value.*

It may seem that you are assuming here that anything that you are going to add to your Event entity is of type String. This does not turn out to be true. Remember that you can override the default setters in your `NSManagedObject` subclass. So, although it is true that the “database” view of the newly added attribute is that it is a String, you can write methods for getting and setting a value of whatever type you want. There are, if anything, too many things you can do with just a name of an attribute. Since you do not want to change the Event entity, you cannot add the attributes to the diagram that the `xcdatamodel` gives you, but almost everything else will work. Want to use `NSBindings`, recently brought to the iPhone with OS 3.0, on a UI element? The keys for the bindings are just strings, and they go through the same key/value interfaces that you have implemented on your event. To make all the bindings work, you may have to implement code for a relatively new addition to Core Data, that being KVO. But even without that, you will be surprised by how many things will just work.

Remote Databases: It's All Net!

One way to give yourself some flexibility while adding power to your application is to not just store your users' data on the iPhone but store it on the Internet as well. Given the pervasive connectivity of the iPhone, it is also easy to do.

You can set up connections to remote databases in a few ways. In the first application I had published on the App Store, it would upload only very small sets of numbers. I packaged these into URLs and opened a connection to a web application that I had running. The web application would peel off the GET parameters of the URL and store them in a MySQL database. The web application was a trivial Perl CGI, and there was not much to it.

Later I used the `NSMutableURLRequest` class so that data could be sent up as a POST. This enables the transfer of more data. For example, since I use WebObjects applications on the server side, I use the `NSPropertyListSerialization` class on an iPhone to serialize an arbitrarily complex dictionary or array. I can pass the resulting plist up as a POST request, and the WebObjects app can read the plist directly. One could just as easily serialize the data on the iPhone as XML and parse the XML on the server.

This code will take a dictionary, which can include arrays, other dictionaries, and other objects and generate a plist for it, which can be sent to a remote machine.

```
- (void)exportData:(id)sender {
    NSData * plistData = [NSPropertyListSerialization
dataFromPropertyList:(id)[NSDictionary dictionaryWithObject:YourDictionaryHere
forKey:@"parameters"] format:NSPropertyListXMLFormat_v1_0 errorDescription:nil];

    NSMutableURLRequest * request = [[NSMutableURLRequest alloc] init];
    [request setURL:[NSURL URLWithString:@"YourURLHere"]];
    [request setHTTPMethod:@"POST"];
    [request setHTTPBody:plistData];
}
```

```

        NSURLConnection * theConnection = [[NSURLConnection alloc]
initWithRequest:request delegate:self];

        [theConnection release];
    }
}

```

// We need to implement these two methods in this class if we want to be notified of the success or failure of the NSURLConnection above

```

//
-(void)connection:(NSURLConnection *)connection didReceiveResponse:(NSURLResponse
*)response { }
-(void)connection:(NSURLConnection *)connection didFailWithError:(NSError *)error { }

```

You may believe that a plist (or property list) is an Apple-only technology, but there is a CPAN module, `Mac::PropertyList::SAX`, that gives you exactly what you need:

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
"http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
  <key>parameters</key>
  <array>
    <dict>
      <key>X</key>
      <real>127.17512512207031</real>
      <key>Y</key>
      <real>394.22320556640625</real>
    </dict>
    <dict>
      <key>X</key>
      <real>161.1126708984375</real>
      <key>Y</key>
      <real>193.724853515625</real>
    </dict>
    <dict>
      <key>X</key>
      <real>57.804325103759766</real>
      <key>Y</key>
      <real>371.4683837890625</real>
    </dict>
  </array>
</dict>
</plist>

```

This structure can be read with a Perl CGI. The script has some funkiness from metacharacters in the plist, and when I wrote this script, I used the `-best-possible-way=false` flag:

```
% cat test.cgi
#!/usr/bin/perl

use Mac::PropertyList::SAX qw( parse_plist );
use CGI;

my $q = CGI->new;

print $q->header();
print $q->start_html('Your Title Here');

$r = $q->Dump;

$r =~ s/\&lt;/>/g;
$r =~ s/\&gt;/>/g;
$r =~ s/\&quot;/\'/g;
$r =~ s/<br \>/g;

if ($r =~ /<\!DOCTYPE plist PUBLIC/) {
    $r = $&.$';
    if ($r =~ /<\plist>/) { $r = $`.$&; }
}

my $data = parse_plist( $r );

$val = $data{'parameters'};
for ($idx = 0; $idx < scalar(@$val); $idx++) {
    print "coord[".$idx."] = (\".$val[$idx]{'targetX'}.\",
    ".$val[$idx]{'targetY'}.)\".$q->br;
}
```

When you send this plist in a POST to this script, the resulting page will be as follows:

```
coord[0] = (127.17512512207031, 394.22320556640625)
coord[1] = (161.1126708984375, 193.724853515625)
coord[2] = (57.804325103759766, 371.4683837890625)
OK
```

There is something to think about when you are transmitting data across the network. You must be careful about whether the UI of the app reflects the network connectivity. If the app transmits data, you may want to not have side-effects of the transmission, such as progress bars, in the UI of your application. The Human Interface Guidelines from Apple say that if connectivity is apparent in your application, then you must alert the user when the network is not available.

Apple's interpretation of this can seem odd. If you have a game and you load high scores onto a server, think about what will happen if the network is not available. If you are showing the user something from the online system when they get a high score, for example, then you may have a problem. Every time your application is being used and the network is or becomes unavailable, Apple is going to say that you need to notify the user. They may hit high scores only once a month, and they may end up getting this notification every time they drive under a bridge, but that does not matter. Or, you could not have anything from the Internet visible to the users when they hit a high score, and then you will not need to worry about it.

Or you might be transmitting *analytics*, information about how often your app is being used or how it is being used. If you have something in your UI that tells the user about this or shows them when transmissions are occurring, you could have trouble. Apple has asked that an alert pop up, letting users know that the data will be transmitted, giving them a way to opt out, and letting them know what happens when the network is down. Or you can have analytics data come from your app to the network, say absolutely nothing to the user about it, as a legion of applications on the App Store do, and you have no problem at all.

It may be smartest to assume that, if the network is down, you should be able to wait an arbitrarily long time before you are able to transmit, and it may be best to not involve the user in the decision about when to transmit. It is easiest to get through the App Store check if you design your application so that a lack of connectivity has no visible effect.

Also, I wish that we could stop there and just ignore the fact that using encryption on an iPhone can be a problem. Unfortunately, if you want to use a 128-bit HTTPS connection in your application and you want to sell or provide it for free outside the United States, Apple wants you to have a license to export it. You may be able to transmit data in the clear or with nonrestricted encryption technologies. All I can say is that I am fairly bureaucracy-phobic, but it is not as hard to deal with the U.S. Commerce Department as you think it might be. You can even get an account on an online system for requesting licenses. I wish we did not have to deal with this, but Apple has set up the rules for the App Store the way it wants. Right now, it behooves us to play the game their way. At some point, we may become more interested in other technologies or find other ways to distribute applications. Until then, here we are.

Summary

Core Data makes it possible to use the database on the iPhone to store information without resorting to SQL. It can be easy to work with if you avoid falling into a few traps. Migrating data schemas leads to some complications, and I have demonstrated two different ways of dealing with the issues. You can use the automatic schema migration, as is provided by Xcode, and you can create a generic object that will allow you to hold many different kinds of objects in one type of data object.

The KVC protocol is simple, but it is a powerful tool for abstracting and separating concerns in an application. Data schemas should be kept as simple as possible. Users have complex needs, but smart designs can let the user interfaces change while the model objects remain unchanged. Different kinds of views can be presented, but the information coming back from them can be kept simple. Different applications can be served by one database schema. As users' needs evolve and applications become more complex, that simplicity will make progress easier.

And we have information, of course, that we want to share. The “mostly connected” nature of the iPhone makes it possible to share data across the network at any time, but applications have to be agile to deal with the real world, where people move around faster than software can figure out how to switch their networks, where sun spots cause

outages, and where people cannot really ever be “always connected.” Smart designs can make our applications agile enough to cope with interruptions without making the user wait, watching a spinning ball.

Designing data applications will never be easy, but with all the kinds of games, puzzles, tools, “find it” apps, and “track it” apps that the world seems to want, the iPhone does make it fun.

Steve Finkelstein



Company: *Lime Medical LLC*

Location: *New York, New York*

Former Life As a Developer: *I'm language and platform agnostic and embrace any technology that is best fit for the job at hand. One day I might be fiddling with automation on my FreeBSD/Linux servers using Perl – other days I'll be configuring Cisco Catalyst switches. Some of my specialties include backend server programming (C, Objective-C, PHP, Perl, SQL), Web Services (SOAP, REST), client-side programming (Cocoa, Cocoa-Touch, JavaScript, CSS, HTML). I'm mostly concentrating on iPhone and Web development at the time of this writing, but I also have vast experience in working for media companies such as About.com (part of The New York Times Company) and Community Connect Inc.*

Life as an iPhone Developer: *My first application is not available at the time of this writing. It will be a medical charge capture application and released late summer.*



What's in This Chapter: *This chapter focuses on strategies for taking an application that's dependent on network connectivity, and having it function while it is offline. We'll be building a sample e-mail client that has one task in mind, drafting and sending messages. We will show how to work with the System Configuration Framework and Apple's Reachability class in order to detect when we're offline. We'll also use some fantastic open source libraries such as Three20, FMDB and SKPSMTP in order to achieve our goals. Our Offline Mailer will continue to function independently of network status.*

Key Technologies:

- **System Configuration Framework**
- **Three20**
- **FMDB**
- **SMTP**

Smart In-Application E-mail with Core Data and Three20

The first time I purchased a cell phone, it resembled something Zach Morris would wield in an episode of *Saved By the Bell*. (In the event the name Zach Morris does not spark some neurons, think of a device that's discernibly colossal in size.) Those days, mobile devices were in a primitive state. Extremely fundamental usage was all that was incorporated into the hardware. You needed to be able to make and receive a phone call, and as long as you were capable of doing that, you were thrilled!

It has been a long time since the advent of the “Zach Morris phone.” The transformation has been nothing short of brilliant. Our phones are mini-computers that we use not only to communicate but as a tool to keep our life organized while we're on the go. I cannot even begin to fathom how my everyday life would continue to exist without the iPhone. The iPhone isn't just a tool that facilitates phone calls to your mother who complains that you never reach out to her. It's a behemoth of a platform that's capable of gaming, keeping up with social media, following the news, managing your diabetes, sending and receiving e-mail, and much more.

I personally am a zealot when it comes to the likes of applications such as Facebook, Instapaper, Things, NYTimes, Twitterrific, Mail, and Wikipanion. Just a side note, I love Tweetie and Twitterrific so much that I typically flip a coin on a daily basis when deciding what Twitter client to use. The one prudent point I want to make here, though, is the following: all of these applications have one thing in common. Can you think of what the common functionality is? It might not be apparent at first as you think, “What does a news app have in common with Facebook?” The answer is straightforward, though—these apps continue to work when you're in an area without Edge, 3G, or WiFi coverage.

I, for one, took it for granted that every single time I would go through the Lincoln Tunnel or ride the subway in New York City that I could continue reading the news on the

NYTimes App. And I took it for granted that I could write an important e-mail for work and have it delivered to a remote SMTP server once my connectivity had resumed. I also adore the capability that I have to flag articles from folks I follow on Twitter and save them to Instapaper. Then, once I'm airborne to a foreign destination, I'm reading everything while in airplane mode. Although the aforementioned applications are nice to have, what do we do when a mission-critical application is not capable of functioning because a physician practices in an outpatient facility that lacks any network connectivity? The physician might want to take notes or look up allergies that a particular medication might give to the patient they're visiting. I can personally vouch for this particular use case—some demographics are substantially more forgiving than others. Depending on who you're building applications for, not having the capability to continue operating an application might mean the difference between retaining a paying customer and having them go to your competitor.

Planning a Simple Offline SMTP Client

In this chapter, I'm going to take you—a resilient iPhone programmer—on a journey to better your application with ideas on how to take it offline. Since we iPhone developers never give up on a problem no matter how egregious it might appear on the surface, you're going to delve into a sample application I had some fun building. The example that I have built is called OfflineMailer. This OfflineMailer implements the very basics of an SMTP client. It meets the following criteria:

- Capable of sending an e-mail to multiple recipients in your Address Book.
- Capable of taking your messages and storing them in persistent storage for later viewing.
- Capable of determining the availability of an Edge, 3G, or WiFi network.
- Capable of taking any draft messages written in offline mode and keeping them around until you have network connectivity available again. Once network connectivity is available, you will send the messages.
- Simple viewing of messages in an offline queue and of sent messages.

Given the stringent criteria I have to meet working on medical applications, I would have loved to have a chapter such as this available for my own personal reading when I first started mobile programming. Although it appears complicated on the surface, you'll quickly see how trivial it really is to build this type of application. The best part of it all is that I'm going to provide a brief introduction to one of my favorite open source iPhone projects—Three20 by the venerable Joe Hewitt (<http://github.com/joehewitt/three20/tree/master>). In addition to Three20, I'll cover the other open source projects I've incorporated into the demo application.

Three20 is an Objective-C library that has very well-written classes that include the likes of a photo viewer and an HTTP disk cache in addition to the capability to style labels and UIViews. Joe Hewitt is one of the original authors of Firefox, is the creator of Firebug, and is the engineer of the Facebook iPhone App. Simply put, the guy is brilliant, and in the years I've been developing, his work has affected me directly.

I originally began writing this project by using a notoriously powerful open source project known as FMDB. FMDB, written by Gus Mueller, is a bunch of well-written Objective-C classes around SQLite. With the iPhone 3.0 SDK now available and not covered by the NDA, I've rewritten the chapter to incorporate Core Data. For those of us who must still support 2.x devices, I will provide the original code that I used with FMDB to you. The fundamentals for the two approaches remain closely knit in theory. If you're building an application for 3.0, I highly encourage the route of Core Data for reasons that I'll talk about in this chapter.

Finally, one of the most critical components of the project in this chapter is an open source package named SKPSMTP, released by developer Ian Baird. SKPSMTP allows for sending SMTP e-mails from within your iPhone application.

If you've noticed a common trend, I'm a huge advocate of open source, and I embrace it in my work whenever I can. Open source has been a prominent player in helping me learn how experts write professional software. As long as you follow projects that are credible, open source can change your outlook on how you write software in addition to helping you improve the software you're already working on.

Creating the User Interface

I'm going to take for granted that you already know how to create basic user interfaces for your iPhone. If you need some help with building user interfaces, either programmatically or with Interface Builder, I highly recommend picking up a copy of *Beginning iPhone Development: Exploring the iPhone SDK* (Apress, 2008) by Dave Mark and Jeff LaMarche. Dave and Jeff are pioneers in the software vertical and make learning how to program on the iPhone really fun. They're also my inspiration and the reason I've had the opportunity to write this chapter.

Diving into Xcode

Let's kick things off with Xcode. Xcode is the lovable IDE that you're going to build the sample application in. I've avoided building any of the user interface elements in Interface Builder to make this chapter slightly easier to follow. This does not by any means imply that I do not support Interface Builder. Quite the contrary, I use it extensively whenever I can. With that said, I do prefer building my views with code whenever the interfaces are trivial.

In Xcode, create a new project, and choose the Navigation-Based Application template, as shown in Figure 7-1. I named my application OfflineMailer, so I suggest you do so also in order to make identifying symbols from the text easier.

NOTE: If you do not have MacFUSE and its associated developer tools installed, you will not see the respective templates available in your Xcode build. Do not worry—that is not a requirement for this demonstration.

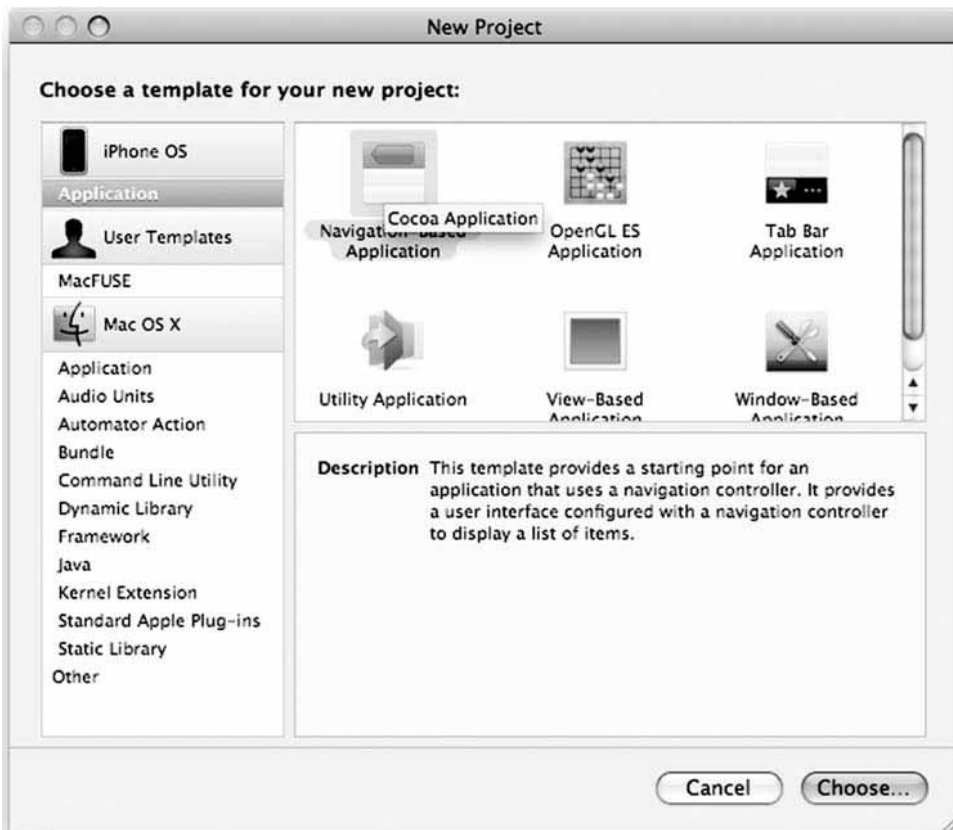


Figure 7-1. Choosing the Navigation-Based Application for the sample application

Next, you need to add a few packages to your application including both FMDB (downloadable from <http://code.google.com/p/flycode/source/checkout>) and Apple's Reachability set of classes. Reachability is a set of classes that Apple provides as a sample to determine the network state of an iPhone or iPod touch device. I use this set of classes exclusively in my projects, because they're well written and plug in to any existing project with ease. If you have an iPhone developer account, the sample code is available at <http://developer.apple.com/iphone/library/samplecode/Reachability/index.html>. Otherwise, feel free to grab it from the source code that goes with this chapter.

In your Groups & Files pane, add a new group underneath the Classes folder. Let's call it Network. In Network, you need to add two files from Apple's sample code—Reachability.h and Reachability.m. Let's move forward to some code now, shall we?

Setting Up Instance Variables in OfflineMailerAppDelegate.h

In `OfflineMailerAppDelegate.h`, you have some instance variables, properties, and instance methods to add. Make sure your code looks like Figure 7-2.

```

9  #import <UIKit/UIKit.h>
10 #import "Reachability.h"
11
12 @interface OfflineMailerAppDelegate : NSObject <UIApplicationDelegate> {
13
14     // Core Data Additions
15     NSManagedObjectContext *managedObjectContext;
16     NSManagedObjectContext *managedObjectContext;
17     NSPersistentStoreCoordinator *persistentStoreCoordinator;
18
19     UIWindow *window;
20     UINavigationController *navigationController;
21
22     // Network status
23     NetworkStatus remoteHostStatus; // server status.
24     NetworkStatus internetConnectionStatus; // carrier data network.
25     NetworkStatus localWiFiConnectionStatus; // wifi network.
26     BOOL hasNetworkConnection;
27 }
28
29 @property (nonatomic, retain) IBOutlet UIWindow *window;
30 @property (nonatomic, retain) UINavigationController *navigationController;
31
32 @property NetworkStatus remoteHostStatus;
33 @property NetworkStatus internetConnectionStatus;
34 @property NetworkStatus localWiFiConnectionStatus;
35 @property (assign) BOOL hasNetworkConnection;
36
37 @property (nonatomic, retain, readonly) NSManagedObjectContext *managedObjectContext;
38 @property (nonatomic, retain, readonly) NSManagedObjectContext *managedObjectContext;
39 @property (nonatomic, retain, readonly) NSPersistentStoreCoordinator *persistentStoreCoordinator;
40
41 @property (nonatomic, readonly) NSString *applicationDocumentsDirectory;
42
43 - (IBAction)saveAction:sender;
44
45 #pragma mark -
46 - (void)reachabilityChanged:(NSNotification *)notification;
47 - (void)updateNetworkStatus;
48
49 @end

```

Figure 7-2. This is what `OfflineMailerAppDelegate`'s header file looks like when it's completed.

Here are some critical components to the architecture. You've added three instance variables to track the network state of your application. `NetworkStatus` is an enum defined by the `Reachability` set of classes. The enum provides you with an elegant interface that returns values for the different network states your device may be in. An iPod touch will never be able to speak through a carrier data network interface, but it will be capable of obtaining a WiFi connection. You also set a Boolean here that simply holds the state of the network. I'll get to what exactly `managedObjectContext`, `managedObjectContext`, and `persistentStoreCoordinator` are soon.

Finally, you set two instance methods:

```
(void)reachabilityChanged:(NSNotification *)notification;
(void)updateNetworkStatus;
```

The `reachabilityChanged:` message gets invoked whenever the status of your network changes. This method is responsible for dispatching a message to `updateNetworkStatus` so that it may update your ivar (instance variables) states accordingly. Figure 7-3 shows what both of those methods look like in `OfflineMailerAppDelegate.m`.

```

236 #pragma mark -
237 #pragma mark Network Connectivity
238 - (void)reachabilityChanged:(NSNotification *)notification
239 {
240     [self updateNetworkStatus];
241 }
242
243 - (void)updateNetworkStatus
244 {
245     // Query the SystemConfiguration framework for the state of the device's network connections.
246     self.remoteHostStatus = [[Reachability sharedReachability] remoteHostStatus];
247     self.internetConnectionStatus = [[Reachability sharedReachability] internetConnectionStatus];
248     self.localWiFiConnectionStatus = [[Reachability sharedReachability] localWiFiConnectionStatus];
249
250     if (self.remoteHostStatus == NotReachable || self.internetConnectionStatus == NotReachable) {
251         // we aren't reachable.
252         self.hasNetworkConnection = NO;
253     } else {
254         self.hasNetworkConnection = YES;
255     }
256 }
257
```

Figure 7-3. *The `reachabilityChanged:` and `updateNetworkStatus` methods*

`NSNotification` is one of my favorite classes that is available in the Foundation framework. Notification programming is a very powerful tool to use in iPhone and Mac development. If you've ever heard of the Observer pattern, then you're already ahead of the game in understanding what behavior this family of classes has. In the application delegate, you registered with the `NSNotificationCenter`, a singleton object that exists throughout the entire application, to invoke the selector `reachabilityChanged:` whenever the `kNetworkReachabilityChangedNotification` is triggered. That notification name is defined in `Reachability.m`.

Ultimately what happens is that every time the network state changes, you invoke `updateNetworkStatus`. This method will set the network state accordingly. Later whenever you want to know whether you have access to the network, you simply query the application delegate with the following code:

```

if (appDelegate.hasNetworkConnection) {
    // We have network connectivity!
} else {
    // We lack network connectivity!
}

```

Simple, huh? I told you this would be easier than you originally thought it would be. Later, you'll also see how I include a visual representation of a light bulb that imitates the current state of the network. The light bulb will glow a vibrant yellow when you have access. It will dim whenever the network state changes on the fly. I'll show you how that is done later as well.

When the application finishes launching, you do a few other things in `applicationDidFinishLaunching:` to set up your app. Let's take a look at what the code in `applicationDidFinishLaunching:` looks like; see Figure 7-4.

```

42 - (void)applicationDidFinishLaunching:(UIApplication *)application {
43
44     // register for offline/online notifications.
45     [[NSNotificationCenter defaultCenter]
46      addObserver:self
47      selector:@selector(reachabilityChanged:)
48      name:@"kNetworkReachabilityChangedNotification"
49      object:nil];
50
51     AccountViewController *rootViewController =
52     [[AccountViewController alloc] init] autorelease];
53     [rootViewController setManagedObjectContext:[self managedObjectContext]];
54
55     [[DataManager sharedDataManager] setManagedObjectContext:[self managedObjectContext]];
56
57     // SMTP might not require authentication, so we only check if there
58     // is a remote hostname available.
59     NSString *remoteServer = [[DataManager sharedDataManager] hostname];
60
61     // Here we ensure we have the settings we need to proceed.
62     if (remoteServer == nil)
63     {
64         SettingsViewController *settingsController = [[SettingsViewController alloc] init] autorelease];
65         navigationController =
66         [[UINavigationController alloc]
67          initWithRootViewController:settingsController];
68         navigationController.navigationBarHidden = YES;
69     } else {
70         navigationController =
71         [[UINavigationController alloc]
72          initWithRootViewController:rootViewController];
73
74         [[Reachability sharedReachability] setHostName:remoteServer];
75         [self updateNetworkStatus];
76     }
77
78     //BOOL success = [self initDatabase];
79
80     // Note, we might not want to throw an exception if the database isn't
81     // reachable in a production application. We could instead let the application
82     // continue to launch as expected. However, when offline, we need to take heed
83     // and let our users know.
84     if (!success) {
85         NSLog(@"Failed to initialize database.");
86     }
87
88     [window addSubview:navigationController.view];
89     [window makeKeyAndVisible];
90 }

```

Figure 7-4. *OfflineMailerAppDelegate.m's applicationDidFinishLaunching: method*

Initializing the UIApplication Delegate

`OfflineMailerAppDelegate` conforms to the `UIApplicationDelegate` protocol. As such, it receives the `applicationDidFinishLaunching:` notification once an application has launched and initializes. Delegates implement this method to typically set up the window and its respective subviews. In addition to that, you write some code of your own, as described next.

First, you check whether the user who's using your application has gone into the Settings application to set up their mail server settings. Here the user is expected to know the hostname of their SMTP server. SMTP is the acronym for Simple Mail Transfer Protocol, the protocol used to relay e-mail. The user needs to put in their username and password, where applicable. If the user doesn't put in their SMTP server, you prompt them with a warning on the screen.

To avoid being used as a spam relay, most SMTP servers will enforce user authentication. Also, here you set up a default SMTP port. However, not every SMTP server listens in on 25. For instance, my ISP blocks all outbound port 25 connections that use the TCP protocol. TCP/25 is the standard protocol/port that's defined in RFC 821 (<http://www.faqs.org/rfcs/rfc821.html>) for SMTP. For me to write and test this application, I had to open my SMTP server to listen on TCP port 2500. How to do that is unfortunately beyond the scope of this chapter. There are plenty of informative web sites and books that detail how to do this in the server of your choice, my preference being Postfix.

One other note I'd like to add here is that the iPhone developer community has gone through extensive back and forth threads on whether settings should be included directly inside your application or in the Settings application. I advocate putting Settings that are rarely changed into the Settings app. Ones that get frequently changed such as turning volume in your app on or off should go directly in the app itself. Craig Hockenberry of Twitterrific fame has put up a great blog post that you can read more about at <http://furbo.org/2009/04/30/matt-gallagher-deserves-a-medal/>. I love this blog post because Craig links to two of my other heroes, Loren Brichter of Tweetie fame and the venerable Matt Gallagher of <http://www.cocoawithlove.com> fame. I highly recommend you check out all three of these programmers—they're super smart and inspire me every time I write any code.

Working with Core Data

Let's now jump into some of the cool parts of this project. In this section, you'll jump in and get your feet wet with Core Data. You initially should have created your application with the "Use Core Data for storage" option, as depicted in Figure 7-5.



Figure 7-5. New project template with “Use Core Data for storage” option

What this does is create several utility methods in your AppDelegate. The code for the methods looks something like this:

```
/**
 *applicationWillTerminate: saves changes in the application's managed object context
 *before the application terminates.
 */
- (void)applicationWillTerminate:(UIApplication *)application {
    NSError *error;
    if (managedObjectContext != nil) {
        if ([managedObjectContext hasChanges] && ![managedObjectContext save:&error]) {
            // Handle error.
            NSLog(@"Unresolved error %@, %@", error, [error userInfo]);
            exit(-1); // Fail
        }
    }
}

/**
 *Returns the managed object context for the application.
 *If the context doesn't already exist, it is created and bound to the persistent store
 *coordinator for the application.
 */
- (NSManagedObjectContext *)managedObjectContext {
```

```

if (managedObjectContext != nil) {
    returnmanagedObjectContext;
}

NSPersistentStoreCoordinator *coordinator = [selfpersistentStoreCoordinator];
if (coordinator != nil) {
    managedObjectContext = [[NSManagedObjectContextalloc] init];
    [managedObjectContextsetPersistentStoreCoordinator: coordinator];
}
returnmanagedObjectContext;
}

/**
 Returns the managed object model for the application.
 If the model doesn't already exist, it is created by merging all of the models found in
 the application bundle.
 */
- (NSManagedObjectModel *)managedObjectModel {
    if (managedObjectModel != nil) {
        returnmanagedObjectModel;
    }
    managedObjectModel = [[NSManagedObjectModelmergedModelFromBundles:nil] retain];
    returnmanagedObjectModel;
}

/**
 Returns the persistent store coordinator for the application.
 If the coordinator doesn't already exist, it is created and the application's store
 added to it.
 */
- (NSPersistentStoreCoordinator *)persistentStoreCoordinator {
    if (persistentStoreCoordinator != nil) {
        returnpersistentStoreCoordinator;
    }

    NSURL *storeUrl = [NSURLfileURLWithPath: [[selfapplicationDocumentsDirectory]
stringByAppendingPathComponent: @"offlinemailer.sqlite"]];

    NSError *error;
    persistentStoreCoordinator = [[NSPersistentStoreCoordinatoralloc]
initWithManagedObjectModel: [selfmanagedObjectModel]];
    if
    ([persistentStoreCoordinatoraddPersistentStoreWithType:NSSQLiteStoreTypeconfiguration:nil
ilURL:storeUrl options:nilerror:&error]) {
        // Handle error
    }

    returnpersistentStoreCoordinator;
}

```

You might be thinking there is a great deal of new stuff here. And you're right—if you've never built a Mac application with the Core Data stack before, there is quite a bit of new API code being thrown at you here. Although the complexity of what Core Data achieves underneath the hood is massive, understanding the API isn't all that tedious.

Understanding the Core Data Stack

Let's chat a few about the relevant Core Data stack objects here including the managed object model (MOM), managed object context (MOC), and the persistent store coordinator. Although one chapter, let alone a few paragraphs, cannot explore all of Core Data, I plan on taking you on a quick tour of the technology to help you understand what exactly Core Data is and how you can use it. There's an abundance of information on the Internet, including the ADC, which covers Core Data in detail. If you're interested in learning more about the topic after you've read this chapter, I highly recommend searching for Bill Dudney's and Marcus Zarra's work on Core Data. Both of these developers are highly regarded in the Mac and iPhone communities. I've had the pleasure of learning how to write my first iPhone application sitting in Bill Dudney's class over at Pragmatic Studios.

Ultimately these objects work in a hierarchy that is responsible for retrieving data from persistent storage, modifying the data, ensuring integrity, and finally presenting it to the application in a requested context. Core Data also gives the fascinating capability of working very aggressively to cache objects whenever it can. This works really well when you're working with a large data set. Regarding the persistent storage, Core Data is capable of using a SQLite back end, memory, or binary. For our purposes, we'll use a SQLite back end, which is evident in the `persistentStorageCoordinator` accessor method.

The `persistentStorageCoordinator` speaks to a persistent object store underneath the hood. Don't worry about the persistent object store, though, because you won't be accessing it directly here. The persistent storage coordinator will take care of that for you. All you have to know is that the persistent storage coordinator communicates with the managed object model to help you understand what the data looks like that it is asked to delegate to the persistent object store (POS).

The MOM contains information to your model classes, which I'll cover shortly. The object you will interact with mostly is the MOC. You ask the MOC to grab your model objects for you. You manipulate your objects through it including insertions and deletions. You also eventually ask the managed object context to save your data. Once that happens, the managed object context shoots all the information it needs to down the stack that composes Core Data. Reference the previous code to see how that stack is built and communicates with each other.

I can't express how condensed the overview of Core Data is here. It doesn't do this framework justice. There are books written on the topic alone, so make sure you do your Googling if you want to learn more.

Adding Three20

While I'm here, you should also go ahead and get your project set up with the Three20 library in addition to the SystemConfiguration framework. Joe Hewitt has posted awesome instructions on how to add Three20 to a project. You will need to have Git available (downloadable from <http://code.google.com/p/>

git-osx-installer/) if you want to clone the Three20 repo. You're also more than welcome to just follow along in the code I've already provided. Here's what you need to do in order to install it in your own projects:

1. Clone the three20 Git repository: `git clone git://github.com/joehewitt/three20.git`. Make sure you store the repository in a permanent place because Xcode will need to reference the files every time you compile your project.
2. Locate the Three20.xcodeproj file under three20/src. Drag Three20.xcodeproj, and drop it onto the root of your Xcode project's Groups & Files pane. A dialog box will appear. Make sure "Copy items" is deselected and that Reference Type is set to Relative to Project. Then click Add.
3. Link the Three20 static library to your project. Click the Three20.xcodeproj item that has just been added to the sidebar. In the details pane, you will see a single item: `libThree20.a`. Select the check box on the far right of that item (Figure 7-6).

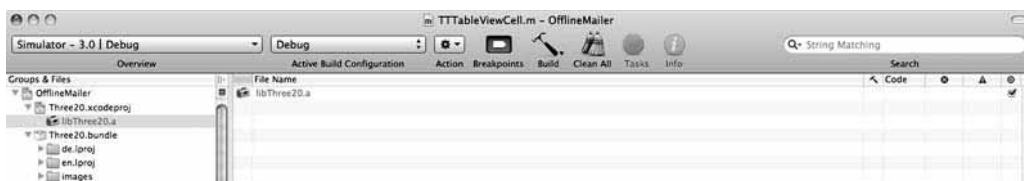


Figure 7-6. *libThree20.a* is added as a target to your application. Take note of the check box that's enabled underneath the target symbol in the top right.

4. Add Three20 as a dependency of your project, so Xcode compiles it whenever you compile your project. Expand the Targets section of the sidebar, and double-click your application's target. On the General tab, you will see a Direct Dependencies section. Click the + button, select Three20, and click Add Target (see Figure 7-7).

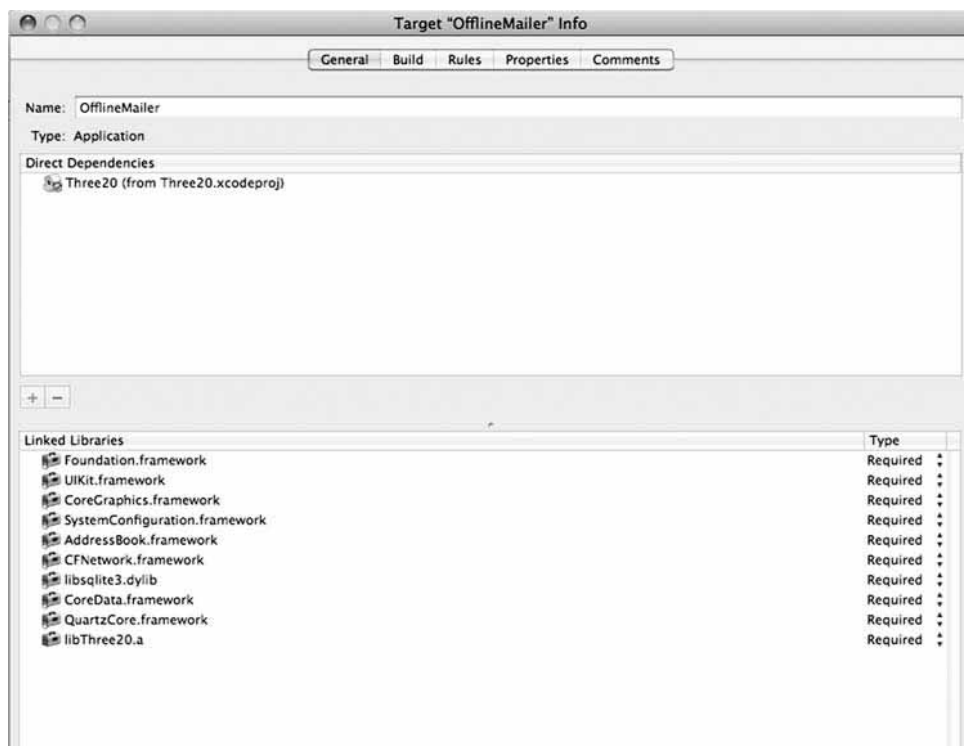


Figure 7-7. *Three20 is a direct dependency to the application.*

5. Add the bundle of images and strings to your app. Locate `Three20.bundle` under `three20/src`, and drag and drop it into your project. A dialog box will appear. Make sure `Create Folder References` is selected, `"Copy items"` is deselected, and `Reference Type` is set to `Relative to Project`. Then click `Add`.
6. Add the `CoreAnimation` framework to your project. Right-click the `Frameworks` group in your project (or equivalent), and select `Add ► Existing Frameworks`. Then locate `QuartzCore.framework`, and add it to the project.
7. Finally, tell your project where to find the `Three20` headers. Open your project settings, and go to the `Build` tab. Look for `Header Search Paths`, and double-click it. Add the relative path from your project's directory to the `three20/src` directory.
8. While you are in `Project Settings`, go to `Other Linker Flags` under the `Linker` section, and add `-ObjC` and `-all_load` to the list of flags.
9. You're ready to go. Just `#import "Three20/Three20.h"` anywhere you want to use `Three20` classes in your project.

I have made the most current source of Three20 available with the sample code provided for this chapter. However, by the time you read this chapter, it's possible that new features and bug fixes have been submitted. I highly encourage you to visit the site's home page on GitHub and check it out.

Journeying Through the User Interface

Enough with installing libraries. Let's get back into code! If you recall in my musings before, once settings are properly added to the Settings of the application, you will bring up the initial screen of your application. `AccountViewController.h` and `AccountViewController.m` are responsible for this, as shown in Figure 7-8.



Figure 7-8. *The target device has at least one form of network connectivity.*



Figure 7-9. *The application does not have network connectivity and is thus considered offline.*

The light bulb is glowing yellow because this screenshot was taken while my simulator had access to the Internet. Should you shut off the Internet while the app is running, it would look like Figure 7-9.

The code for making that happen is trivial. Since you want to update the user interface for the `UIViewController` whenever the network status is changed, you create a superclass called `OfflineViewController`. The source for `OfflineViewController` is available in its respective classes, `OfflineViewController.h` and

OfflineViewController.m. The most important methods to take heed of are the following:

```
// Implement viewDidLoad to do additional setup after loading the view.
- (void)viewDidLoad {
    [super viewDidLoad];

    [[NSNotificationCenter defaultCenter]
     addObserver:self
     selector:@selector(reachabilityChanged:)
     name:@"kNetworkReachabilityChangedNotification"
     object:nil];

    [[Reachability sharedReachability] setHostName:[DataManager sharedManager]
    hostName];
    [self updateNetworkStatus];
}

// for subclasses to implement
- (void)reachabilityChanged:(NSNotification *)note
{}

// for subclasses to implement
- (void)updateNetworkStatus
{}

```

Every UIViewController that subclasses OfflineViewController should implement these methods. They will be invoked every time the network status changes. The respective code in the AccountViewController class is implemented like the following:

```
- (void)updateNetworkStatus
{
    NSLog(@"Network Status Has Been Changed");
    currentNetworkStatus = [[Reachability sharedReachability] remoteHostStatus];
    [lightBulbView setImage:[self lightBulb]];

    if (currentNetworkStatus == ReachableViaWiFiNetwork || currentNetworkStatus ==
    ReachableViaCarrierDataNetwork) {
        // Check the offline queue since we're on the network.
        NSInteger queueCount = [[DataManager sharedDataManager]
        numberOfMessagesInQueue];
        if (queueCount) {
            [[DataManager sharedDataManager] flushQueue];
        }
    }
}

- (UIImage *)lightBulb
{
    return (currentNetworkStatus == NotReachable)
        ? [UIImage imageNamed:@"light_bulb_off.png"]
        : [UIImage imageNamed:@"light_bulb_on.png"];
}

```

There is a reference to a class in the previous code that I have not yet discussed by the name of DataManager. For now, all you need to know is that DataManager is the class that's responsible for proxying all the communication to the database and network.

Whenever the network state changes, `NSNotificationCenter` fires off a message to everyone observing the `kNetworkReachabilityChangedNotification` notification. Remember that both the application delegate and any class that subclasses `OfflineViewController` will be an observer. Here all you do is simply swap images when the network's state changes. I set up an ivar with a coinciding property named `lightBulbView`. `lightBulbView` is an instance of `UIImageView` that you can access anywhere throughout the instance of `AccountViewController`. When the network state changes, you invoke the following:

```
[lightBulbView setImage:[self lightBulb]];
```

Again, it's far from rocket science. And the purpose of this chapter is not to teach UI design. In fact, I've intentionally left the UI here as cookie-cutter as possible so that you can concentrate on code. That, and the real secret is I make a better ballerina than a designer. I have no business touching software such as Photoshop or Illustrator; however, there are plenty of hungry graphic designers out there if you're in a similar position. I also use sites such as <http://www.istockphoto.com> and <http://www.smashingmagazine.com> extensively in order to pick up visually stimulating icons and graphics.

Just to quickly go over what's happening on the screen here, you have a few key elements. On the `UIToolbar` on the bottom left, you have a compose message button in addition to the aforementioned network connectivity light bulb icon. The `UITableView` consists of two key players: `Offline Queue` and `Sent Mail`. These entities will also display a visual indicator for the number of items available in either the offline cache or the number of messages sent over the network. Since you haven't written the first message yet, you do not display a count. You will find this very similar to Apple's `Mail.app` behavior. Finally, the title simply consists of the username I use for my SMTP server followed by the SMTP server's hostname.

Managing Top-Level Data with DataManager

When building this demo, I needed a convenient class that would handle the management of delivering messages. The class would also contain utility methods that would assist view controllers in knowing how many messages are in the offline queue and how many have been sent over the wire.

Before diving into the `DataManager` class, I should make it a point that some folks are strongly opinionated about solving the common problem of putting top-level data into a singleton the way I'm doing so here. The pros and cons for doing so are outside the context of this chapter. For this chapter's purposes, however, this suits the architecture well. I will leave you with another great article by Matt Gallagher that further explores this topic: <http://cocoawithlove.com/2008/11/singletons-appdelegates-and-top-level.html>. Now, let's dip into both `DataManager.h` and `DataManager.m`.

The first requirement of the `DataManager` class was that it should be instantiated only once. That's where the singleton design pattern comes in and saves the day. (See the *Cocoa Fundamentals Guide* at <http://developer.apple.com> for more information about

singletons.) Finally, the class could also be accessed by multiple threads, so you need to make sure you set up locks where applicable. Let's get to some code to see the nitty-gritty details of how the DataManager class is created:

```
static DataManager *dataMgr = nil;
```

First you declare a static instance of the DataManager class named dataMgr. If you recall your basic C programming, you declare this static variable so that the dataMgr variable is not available to other source files. It is locally scoped to the DataManager class.

```
// Initialize the singleton instance if needed and return
+(DataManager *)sharedDataManager
{
    @synchronized(self) { // thread safe init
        if (dataMgr == nil) {
            [[self alloc] init];
        }
    }
    return dataMgr;
}

+ (id)allocWithZone:(NSZone *)zone
{
    @synchronized(self) {
        if (dataMgr == nil) {
            dataMgr = [super allocWithZone:zone];
            return dataMgr; // assignment and return on first allocation
        }
    }
    return nil; //on subsequent allocation attempts return nil
}
```

The sharedDataManager method is how you access the singleton object. Any class that wants an instance of DataManager will simply call [DataManager sharedDataManager]. They will not need to worry about multiple copies or data corruption when accessing the DataManager from multiple threads. The @synchronized() directive sets a lock on a section of code so that a single thread can execute the rest of the instructions in the block without worrying about other threads accessing the data. Any secondary threads trying to access the method will be blocked until the thread accessing the block of code exits the last statement in the @synchronized() block. The self argument passed to the @synchronized() directive is the actual mutex that the lock is set on. If you're familiar with POSIX threads, better known as *pthread*s, you should be familiar with mutexes. You will find the rest of the auxiliary code for a singleton class available in the source code.

```
- (void)loadDefaultSettings
{
    NSUserDefaults *defaults = [NSUserDefaults standardUserDefaults];

    self.hostName = [defaults objectForKey:@"hostName"];
    self.smtpPort = [defaults objectForKey:@"smtpPort"];
    self.smtpPassword = [defaults objectForKey:@"smtpPassword"];
    self.smtpUserName = [defaults objectForKey:@"smtpUserName"];

    if (self.smtpPort == nil) {
        self.smtpPort = [NSNumber numberWithInt:DEFAULT_SMTP_PORT];
    }
}
```

```
    }  
}
```

The `DataManager` is also responsible for loading the default settings from `NSUserDefaults`. This is only ever a read-only operation for our purposes. If you pay close attention, you might notice something similar between the `NSUserDefaults` class and the `DataManager`. That's right, the `NSUserDefaults` class is also a singleton class. Notice the choice of selector name for accessing an instance of the class?

Now that you have established the bare bones of the `DataManager`, let's take a subtle fork in the road before digging into the meat of the class. For the rest of the code to make sense, you need to take a look at Three20 classes that communicate with the `DataManager`.

Diving into Three20 and TTMessageController

Let's get back to the `AccountViewController.m` file. This is the initial screen where user interaction happens. When building this demo, I wanted to ensure the experience for you would be smooth and simple. That's something that even the most advanced power users can appreciate. In doing so, I needed a clean interface that would allow me to compose my e-mail messages. I knew that at the time of this writing, the SDK did not allow me to access the Mail application without also leaving my application. This is when I ran into Joe Hewitt's exemplary open source library named Three20.

Three20 is a set of Objective-C classes that you can reuse in your applications to either enhance its UI, add a photo picker that looks like Apple's, style your labels like you would with CSS, and do many other outstanding things. It's simply brilliant how trivial it was for me to build a compose e-mail view with the powerful functionality I required for this demo, including the capability to add multiple recipients, search my Address Book, add a subject, and be able to write text that I can easily scroll through. To do that, you'll use the Three20 class `TTMessageController`. Here is how you instantiate it when the `composeMessage: selector` is invoked:

```
- (void)composeMessage:(id)sender  
{  
    id recipient = [[[TTTableField alloc] initWithText:nil url:TT_NULL_URL]  
autorelease];  
    TTMessageController* controller = [[TTMessageController alloc]  
initWithRecipients:[NSArray arrayWithObject:recipient]];  
    self.messageController = controller;  
    messageController.delegate = self;  
    messageController.dataSource = dataSource;  
  
    [controller release];  
  
    [self presentViewController:messageController animated:YES];  
}
```

Initially, you do not want any recipients to populate your recipient row. So, you use an instance of `TTTableField` as a “dummy” placeholder. The `TTMessageController` needs to be initialized with the `initWithRecipients: methods`, and here is where you provide the

recipient object. You assign the instance of `AccountViewController` as a delegate, and you assign a `dataSource` of `TTAddressBookDataSource`. `TTAddressBookDataSource` is not part of the Three20 library. It is a class I've put together so that you can take advantage of pulling contacts from your Address Book. That adds a more realistic touch to the application. `TTAddressBookDataSource` resides in `Classes > Data Management > TTAddressBookDataSource.h/TTAddressBookDataSource.m`. One of my other favorite tricks is to use Cmd+D. There you can type in any file or symbol name you're searching for.

The most interesting method in `TTAddressBookDataSource` looks like the following:

```
+ (TTAddressBookDataSource *)abDataSourceForSearch:(BOOL)forSearch
{
    ABAddressBookRef addressBook = ABAddressBookCreate();
    NSArray *peopleArray = (NSArray *)ABAddressBookCopyArrayOfAllPeople(addressBook);
    NSMutableArray *allContacts = [NSMutableArray array];

    for (id person in peopleArray) {
        if ([([NSString *)ABRecordCopyValue(person, kABPersonOrganizationProperty)
autorelease]) continue;
        NSMutableString *firstName = [([NSString *)ABRecordCopyValue(person,
kABPersonFirstNameProperty) autorelease];
        NSMutableString *lastName = [([NSString *)ABRecordCopyValue(person,
kABPersonLastNameProperty) autorelease];
        ABMutableMultiValueRef multiValueEmail = ABRecordCopyValue(person,
kABPersonEmailProperty);

        NSString *email = nil;
        if (ABMultiValueGetCount(multiValueEmail) > 0) {
            email = [([NSString *)ABMultiValueCopyValueAtIndex(multiValueEmail, 0)
autorelease];
        } else {
            continue;
        }

        Contact *aContact = [[[Contact alloc] initWithFirstName:firstName
lastName:lastName email:email] autorelease];
        [allContacts addObject:aContact];
    }

    TTAddressBookDataSource *dataSource = [[[TTAddressBookDataSource alloc]
initWithNames:allContacts] autorelease];

    if (!forSearch) {
        [dataSource rebuildItems];
    }

    CFRelease(addressBook);

    return dataSource;
}
```

What this code does is creates a reference to the Address Book and its respective database. You then return an array of all the people in your Address Book using the following code:

```
NSArray *peopleArray = (NSArray*)ABAddressBookCopyArrayOfAllPeople(addressBook);
```

NOTE: Organizations you add to your Address Book will not be returned by this invocation! That might be clear by the verbose name of the function; however, it was something that had me stumped when I first started writing code with the AddressBook framework. The rest of the code should be rather self-explanatory. There is excellent documentation on programming with the AddressBook framework in the ADC. Feel free to use this as a reference in order to incorporate the AddressBook framework in your application.

Composing and Sending Messages

So up to this point, you have seen the building blocks to put this application together. However, I haven't just yet touched the novel parts of dealing with offline operations. This is where we start chewing bubble gum and taking names.

When you're composing your message, you'll notice that the controller has a built-in search that's very similar to Apple's. You get that great-looking dim overlay on top of your Address Book that sits between the keyboard and the search bar, as shown in Figure 7-10.



Figure 7-10. A rudimentary search user interface that resembles the search used by Apple in the Contacts application

You'll also see an attractive user interface when you enter a contact name in the To field that has a match in the Address Book, as Figure 7-11 reflects.



Figure 7-11. The result of all of the contacts that contain the substring “s” as it is entered into the search field above

Let’s give this a spin. Assuming you have a properly configured SMTP server set up and input into Settings, go ahead and add some dummy contacts into your simulator’s Contacts if you’re running in a simulator environment. If you’re on your actual device, you should be all set there. After composing a message, your screen should look like Figure 7-12.



Figure 7-12. The user interface after one message has been successfully sent over the network

Neat, huh? There's the visual indicator, showing a very Apple Mail-esque blue-pill tablet with the amount of messages sent. Here's what gets fired off after you hit Send:

```
- (void)composeController:(TTMessageController*)controller
didSendFields:(NSArray*)fields
{
    NSMutableArray *contacts = [[NSMutableArray alloc] initWithCapacity:0];
    TTMessageRecipientField *toField = [fields objectAtIndex:0];

    for (id recipient in toField.recipients) {
        Contact *aContact = [dataSource contactWithName:recipient];
        [contacts addObject:aContact];
    }
    [[DataManager sharedDataManager] sendEmailWithFields:fields forContacts:contacts];
    [contacts release];
}
```

Here you use Three20's class `TTMessageRecipientField` in order to establish the list of destination users you'll be messaging. You then go through each individual recipient and instantiate a `Contact` object.

NOTE: The `Contact` class is a trivial resource that I wrote as a wrapper of key/value pairs. The source code is available in `Classes/Model/Contact.h/m`. On the other hand, the `Message` class is a special class in that it is coupled with Core Data. I'll discuss that shortly.

When you're ready, you use `DataManager` as a pass-through proxy to figure out what to do with the message:

```
- (void)sendEmailWithFields:(NSArray *)fields forContacts:(NSArray *)contacts
{
    // message subject
    TTMessageSubjectField *subjField = [fields objectAtIndex:1];

    // email body
    TTMessageTextField *bodyField = [fields objectAtIndex:2];

    // recipients
    NSMutableString *recipients = [[NSMutableString alloc] init];
    NSInteger cnt = [contacts count];
    for (int i=0; i < cnt; ++i) {
        NSString *anEmail = nil;
        if (i == cnt - 1)
            anEmail = [NSString stringWithFormat:@"%@", (Contact *)[contacts
objectAtIndex:i] email]];
        else
            anEmail = [NSString stringWithFormat:@"%@", (Contact *)[contacts
objectAtIndex:i] email]];

        [recipients appendString:anEmail];
    }

    NSDictionary *data = [NSDictionary dictionaryWithObjectsAndKeys:recipients,
@"recipients", subjField.text, @"subject", bodyField.text, @"body", nil];
```

```

    if (self.appDelegate.hasNetworkConnection) {
        NSInvocationOperation *onlineEmailOperation = [[NSInvocationOperation alloc]
initWithTarget:self selector:@selector(emailInvocationOperation:) object:data];
        [networkOperationQueue addOperation:onlineEmailOperation];
        [onlineEmailOperation release];
    } else {

        //
        // get the next id for messageid, its unique identifier.
        //
        NSString *newId = [selfgetNewMessageID];

        //
        // Create a new Message object and add it to the Managed Object Context.
        //
        Message *message = (Message
*)[NSEntityDescriptioninsertNewObjectForEntityForName:@"Message" inManagedObjectContext:m
anagedObjectContext];

        // configure the message object using KVC, a common pattern
        // when using Core Data.
        [message setValue:recipients forKey:@"to"];
        [message setValue:bodyField.text forKey:@"body"];
        [message setValue:subjField.text forKey:@"subject"];
        [message setValue:newId forKey:@"messageID"];
        [message setValue:[NSNumber numberWithInt:0] forKey:@"dateSent"];
        [message setValue:[NSNumber numberWithInt:NO] forKey:@"status"];

        NSLog(@"newId in queue: %@", newId);

        NSError *error = nil;
        if (![managedObjectContextsave:&error]) {
            // handle the error
        } else {
            [[NSNotificationCenter defaultCenter]
postNotificationName:kMessageQueuedSuccessfullyobject:nil];
        }
    }
}

```

The interesting code here is where you refer to your application delegate (appDelegate) and ask it what state of your network is in:

```

If (self.appDelegate.hasNetworkConnection) {
    // do network stuff.
} else {
    // cache stuff.
}

```

Look familiar? That's because it is. And again, it's not rocket science—it's trivial at best. The idea here is to demonstrate how some of the most advanced applications use simple implementation interfaces such as this to operate.

Now in this particular example you were online, so what you do here is instantiate an NSInvocationOperation object with the following:

```
NSInvocationOperation *onlineEmailOperation = [[NSInvocationOperation alloc]
initWithTarget:self selector:@selector(emailInvocationOperation:) object:data];
```

We then add it to our networkOperationQueue:

```
[networkOperationQueue addOperation:onlineEmailOperation];
```

NSInvocationOperation is one of my favorite classes that is available in the Foundation framework. It simply takes one of the selectors and encapsulates the instructions within to run in a threaded environment. And although it's not a true concurrent operation, you have the full benefit of freeing your GUI thread from getting locked because of the work that's going on in the invocation. This gives the user greater feedback in the application and allows them to either continue writing more messages or view their queue.

Our emailInvocationOperation: method looks like the following:

```
- (void)emailInvocationOperation:(id)data
{
    NSAutoreleasePool *aPool = [[NSAutoreleasePool alloc] init];
    NSString *body = [data objectForKey:@"body"];
    NSString *recipients = [data objectForKey:@"recipients"];
    NSString *subject = [data objectForKey:@"subject"];
    NSString *messageID = [data objectForKey:@"messageID"];

    if (!messageID) {
        // No message created yet? Let's create it.
        NSNumber *currentTime = [self currentTimeStamp];
        messageID = [self getNewMessageID];

        Message *message = (Message *)[NSEntityDescription
insertNewObjectForEntityForName:@"Message" inManagedObjectContext:managedObjectContext];

        [message setValue:recipients forKey:@"to"];
        [message setValue:body forKey:@"body"];
        [message setValue:subject forKey:@"subject"];
        [message setValue:messageID forKey:@"messageID"];
        [message setValue:currentTime forKey:@"dateSent"];

        NSLog(@"messageID in emailInvocationOperation: %@", messageID);

        NSError *error = nil;
        if (![managedObjectContext save:&error]) {
            // handle the error
        }
    }

    SKPSMTPMessage *smtpMsg = [[SKPSMTPMessage alloc] init];
    smtpMsg.fromEmail = @"you@example.com ";
    smtpMsg.toEmail = recipients;
    smtpMsg.relayHost = self.hostName;
    smtpMsg.requiresAuth = YES;
    smtpMsg.login = @"self.login ";
    smtpMsg.pass = @"self.password";
    smtpMsg.subject = subject;
    smtpMsg.wantsSecure = YES;
```

```

//
//Upon a successful callback of messageSent:
// we want to update this particular row from the database.
//
smtpMsg.messageID = messageID;

smtpMsg.delegate = self;
NSMutableDictionary *plainText =
    [NSMutableDictionary
dictionaryWithObjectsAndKeys:@"text/plain",kSKSMTPPartContentTypeKey,
body,kSKSMTPPartMessageKey,@"8bit",kSKSMTPPartContentTransferEncodingKey,nil];

smtpMsg.parts = [NSArray arrayWithObjects:plainText,nil];
[smtpMsg send];

[smtpMsg release];
[aPool drain];
}

```

What's happening here is the `emailInvocationOperation:selector` is being invoked on a secondary thread. As such, I like to create an additional instance of `NSAutoreleasePool` since I'll be allocating some objects in this thread and I'd like the autorelease pool to get drained when execution finishes here.

When you invoked `emailInvocationOperation`, you passed along a dictionary of data about the message.

This metadata gives you the opportunity to do your record keeping to know what message you're currently dealing with. Speaking of the `Message` class, you're probably asking yourself where this class came from and why I haven't spoken about it yet. If you're asking that, you're asking the right questions. Let's jump into your first glance at an implementation of `NSManagedObject`.

Creating the Core Data Model

When you created the project as a Core Data-enabled project, Xcode's template magic should have created a file called `OfflineMailer.xcdatamodel`. Double-click it, and you should get something that looks like Figure 7-13.

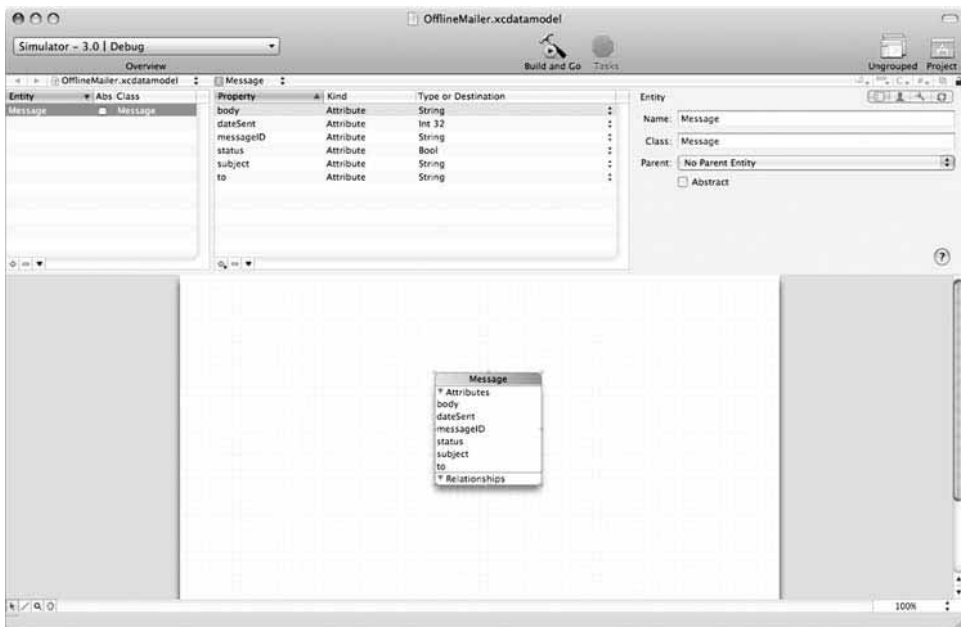


Figure 7-13. The simple Core Data model

If there were a few more entities (entities in Core Data are better known as *models*), I would have something that resembles an ERD diagram. In this figure, I have created a Message entity by clicking the + button in the upper-left corner of Figure 7-13. The Message entity has several attributes (better known as *properties*). Basically, try to think of entities as eventually being turned into tables in a SQLite implementation with attributes acting as columns.

As depicted in Figure 7-13, you can see I added several attributes to the Message entity. These properties include a body, dateSent, messageID, status, subject, and a “to” field. Note that each one has a correlated type. Also, notice I disabled the Optional field in all of the attributes, as depicted in Figure 7-14.

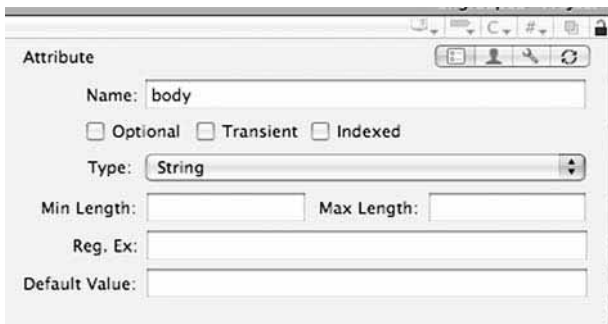


Figure 7-14. The Attribute pane of the Core Data model indicating that the attribute “body” is not optional.

There's a great deal of information I can get into about what's going on here, but you must remain focused on the task at hand. I haven't even scratched the surface of Core Data by not mentioning relationships and their relevance between entities. It makes me want to jump into another several chapters dedicated to Core Data, but that's a luxury I cannot afford.

What's most important is for me to show you how to create your `Model` class after creating this entity. What you should do next is click the `Message` entity, then press `Cmd+N`, and select `Managed Object Class` from the menu, as depicted in Figure 7-15.



Figure 7-15. *Selecting Managed Object Class as an available template*

You'll notice a new `Message` class has been created for you. All this work has been done with very little effort. There shouldn't be anything new to you in the generated class except for `Message`'s base class (`NSManagedObject`) and perhaps the `@dynamic` directive. All the `@dynamic` directive does is promise the compiler (or Core Data in this case) that you'll supply the implementation for the properties at compile time. `NSManagedObject` is simply a generic class that Core Data molds as a model for you.

Now you'll dive back into `emailInvocationOperation`. You're about to inhale the fresh aura that Core Data emits. In the next several lines, you're going to avoid the hassle of going back and forth between objects and SQL. That is a process that can grow tedious and error-prone. This is handled much more elegantly with the advent of Core Data.

First, you're checking to see whether you have a `messageID`. The `messageID` helps you identify a unique message in a pool of messages. If you have experience with databases, this is a fairly similar approach to using an autoincrement field. The code for `getNewMessageID` looks like the following:

```

- (NSString *)getNewMessageID
{
    NSString *newId = @"1";

    NSEntityDescription *entity =
        [NSEntityDescription entityForName:@"Message"
inManagedObjectContext:managedObjectContext];
    NSFetchRequest *fetchRequest = [[NSFetchRequest alloc] init];
    [fetchRequest setEntity:entity];
    NSString *predicateString = @"messageID = max(messageID)";
    NSPredicate *predicate = [NSPredicate predicateWithFormat:predicateString];
    [fetchRequest setPredicate:predicate];

    NSError *error = nil;
    NSArray *allMessages = [managedObjectContext executeFetchRequest:fetchRequest
error:&error];
    if ([allMessages count] > 0) {
        Message *message = [allMessages objectAtIndex:0];
        NSString *messageID = [message valueForKey:@"messageID"];
        NSNumberFormatter *numberFormatter =
            [[[NSNumberFormatter alloc] init] autorelease];
        NSNumber *num = [numberFormatter numberFromString:messageID];
        newId = [NSString stringWithFormat:@"%d", [num intValue] +1];
    }

    [fetchRequest release];

    return newId;
}

```

This method contains some new classes, so let's go over them briefly. First you use `NSEntityDescription` to describe what objects you're particularly interested in working with. In this case, it's the only Core Data class you have, so you set the entity name `Message`. The `NSFetchRequest` is where you retrieve our data from storage. It's also where you can set a predicate using `NSPredicate`.

If you're a database person, it might help you to think of the `NSPredicate` class as a wrapper around the SQL keyword `WHERE`. It is used to look for a specific domain of objects that are available from a much larger pool of objects. I highly recommend looking up the documentation for this class and getting better acquainted with it.

It's fairly simple to use; as you can see in the `predicateString` variable, you set a format of `messageID = max(messageID)`. What this does is retrieve one instance of message (if you have one) with the highest `messageID` value. You simply take that value, increment it by one, and return it. This will give the caller the capability to set the value for the next Message object.

After you fetch the new ID, you use `NSEntityDescription` to inject a new Message object into your managed object context. The entity you're creating is named in `insertNewObjectForEntityForName:`. You then use a common pattern of Key Value Coding (KVC) to properly initialize your message object. Finally, you save the state of the context. Note that I'm not doing any real error handling here besides leaving a template for where you would normally do it. If this were a production application, you most certainly would want to fill in this code here in the event of an error.

Hacking SKPSMTPMessage to Support Threaded Message Sending

You've gotten this far, but you have yet to see the workhorse of what's responsible for handling the gamut of paramount details behind the scenes. The SKPSMTPMessage is an open source library that is a simple but well-working wrapper that sends messages via SMTP on the iPhone and Mac.

CAUTION: Applications that tend to replicate Apple's functionality such as Mail.app stand the risk of being rejected from the App Store. We use this simply as a demo for the book.

The instantiation of the SKPSMTPMessage class is straightforward. You should be able to understand its usage by reading the source in `DataManager.m`. I made some minor modifications to SKPSMTPMessage in order to make it work with this app. First, I do the following:

```
smtpMsg.messageID = messageID;
```

I added the following instance variable to SKPSMTPMessage.h:

```
NSString *messageID;
```

The reason for this is simple—after the message delegate receives a `messageSent:` or `messageFailed:` callback that is defined in the SKPSMTPMessageDelegate, I need to know which message it is. You could have an infinite amount of messages queued (although your ISP might end up knocking on your door if you should ever try to reach this egregious quantity...), and whenever one fails or succeeds, you need to update your database.

Setting Up the NSRunLoop on SKPSMTPMessage

There's one more change you have to make to SKPSMTPMessage that isn't as straightforward to understand without a brief explanation, so I'll talk about that here. The SKPSMTPMessage class includes a multitude of sources such as `NSOutputStream`, `NSInputStream`, and `NSTimer` that work just fine in a main thread but require slightly more configuration in a secondary thread. The reason for that is most of the time there is an object that works behind the scenes that manages these input sources for you without you even knowing. It all happens automatically if you create your application with one of Xcode's sample templates and an instance of `UIApplication` is created for you. That magic object is known as `NSRunLoop`.

When you manage secondary threads with input sources and instances of `NSTimer`, you need to create your own `NSRunLoop`, and you need to run it periodically to check for events. `NSTimers` that get fired off without a properly configured `NSRunLoop` will never work. So, in order to make this work, I simply created an `NSRunLoop` that runs in a loop and processes the input sources until the thread exits:

```
while (self.runningLoop) {
    [r1 runUntilDate:[NSDate dateWithTimeIntervalSinceNow:1]];
}
```

That should satisfy SKPSMTPMessage in a secondary thread. Let's do something slightly more interesting now. Let's fire up the application, let's turn off the WiFi/Internet, and let's send a message with no connectivity. In fact, write two or three messages and see what happens. Your screen should look something like Figure 7-16.



Figure 7-16. An offline state of the application, with new messages incrementing the Offline Queue count

NOTE: The blue-like pill is a nifty user interface element you see available in many applications such as Apple's Mail.app and many third-party apps in the App Store. To see how it's built, take a look at `Classes/View/Elements/BlueBadge.h/m`. The source is freely available by developer Leon Ho on <http://github.com/leonho/iphone-libs/tree/master>.

Don't you just love it when you're interacting with an application that doesn't crash or give you an obtrusive UIAlertView complaining that there isn't any Internet available? It's a beautiful and simple concept. Let's see what's going on to make this happen:

```
if (self.appDelegate.hasNetworkConnection) {
    NSInvocationOperation *onlineEmailOperation = [[NSInvocationOperation alloc]
initWithTarget:self selector:@selector(emailInvocationOperation:) object:data];
    [networkOperationQueue addOperation:onlineEmailOperation];
    [onlineEmailOperation release];
} else {
    //
```

```

// get the next id for messageid, its unique identifier.
//
NSString *newId = [self getNewMessageID];

//
// Create a new Message object and add it to the Managed Object Context.
//
Message *message = (Message *)[NSEntityDescription
insertNewObjectForEntityForName:@"Message" inManagedObjectContext:managedObjectContext];

// configure the message object using KVC, a common pattern
// when using Core Data.
[message setValue:recipients forKey:@"to"];
[message setValue:bodyField.text forKey:@"body"];
[message setValue:subjField.text forKey:@"subject"];
[message setValue:newId forKey:@"messageID"];
[message setValue:[NSNumber numberWithInt:0] forKey:@"dateSent"];
[message setValue:[NSNumber numberWithBool:NO] forKey:@"status"];

NSLog(@"newId in queue: %@", newId);

NSError *error = nil;
if (![managedObjectContext save:&error]) {
    // handle the error
} else {
    [[NSNotificationCenter defaultCenter]
postNotificationName:kMessageQueuedSuccessfully object:nil];
}
}

```

Some of this code should look familiar. That's because it is—I already went over it. The only difference here is you've failed a network connection check for which the SystemConfiguration framework has so kindly been programmed to report.

The most important thing to note here is to set the status of [NSNumber numberWithInt:NO] in the Message object. If you recall, status is the attribute you set as a BOOL in the entity. This status will help you determine whether a message has already been sent to its recipients. With a BOOL of NO, the message is still in the queue.

Switching the Bits Back to Online Mode

So, you have a bunch of messages queued now. That's fine and dandy, but it does you no use unless your recipients ultimately receive the message you've crafted to them during your flight to the sunny beaches of Antigua where you're going to build your ultimate iPhone app from a beach-based villa, right? OK, OK, we can dream.

So, let's get back into reality here and reenable the Internet connection, or whatever means you're using to connect. After reconnecting, you should see something like Figure 7-17.



Figure 7-17. *Returning online after spending some time in an offline state*

You guessed it—you have a notification that is listening and waiting for a change in the network state. Upon reconnecting, you see whether there are any messages in the offline queue and take the appropriate action:

```
- (void)updateNetworkStatus
{
    currentNetworkStatus = [[Reachability sharedReachability] remoteHostStatus];
    [lightBulbView setImage:[self lightBulb]];

    if (currentNetworkStatus == ReachableViaWiFiNetwork || currentNetworkStatus ==
ReachableViaCarrierDataNetwork) {
        // Check the offline queue since we're on the network.
        NSInteger queueCount = [[DataManager sharedDataManager]
numberOfMessagesInQueue];
        if (queueCount) {
            [[DataManager sharedDataManager] flushQueue];
        }
    }
}
```

and finally flushQueue:

```
- (void)flushQueue
{
    NSLog(@"Flushing queue");

    if (self.appDelegate.hasNetworkConnection) {
        NSArray *messages = [self getResultSetFromQueue];
        for (Message *message in messages) {
            NSMutableDictionary *data = [NSMutableDictionary new];
            [data setValue:[message valueForKey:@"body"] forKey:@"body"];
            [data setValue:[message valueForKey:@"to"] forKey:@"recipients"];
        }
    }
}
```

```

        [data setValue:[message valueForKey:@"messageID"] forKey:@"messageID"];
        [data setValue:[message valueForKey:@"subject"] forKey:@"subject"];

        NSInvocationOperation *onlineEmailOperation = [[NSInvocationOperation alloc]
initWithTarget:self selector:@selector(emailInvocationOperation:) object:data];
        [networkOperationQueue addOperation:onlineEmailOperation];

        [onlineEmailOperation release];
        [data release];
    }
}
}

```

Here you query for all the objects that have not yet been sent (remember the status attribute I spoke about earlier?) using the `-getResultSetFromQueue` method:

```

- (NSArray *)getResultSetFromQueue
{
    NSFetchRequest *request = [[NSFetchRequest alloc] init];
    NSEntityDescription *entity = [NSEntityDescription entityForName:@"Message"
inManagedObjectContext:managedObjectContext];
    [request setEntity:entity];

    NSPredicate *pred = [NSPredicate predicateWithFormat:@"status == NO"];
    [request setPredicate:pred];

    NSError *error = nil;
    NSArray *fetchResults = [managedObjectContext executeFetchRequest:request
error:&error];
    if (fetchResults == nil) {
        // handle error here.
    }

    [request release];

    return fetchResults;
}

```

`-getResultSetFromQueue` should start looking awfully familiar to you if it hasn't already. What you're doing in here is very similar to what you saw before in the `getNewMessageID` method. This time, you're simply setting the predicate to look like the following:

```
NSPredicate *pred = [NSPredicate predicateWithFormat:@"status == NO"];
```

For each individual message, you wrap the data into an `NSInvocationOperation` selector and add it to the `networkOperationQueue`.

NOTE: Some MTAs might raise a suspicious flag if you queue a high number of messages and send them all simultaneously without any throttling. I've left a comment here where you could set a throttle in order to avoid hearing from the BOFH that's administrating the recipient's or sender's mail server.

Finally, once a message is sent, the `messageSent:` delegate is invoked on the main thread:

```

- (void)messageSent:(SKPSMTPMessage *)smtpMessage
{
    NSLog(@"delegate - message sent for message id: %@", [smtpMessage messageID]);

    NSString *messageID = [smtpMessage messageID];

    // retrieve Message based on ID
    Message *message = [self getMessageWithID:messageID];

    // Update status to sent and current timestamp.
    NSNumber *currentTime = [self currentTimeStamp];

    [message setValue:currentTime forKey:@"dateSent"];
    [message setValue:[NSNumber numberWithInt:YES] forKey:@"status"];

    NSError *error = nil;

    if (![managedObjectContext save:&error]) {
        // handle error;
    }

    // post a notification to alert the client that the message has been sent.
    [[NSNotificationCenter defaultCenter] postNotificationName:kMessageSentSuccessfully
    object:nil];
}

```

Here you simply update the timestamp of the message you're sending once it has been successfully sent. You also make sure to set the status to a Boolean YES so you do not resend it in the future. After that is done, you remember to save the `managedObjectContext`. Finally, you post a notification so that the `AccountViewController` can invoke `reloadData` on its `tableView` and your badges can redraw themselves with the correct counts.

You also now have the capability of persisting messages queued and sent whenever your application is shut down. At your convenience, you can log back in and reread what you've already sent and what you have in the queue (if you're offline).

Summary

You worked with several fascinating technologies in this chapter, most of them open source with very relaxed licenses. I was able to build this demo without having to write too much of my own code. I'm a huge advocate of not reinventing what smarter people have already done better than I could. In this chapter, you worked with Three20, SKPSMTP, and various Cocoa Touch technologies.

You had the opportunity to work with Core Data and understand the basic objects that are used frequently with the entire Core Data stack. You also took a look at how you can leverage Core Data in order to make your application usable even when you're entirely dependent on Internet connectivity to do anything useful. Adding even subtle offline caching can dramatically increase the sales of your next iPhone app, and your users will appreciate it.

I hope this introduction to persisting data using Core Data and working with offline applications has inspired you to build the next killer application that I can use while I'm riding on the train, flying on a plane, or commuting home on the bus through the Lincoln tunnel. To keep up with my updates, you may follow me on Twitter via @stevefink or subscribe to my blog where I'll be holding tech-related musings (it's currently under construction at the time of this writing) via <http://www.stevefink.net>.

Florian Pflug

Peter Honeder



Company: *Florian is working self-employed as a software developer. Peter is co-owner of “Honeder Lacher Wallner Softwareentwicklung OEG”, a software development company.*

Location: *Vienna, Austria*

Former Life As a Developer:

Florian:

- **C and C++ programming, mostly on Unix-based systems.**
- **Network programming, using C, C++ or one of various scripting languages like e.g. Ruby.**
- **Database Design and Development, mostly using PostgreSQL (www.postgresql.org)**
- **GUI programming using the Qt toolkit (www.qtsoftware.com)**

Peter:

- **Multi-platform software development (mostly Windows, Mac, Linux)**
- **Network Programming**
- **GUI programming with Qt**
- **Developing Plug-Ins for Adobe InDesign**

Life as an iPhone Developer: Our Applications:

- iTap



- iTap Volume

What's in This Chapter:

- *And introduction to iTap and the main challenges we faced*
- *A discussion of WiFi networking on the iPhone, focused on the requirements of iTap and similar applications*
- *Auto-discovery of other WiFi devices. We look at both Bonjour and the proprietary solution of iTap.*
- *A few programming tricks, most notably using notifications to achieve better modularization*

Key Technologies:

- *BSD networking API*
- *Core Foundation networking API*
- *Multicasting*
- *Notifications*

How iTap Tackles the Challenges of Networking

Both of us have always been intrigued by developing applications for mobile devices. In the past, we looked at all kinds of mobile devices, but they either lacked major hardware or software features or did not provide convenient means of selling the application. All the rumors regarding Apple months before actually presenting the App Store and the new iPhone 3G to the public made us curious.

Our first step into iPhone development was to apply for the iPhone development program on Apple's web site. Although our initial steps were taken at the end of May 2008, our application was not accepted until the second week of August. The ability to test programs live on our first iPhone increased our motivation to start development.

iTap was born around the end of August 2008 when the first initial prototype was finished and showed a lot of potential. What brought us to develop iTap was a simple requirement of our own: the ability to control our computer while using it to watch movies on a beamer without getting up from the couch.

Completely separately, we thought about developing such an application every time we watched movies with our girlfriends or other friends. Without some kind of wireless keyboard and mouse, having to get up each time you want to turn the volume up or down gets annoying over time. Being software developers, instead of simply buying such devices, we asked ourselves, "Couldn't we somehow use the iPhone for that?" Equipped with an advanced touchscreen, WiFi networking, and a rich API, the iPhone turned out to be the optimal device for this endeavor.

As you can see in Figure 8-1, the main GUI of iTap is very simple. After loading, it shows instructions for the available gestures that fade out after some seconds to clean the screen. The image on the right is a bit more complicated; it shows the iPhone keyboard and some extra buttons such as cursor keys, the ESC key, and multimedia controls.

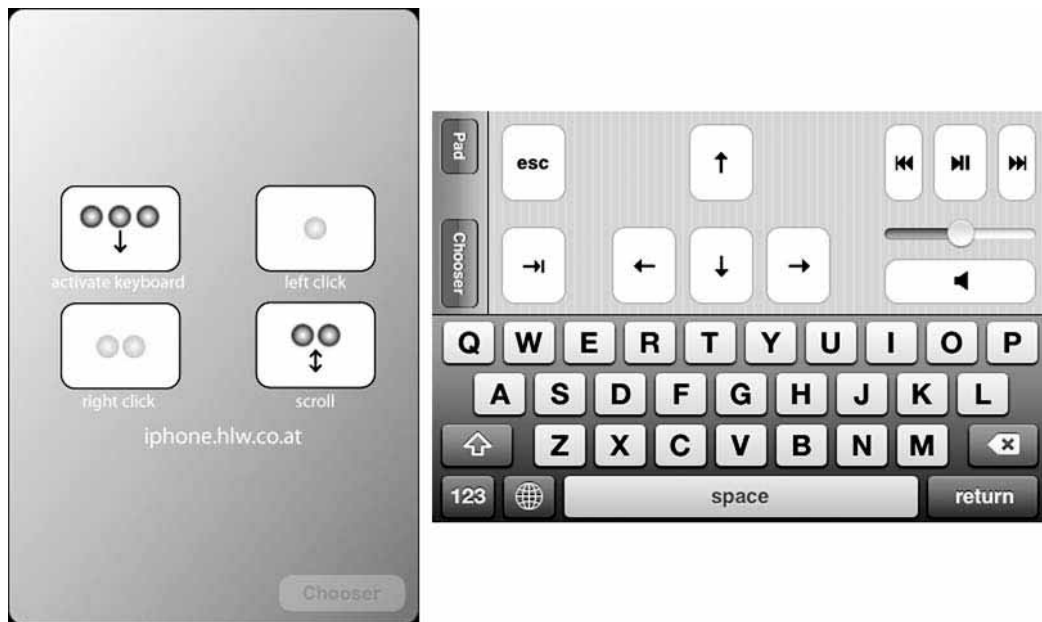


Figure 8-1. *iTap's main GUI*

Meet iTap and iTap Receiver

You can very likely already imagine how iTap works, but in Figure 8-2 we will provide an overview of each of the components involved.



Figure 8-2. *Overview of the connection between iTap and the iTap receiver*

iTap

Effectively, iTap turns any iPhone or iPod touch into a wireless keyboard and touchpad. The iTap iPhone application translates the user's intended pointer movements or keypresses into network packets sent over the WiFi to a computer.

Since neither Mac OS X nor Windows support WiFi-based input devices, the iTap receiver application needs to run on the computer to be controlled. The receiver receives the network packets sent by iTap and synthesizes appropriate mouse movement or keypress events.

This way, users can remote control any application on their computers, most notably media players of all sorts and presentations in Keynote or PowerPoint.

Our iTap receiver is a multiplatform application that runs on Mac OS X as well as Windows.

iTap Receiver

The iTap receiver provides very few interactive features. It mainly allows you to see a running iTap on an iPhone or iPod touch within a list of visible devices. Using the Pair button, you grant a device permission to control the computer (see Figures 8-3 and 8-4.)



Figure 8-3. Receiver main screen with one device paired (Mac)

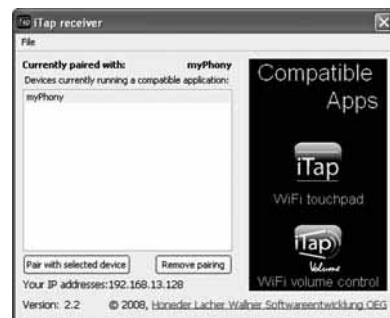


Figure 8-4. Windows version of the iTap receiver

Later we added other features to the receiver. Most notably, we display the IP addresses of the host where the receiver runs and provide better methods for synthesizing keys on Mac OS 10.4, Mac 10.5, and Windows.

Communication between the iTap receiver and iTap on the device is done through UDP packets. Many problems and difficulties that severely impact the complexity of support issues arise because of different and unpredictable network configurations.

You can see that this discussion of features and usability requirements leads to problems especially in the area of networking. We will discuss our solutions to these problems in the later parts of this chapter in more detail.

How the Idea for iTap Emerged and Evolved

Our first iTap prototype showed potential but had many drawbacks. Most notable were the lack of keyboard controls and a network implementation that simply used broadcasts to send all the data. We concentrated on networking and usability for the first version, and thus we added keyboard controls with the first update in the App Store some weeks after the initial release.

Many discussions regarding which features to include in the first version spread out during a break from our day jobs on holiday on a Spanish island during September. A temperature of nearly constant 23 degrees Celsius (73 Fahrenheit) and the possibility of going to the beach any time combined with working on the terrace created many new ideas and sped up the development of iTap drastically. We can only recommend creating first versions of iPhone projects during a holiday without business distractions.

One of the most important things we focused on was usability. It is a key to success for nearly all iPhone applications. Looking back to the beginning of our development clearly shows that much of the positive feedback from our users came from the fact that we concentrated on usability.

The Main Challenges

Compared to other projects we did in our pasts of long-time software development, iTap was a small-scale project. We'll summarize the main challenges because, even for a small project like iTap, it is never wise to underestimate the quality requirements that are necessary to be successful.

No Physical Buttons on the iPhone

iPhones have really small screens. Even for applications like iTap, the screen is not very large if stuffed with all kinds of controls for settings, mouse buttons, special functions like scrolling, and so on. The solution was simply to make use of as many gestures as possible to provide the user with a good set of features while having the screen as empty as possible.

It may not sound obvious, but an empty screen is exactly what we tried to achieve with iTap. Only a completely empty screen would provide the user with the possibility of moving the mouse cursor while looking at his TV screen (and not at his iPhone) and not accidentally hitting buttons at the same time. We designed iTap to be as easy to use as possible in the dark and without looking at it.

Not only did this decision influence how the GUI was presented to the user, but we even restricted the application to not automatically rotate its GUI when the iPhone is rotated. It was quite logical to us that without looking at your device you would not want it to rotate automatically.

Our feature set for the mouse controls always included at least left button and right button presses as well as moving the mouse and two-finger scrolling. Right button clicks

are triggered using a two-finger touch. We quickly developed a small state machine for the first version that included all of these features, but thorough testing on all platforms showed that tuning thresholds for the different inputs was difficult but necessary to provide good user experience.

To facilitate multitouch gestures, we created a user interface that showed a mock-up of an actual notebook touchpad. In addition to left/right button clicks and two-finger scrolling gestures, we implemented a three-finger downward swipe gesture to activate the virtual keyboard.

To provide users with the choice of how to create mouse events, we allow iTap to display a single button or two buttons (left and right). This is not our default configuration for iTap, but Windows users especially sometimes prefer seeing buttons instead of working only with gestures.

Third-Party Applications Cannot Use USB or Bluetooth

Both USB and Bluetooth are interfaces largely inaccessible from third-party applications. Especially for communicating with other computers in a network, the only option is WiFi.

Although iPhone OS 3.0 supports peer-to-peer networking via Bluetooth, this support is mostly limited to networking between iPhones (and iPod touches). The only exception is the tethering option in iPhone OS 3.0, which establishes a network connection between the iPhone and a computer to allow the computer to share the iPhone's Internet connection. Since this network connection is very similar to a WiFi connection from an application's point of view, iTap is able to use it to communicate with the computer.

iPhone applications, however, cannot exercise any control over this connection, and hence the burden of setting it is solely on the user. Given these constraints, WiFi remains the most important means of communication between the iPhone and a computer.

Getting iTap to work on any WiFi network that users may have at home or at work quickly emerged as our main concern. Our first prototypes simply sent all data using broadcasts, which is an easy way to do prototyping. UDP broadcasts provided a way to test all the usability features of iTap without spending a single hour on configuration or complicated autodiscovery.

Broadcasts on the other side quickly emerged to be a big problem for practical use on wireless networks. One of the main reasons is that access points can and will delay sending broadcasts because they try to send them blocked and with less priority. This results in strange behavior when sending mouse movement events; specifically, they start to lag, and this delay makes such an application nearly unusable.

An interesting observation was that controlling a computer connected by a cable to the access point where the iPhone was connected through WiFi resulted in very low network delays even when sending everything as broadcasts. But controlling, for example, a

notebook computer connected to the same WiFi network instead of by cable immediately had severe latency problems in the order of 20ms to 100ms delays.

The solution to this problem was simple; just perform autodiscovery using broadcasts or multicasts and then communicate directly between iTap and its receiver application.

Directly in this context means sending data directly to the other peer's IP address.

Supporting Both Mac and PC

iTap needs the iTap receiver to handle data sent from the device through the network and to synthesize input. The nature of supporting more than one platform creates a whole new set of challenges that can be tracked down differently.

One approach is of course to develop two separate applications and to optimize the visual appearance and installation process of each application for the respective platform. This essentially doubles the effort required to develop and maintain the application compared to supporting only a single platform.

Because we have developed many multiplatform applications in the past, it was no question that we were going to use all the tools, libraries, and possibilities available to create applications looking, installing, and behaving like native ones but also share as many lines of code as possible between the different platforms.

The following were the requirements of the iTap receiver with respect to multiplatform development:

- Easy to install
- Simple to use (and no alien platform look and feel)
- Multiplatform (at least Mac OS X and Windows)

These requirements set strict limitations on which technologies to use and how to implement a version to be released. Simplicity during the install process already limits how it will be distributed through our web site and how it appears to the user. The Mac OS X version comes as a DMG image where you can simply install the iTap receiver by dragging it to your Applications folder. To enhance user convenience, the Applications folder is already prelinked to the default view of the DMG file.

The Windows version is deployed with an installer where we used the Windows Installer XML (WIX) toolkit (<http://wix.sourceforge.net/>) originally developed by Microsoft to create a proper Windows Installer .msi file. The WIX toolkit additionally provides features to add firewall exceptions to the Windows integrated firewall, which proved to be a very important feature to reduce the number and complexity of support requests.

Qt was our choice for the GUI development. It provides all the features required for well-designed applications that are simple to roll out. We also both already had experience developing applications based on Qt from past business software projects that came in handy.

Multiplatform requirements also set limits on how to develop the networking protocol to run on OS X, Windows, and the iPhone itself and on how to implement all kinds of GUIs for OS X and Windows. Implementing all networking code in C++ classes using BSD sockets, which are available on all three platforms, solved many of our problems.

User-Friendliness Demands Autodiscovery of Computers and Devices

The usability of both iTap and the iTap receiver is very important. In the optimal case, they simply find each other; you pair your device with the receiver and are immediately able to control your computer.

iTap is supposed to work out of the box, without any hassles involved in network configuration and especially without entering any kind of IP address. Because we cannot predict which kind of router or access point a user connects to, you can imagine that there were many problems to solve. Not only do home networks provide enough difficulties for proper network communication, there are also university campus networks and company networks with severe restrictions that are even more difficult to solve. You can read our solutions to many of these problems in the following sections of this chapter.

WiFi Networking on the iPhone from a Programmer's Perspective

Before we cover the details of the different networking APIs on iPhone OS, we'll first give you the big picture (see Figure 8-5).

CoreFoundation Higher-Level APIs		Objective-C Higher-Level APIs
CFSocket	DNS-SD	NSSocketPort
UNIX Sockets		

Figure 8-5. Relationship of the different networking APIs on iPhone OS

With iPhone OS having inherited its networking stack from BSD Unix (as did Mac OS X), it comes as no surprise that the principal networking API of iPhone OS is just the same socket-based API initially invented for Unix.

Since Unix predates the widespread adoption of graphical user interfaces and therefore event-driven programming, BSD sockets don't integrate well with such applications. iPhone OS therefore offers two flavors of wrappers around raw BSD sockets. The first, CFSocket, is part of the C-based Core Foundation framework. The second is its

Objective-C counterpart called `NSSocketPort`, which is part of the Objective-C-based Foundation framework.

On top of these, both the Core Foundation framework and the Foundation framework offer additional support for network programming. For example, both frameworks make it quite easy both to browse services offered via Bonjour and to publish your own services.

About the Sample Code

Most of the code presented throughout this chapter is in the form of small utility functions. All these functions are assumed to be part of the class `NetworkDiscovery`, a complete implementation of which is included in the sample code posted on this book's home page. Figures 8-6 and 8-7 show the sample application in action, first detecting an instance of the iPhone Simulator on the same network and then running without WiFi connectivity.

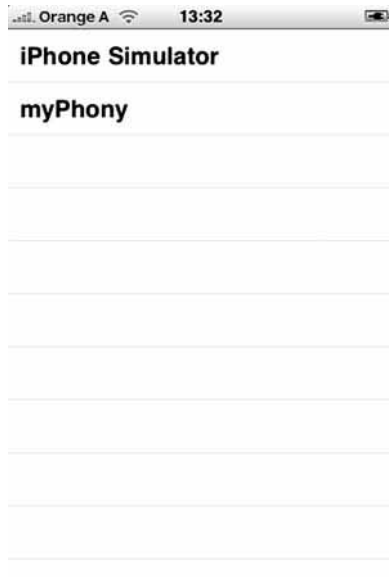


Figure 8-6. The sample application detecting itself and an iPhone Simulator on the network

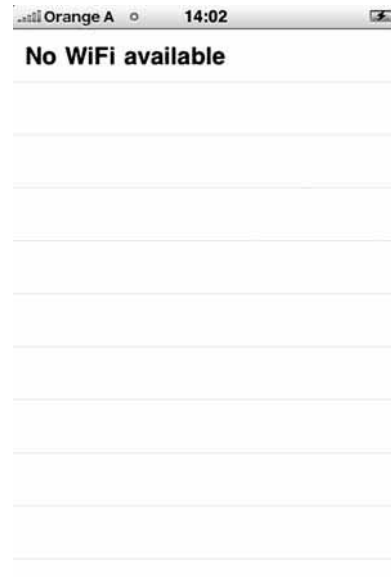


Figure 8-7. The sample application warning about WiFi unavailability

Over the course of the next two sections, we'll show how to gradually develop this class until it has the ability to send and receive datagrams via a WiFi network, to enumerate the available network interfaces, and to send *multicast* datagrams to all other devices on the network. To keep the printed code samples concise, we've stripped them of all error handling and all comments. Where possible, each piece of sample code resembles exactly one of the functions provided by this class, though some longer functions are discussed in more easily digestible pieces.

Listing 8-1 shows the declaration for this class. We've chosen to reproduce the full declaration here, even though the purpose of the individual instance variables will

become clear only over the course of the next sections. This avoids having to update the declaration with each bit of sample code separately.

Listing 8-1. The Declaration of *NetworkDiscovery* Without Its Member and Class Functions

```
@interface NetworkDiscovery : NSObject {
    NSMutableDictionary* peers;

    int socket_bsd;
    CFSocketRef socket_cf;
    CFRunLoopSourceRef socket_runloopsource;

    NSTimer* timer;
    BOOL previousHelloResult;
}
```

While compiling the sample code for this chapter, we had to balance our urge to include as much of iTap's networking code with the need to keep the code as concise and easy to understand as possible. The biggest deviation between the actual code found in iTap and the sample code presented here is in the choice of language. The networking subsystem of iTap is mostly written in C++ since that allowed us to share more code between iTap and the receiver application. The sample code, on the other hand, uses Objective-C exclusively to reach an audience as broad as possible.

Introducing Sockets

Let's now take a closer look at the socket-based networking API iPhone OS inherited from Unix. Following the gist of Unix, this API models a network connection as a file-like entity that can be read from and written to just like any regular file. A file handle referring to a network connection instead of an on-disk file is called a *socket*. Sockets come in different flavors, depending on their underlying network protocol. The flavor is determined at creation time by the socket's *address family*, *socket type*, and *protocol*.

Address Family

The address family distinguishes between protocols with different address formats. We'll always be using `AF_INET`, which selects the Internet Protocol (IP) with the usual 32-bit IP addresses. To use IPv6, you'd use `AF_INET6` instead.

Socket Type

The socket type selects the kind of network connection represented by the socket. `SOCK_STREAM` requests a connection resembling a stream of single bytes, received in the same order they are sent. `SOCK_DGRAM`, on the other hand, requests a datagram-oriented connection that transmits datagrams as a whole, guaranteeing neither that they will be received at all nor in which order. For sockets using the Internet address family (`AF_INET`), `SOCK_STREAM` will trigger the use of TCP for the connection, while `SOCK_DGRAM` sockets will use UDP.

Protocol

In theory, this parameter allows you to choose the precise protocol used to transfer the data over the network. However, for AF_INET-type sockets, your only choice is between TCP and UDP, which is already determined by the socket type. You should therefore just use zero to let iPhone OS select the appropriate protocol.

Creating a Socket

Sockets are created with the function `socket()` taking the address family, the socket type, and the specific protocol as parameters. Here is how you'd create a datagram socket using the Internet Protocol address family:

```
socket_bsd = socket(AF_INET, SOCK_DGRAM, 0);
```

Local and Remote Addresses

Resembling a network connection, a socket usually has two associated addresses. The *local address* is the address sent data originates from, while the *remote address* is the address to which it is sent. Conversely, a socket receives only that data sent from the remote address to its local address. Each address family has an associated datatype used to represent addresses of this family. For IP addresses, that datatype is `struct sockaddr_in`. If you were using IPv6, the correct datatype would be `struct sockaddr_in6`. A generic socket address structure, called `struct sockaddr`, is used in declarations of functions meant to be used with different address families. When calling such functions, you nevertheless have to pass a pointer to a specific kind of socket address structure. In our case, that will be `struct sockaddr_in`. To help such functions find the actual kind of socket address passed, all of the socket address structures store their length in the first field and their address family in the second (called `sin_len` and `sin_family` for `struct sockaddr_in`). As the names imply, the former contains the size of the structure (`sizeof(struct sockaddr_in)` in our case), while the latter contains the address family the address belongs to (`AF_INET` in our case).

Since sockets represent single network connections, not whole devices, the IP address alone is only one part of a socket's address. The other, the *port*, ranges from 1 to 65535.

Although an IP socket address structure contains a field `sin_addr`, for historic reasons this field does not directly contain the IP address. Instead, it contains a structure called `in_addr` with a single field called `s_addr` that contains the actual address as a 32-bit integer value.

To access the port, you don't have to jump through such hoops. The field `sin_port` directly contains the port as a 16-bit short integer value. Listing 8-2 contains the implementation of a helper function that fills out a `struct sockaddr_in`.

CAUTION: Both `sin_addr` and `sin_port` *always* store their values in *network byte order* independent from the byte ordering used otherwise. To convert between this and the iPhone's native byte ordering, use one of these four functions: `htonl()`, host-to-network for 32-bit integers; `ntohl()`, network-to-host for 32-bit integer; `htons()`, host-to-network for 16-bit short integer; or `ntohs()`, network-to-host for 16-bit short integers.

Listing 8-2. Filling Out a struct `sockaddr_in`

```
+ (void)sockaddr_in:(struct sockaddr_in*)sa_in setAddress:(in_addr_t)addr ➡
port:(in_port_t)port {
sa_in->sin_len = sizeof(struct sockaddr_in);
sa_in->sin_family = AF_INET;
sa_in->sin_addr.s_addr = htonl(addr);
sa_in->sin_port = htons(port);
}
```

BYTE ORDERING

On most CPUs on the market today, memory is addressed in quantities of *bytes*; each position in memory contains exactly one byte of data. For data types needing more than one byte of memory, like port numbers (which range from 1 to 65535 and hence need two bytes of storage) or IP addresses (which need four bytes), different *storage layouts* of the individual bytes are possible. The bytes constituting an integer are commonly sorted by powers of two corresponding to the bits they contain.

Ordering the bytes by *ascending* powers of two is called *little endian* for “little end first.” 0xABCD would be stored as 0xCD followed by 0xAB, for example, since the byte 0xCD represents the powers 2^0 to 2^7 , while 0xAB represents the powers 2^8 to 2^{15} .

Ordering the bytes by descending powers of two is called *big endian* for “big end first.” 0xABCD would now be stored as 0xAB followed by 0xCD. Since this byte ordering is often used when transferring data over a network, it is also commonly referred to as *network byte ordering*.

Note that these ordering issues also apply to IP addresses since they can be interpreted as integers. To get the integer representation of an IP address, simply write the individual octets out in binary (or hexadecimal) and join them together to obtain one large number. For example, the address 1.2.3.255 corresponds to the integer 0x010203FF. This address would therefore be stored as 0xFF 0x03 0x02 0x01 on little endian systems and as 0x01 0x02 0x03 0xFF on big endian ones.

Setting a Socket's Local and Remote Address

The BSD networking API offers two functions to set a socket's local and remote address:

- `bind()` sets the local address. To receive data directed to any one of the active network interfaces, use the special value `INADDR_ANY` instead of an actual IP address. If your application does not require a specific port, you may put zero in the field `sin_port`. iPhone OS will assign a random port to your application in this case. Listing 8-3 gives a practical example.
- `connect()` sets the remote address. In the case of a stream-oriented socket, a connection will immediately be opened, and the returned value will indicate whether establishing that connection was successful. For datagram-oriented sockets, calling this function will not trigger any network traffic. The returned value indicates only whether the address was valid. For datagram-oriented sockets, `connect()` is probably useful if you use the sockets to communicate with only a single peer.

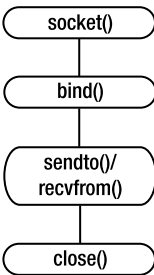
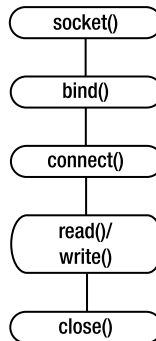
Listing 8-3. *Setting Up a Socket to Receive Datagrams Sent to Port 1234 on an Arbitrary Interface*

```
struct sockaddr_in addr;  
[NetworkDiscovery sockaddr_in:&addr setAddress:INADDR_ANY port:1234];  
bind(socket_bsd, (const struct sockaddr*)&addr, sizeof(addr));
```

Sending and Receiving Data

With a socket being just a special kind of file handle, you can use the usual `read()` and `write()` system calls to send and receive data. For datagram-oriented connections, you might want to use `sendto()` and `recvfrom()` instead. `sendto()` allows you to specify a different remote address for each datagram you send, while a plain `write()` will always use the address set by `connect()`. `recvfrom()` serves a similar purpose, returning the source address of a datagram alongside its content. Figure 8-8 shows the two most common uses of the API for datagram-oriented sockets. We use `sendto()` in Listing 8-4 to implement `sendData:toAddress:port:`, which sends an `NSData` instance over the network.

NOTE: You can still use `sendto()` and `recvfrom()` even if you set a remote address using `connect()`. The use of `recvfrom()` is limited in this case, however, since such a datagram socket will drop all datagrams arriving from sources other than its remote address.

Multiple Peers**Single Peer****Figure 8-8.** Communication with multiple peers vs. communications with a single peer**Listing 8-4.** Using *sendto* to Transmit the Contents of an *NSData* Instance

```

- (BOOL)sendData:(NSData*)data toAddress:(in_addr_t)a port:(in_port_t)p{
    struct sockaddr_in addr;
    [NetworkDiscovery sockaddr_in:&addr setAddress:a port:p];

    return (sendto(socket_bsd, data.bytes, data.length, 0,
(const struct sockaddr*)&addr, sizeof(addr)) == data.length);
}

```

Raw BSD Sockets vs. User Experience

Achieving the best possible user experience is an important factor in the success of an iPhone application. One key factor of good user experience is to never take the control out of the user's hands, which turns out to be hard to achieve using raw BSD sockets alone. Let's look into the problem before we turn to the next section for a solution.

Studying the `read()` and `write()` system calls in more detail quickly brings up the question of how these system calls react if the desired requests cannot be carried out immediately. For example, consider what happens if your application calls `read()` (or `recvfrom()`, for that matter) but no datagram is currently available to be delivered to your application.

By default, `read()` will *wait* for data to arrive before returning to your application. During this waiting period, your application will *not* be reacting to any user input; after all, from its point of view, it's still executing the `read()` system call and has never returned to the run loop.

Alternatively, you can set the `O_NONBLOCK` flag of the socket via the `F_SETFL` command of the `fcntl()` system call. This will cause system calls to return the error `EAGAIN` instead of waiting. However, you will still get no indication of when the next piece of data will arrive (in case of a read) or when the network will be ready to send more data (in case of a write). Your only option is to repeatedly retry the operation until you either give up or it eventually succeeds.

Neither option fits the event-driven programming model of iPhone applications very well. For sending datagrams, however, using `sendto()` (or `write()`) is an acceptable choice as long as the rate of datagrams is low enough for the network interface to keep up.

Using CFSocket to React to Networking Events

As you saw earlier, raw BSD sockets do not fit very well into the event-driven world of iPhone programming. Luckily, Core Foundation provides a bridge between these worlds in the form of CFSocket.

A CFSocket object exposes its underlying BSD socket via the function `CFSocketGetNative()` and can even be created from a raw BSD socket with `CFSocketCreateWithNative()`. This gives you the option to either use CFSocket as a mere adapter to integrate a raw BSD socket into your application's run loop or instead use the Core Foundation wrapper functions extensively and revert to the BSD API only when absolutely necessary.

We've taken the first approach throughout this chapter since we will require some functionality that is not accessible from higher layers. For your own projects, we advise you to stick with whatever makes your code more consistent.

Reacting to Incoming Datagrams

When you create a CFSocket, you need to provide a callback function to be called whenever this occurs:

- The new data arrives and is ready to be read (`kCFSocketReadCallBack`). The callback has to read the data itself, and the necessary call to `read()` is guaranteed not to block your application.
- The new data arrived and was read (`kCFSocketDataCallBack`). The data is passed to the callback as a `CFDataRef`.
- The socket is ready to queue more data for transmission (`kCFSocketWriteCallBack`). The next `write()` is guaranteed not to block your application.
- A new peer establishes a connection (`kCFSocketAcceptCallBack`). The socket representing the newly established connection is passed to the callback as a pointer to a `CFSocketNativeHandle`. You can use `CFSocketCreateWithNative` to create a CFSocket from that. This event occurs only on stream-oriented sockets.
- Trying to establish a connection to a peer (`kCFSocketConnectCallBack`) ends. This event occurs as a result of trying to establish a connection in the background using `CFSocketConnectToAddress()`. If an error occurs, a pointer to an `SInt32` containing the error code is passed to the callback. This event occurs only on stream-oriented sockets.

To aid you with accessing per-socket data structures inside your callback function, CFSocket provides a way to pass an arbitrary pointer to your callback. You specify this pointer via the `info` member of the `CFSocketContext` structure passed to `CFSocketCreate` or `CFSocketCreateWithNative`. To have CFSocket track how many references it holds to the entity pointed to by `info`, you need to provide pointers to a retain function and a release function in the respective fields of `CFSocketContext`. For example, if `info` pointed to a Core Foundation object, you might set the retain field to point to `CFRetain()` and the release field to `CFRelease()`.

In the example in Listing 8-5, we use the `info` field to store a reference to the Objective-C object owning the socket. Since the socket is deleted during deallocation of the object, we do not require Core Foundation to track its references to it and hence just set the retain and release members to `NULL`.

Inside the callback shown in Listing 8-6 (which must be a plain C function), we convert `info` back to an Objective-C reference and send it the message `onDatagram:fromAddress:port:`.

NOTE: A lot of basic Core Foundation data types are bridged toll-free to their Objective-C counterparts, meaning you can simply treat the Core Foundation reference as an Objective-C object reference. We use this in the code samples to convert between `CFDataRef` and its Objective-C counterpart `NSData`.

Listing 8-5. *Creating a CFSocket from a Raw BSD Socket to Handle Incoming Datagrams*

```
CFSocketContext ctx;
ctx.version = 0;
ctx.info = self;
ctx.retain = NULL;
ctx.release = NULL;
ctx.copyDescription = NULL;
socket_cf = CFSocketCreateWithNative(
    kCFAllocatorDefault,
    socket_bsd,
    kCFSocketDataCallBack,
    NetworkDiscovery_CFSocketCallBack,
    &ctx
);
```

Listing 8-6. *The Callback Function*

```
static void NetworkDiscovery_CFSocketCallBack(
    CFSocketRef,
    CFSocketCallBackType callbackType,
    CFDataRef addr,
    const void *data,
    void *info
) {
    const struct sockaddr_in* src=(const struct sockaddr_in*)CFDataGetBytePtr(addr);
    if (callbackType == kCFSocketDataCallBack)
        [(NetworkDiscovery*)info onDatagram:(NSData*)data
        fromAddress:ntohl(src->sin_addr.s_addr)
```

```

    port:ntohs(src->sin_port)];
}

```

Querying the Network Configuration

Being able to just send and transmit data over the network might not be sufficient for your application, and it certainly isn't for iTap, as you will see when we go into the details of autodiscovery. We'll therefore look into how to use the BSD networking API to query the network configuration. For the sake of brevity, we'll concentrate on two specific tasks: getting the names of all the available network interfaces and querying their flags.

Introducing IO Controls

At first glance, functions to query or modify the network configuration seem to be suspiciously absent from the BSD networking API. They all deal with either setting up or shutting down sockets (like `socket()`, `bind()`, `close()`, ...) or data transmission (like `read()`, `write()`, `sendto()`, ...).

However, once again, the key is to take the Unix inheritance of this API into account. On Unix, a file handle is not only something you can read from and write to but also something you can send certain requests to. These requests are called *IO controls* and are invoked with the function `ioctl()`. And they turn out to be the key to getting at the iPhone's network configuration.

As you can see in Listing 8-7, the IO control mechanism is designed to be quite generic because the number and types of a request's parameters depend on the request in question. We will now look into how to call a specific request: the one that lists all available network interfaces.

Listing 8-7. *The `ioctl()` Function*

```
int ioctl(int socket_bsd, unsigned long request, ...);
```

Querying the Names of the Available Interfaces Using the SIOCGIFCONF IO Control

SIOCGIFCONF requires a single parameter that must be a pointer to a structure fittingly called `ifconf`. Take a look at Listing 8-8 to see how this structure is defined in the header `net/if.h`.

Listing 8-8. *The Definition of `struct ifconf`*

```

struct ifconf {
    int ifc_len; /* size of associated buffer */
    union {
        caddr_t ifcu_buf;
        struct ifreq *ifcu_req;
    } ifc_ifcu;
};

```

```
#define ifc_buf ifc_ifcu.ifcu_buf /* buffer address */
#define ifc_req ifc_ifcu.ifcu_req /* array of structures returned */
```

As a rule, IO controls do not allocate memory, not even to store the results they return. Similarly, they do not take ownership of pointers you pass to them. This is the reason for the rather strange definition of struct ifconf. When making the SIOCGIFCONF request, you need to provide the IO control with a memory area to store its results in. To set that buffer, you use the union member ifc_ifcu.ifcu_buf or its abbreviation ifc_buf. After the IO control returns, this buffer will contain one struct ifreq struct after the other, one for each network interface on the device. You can access the first of these structures with ifc_ifcu.ifcu_req or its abbreviation ifc_req.

Unfortunately, accessing the subsequent structure is a bit more involved. Let's first show you part of the definition of struct ifreq, reproduced in Listing 8-9.

Listing 8-9. The Definition of struct ifreq

```
struct ifreq {
    char ifr_name[IFNAMSIZ]; /* if name, e.g. "en0" */
    union {
        struct sockaddr ifru_addr;
        /* Other union members not relevant for SIOCGIFCONF */
    } ifr_ifru;
#define ifr_addr ifr_ifru.ifru_addr /* address */
/* Other abbreviations for union members not relevant for SIOCGIFCONF */
```

If our earlier discussion of address families and their associated socket address structure is still fresh on your mind, you will find this ifru_addr member highly suspicious. After all, struct sockaddr itself isn't capable of holding *any* socket's address; it's merely used as a placeholder for one of the actual address structures like struct sockaddr_in.

Consequently, *none* of the structures returned by SIOCGIFCONF *actually* contains a struct sockaddr. Instead, each structure contains a socket address structure matching the address family of the interface described by this structure. Since the length of these socket address structures differs, so does the length of the ifreq structures containing them. To be able to scan the ifreq structures placed into the buffer we provided, we'll need to find a way to compute their lengths, preferably without hard-coding knowledge about each and every socket address structure into our application.

Luckily, looking back at the discussion of address families not only uncovers the problem but also provides a solution. Remember that *every* address structure, no matter what address family it belongs to, stores its own length in its first field. Even the placeholder struct sockaddr declares that field, naming it sa_len.

You can therefore use this field to move from one structure in the buffer to the next; you simply need to move the pointer by IFNAMSIZ bytes plus whatever the sa_len field says is the size of the address structure. Using the second field common to every address structure, the address family, you can skip interfaces that do not have an Internet address assigned.

To actually invoke the SIOCGIFCONF IO control, you have to make two additional decisions. Since you'll need to provide a buffer to the IO control to store the result in,

you'll have to decide how much memory to allocate. Unfortunately, you cannot query the size required to store the information for all available interface. You therefore need to call the IO control repeatedly with increasing buffer sizes until the empty space left at the end of the buffer is larger than any additional struct `ifreq` might be. Only then can you be sure not to miss an interface because of a too small buffer.

You'll also need a socket to perform the IO control on. In our sample code, we simply create a dummy datagram socket without any addresses assigned, but you could also reuse an existing datagram socket.

The sample code for this chapter found on the home page contains the function `interfaceNamesAddresses`, which returns an NSArray of NSDictionarys each containing the name and address of one network interface.

Querying an Interface's Flags Using the SIOCGIFFLAGS IO Control

Just having a name and an associated address is not all there is to an interface. Networking interfaces might have additional capabilities and be in different states (for example, active or inactive). These properties are represented by the interface flags listed in Table 7-1, queried with the help of SIOCGIFFLAGS, as shown in Listing 8-10.

Table 7-1. *Interface Flags*

Name	Description
IFF_UP	The interface is active.
IFF_LOOPBACK	The interface does not represent a real network. Usually this has the address 127.0.0.1.
IFF_POINTOPOINT	The interface represents a remote connection to a single peer. Such interfaces are used to connect to the Internet via the carrier and for VPN connections.
IFF_MULTICAST	The interface allows sending datagrams to groups of hosts.

Listing 8-10. *Using the SIOCGIFFLAGS to Query an Interface's Flags*

```
+ (short) interfaceFlags:(NSString*)interface {
    int sock = socket(AF_INET, SOCK_DGRAM, 0);

    struct ifreq req;
    [interface getCString:req.ifr_name
                     maxLength:IFNAMSIZ
                     encoding:NSUTF8StringEncoding];

    ioctl(sock, SIOCGIFFLAGS, &req);
    return req.ifr_flags;
}
```

Other Interesting IO Controls

Table 7-2 lists other IO controls concerned with retrieving the configuration details of a specific interface.

Table 7-2. *Network-Related IO Controls*

Name	Description
SIOCGIFCONF	Gets the list of available interfaces
SIOCGIFFLAGS	Gets the flags of a specific interface
SIOCGIFADDR	Gets the address of a specific interface
SIOCGIFNETMASK	Gets the netmask of a specific interface
SIOCGIFBRDADDR	Gets the broadcast address of a specific interface

Contacting All Devices on the Network

Usually, sockets model network connections between exactly two peers. But to *discover* possible peers, you need a way to contact all devices at once, without knowing their IP addresses in advance.

Multicasts provide a way to do just that. Additionally, to the unique IP address assigned to every device, a device on a network may join one or more so-called multicast groups. These groups correspond to special IP addresses in the range 224.0.0.0 to 239.255.255.255. Once joined, the device receives not only those datagrams targeted at its own IP address but also those targeted at the group's multicast address.

A special multicast address, 224.0.0.1 (INADDR_ALLHOSTS_GROUP), corresponds to the all-hosts multicast group. This group includes all devices on a network segment without requiring the device to explicitly join.

Multicasts are sent by simply passing a multicast address as the destination address to `sendto()`. Without further specification, however, the iPhone will *only* send the multicast datagram out on the default network interface, *not* on all available ones! In our experience, this causes the multicast to not get sent to the WiFi network under some circumstances, even though the iPhone shows the WiFi connection to be active.

To work around this, you need to manually override the network interface used for multicasts. This can be done on a per-socket basis by modifying the `IP_MULTICAST_IF` option of the socket using the `setsockopt()` function provided by BSD networking API, as shown in Listing 8-11.

Listing 8-11. Using `IP_MULTICAST_IF` to Override the Outgoing Network Interface Before Sending a Multicast

```

- (BOOL)multicastData:(NSData*)data toGroup:(in_addr_t)mcGroup ➡
port:(in_port_t)dstPort onInterfaceWithAddress:(in_addr_t)ifaceAddr {
    struct in_addr sin_addr;
    sin_addr.s_addr = htonl(ifaceAddr);
    setsockopt(socket_bsd, IPPROTO_IP, IP_MULTICAST_IF, &sin_addr, sizeof(sin_addr));

    return [self sendData:data toAddress:mcGroup port:dstPort];
}

```

Refer to the sample code for this chapter on the book's home page for the function `multicastData:toGroup:port:` that sends the multicast out on all available networking interfaces by combining `multicastData:toGroup:port:onInterfaceWithAddress:` with `interfaceNamesAddresses`.

Detecting WiFi Availability

The iPhone SDK contains the `SCNetworkReachability` framework to help you determine whether a certain peer is reachable or not given the current network configuration. By being integrated with the application's run loop, `SCNetworkReachability` not only supports one-time queries of a certain peer's state but also lets you register a callback function to be called whenever that state changes.

However, as powerful a tool `SCNetworkReachability` is for monitoring a *single* peer, it is not as well suited for monitoring the *general* availability of a WiFi network. Essentially, `SCNetworkReachability` answers questions like “Will datagrams originating from a certain *local address* be able to reach a certain *remote address*?” Even though these can be extended to “Will datagrams originating from any local address be able to reach a certain remote address?” and “Is a certain local address assigned to this device at all?” by leaving one of the addresses blank, these questions still focus on the reachability of individual addresses.

NOTE: The reachability status is never probed for by actually sending out packets. It instead represents a theoretical result based on the device's routing tables and VPN configuration. Therefore, although a negative reachability result *does* guarantee that datagram transmission will fail, a positive one *does not* guarantee they will succeed.

We'll therefore turn to the BSD networking API for a different approach to detecting WiFi availability. We've already established how to enumerate all available network interfaces and their assigned addresses, implemented in the function `interfaceNamesAddresses`. Furthermore, we've shown how to query an interface's flags by using the `SIOCGIFFLAGS` IO control. The following facts allow us to judge WiFi availability by scanning for an active, non-point-to-point network interface that supports multicasting and isn't a loopback interface:

- The network connection to the carrier (via GRPS, EDGE, or UMTS) is established via PPP and hence has the flag `IFF_POINTOPOINT` set.

- The WiFi interface always supports multicasting and hence has the flag `IFF_MULTICAST` set.
- The loopback interface with the address 127.0.0.1 has the flag `IFF_LOOPBACK` set.
- The flag `IFF_UP` shows whether an interface is activated or deactivated.

Since the interfaces fulfilling these criteria are exactly the ones we'd want to send out multicast datagrams on, we can conveniently integrate both functionalities into one function `multicastData:toGroup:port:`, shown in Listing 8-12. The function returns YES if it managed to multicast the datagram on at least one suitable interface, which is exactly the indicator for WiFi availability we need.

Listing 8-12. *Multicasting on All Suitable Interfaces and Detecting WiFi Availability*

```
- (BOOL)multicastData:(NSData*)data toGroup:(in_addr_t)mcGroup
port:(in_port_t)dstPort {
    BOOL result = NO;

    short flags_on = IFF_MULTICAST | IFF_UP;
    short flags_off = IFF_POINTOPOINT | IFF_LOOPBACK

    for(NSDictionary* ifc in [NetworkDiscovery interfaceNamesAddresses]) {
        NSString* ifc_name = [ifc objectForKey:@"name"];
        short ifc_flags = [NetworkDiscovery interfaceFlags:ifc_name];
        if (((ifc_flags & f_on) == flags_on) && !(ifc_flags & f_off)) {
            NSNumber* ifc_addr = (NSNumber*)[ifc objectForKey:@"address"];
            BOOL ifc_result = [self multicastData:data
                                toGroup:mcGroup
                                port:dstPort
                                onInterfaceWithAddress:ifc_addr.unsignedIntValue];
            result = result || ifc_result;
        }
    }
    return result;
}
```

Playing by the Power Management Rules

On a portable device like the iPhone, sophisticated power management throughout the whole operating system is an important part of the overall user experience. Since the WiFi radio is amongst the biggest consumers of power, applications using the WiFi extensively need to respect a few power management rules to provide the best user experience possible.

Informing iPhone OS About Your Application's Networking Requirements

By default, iPhone OS won't assume that your application depends on the availability of a WiFi network. This has multiple consequences:

- If the iPhone is not already connected to a WiFi network at the time your application is launched, iPhone OS will not make any effort to establish such a connection while your application is running. This is true even if your application tries to connect to peers that are unreachable without an active WiFi connection.
- If your application runs for longer than about 30 minutes, iPhone OS might shut down the WiFi radio. Open connections to peers and even active transmissions do not stop it from doing so.

If your application is of limited or no use without an active WiFi connection, you will need to convince iPhone OS to make a bigger effort to provide one. You do that by adding the key `UIRequiresPersistentWiFi` with the boolean value `true` to the `Info.plist` file of your application.

Doing so has two effects:

- If no WiFi connection exists at the time your application is launched, iPhone OS will either connect to an available WiFi network automatically or ask the user to choose from the list of available networks. The user does have the ability to cancel this selection process, though, in which case no WiFi will be available to your application.
- The WiFi radio will not be shut down while your application is running. It will, however, be put into a power-saving mode if you cease sending and receiving data.

However, setting this key has the potential of substantially increasing the power consumption of the iPhone while your application is running. To alleviate this effect, it is important to cease sending and receiving data whenever possible.

Minimizing Power Consumption While the iPhone Is Locked

Without special consideration, the currently active application will keep running while the iPhone is locked. However, since your application is neither visible nor operable while the device is locked, continuing to use the WiFi network during that time will drain the battery without any benefits for the user.

If locking is imminent, `UIApplication` will send the message `applicationWillResignActive` to its delegate. This method should do its best to prevent any part of the application from using the WiFi network unless there is a clear advantage of doing so even while locked.

Upon unlocking, `UIApplication` sends `applicationDidBecomeActive` to its delegate. This is the place to realow transmission and reception of packets.

Implementing your network protocol as a singleton class (a class with only one instance, like for example `UIApplication`) makes it easy to do that. For example, the `NetworkDiscovery` class discussed throughout this chapter provides two class-level

functions called setup and shutdown, which we use in Listing 8-13 to create and destroy the one instance of this class.

Listing 8-13. *Ceasing to Transmit and Receive Datagrams While the Device Is Locked*

```
- (void)applicationWillResignActive:(UIApplication *)application {  
    [NetworkDiscovery shutdown];  
}  
  
- (void)applicationDidBecomeActive:(UIApplication *)application {  
    [NetworkDiscovery setup];  
}
```

The Networking Subsystem of iTap

We will now give you a tour through the code of one of iTap's core components: the networking subsystem. After reading the previous sections, you are well adept at the inner workings of the iPhone networking APIs. Having discussed most of the core networking-related function of iTap, we will now focus on the bigger picture. We'll explain some of the design decisions we faced while implementing iTap and show you how we integrated the networking component into the rest of the application. Our code samples will again closely follow the downloadable version of the sample code.

To use Bonjour or Not to Use Bonjour

One of the first decisions we faced while designing the networking subsystem of iTap was whether to use Bonjour for autodiscovery or to implement our own protocol for that. Here is what Bonjour has to offer to the programmer:

- The ability to publish services—identified by a type and a name—on connected networks
- The ability to browse for services published by others
- Notifications if new services are added or removed by others
- The same APIs work on both Mac OS X and iPhone OS

This functionality is available via both an Objective-C API consisting of the classes `NSNetService` and `NSNetServiceBrowser` and a C-based API called `CFNetService`, which is part of Core Foundation. The service browsing and publishing parts of Bonjour are based on an extension of the DNS protocol used to translate names to IP addresses on the Internet called DNS-SD. A third API, also available on both iPhone OS and Mac OS X, provides raw access to DNS-SD.

NOTE: Since Apple provides extensive documentation and sample code covering both APIs, we will not provide additional code samples here.

Since supporting both Mac OS X and Windows was one of the goals of iTap right from the start, let's take a look at the state of Bonjour on Windows:

- Although Bonjour is an integral part of Mac OS X and iPhone OS, it isn't on Windows. To use Bonjour in a Windows application, you either need to require your users to download and install Bonjour for Windows themselves or include that step in the installation process of your application. This would preclude any "download-and-run" version of the iTap receiver for Windows.
- Both higher-level APIs to Bonjour are too deeply tied to other core frameworks on Mac OS X and iPhone OS to be usable on Windows. The only API remaining is the raw DNS-SD one. This is what the iTap receiver would need to use if iTap were based on Bonjour.

In the end, we thought that although Bonjour might have some technical merits, from a user's point of view rolling our own solution was clearly beneficial. Forcing our Windows users to install a whole new system component just to use our receiver application just didn't seem right. Besides, each additional component used is an additional component to support. Since we anticipated that supporting iTap in all kinds of different networking environment would not be an easy task, we were reluctant to add yet another possible source of problems.

Using Notifications to Communicate Between Components

To keep the code of a larger application as easy to understand and extend as possible, you will usually strive to separate the application into separate components. Ideally, these components are largely self-contained and able to perform their task with as little knowledge about other parts of the application as possible. Trying to adhere to this ideal proves to be difficult in practice, though. For example, take a look at our little sample application called Discover.

This application contains two components: the `NetworkDiscovery` class introduced earlier and a `UITableView` plus its view controller `NetworkDiscoveryPeerTable`. It's the responsibility of `NetworkDiscovery` to monitor WiFi availability and to detect other instances of Discover running on the same network. The table view and its associated view controller are responsible for visualizing this information.

Given this separation of responsibilities, you need a way for the `NetworkDiscovery` class to inform `NetworkDiscoveryPeerTable` about changes to the list of peers or to WiFi availability.

You could of course let `NetworkDiscovery` store a reference to the instance of `NetworkDiscoveryPeerTable` somewhere and simply send that instance some message to signal an event. But doing so would mean abandoning the modularity of the application that you seek to achieve. For example, imagine the application contained some button that you wanted to be visible only if at least one other device is detected. Since that button would presumably not be managed by our table view controller, you'd

need to extend `NetworkDiscovery` to store a reference to a second object too and to signal events to both of them.

Fortunately, the iPhone SDK provides a much better solution for propagating such events between different components, called *notifications*.

Notifications are arbitrary strings that are posted to some notification center. Other components may indicate their interest in certain notifications by registering as an *observer*, specifying an object instance and an Objective-C message selector. The notification center will then send the specified message to the specified object instance if the observed notification is posted.

Although it is possible to create an arbitrary number of notification centers, it is usually sufficient to just use the application's default notification center returned by `[NSNotificationCenter defaultCenter]`.

Listing 8-14 shows how to register as an observer, requesting the function `onPeersChanged:` to be called should the notification `PeersChanged` be posted by any other component. By specifying something other than `nil` for the parameter `object:`, you could restrict your observation to notification posted by a specific sender.

Listing 8-14. Registering as an Observer

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

`NetworkDiscovery` would then need to post said notification to signal a change to the list of peers to only every other component in the application. Listing 8-15 shows an example of how to post such a notification.

Listing 8-15. Posting a Notification

```
[[NSNotificationCenter defaultCenter] postNotificationName:@"PeersChanged"
object:self];
```

NOTE: Notifications are delivered immediately upon being posted. The `postNotificationName:object:` functions returns only *after* delivering the notification to all registered observers.

Our Custom Autodiscovery Solution

Having decided to roll our own autodiscovery solution, the next step was to design one. To reduce the amount of possible problems with strange network configurations and strict firewall policies, we've tried to keep our protocol as simple as possible.

Each device uses multicasting to send its name out periodically on all available network interfaces to the all-hosts multicast group. The previously introduced function `multicastData:toGroup:port:` handles that job nicely. The system header `netinet/in.h`

even defines the constant `INADDR_ALLHOSTS_GROUP` containing the multicast address of the all-hosts group in a form suitable for our function. Should `multicastData:toGroup:port:` be unable to find any suitable network interface to send the datagram on, we alert the user that no WiFi network connection is currently available. The sample code developed so far allows for a quite concise implementation of this algorithm, as you can see in Listing 8-16.

Listing 8-16. Sending the Device Name Out on All Network Interfaces and Triggering a Notification Should That Fail

```
BOOL result = [self multicastData:[[[UIDevice currentDevice] name]
                                   dataUsingEncoding:NSUTF8StringEncoding]
              toGroup:INADDR_ALLHOSTS_GROUP
              port:DISCOVERY_PORT];

if (result && !previousHelloResult)
    [[NSNotificationCenter defaultCenter] postNotificationName:@"WiFiAvailable"
                                                             object:self];
else if (!result && previousHelloResult)
    [[NSNotificationCenter defaultCenter] postNotificationName:@"WiFiNotAvailable"
                                                             object:self];

previousHelloResult = result;
```

Upon receiving a datagram, we extract the device name and store it together with the time of reception. If the device name is already known, we overwrite the previously stored timestamp. `NSMutableDictionary` turns out to be the most convenient way of storing these names and their associated times. Listing 8-17 shows the corresponding source code.

Listing 8-17. Reception of a Datagram

```
NSString* peer = [[NSString alloc] initWithData:data
                                                  encoding:NSUTF8StringEncoding]
                autorelease];
NSNumber* time = [NSNumber numberWithDouble:CFAbsoluteTimeGetCurrent()];

BOOL peer_added = ([peers objectForKey:peer] == nil);
[peers setObject:time forKey:peer];

if (peer_added)
    [[NSNotificationCenter defaultCenter] postNotificationName:@"PeersChanged"
                                                             object:self];
```

Finally, we check periodically to see whether any device's timestamp is older than some multiple of the sending interval. Should this be the case, we assume the device has vanished and remove it from the list. Since `NSMutableDictionary` instances must not be modified while they are traversed, the code in Listing 8-18 builds a new `NSMutableDictionary` and swaps it with the original one at the end.

Listing 8-18. Purging of Vanished Devices from the Peers List

```
CFAbsoluteTime now = CFAbsoluteTimeGetCurrent();
NSMutableDictionary* peers_new = [[NSMutableDictionary alloc] init];
for(NSString* peer in peers) {
    CFAbsoluteTime time = [((NSNumber*)[peers objectForKey:peer]) doubleValue];
    if (time > now - 3.0*DISCOVERY_INTERVAL)
```

```

        [peers_new setObject:[peers objectForKey:peer]
                        forKey:peer];
    }
    BOOL peer_removed = (peers.count != peers_new.count);
    [peers release];
    peers = peers_new;

    if (peer_removed)
        [[NSNotificationCenter defaultCenter] postNotificationName:@"PeersChanged"
                                                object:self];

```

Summary

Implementing a WiFi touchpad application like iTap poses unique challenges, both from a technical as well as from a user interface point of view.

From a user interface perspective, the most challenging aspects of the iPhone platform are probably the lack of physical buttons and the limited screen real estate compared to a full-blown computer. Power consumption was also a big concern of ours, since users do not react too well to battery-draining applications.

From a technical perspective, we had to deal with subtle differences between the vast number of WiFi networks out there, while still autodetecting available devices and computers in nearly all cases and staying compatible with both Mac OS X and Windows.

To conserve screen real estate and to allow our users to use the iTap touchpad blindly, we've adhered to a reductionist user interface philosophy, relying heavily on gestures instead of on-screen buttons.

On the technical side, we bypassed most of the higher-level layers of the iPhone networking stack including Bonjour and got our hands dirty with raw POSIX UDP sockets and multicasting. This allowed us to engineer a network protocol for iTap that both supports autodetection and works with Windows as well as Mac OS X. To get multicasting to work reliably, we had to turn to the introspection capabilities of the POSIX API to enumerate the available network interfaces and query their states. This work also allowed us to build a WiFi detection algorithm more tailored to iTap's needs than the `SCNetworkReachability` framework is.

Although not strictly dealing with iPhone development per se, the Qt toolkit (www.qtsoftware.com) proved to be the perfect toolkit to implement our receiver application with. Although we still had to write a fair amount of platform-specific code to support the iTap receiver on both Mac OS X and Windows, using the Qt GUI toolkit saved us from having to code two completely separate applications for these two platforms.

Last but not least, we've found `NSNotification` and friends to be an invaluable tool in the struggle to keep our code clean and modular.

We hope that we've managed to pass some of the knowledge we gained while developing iTap on to you, and we hope to see a great number of new and exciting iPhone applications soon!

Jonathan Saggau



Company: Sounds Broken inc

Location: New York, NY

Former Life As a Developer: *I started writing code in BASIC on the TANDY 1000HX my parents bought when I was young. Throughout High School in Iowa, I was keenly interested in both computers and music, programming video games for the TI-82 and writing a checkbook balancer in Pascal. Throughout my undergraduate and graduate studies in music composition, I used Mathematica and Python to generate musical possibilities, eventually developing a psycho-acoustical model as a means to model my orchestrations of acoustic music and to generate electronic sound as well as an automated musical pattern recognition engine. After graduate school, I became interested in Cocoa programming through PyObjc and started taking classes at the Big Nerd Ranch in Atlanta. Since then, I have been freelancing as a Cocoa and iPhone developer and loving every minute of it.*

Life as an iPhone Developer: *I'm a subcontractor on several projects through my own company and, more often, through a private development firm. My company's first solo application, gogoDocs, should be on the app store very soon. It is an online and offline reader for the popular Google docs service. Come visit us at gogodocs.com for updated information.*



What's in This Chapter: *Perceived speed and interface responsiveness on the iPhone, especially when dealing with data stored on the Internet or when dealing with large data sets can present serious challenges. In this chapter, you'll step through the development of two applications, one that deals with downloading*

and displaying stock price information stored remotely on yahoo.com and another that implements a large image viewer in a scroll view.

Key Technologies:

- *Optimization techniques*
- *Concurrency*
- *UIScrollView*
- *NSOperation / NSOperationQueue*
- *Open source technologies: Plausible Blocks and Core-Plot*

Fake It 'Til You Make It: Tips and Tricks for Improving Interface Responsiveness

Why do some native applications seem so fast while others do not? There is an old adage in auto racing. “Speed is money. How much do you want to spend?” It doesn’t take long for iPhone programmers to rub up against a similar problem, one perhaps expressed as, “Speed is time. How much do you have left to spend before release?” Given the limitations of processor power, RAM, and network bandwidth, not to mention battery drain, writing iPhone applications that display lots of data is hard. Clever caching, prefetching of data, and optimized drawing are the keys to removing the variable response times that make an app that’s consuming nonlocal or large amounts of data seem slow to the user.

How can you avoid a “death by 1,000 paper cuts” user experience when you have a lot of data to display? Most of the applications that Apple ships on the iPhone access network services, and many of them deal with large data sets. Mail pulls and caches potentially large amounts of data from your mail server, the Maps application loads tiles from Google Maps, and the Weather application requests the latest weather on demand; even the Calendar and Contacts applications can sync with data stored on servers hosted by Microsoft, Google, Yahoo, and Apple. Many well-reviewed third-party applications also pull large quantities of data from the cloud in one way or another. Facebook, Pandora, AIM, Yahoo Instant Messenger, and many others have developed offerings that are robust and responsive. Writing an application for the iPhone that displays large amounts of potentially nonlocal data is not easy. You’ve probably experienced an application that seems to start and stop working depending on your network connection or how much information you’ve loaded. Users of native iPhone

applications have different expectations with regard to interface responsiveness than they do when browsing the Web. It's difficult to satisfy a user who tolerates multiple page loads while using a browser but who may not tolerate a slow-scrolling table view or a view that takes a few seconds to download data and render in a native application.

I am keenly interested in iPhone application responsiveness. As a freelance iPhone developer primarily writing various applications that connect to database or HTTP servers, I have had a lot of opportunity to watch my first stabs at cloud-based applications seem abysmally slow to respond to user input because the app is busy downloading or parsing data. Since a lot of my time is spent talking to servers, I have amassed a fair number of tricks to make applications seem faster than they really are, from prefetching data to caching to drawing to off-screen contexts in separate threads. I am excited to share some of those tricks with you.

In this chapter, I'll show how to improve the responsiveness of two projects. The first project starts out as an app that displays historical AAPL stock information from Yahoo.com and graphs closing prices over time similar to Apple's own Stocks application. As you add functionality, I'll discuss some strategies as well as some of the trade-offs involved with various methods of caching information from remote data sources. By the time you're done, the application will cache and update the stock prices of several stocks while remaining usable and responsive to the user. The second project deals with displaying large amounts of information in a scroll view that is generated and drawn programmatically. In that project, I'll show how to solve some common performance and user experience problems related to drawing large amounts of data.

Plotting of Historical Stock Prices with AAPLot

In this section, you'll start with a simple application that charts the last few months of Apple stock prices. You can find the code in 01AAPLPlotFirstPlot in the book's download.

AAPLot uses a simple web service from Yahoo.com to download historical stock data in comma-separated format. Type the following URL into a web browser <http://ichart.yahoo.com/table.csv?s=AAPL&a=3&b=19&c=2009&d=6&e=12&f=2009&g=d&ignore=.csv>. You should see text that looks something like this:

```
Date,Open,High,Low,Close,Volume,Adj Close
```

```
2009-06-18,136.11,138.00,135.59,135.88,15237600,135.88
2009-06-17,136.67,137.45,134.53,135.58,20377100,135.58
2009-06-16,136.66,138.47,136.10,136.35,18255100,136.35
2009-06-15,136.01,136.93,134.89,136.09,19276800,136.09
2009-06-12,138.81,139.10,136.04,136.97,20098500,136.97
2009-06-11,139.55,141.56,138.55,139.95,18719300,139.95
2009-06-10,142.28,142.35,138.30,140.25,24593700,140.25
2009-06-09,143.81,144.56,140.55,142.72,24152500,142.72
2009-06-08,143.82,144.23,139.43,143.85,33255400,143.85
```

Most of the work for AAPLot is concentrated in two objects: APYahooDataPuller, which downloads, parses, and stores the data from Yahoo.com, and AAPLotViewController,

which displays the data in a plot. Listing 9-1 shows the method from APYahooDataPuller that constructs a URL with a target start date and an end date.

Listing 9-1. *Constructing a URL String to Retrieve Stock Data from Yahoo.com*

```
-(NSString *)URL;
{
    unsigned int unitFlags = NSMonthCalendarUnit | NSDayCalendarUnit |
    NSYearCalendarUnit;

    NSCalendar *gregorian = [[NSCalendar alloc] \
                             initWithCalendarIdentifier:NSGregorianCalendar];

    NSDateComponents *compsStart = [gregorian components:unitFlags
    fromDate:targetStartDate];
    NSDateComponents *compsEnd = [gregorian components:unitFlags
    fromDate:targetEndDate];

    [gregorian release];

    NSString *url = [NSString
    stringWithFormat:@"http://ichart.yahoo.com/table.csv?s=%@", \
                                                             [self
    targetSymbol]];
    url = [url stringByAppendingFormat:@"a=%d&", [compsStart month]-1];
    url = [url stringByAppendingFormat:@"b=%d&", [compsStart day]];
    url = [url stringByAppendingFormat:@"c=%d&", [compsStart year]];

    url = [url stringByAppendingFormat:@"d=%d&", [compsEnd month]-1];
    url = [url stringByAppendingFormat:@"e=%d&", [compsEnd day]];
    url = [url stringByAppendingFormat:@"f=%d&", [compsEnd year]];
    url = [url stringByAppendingString:@"g=d&"];

    url = [url stringByAppendingString:@"ignore=.csv"];
    url = [url stringByAddingPercentEscapesUsingEncoding:NSUTF8StringEncoding];
    return url;
}
```

On application launch, the AAPLotViewController creates an APYahooDataPuller instance. It downloads and parses the CSV data and then calls the APYahooDataPullerDelegate method `dataPullerDidFinishFetch:` of the AAPLotViewController. The view controller then draws a plot into a layer of its view.

NOTE: I'll be using quite a lot of free and open source code in the examples for this chapter, all of which have licenses that allow for redistribution and commercial use. The plotting library used in AAPLot is from Core Plot, which is an impressive new project by a group of developers interested in graphing, charting, and plotting for the iPhone and the Mac. During WWDC 2009, Apple sponsored a code-a-thon to jump-start its development. One of its stated goals is to maintain a tight integration with Apple's core technologies like Core Animation, Core Data, and Cocoa Bindings. You can read more and download the latest code at <http://code.google.com/p/core-plot/>.

Build and run the AAPLot example. Depending on whether you have an Internet connection, you should see something that looks like one of the two images shown in Figure 9-1.



Figure 9-1. AAPLot with and without an Internet connection

It's already a modestly useful application. You might want to add some text to warn the user if there was a problem while trying to retrieve the graph from the Internet. You could also remove the empty graph from the UI when there isn't a connection, call it a day, and release. You certainly wouldn't be the first to be tempted to do that. Even Apple's Stocks application is not usable without an Internet connection, as shown in Figure 9-2.



Figure 9-2. Apple's Stocks application as it appears without an Internet connection

Storing Data Between Runs

My company's soon-to-be-released application, gogoDocs, is an online and offline reader for documents stored on the popular Google Docs service. When you start the application, it reads a list of the user's documents from a plist stored on disk the last time it ran and displays the information in a table view before attempting to fetch an updated feed. In earlier development versions, we didn't show the cached list on launch unless the user was offline, thinking that the user would rather see the new data that was soon to be downloaded. This proved to be a mistake. Beta testers were much happier with the application when we loaded the cached feed on launch, allowing the user to interact with the application while any new information downloads in the background. This taught us that stale information is often better than no information. A simple, and often big, usability win is to cache any downloaded information to disk and present that data to the user as a placeholder before attempting to download any new data. This will make the application appear to load faster because the user will not have to wait for new data to download before interaction with your app; they can view and possibly interact with the data that your application downloaded last time it was run. With data that can get stale fairly quickly, like stock prices, it is still better to show the user something rather than nothing, while perhaps signaling in an unobtrusive way that the data is a little stale.

To add caching logic to the AAPLot application, you will add a mechanism to save to and load from disk a given set of financial data. On launch, you'll show the cached data and then attempt to download new data. If you are able to get new data, you'll compare it to your old data, and you'll overwrite the old data and update the UI only if it's stale.

WRITING TO THE IPHONE'S NAND FLASH MEMORY

With the iPhone's NAND flash memory, writing is expensive both in terms of speed and in terms of hardware lifetime. It will eventually wear out with use. Apple recommends that you write to disk only when necessary. Since our application checks to see whether the data is stale, it is unlikely to download stock data more than once or twice per day, so you can reasonably store it to disk when it arrives. If your data were more often malleable, you might consider storing it only when the application closes or if you ran out of memory. In gogoDocs, we only download and write the updated feed if its last changed date is later than that of our cached information. This keeps the application from making unnecessary writes to the flash memory. Apple supplies a convenience method in your application's delegate where you can save data before the app closes:

```
-applicationWillTerminate:
```

Using Plists to Persist Data

Dumping NSDictionary objects to plists has proven to be a simple way of persisting small amounts data that I use often. It's especially useful when you have control over the server because you can have the server send you a plist that you can persist more or less directly. In an open source library called TouchEngine for communicating with Google App Engine that I'm working on with Noah Gift, a great Python programmer, we chose to use plists as our communication medium, and we automatically cache any plist that we get on the iPhone from the Google servers. We are automatically loading the cached data from the plists before we fetch new data. TouchEngine is available from Google Code at <http://code.google.com/p/touchengine/>.

In AAPLot, you are already using an array of NSDictionary objects to store your data within the APYahooDataPuller, so it is trivial to persist them because an NSDictionary or an NSArray can be written to disk as a property list as long as it contains only property list objects (instances of NSData, NSDate, NSNumber, NSString, NSArray, or NSDictionary). The NSDecimalNumbers are subclasses of NSNumber, so you can store those with one caveat: they're going to get converted to floating point first, which will reduce their precision. For demonstration purposes, I'll just round them when reading them back in. The precision you lose might cause a graph line to move by a pixel, which isn't a big deal for this application. Let's add some caching methods to APYahooDataPuller.

First you'll add a method called `plistRep` that returns a dictionary representation of the APYahooDataPuller's data. Then you'll add a method that writes that dictionary to a file, calling the built-in NSDictionary `writeToFile:atomically:` method. You should also take this opportunity to further modify APYahooDataPuller to better model your new strategy. Since you are caching the `startDate` and `endDate` values to disk and will need them for comparison later, you will want to add a few instance variables to track the dates you want from the server and also the symbol you're looking for, which may be different from those you're loading from the cache, and you'll also want to change the designated initializer accordingly. You should change the behavior with respect to notifying the delegate. Since you are caching financial data, it's possible that the target `startDate`, `endDate`, and `symbol` will match that which is already cached. If that is the case, you

won't need to reload the graph, and you should probably not even notify your delegate. You'll change the interface with the delegate so that you notify only when the financial data changes as a result of a fetch.

Listing 9-2 shows the code for inserting the instance variables of the APYahooDataPuller object into a dictionary and then writing that dictionary to a plist on disk, which you can load the next time the user runs the application.

Listing 9-2. *Inserting Instance Variable Values into an NSDictionary Object and Writing It to a Plist on Disk*

```
- (NSDictionary *)plistRep
{
    NSMutableDictionary *rep = [NSMutableDictionary dictionaryWithCapacity:7];
    [rep setObject:[self symbol] forKey:@"symbol"];
    [rep setObject:[self startDate] forKey:@"startDate"];
    [rep setObject:[self endDate] forKey:@"endDate"];
    [rep setObject:[self overallHigh] forKey:@"overallHigh"];
    [rep setObject:[self overallLow] forKey:@"overallLow"];
    [rep setObject:[self financialData] forKey:@"financialData"];
    return [NSDictionary dictionaryWithDictionary:rep];
}

- (BOOL)writeToFile:(NSString *)path atomically:(BOOL)flag;
{
    NSLog(@"writeToFile:%@", path);
    BOOL success = [[self plistRep] writeToFile:path atomically:flag];
    return success;
}
```

Saving Data to the iPhone Application Sandbox

When your application is installed on the iPhone or the iPhone Simulator, its sandbox includes several directories. Take a look at Figure 9-3. The Library directory includes a Preferences directory where preferences are stored as plist files. The Caches directory stores cached data between runs but is not backed up when iTunes connects to the phone. The tmp directory holds temporary files while the application is running and is cleared between runs. The Documents directory contains user data. It is backed up by iTunes during a sync and restored during a restore from backup. I tend to use the Documents directory for cached data that isn't too big, like plist files, as well as data the user generates because I prefer the user to be able to use my and my clients' applications directly after a restore from backup. The iTunes sync time for your application will increase the more information you store in the Documents directory, so try to avoid caching really large files there if you can help it.

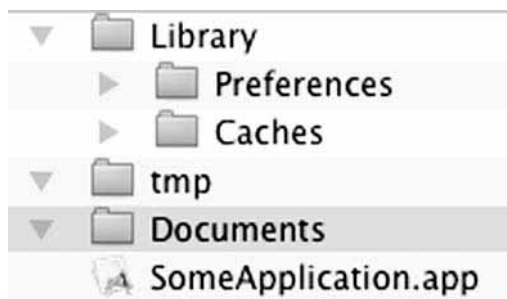


Figure 9-3. *iPhone application directory structure*

You can get the path to the Documents directory using this code snippet:

```
NSArray *paths = NSSearchPathForDirectoriesInDomains(NSDocumentDirectory,  
NSUserDomainMask, YES);  
NSString *documentsDirectory = [paths objectAtIndex:0];
```

For AAPLot, you append AAPL.plist to the path stored in the documentsDirectory variable when you store the plist data file.

Build and run the application while connected to the Internet. It should look about the same as before. Now disable your Internet connection and run the application again. The graph should render just as it did before using the data that was cached to disk on the first run. If the graph does not draw when you run the application without an Internet connection, you're likely reinstalling the application and overwriting the Documents folder with an empty one each time you install it. Instead of running the app using Xcode's build and run, try running the application by touching or clicking it on the phone or in the simulator without reinstalling. If your application stores more critical data, perhaps business documents, your users will appreciate having their content available to them at any time, regardless of their Internet connection status.

Shipping AAPLot with Placeholder Data

You never get a second chance to make a first impression. If a user downloads your application on the App Store and then finds himself without an Internet connection the first time he uses it, having nothing to look at can be disappointing. That user may never run your application again. Many applications would benefit from having some kind of default local data, even if it is just something to show the user what it will look like when they are able to get fresh data. One fantastic application that I use often, FileMagnet from Magnetism Studios (<http://www.magnetismstudios.com/FileMagnet>), is a file viewer that makes it simple to synchronize files from your computer to your iPhone for viewing. They ship the application and each update with a document outlining what's new in the application. This allows a new user to experience the application in action before importing any documents while at the same time allows the veteran user see what new features are available. It's a very nice touch.

To ship a default version of the AAPL.plist with your application, you will first need to retrieve one from the simulator. The iPhone Simulator loads its library of applications and data from your home directory in `~/Library/Application Support/iPhone Simulator/User/Applications/`. Each application is housed in a directory named with a UUID. The easiest way to find your AAPL.plist is to empty this directory, build and run your application, and then retrieve it from the newly created directory. The iPhone Simulator will empty the directory for you. Open the iPhone Simulator, and then select Reset Content and Settings from the iPhone Simulator menu.

Make sure your Internet connection is live, and build and run the application in the simulator. You'll find AAPL.plist in the `~/Library/Application Support/iPhone Simulator/User/Applications/SOMELONGUUID/Documents/` directory. Copy it into the AAPLot code directory. Now add it as a resource in Xcode. You can set Reference Type to Default. Make sure that Add To Target is also selected so Xcode knows to copy it during the build. See Figure 9-4.

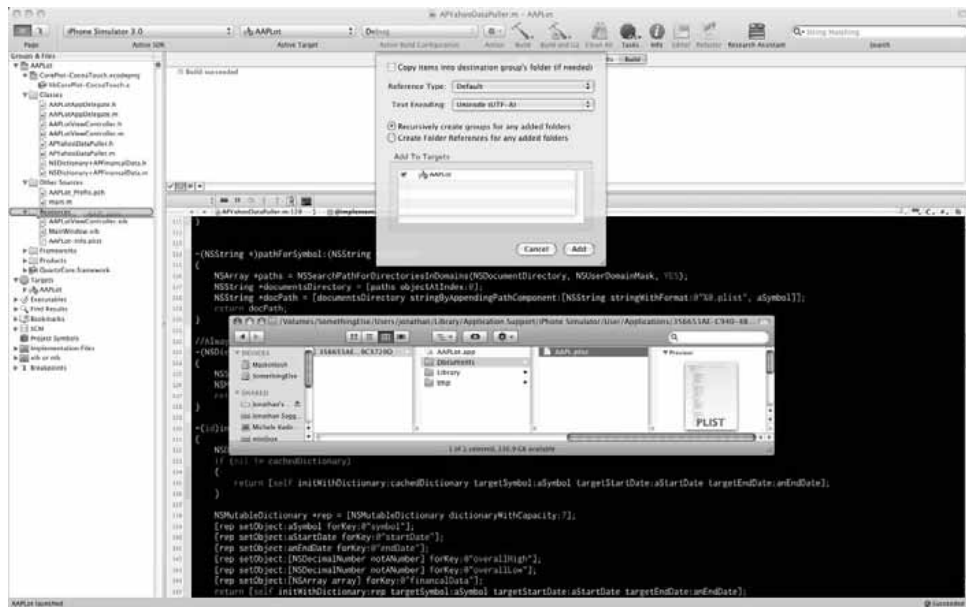


Figure 9-4. Adding AAPL.plist to AAPLot in Xcode

Now you need to write a method that checks to see whether AAPL.plist is in the Documents directory, and if it is not, you should instead load the plot from the application's resources folder.

```
-(NSString *)faultTolerantPathForSymbol:(NSString *)aSymbol
{
    NSString *docPath = [self pathForSymbol:aSymbol];
    if (![NSFileManager defaultManager] fileExistsAtPath:docPath) {
        //if there isn't one in the user's documents directory, see if we ship with this
        data
        docPath = [[[NSBundle mainBundle] resourcePath] \
```

```

        stringByAppendingPathComponent:[NSString
stringWithFormat:@"%@.plist", aSymbol]];
    }
    return docPath;
}

-(NSDictionary *)dictionaryForSymbol:(NSString *)aSymbol
{
    NSString *path = [self faultTolerantPathForSymbol:aSymbol];
    NSMutableDictionary *localPlistDict = [NSMutableDictionary
dictionaryWithContentsOfFile:path];
    return localPlistDict;
}

```

Remove all applications as before from the simulator so you can see how the application behaves when it is used for the first time. Now disable your Internet connection again. Build and run. The default AAPL.plist should load even though the application is freshly installed with no previously fetched data. You can find the version of AAPLOT that includes all of these caching changes in Examples/03AAPLOTDefaultData of the book's download.

In a shipping application, indicating to the user that the data they're seeing is stale and warning them that the application would really benefit from an Internet connection is a good idea. See Apple's Reachability sample code at <http://developer.apple.com/iphone/library/samplecode/Reachability/index.html> for information on how to test for the availability of a server on the Internet. See also the Human Interface Guidelines for the iPhone available at <http://developer.apple.com/iphone/library/documentation/userexperience/conceptual/mobilehig/Introduction/Introduction.html>.

Extending the App for Multiple Stock Graphs: StockPlot

Now I'll show how to reuse some of the objects from the AAPLOT application in an app called StockPlot that loads a whole bunch of stocks into a table that the user can select to push a graph onto the screen. Things get rather more complicated when there is a lot of data to download and store. The gogoDocs application I mentioned before downloads and caches a list of preferred documents each time the user is connected to the Internet to make sure important documents are always available offline. When we decided to add this multiple document download feature to the application, which was previously downloading one document at a time on demand, the UI would grind to a halt.

The earlier strategy of download, then parse, then cache, and then display from the AAPLOT application might not hold up when you try it with a lot of stocks at the same time. Let's also see what happens when you try to load graphs in response to user input. You can find StockPlot in Examples/04StockPlotConcurrentDownloads.

StockPlot will ship with Yahoo, Microsoft, Google, and Apple stock data and will attempt to download more than a dozen other technology companies' prices on launch. The

RootViewController of the project handles table view loading and APYahooDataPullerDelegate duties. It loads a summary of whatever data it can find in its array of APYahooDataPuller objects, which it creates at launch. Each APYahooDataPuller object in the array will continue to act just like it did in AAPLot by loading from disk, downloading, and notifying of changed data; you'll just have a large number of them. The RootViewController object also has a small amount of code to limit the number of concurrent downloads to three connections at a time. Build and run it on the simulator. It should look like Figure 9-5. If you're online (and you don't blink), you will see the little exclamation point cautionary icons that you're using to indicate stale data in the table cells replaced by progress indicators while the corresponding APYahooDataPuller object downloads; then they disappear once fully loaded. If you click a table cell, the now-familiar graph is rendered and animated onto the screen through the canonical UINavigationController view controller–pushing methods.

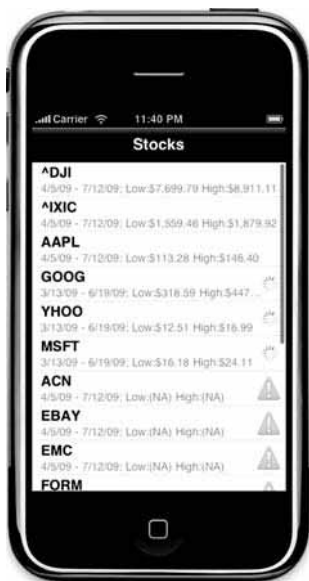


Figure 9-5. *StockPlot loading data from Yahoo.com*

Now install and run it on your device. It seems like it's pretty slow to download, huh? It's nothing like the simulator experience. The user interface even freezes in fits and starts while the data comes in; you can't even scroll the table view most of the time. Once everything is downloaded, the interface is really responsive. You could choose to download only the data you need for the table view on launch, but that would only push the lack of responsiveness somewhere else, which would bring you dangerously close to death by 1,000 paper cuts. It would probably take quite a while to load the data for a given graph on demand. You should also try to build and run a release configuration to see whether perhaps the sluggish UI has anything to do with a certain lack of compiler optimizations. Nope. Let's run StockPlot in the Shark profiler to see whether we can find out what's slowing things down.

During the development of gogoDocs, we had a similar problem. On the device, the UI would freeze periodically. After running the application through Shark, we saw that the parsing of access control (document sharing and outside access) XML data from Google was freezing the UI intermittently because that was all running on the main thread and getting in the way of UI drawing. There is nothing more disheartening than watching your previously responsive app suddenly start to stutter while scrolling through a table view. We ended up moving the ACL downloading and parsing into a background thread. Now the UI is very smooth.

Shark is Apple's profiler. Attach it to a running process, and it takes a snapshot of what portion of your binary is running at regular intervals. Shark shows you a sort of weighted statistical table of how many times through a given method or line of code it counted. The more times it sampled your code in a given area, the more time your code spends in that area. You should always run Shark on a release build of your application because you will want to profile the compiler-optimized code with which your application will ship. There is one problem. The default settings for release also strip debugging symbols from your binary, which makes Shark look more like a hexadecimal puzzle game for those who can solve Rubik's Cube, of which I am certainly not one. Now copy the release build configuration into one called Profiler by opening the Project menu in Xcode and clicking Edit Project Settings; then select the Configurations tab and duplicate the Release configuration.

Then in the Build tab, deselect the boxes for stripping debug symbols, as shown in Figure 9-6.

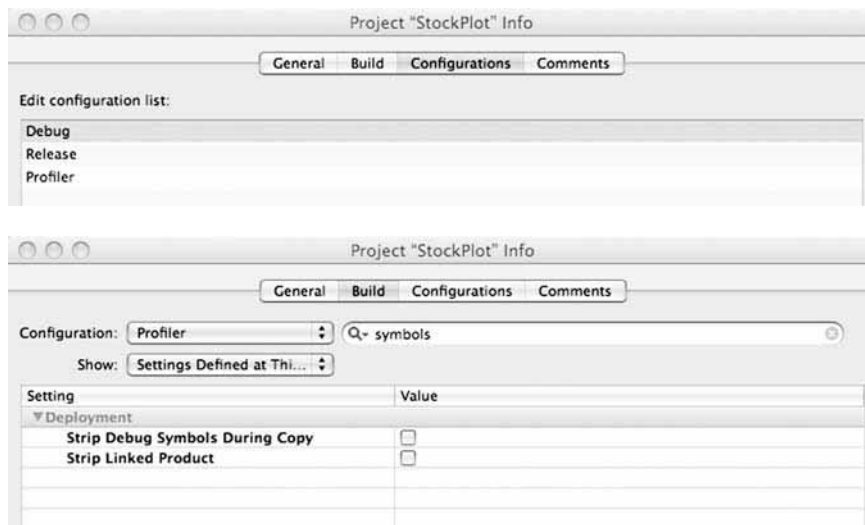


Figure 9-6. Removing symbol stripping from the StockPlot build settings

You have one thing left to do before you can test the downloading problem. Once your application is running, it takes a little while to attach the profiler. To test the problematical code, you need a way of attaching Shark at the very start of the application run. Although it might seem easy to drop a breakpoint in gdb in your `main()`

function, I have had some trouble getting Shark to connect while gdb is also attached. Instead, you'll drop a ten-second `sleep()` call in `applicationDidFinishLaunching`. That should give you enough time to attach Shark.

You can usually find `Shark.app` in `/Developer/Applications/Performance Tools/`. Run it, and select `Network/iPhone Profiling` from the Sampling menu. Delete the copy of `StockPlot` with cached data from your phone by using the Xcode organizer or directly on the iPhone. Connect your iPhone to your computer, and build and run the application. Once it's running (and sleeping), you can select the check mark in the menu next to the name of your iPhone, select `TimeProfile (WTF)` from the Config drop-down, and select `StockPlot` from the Target drop-down. As soon as you see log messages indicating download activity, hit the Start button; once the messages stop, hit the Stop button. If you are a coffee drinker, now is a good time to go make a cup. This part takes a little time because a lot of the processing that Shark needs in order to figure out what happened during the profiling run is actually performed on the device itself.

After you finish your coffee and once Shark and your iPhone are finished, you might see a window with all kinds of hexadecimal jibberish that I promised wouldn't happen. If so, you will need to symbolicate the time profile by telling Shark where the symbol-rich binary is located on your filesystem. Select `File > Symbolicate`, and then navigate to the build directory corresponding to your Profiler build settings, as shown in Figure 9-7. Make sure you see "type: ARM" on the window when you select it. Now Shark should have familiar method names. Poking around in the trace, you can see that most of the work is being done in parsing the comma-separated strings and writing the plists to disk. That makes sense. You're using an asynchronous download, so that shouldn't freeze your UI, but the string parsing and caching to disk blocks the main thread. See Figure 9-8.

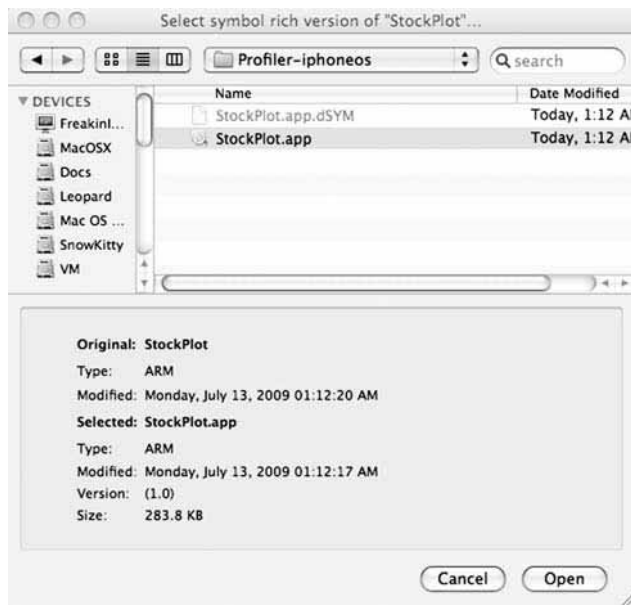


Figure 9-7. Pointing Shark to a symbolicated binary

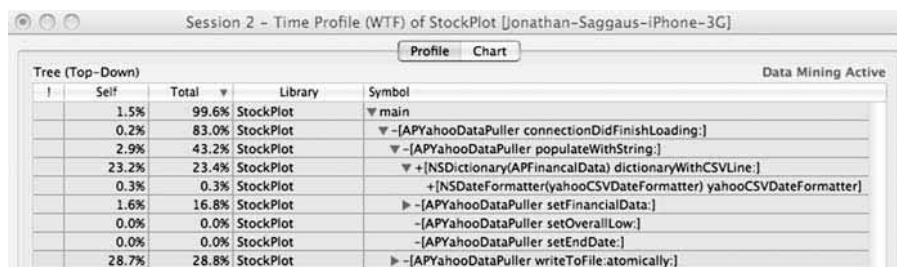


Figure 9-8. Shark profile showing the data parsing and plist writing bottlenecks

Even if you ship with 20 plists (which I probably would for this application), they'll almost definitely be stale once the application gets into users' hands. You don't want your application to be this unresponsive the first time it is run. What can you do about this? You have several options. At the moment, you're downloading and parsing all three months worth of data from Yahoo, because that's the easiest thing to do. You could figure out how much data you already have on disk and download only the missing data. You are also spending a fair amount of time converting `NSNumber`s to the `NSDecimalNumbers` you need for Core Plot. You could change Core Plot to accept `NSNumber`s, or you could change your storage to `CoreData`, which retrieves `NSDecimalNumbers` without conversion. The problem with these optimizations, some of which you may choose to do before shipping, is that they will all incur unpredictable amounts of overhead on the main thread. Thus, you would have to test a lot of use cases. It may also prove difficult to predict just how much data you'll need. If your user uses your application often enough to pull down small chunks (in this case, fewer days) of data, which is not guaranteed, you might do well to avoid downloading duplicate data. You might also want to allow the user to add stocks to plot, which would definitely require a lot of parsing the first time the stock data is downloaded; it also adds yet another stock to the queue on application launch. Perhaps you would do well to try to pull the processing off of the main thread so you can unblock the user interface once and for all, thus freeing yourself from all of these problems at once.

Concurrency

*"Cause it's gonna be the future soon. And I won't always be this way.
When the things that make me weak and strange get engineered away."*

—Lyrics for *The Future Soon* by Jonathan Coulton

Wait a tick. Did I just suggest multithreading?

OK, threading is hard, but the engineers at Apple and elsewhere keep making it easier for us. We have all of these cores on our desktops because the hardware engineers keep slicing silicon, so concurrency keeps getting more and more important. Who

knows? Perhaps one day we'll all be walking around with multicore processors in our phones. If so, you'll be ready to write software for them.

NSOperation, NSOperationQueue, and Blocks

NSOperationQueue and NSOperation remove much of the pain of multithreading. NSOperationQueue is a sort of thread pool that maintains an array of pending NSOperation objects that it schedules to run in a background thread based on a number of factors from system hardware to system state to the relative priority of a given NSOperation. You can even declare one NSOperation dependent on the completion of another. You normally subclass NSOperation to override one method, `main`, which is where you put the work you want on a background thread. It's called when the operation is run. The only things we as programmers have to be wary of in this situation are the usual data access caveats. Try not to mutate data at the same time you're reading it. There are tools for this, too. We can use the various permutations of `performSelectorOnMainThread:`, and `@synchronized()` directives are useful, too. Before you dig in, I highly recommend reading Apple's concurrency document at <http://developer.apple.com/Cocoa/managingconcurrency.html>.

There is a helpful tool in other languages for this kind of problem called *blocks*. Blocks are another name for closures, with which you may have familiarity with from using Ruby, LISP, Python, Smalltalk, and others. They're like function pointers that take a (usually const) snapshot of their local stack variables so you can run them later with the information you shove in them now. They're little portable units of computation that carry their state around and are extraordinarily useful with concurrent operations. Because they have a snapshot of their state, they're easier to deal with in a concurrent environment. Useful though they may be, they don't officially exist yet. They're being added to Objective-C by the folks who are bringing us the open source Clang and LLVM projects. (See <http://lists.cs.uiuc.edu/pipermail/cfe-dev/2008-August/002670.html> and <http://www.macresearch.org/cocoa-scientists-part-xxvii-getting-closure-objective-c>.) There is no guarantee, though it seems likely, that Apple will bring them to the iPhone.

These additions to the Objective-C language and runtime are free and open source, and they've been implemented in GCC 4.2, so it is actually quite possible to backport them to the iPhone, so of course they have been. Plausible Blocks from Plausible Labs is available at <http://code.google.com/p/plblocks/> and is, as of this writing, shipping its second beta of a gingerly patched version of the standard, stable GCC 4.2 compiler that ships with the OS X Leopard (10.5) and iPhone software development kits. I have found it to be very stable, and it works with both iPhone OS 3.0 and 2.2.1 targets. There is some example code for their use on the primary author's GitHub repository available at http://github.com/landonf/block_samples/tree/master. Next you'll install the Plausible Blocks compiler and add its static framework to your project so you can easily place your downloading, parsing, and saving code in a block to be executed by an NSOperation to be scheduled by an NSOperationQueue (in the house that Jack built). If or when Apple does add blocks support to the iPhone, switching from Plausible Blocks will be simple. You'll revert to Apple's compiler and remove the Plausible blocks framework

from your project. All of the things that make your application weak and strange are being gradually engineered away, and you're even using future technology!

Installing the Plausible Blocks Compiler and Adding It to the Project

First, download the latest disk image of the Plausible Blocks compiler and frameworks from <http://code.google.com/p/plblocks/downloads/list>. Mount the disk image, and run the included package. This installs the patched compiler as an Xcode plug-in.

Now copy the iPhone Runtime folder, which includes the static framework you'll need to link against, into the StockPlot project. Double-click the StockPlot target, select the General tab, and click the plus (+) button in the lower-left corner of the window to add a new linked library. Click Add Other in the resulting sheet, and navigate to and select the framework for addition, as shown in Figure 9-9.

Now you need to tell Xcode to use the special compiler. Double-click the StockPlot target to open the Build Settings window. Select the Build tab. Select All Configurations from the upper-left drop-down. Now select the GCC 4.2 (Plausible Blocks) compiler, as shown in Figure 9-10. You now have blocks support.

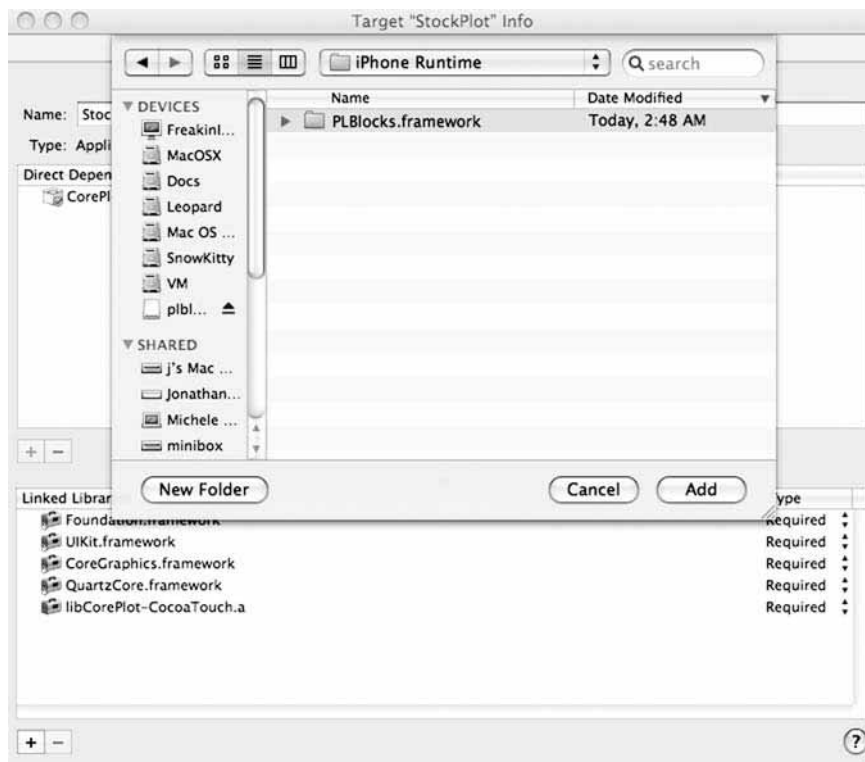


Figure 9-9. Adding the *PLBlocks.framework* to the *StockPlot* project

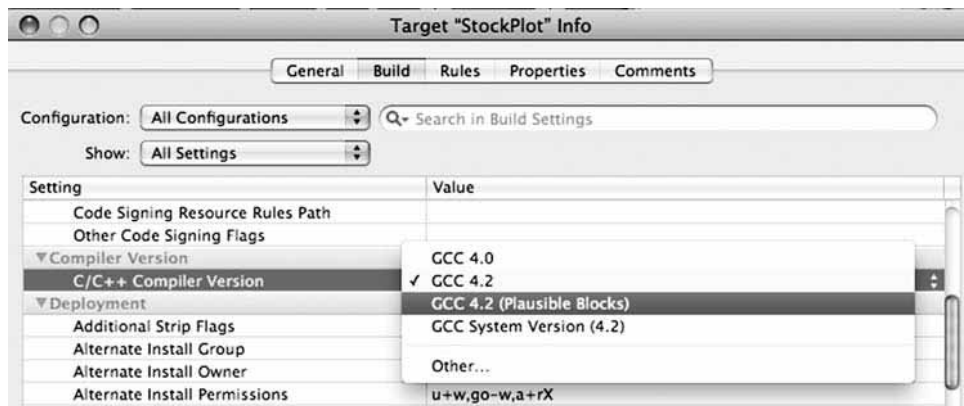


Figure 9-10. Switching to the Plausible Blocks patched version of GCC

You'll use some convenience categories and objects from the Plausible Blocks sample code mentioned earlier. They're included with the sample code in `Examples/05StockPlotParallelDownloads` in files called `NSThread+PLBlocks.h/m` and `NSOperationQueue+PLBlocks.h/m`. Add them to the StockPlot project.

Using Blocks, NSOperation, and NSOperationQueue in StockPlot

To get asynchronous downloading, parsing, and saving, the first thing you need to do is make something synchronous. Go figure. The downloading code is using `NSURLConnection` to download the data asynchronously from Yahoo. `NSURLConnection` doesn't like to be launched asynchronously from any thread other than the main thread because that would be silly. This isn't a big deal, because you're going to place all downloading, parsing, and saving in a background thread using the `NSOperation/NSOperationQueue` objects. This has the added benefit of making the downloading code simpler. Instead of asynchronously adding data to an `NSMutableArray` object and defining a bunch of `NSURLConnectionDelegate` methods, you need only call the `NSURLConnection sendSynchronousRequest:returningResponse:error:` method. It blocks execution while downloading and can be run from a nonmain thread, which is exactly what you want. Every time you call a delegate method from the background thread, you make sure that the delegate gets called on the main thread. Usually, you would use the `performSelectorOnMainThread:... family of calls`, but it's easier to wrap them in a block and have the new category on `NSThread` execute the block on the main thread. Listing 9-3 shows the new "blockified" `fetchIfNeeded` method.

Listing 9-3. *APYahooDataPuller.m*

```
-(void)fetchIfNeeded
{
    if ( self.loadingData ) return;

    //Check to see if cached data is stale
    if ([self staleData])
```

```

{
    self.loadingData = YES;
    NSString *urlString = [self URL];
    NSLog(@"Fetching URL %@", urlString);
    NSURL *url = [NSURL URLWithString:urlString];
    NSURLRequest *theRequest=[NSURLRequest requestWithURL:url

cachePolicy:NSURLRequestUseProtocolCachePolicy
                                timeoutInterval:60.0];
    // create the connection with the request
    // and start loading the data
    NSURLResponse *theResponse;
    NSError *theError;
    [self downloadWillStart];
    self.receivedData = [NSURLConnection sendSynchronousRequest:theRequest
                                        returningResponse:&theResponse
                                        error:&theError];

    if(theError)
    {
        self.loadingData = NO;
        self.receivedData = nil;
        NSLog(@"err = %@", [theError localizedDescription]);
        [[NSThread mainThread] pl_performBlock: ^{
            if(delegate && [delegate
respondsToSelector:@selector(dataPuller:downloadDidFailWithError:)])
            {
                [delegate performSelector:\
                    @selector(dataPuller:downloadDidFailWithError:)
                    withObject:self
                    withObject:theError];
            }
        }];
        [self connectionEnded];
    }
    else
    {
        self.loadingData = NO;
        NSString *csv = [[NSString alloc] initWithData:self.receivedData
encoding:NSUTF8StringEncoding];
        [self populateWithString:csv];
        [csv release];
        self.receivedData = nil;
        [self writeToFile:[self pathForSymbol:self.symbol] atomically:NO];
        [self connectionEnded];
    }
}
}
}

```

This method is called from the RootViewController's `updateDownloadStatus` method from within a `pl_addOperationWithBlock` method that has been added to `NSOperationQueue`. This method adds a `PLBlockOperation` to the queue and schedules it for execution. The `NSOperation` object subclass `PLBlockOperation` that gets instantiated here copies the block you pass it into an ivar and simply executes it in its main method. (Blocks are also Objective-C objects, but they must be copied rather than retained.) Since all the stack variables are copied into the block, you don't need to worry if they change or go out of scope before the block is called.

```

-(void)updateDownloadStatus
{
    while ([stocksToDownload count])
    {
        APYahooDataPuller *dp = [stocksToDownload objectAtIndex:0];
        NSOperationQueue *q = [(StockPlotAppDelegate *) [[UIApplication
sharedApplication] delegate]
globalQ];
        [q pl_addOperationWithBlock: ^{
            [dp fetchIfNeeded];
        }];
        NSUInteger idx = [stocks indexOfObject:dp];
        NSUInteger section = 0;
        NSIndexPath *path = [NSIndexPath indexPathForRow:idx inSection:section];
        UITableViewCell *cell = [self.tableView cellForRowAtIndexPath:path];
        if(nil != cell)
            [self setupCell:cell forStockAtIndex:idx];
        [stocksToDownload removeObject:dp];
    }
}

```

Uninstall, build, and run on the device. `NSOperationQueue` tends to be conservative on the iPhone, so you'll probably see stock information downloaded one symbol at a time; the application will remain responsive throughout.

Just for fun, let's reinstall and run it through Shark again. If you've deleted it, add that temporary call to `sleep()` as well.

You can see in Figure 9-11 that the application didn't really run any faster; you've just parallelized it. Multithreading isn't so painful after all. Welcome to the future.

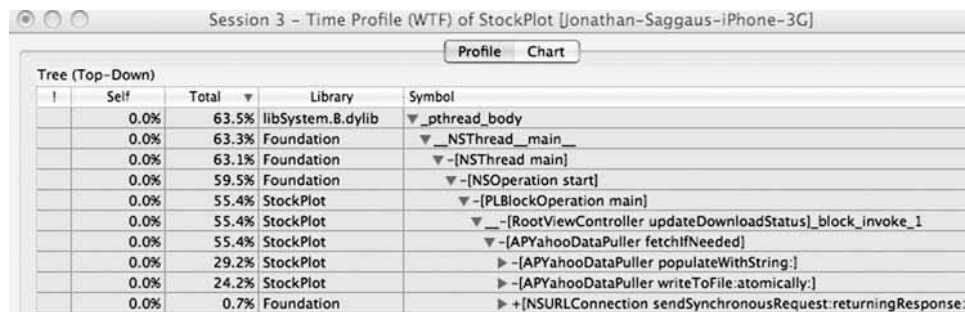


Figure 9-11. This Shark profile shows *StockPlot* running the *PLBlockOperation* in the background.

NOTE: Apple has not officially announced any intention to bring blocks to the iPhone, though it's a fair bet that Apple will do so once blocks are added to the desktop runtime and compiler collection. You should very thoroughly test any application using a nonstandard compiler and be prepared for things to break in spectacular and unexpected ways. That said, Plausible Blocks appears well on its way to release-level stability.

Displaying Large Amounts of Data Efficiently

How easily the iPhone UI can be brought to its knees by performing something as seemingly simple as downloading, processing, and caching data to disk! Now you're going to really make it hurt by throwing it an application that has to work very hard to draw anything at all. One of my clients has an application that pulls potentially dozens of high-resolution images from a database server into an image-browsing view not unlike that of Apple's Photos application. To load *something* into the image viewer quickly, we prefetch a set of low-resolution thumbnail images from the database and scale them to fit the screen. As the user thumbs through the images, the application downloads the current high-resolution image and those to the left and right thereof. As each large image arrives, the application replaces the low-resolution image on screen with the high-resolution image. For the application to remain responsive, image decoding and drawing are handled in a background thread by pushing the images to off-screen `CGContextRefs`. To solve a similar problem in this section, you'll examine a project for drawing a vertical succession of very large images into a zoomable scroll view. So that you might encounter some of the difficulties inherent in dealing with large amounts of data, you'll add an admittedly somewhat contrived requirement: the images cannot be presliced, thumbnailed, or otherwise massaged outside of the device. All drawing code must use the original very large PNG images shown in Figure 9-12 bundled with the application. If you can make this example perform reasonably well, you'll have a reusable framework for drawing any processor-intensive tiled scroll view while maintaining UI responsiveness.

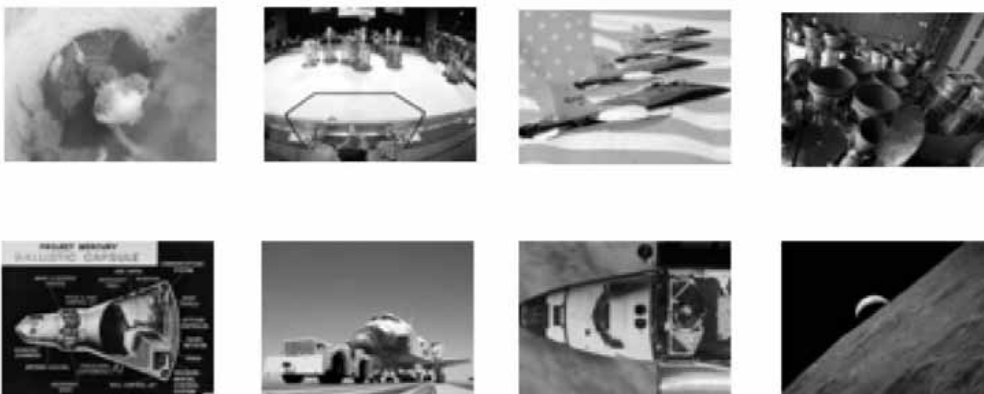


Figure 9-12. Images for *BigViewThing.app* © NASA

You'll begin with a modified version of Apple's `ScrollViewSuite` sample code available at <http://developer.apple.com/iphone/library/samplecode/ScrollViewSuite/index.html> called *BigViewThing*. The original Apple sample is designed to draw view tiles that are chunks of a larger image; it shows how to reuse view objects in two dimensions similar to the way the `UITableView` dequeues and enqueues rows in one dimension. In the case of Apple's sample, the smaller image chunks are meant to ship with the application.

BigViewThing is already partially implemented as a result of being derived from this sample code. It handles double-tap to zoom, suspends tile redraws when the user is interacting with the view, and draws only on-screen tiles. It's in the `Examples/06BigViewThingOriginal` directory of the sample code. Build it and run. There are quite a number of large images in it, so it will take a while to copy over to the device.

Once you have it running on your phone, you'll notice a few issues. Whenever a new tile comes on the screen, it takes a while to render. The image doesn't redraw at higher resolution when you zoom. It remains grainy. Let's profile it in Shark to see what is happening. There is no need to add a `sleep()` to this application because the performance problems appear throughout rather than just on startup. Start the application, and attach Shark. Remember, the longer you sample, the longer you will wait for results, so scroll around enough to get it to draw just a couple of images.

As you can see in Figure 9-13, almost all the application's execution time is being used in decompressing and drawing the PNG images. Our goal with this demonstration application is to simulate what happens when drawing very heavy, data-intensive views. You never know when a user is going to try to load a giant document or image into your application. Some developers, myself included, have run into this problem using Apple's `UIWebView`. It was designed to render small e-mail attachments in various formats in the Mail application and to render web content. Several document reader applications fail when the user tries to load a large document because they are trying to leverage `UIWebView` to draw heavy content. It clearly isn't designed for such content. We ended up writing our own memory- and CPU-optimized document view for some common files in `gogoDocs` because the `UIWebView` could not handle drawing some of the larger ones.

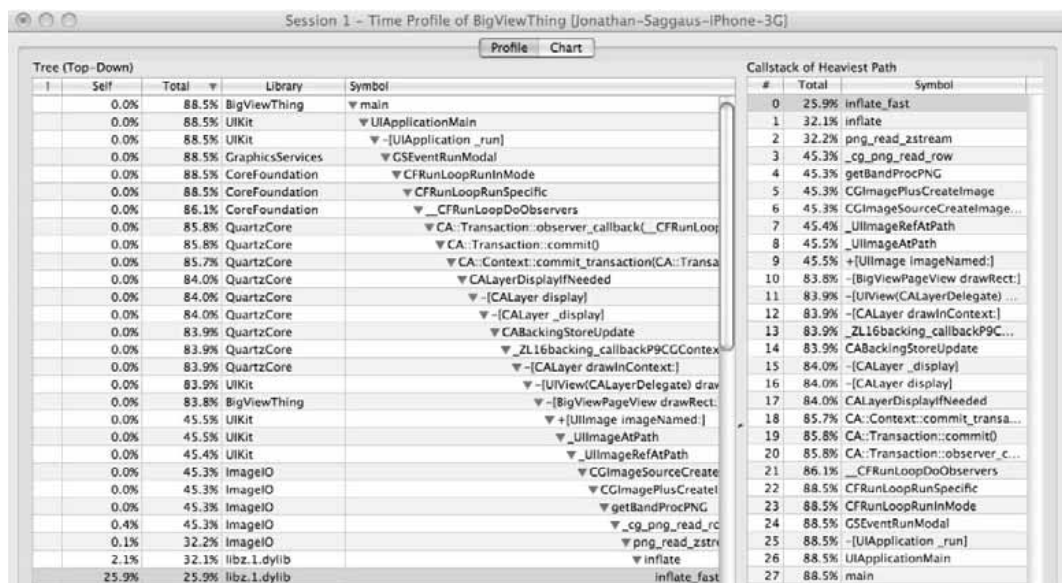


Figure 9-13. This Shark profile shows `-[BigViewPageView drawrect:]` taking up most of the execution time.

Zooming a UIScrollView

One thing that `UIWebView` performs very well is its sharp redrawing of zoomed content after a zoom is finished. In `BigViewThing`, you're currently allowing the scroll view to zoom for you and leaving the content alone when zooming is finished. This default behavior results in an unpleasant grainy appearance because the `UIScrollView` that you use to host the content simply applies a scaling affine transform to the content view. It's also expanding or contracting its own content size relative to the new drawn size of the overall view. `UIScrollView` does this for performance reasons. If it takes three seconds (and it does take that long right now in our application) to draw a view into a given square of pixels, imagine what it might look like to animate resizing by redrawing. One-third of a frame per second is subpar to say the least.

Search the Internet for *UIScrollView zooming reset resolution*, and you'll find a lot of developers pulling their hair out trying to get this to look right. A little caveman `NSLog` experimentation to figure out what the `UIScrollView` is really doing can reveal what's happening under the covers when you, say, pinch to zoom or directly set the `zoomScale` property of a `UIScrollView`.

UIScrollView Zooming Under the Covers

When the user attempts a zoom, `UIScrollView` first checks to see whether `minimumZoomScale` and `maximumZoomScale` are not equal to one another. It also checks the current `zoomScale` to see whether it can zoom. If parameters allow for zooming, `UIScrollView` then asks the `UIScrollViewDelegate` for a view to scale during the zoom with the `viewForZoomingInScrollView:` method call. You return the content view in the `BigViewThing` project. As the zoom scale changes, the `UIScrollView` does two things:

- It sets an affine transform on the view it is zooming to scale it up or down without redrawing. It's a "square" transform that maintains aspect ratio, so there is no distortion.
- It resets its own `contentSize`, `contentOffset`, and `zoomScale` so as to hold the content in place relative to the point about which it is zooming (in the case of pinching, that point was halfway between your fingers when you put them down). See Figure 9-14.

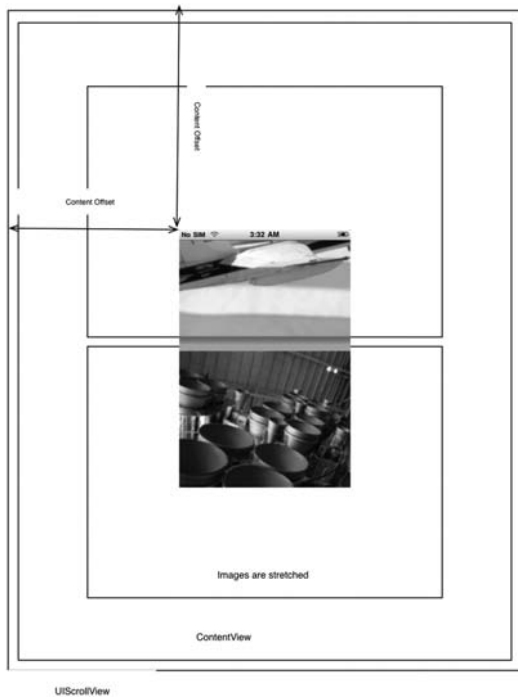


Figure 9-14. *The UIScrollView default zoom simply stretches the ContentView, leaving pixelated images.*

If the zoom was performed with a pinch gesture or through the `setZoomScale:animated:` methods, it calls `scrollViewDidEndZooming:withView:atScale:` on its delegate when the zooming ends. However, it does not call this delegate method if the `animated:` argument was NO because the zoom is set instantly when you call the method. The UIScrollView assumes that you know that it finishes zooming right away in that case. After zooming, the UIScrollView leaves the affine transform on the view, and it leaves the stretched `contentSize`, `contentOffset`, and `zoomScale` in place, which is why the view seems grainy. It's still being stretched when you zoom.

Resetting Resolution in a UIScrollView after a Zoom Operation

Armed with knowledge of some of the internal workings of UIScrollView, you can now reset the drawing after a zoom by implementing and calling an `updateResolution` method when zoom finishes. Updating resolution on the content of a UIScrollView after zoom can be tricky because the state of the UIScrollView is a little awkward and counterintuitive at that point. There is an affine transform scaling the content. The `zoomScale`, `contentSize`, view frame, and `contentOffset` are all set such that they take into account the zoom scale and the transformation on the view. Because the view is being resized by a transform, its frame wasn't changed during zooming. We'll reset the resolution of the zoomed view by removing the stretching affine transform and resizing

its frame so that the number of pixels that it occupies is equal to the number of pixels that are being drawn. You'll need to take care to reset the underlying parameters of the scroll view so as to reposition the view that now has a larger frame so that it appears to simply sharpen in place. The following is a step-by-step algorithm for resetting resolution after a zoom:

1. Take a snapshot of the current (scaled) `contentSize` and `contentOffset`.
2. Take a snapshot of the current (unscaled) content view's frame size; it's being scaled by an affine transform, so its actual frame size is the same as it was before zooming.
3. Take a snapshot of the current minimum and maximum zoom scales.
4. If your scroll view is its own delegate as it is in `BigViewThing`, call `super` to set the minimum and maximum zoom scales both to 1.0 because setting zoom on `self` will eventually call `updateResolution` again; infinite recursion is so last year.
5. Set the current zoom scale to 1.0, which will rescale the content size internally back to the size of the content view, and reset the affine transform on the content view to unity.
6. Calculate new content offset by scaling the stretched/zoomed offset you took a snapshot of in step 1. You want the new content to appear in the same place in the scroll view:
7. `newContentOffset.x *= (oldContentSize.width / contentViewSize.width);`
8. `newContentOffset.y *= (oldContentSize.height / contentViewSize.height);`
9. Divide the old minimum and maximum `zoomScale` by the new zoom scale. This scales the original minimum and maximum zooms relative to the new content size. If minimum zoom were 1.0 and maximum zoom were 2.0, when the user zooms to 2.0 and I reset, my new minimum zoom would be .5, and my new maximum zoom would be 1.0.
10. Set the content view's `frame.size` to the `contentSize` you took a snapshot of in step 1. This is a little counterintuitive. The numeric values of the new content size are being reset to the same values as those the scroll view calculated for the transformed zoomed view but are now reinterpreted in a new overall scroll view frame. Essentially, Apple already did the math for you, so you're reusing its values in a new context.
11. Set the scroll view's `contentSize` to the scaled `contentSize` you took a snapshot of in step 1. This stretches the overall size of the view to match the new zoom level (but without any affine transform applied).

12. Call the `setNeedsLayout` method on the scroll view. This will cause `layoutSubviews` to be called where you can reset the content view's internal subview geometry.

The following is an implementation of the previous steps that you'll add to the `BigViewScrollView`. You'll call it whenever zooming finishes.

```
- (void)updateResolution {
    //LogMethod();
    isdblTapZooming = NO;
    float zoomScale = [self zoomScale];

    CGSize oldContentViewSize = [contentView frame].size;
    //zooming properly resets contentsize as it happens.
    CGSize newContentSize = [self contentSize];

    CGPoint newContentOffset = [self contentOffset];
    float xMult = newContentSize.width / oldContentViewSize.width;
    float yMult = newContentSize.height / oldContentViewSize.height;

    newContentOffset.x *= xMult;
    newContentOffset.y *= yMult;

    float currentMinZoom = [self minimumZoomScale];
    float currentMaxZoom = [self maximumZoomScale];

    float newMinZoom = currentMinZoom / zoomScale;
    float newMaxZoom = currentMaxZoom / zoomScale;

    //don't call our own set..zoomScale, cause they eventually call this method.
    //Infinite recursion is uncool.
    [super setMinimumZoomScale:1.0];
    [super setMaximumZoomScale:1.0];
    [super setZoomScale:1.0 animated:NO];

    [contentView setFrame:CGRectMake(0, 0, newContentSize.width,
newContentSize.height)];
    [self setContentSize:newContentSize];
    [self setContentOffset:newContentOffset animated:NO];

    [super setMinimumZoomScale:newMinZoom];
    [super setMaximumZoomScale:newMaxZoom];

    // throw out all tiles so they'll reload at the new resolution
    [self reloadData]; //calls setNeedsLayout, among other things for
housekeeping
}
```

Build and run `Examples/07BigViewThingZoomAddition` in the simulator. The images should clear up after a zoom. Speaking of the simulator, this demo application takes a very long time to install on the device because it's copying all the images over USB each time. Since you are about to spend some time focusing on a single performance bottleneck in the code, image drawing, you can simulate this slowness in the simulator with a call to `sleep()`. Avoiding the copy of those PNG files will make debugging go a little faster while simulating the problem reasonably well. Also, I tend to forget to remove

these `sleep()` calls when compiling for the iPhone and wonder why everything slows down when I move back to the device, so let's `#define` this one to only compile into the simulator target. Add the following to `drawRect` in `BigPageView.m`:

```
if(!drawingSuspended)
{
    CGContextSetFillColorWithColor(context, [[UIColor whiteColor]
colorWithAlphaComponent:0.5].CGColor);
    CGImageRef tempImage = [UIImage imageNamed:self.imageName].CGImage;
    #if TARGET_IPHONE_SIMULATOR
        sleep(2.5);
    #endif
    CGContextDrawImage(context, tempbounds, tempImage);
    drawnPageOnce = YES;
}
```

Build and run in the simulator. You should see similar sluggishness compared to running on the phone. Let's tackle that problem now.

Drawing into an Off-Screen Context

Given our self-imposed limitations, we can't make the drawing much faster without digging into OpenGL. Even then, you'll have to decode the images and throw them up into texture memory no matter what you do, so the drawing itself would be fast, but you know from the Shark profile that the decoding is what takes a long time. It's time to take the `NSOperationQueue` and blocks magic to the next level and parallelize the drawing work by putting it into a background thread.

NOTE: Danger, Will Robinson! `UIKit` is not thread-safe. Try to draw to screen from another thread, and bad things might happen. Ugly things are almost guaranteed to happen. You can, however, draw your images into off-screen buffers (actually, `cgContexts`) and then grab the pixels that you need to throw on the screen once the buffer is filled with data. There is nothing stopping you from filling that data asynchronously and reading it from the main thread in order to draw it, as shown in Figure 9-15.

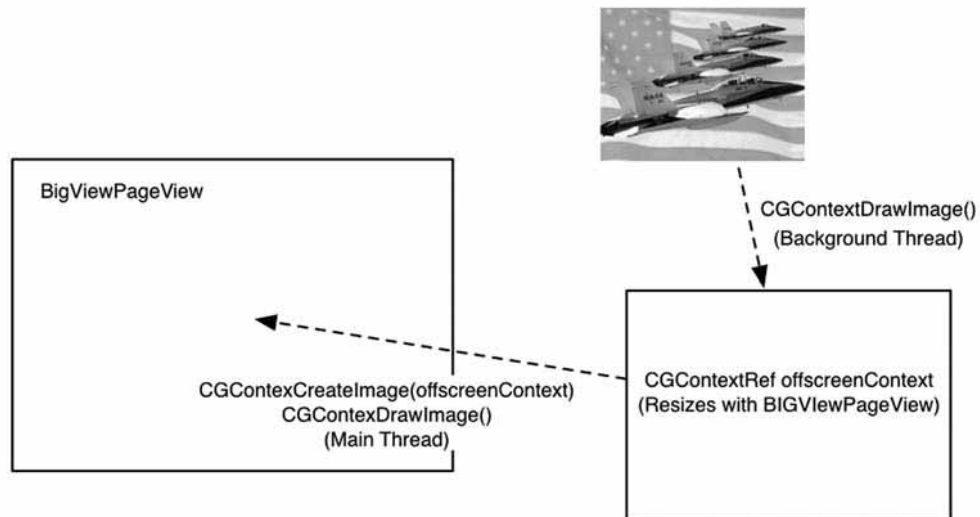


Figure 9-15. *BigViewPageView* draws into an off-screen *CGContextRef* in the background.

Here is the step-by-step exercise:

- a. The first time one of the *BigViewPageView* objects is asked to draw, it will create a *cgContext* type instance variable into which it will quickly draw the half opaque white background that you are currently drawing as a placeholder when the *BigViewPageView* is inactive, like so:

```
-(void)initOffscreenContext // do this on the MAIN thread
{
    CGSize layerSize = [self bounds].size;
    layerSize.height = floorf(layerSize.height);
    layerSize.width = floorf(layerSize.width);

    CGColorSpaceRef colorSpace = CGColorSpaceCreateDeviceRGB();
    CGContextRef ctx = (CGContextRef) [(id) CGContextCreate(NULL, layerSize.width,
layerSize.height, \
                                8, layerSize.width*4, colorSpace,
kCGImageAlphaPremultipliedLast) autorelease];
    CGColorSpaceRelease(colorSpace);
    CGContextTranslateCTM(ctx, 0, layerSize.height);
    CGContextScaleCTM(ctx, 1.0, -1.0);

    //scale is #defined to .94 elsewhere. It causes the images to draw with a little empty
    //space in between each one.
    CGFloat tx = layerSize.width * (1.0 - scale) * 0.5;
    CGFloat ty = layerSize.height * (1.0 - scale) * 0.5;
    CGRect tempbounds = CGRectZero;
    tempbounds.size = layerSize;
    tempbounds = CGRectIntegral(CGRectInset(tempbounds, tx, ty));
    CGContextSetShadow(ctx, CGSizeMake(5,5), 5);
```

```

        CGContextSetFillColorWithColor(ctx, [[UIColor whiteColor]
colorWithAlphaComponent:0.5].CGColor);
        CGContextFillRect(ctx, tempbounds);
        self.offscreenContext = (id) ctx;
    }

```

2. It will draw whatever is in the off-screen context to the screen in `drawRect:`

```

-(void)drawRect:(CGRect)rect
{
    //NSLog(@"drawRect");
    CGContextRef context = UIGraphicsGetCurrentContext();
    CGContextRef osc = (CGContextRef) self.offscreenContext;
    UIGraphicsPushContext(osc);
    CGImageRef tempImage = CGBitmapContextCreateImage (osc);
    UIGraphicsPopContext();
    if(tempImage)
    {
        CGContextDrawImage(context, self.bounds, tempImage);
        CGImageRelease(tempImage);
        drawnPageOnce = YES;
    }
}

```

3. It will generate an `NSOperation` (that calls a block, naturally) that will fill a new `cgContext` with the image data you will need:

```

-(void)createOffscreenCtx
{
    NSOperationQueue *q = [(BigViewThingAppDelegate *) [[UIApplication
sharedApplication] delegate]
globalQ];
    PLBlockOperation *op = [PLBlockOperation blockOperationWithBlock:^(
        //imgRef = [[UIImage imageNamed:imageName] CGImage];
        NSString* bundlePath = [[NSBundle mainBundle] bundlePath];
        UIImage *img = [UIImage imageWithContentsOfFile:[NSString
stringWithFormat:@"%s/%s", bundlePath,
imageName]]];
        CGContextRef imgRef = [img CGImage];

        CGSize layerSize = [self bounds].size;
        layerSize.height = floorf(layerSize.height);
        layerSize.width = floorf(layerSize.width);
        CGColorSpaceRef colorSpace = CGColorSpaceCreateDeviceRGB();
        CGContextRef ctx = (CGContextRef) [(id) CGBitmapContextCreate(NULL,
layerSize.width,
                                layerSize.height, 8,
                                layerSize.width*4, colorSpace,
                                kCGImageAlphaPremultipliedLast) autorelease];
        CGColorSpaceRelease(colorSpace);
        CGContextTranslateCTM(ctx, 0, layerSize.height);
        CGContextScaleCTM(ctx, 1.0, -1.0);

        CGFloat tx = layerSize.width * (1.0 - scale) * 0.5;
        CGFloat ty = layerSize.height * (1.0 - scale) * 0.5;

```

```

        CGRect tempbounds = CGRectZero;
        tempbounds.size = layerSize;
        tempbounds = CGRectIntegral(CGRectInset(tempbounds, tx, ty));
        CGContextSetShadow(ctx, CGSizeMake(5,5), 5);
    #if TARGET_IPHONE_SIMULATOR
        sleep(2.5); //fake slow drawing on the simulator
    #endif
    CGContextDrawImage(ctx, tempbounds, imgRef);
    self.offscreenContext = [[[id] ctx retain] autorelease];
    NSLog(@"Image loaded for %d", pageToDraw);
    //when we're done filling, we need to redisplay content
    [self performSelectorOnMainThread:@selector(setNeedsDisplay) withObject:nil
    waitUntilDone:NO];
    }];
    [q addOperation:op];
}

```

4. When the NSOperation finishes, it will call `setNeedsDisplay` on the view in the main thread so the view knows to draw the image data to screen. You can do this in real time. Drawing from a buffer is fast.
5. Any time the `BigViewPageView` is asked to `drawRect`, it pulls the image data from the current `cgContext` for drawing; it's also filling new `cgContexts` in the background if you change the expected drawing size of the image through some bizarre action like zooming. Before the new buffer is ready, your image will stretch to fill and probably pixelate for a moment while the `NSOperation` is preparing new data.

The sample code in `Examples/08BigViewThingOperationQueueRegular` has all the additional code. It also prints the contents of the `NSOperationQueue` on a timer to show you what is in there. Build and run in the simulator. The application should remain responsive.

Or is it? Every time I zoom in or zoom out on an image, the view pushes another `NSOperation` onto the queue. If you watch the log messages printing the contents of the `NSOperationQueue`, you will see that there are an ever-growing number of operations for each view getting pushed when there is a lot of zooming going on. This makes the app seem like it's updating less and less often. The queue eventually clears but not after drawing a given image several times, usually at zoom levels not currently needed for drawing.

Wouldn't it be nice to be able to cancel only certain pending operations on the `NSOperationQueue`? You can. You just call the `cancel` method on your `NSOperation` object; the queue will eventually (but not immediately) remove it, but it will never actually run it. You can add a weak reference to the `NSOperation` subclass to point back to the `BigViewPageView` object that placed it on the queue and then ask each `NSOperation` that belongs to you to cancel before you add another operation to the queue. This way, you can be sure that there is little wasted CPU time.

NOTE: In this implementation, an operation in progress cannot be canceled, so it's still possible that the queue will have to run two operations for a given view in fairly rapid succession.

Once you have that weak reference, it's easy to create a category on `NSOperationQueue` to cancel all pending `NSOperations` in the queue filtered by an `NSPredicate`.

```
- (void)cancelOperationsFilteredByPredicate:(NSPredicate *)predicate;
{
    NSArray *ops = [[self operations] filteredArrayUsingPredicate:predicate];
    for (NSOperation *op in ops)
    {
        if(![op isExecuting] && ![op isFinished] && ![op isCancelled])
        {
            [op cancel];
        }
    }
}
```

If you notice that the `NSOperation` objects stay in the queue for a while, that is OK. When `NSOperationQueue` decides that it is time to run a given operation, it will call `start` on the `NSOperation` and wait for that operation to finish executing. If `isCancelled` returns YES, the `NSOperation` will tell the `NSOperationQueue` that it is finished right away without ever calling the main method. Add the operation cancellation code into your `BigViewPageView`:

```
-(void)createOffscreenCtx
{
    NSOperationQueue *q = [(BigViewThingAppDelegate *) [[UIApplication
sharedApplication] delegate]
globalQ];
    NSPredicate *filter = [NSPredicate predicateWithFormat:@"SELF.interestedObject ==
%@", self];
    [q cancelOperationsFilteredByPredicate:filter];
    PLBlockOperation *op = [PLBlockOperation blockOperationWithBlock:^(
//BUNCH of drawing code here
)];
    [op setInterestedObject:self];
    [q addOperation:op];
}
```

NOTE: `NSPredicate` is an extraordinarily useful class that uses key-value coding to perform queries on objects. I tend to think of them as structured queries for Cocoa objects, often used to filter arrays based on some parameter or parameters of the objects it contains. Cocoa programmers have enjoyed `NSPredicate`'s power for some time, but it has only recently come to the iPhone in the 3.0 SDK. It's also an important part of the magic of Core Data. You can find more information on `NSPredicate` in Apple's *Predicate Programming Guide* available at <http://developer.apple.com/mac/library/documentation/Cocoa/Conceptual/Predicates/Articles/pUsing.html>.

`BigViewThing` is not finished yet. You've just implemented something similar in behavior to `CATiledLayer`. Perhaps `CATiledLayer` would be even more performant than the `NSOperationQueue` code you're using now. `NSOperation` can have an attached priority. Perhaps you could place a series of low-priority operations on the drawing queue to fill the `cgContext` buffers with a low-resolution version of each image so that the user's off-screen tiles will get drawn in the background using idle CPU cycles, thus removing the gray placeholders. When you zoom back and forth between different levels, you might not really need to rerender each time. Perhaps the default scaling transform from a big zoomed-in image to a small zoomed-out image looks OK to you without a redraw. Buffer size issues aside, perhaps you could allow a delay in redrawing the tiles at a smaller size when the user zooms out by lowering the priority of that operation. That way, operations that dramatically change the user experience will run first.

Observations, Tips, and Tricks

iPhone programming is embedded systems programming. Although you can expect Cocoa Touch devices to become faster and faster over time, programming for the iPhone is closer to that of a Nintendo DS or a LART box than a desktop computer. Our examples will seem slow before you optimize on the new, faster iPhone 3GS, just less so than on the original device. It's always helpful to learn some embedded system programmers' tricks by programming for even more limited devices like LARTs or SBCs. You can often sort of "fake it 'til you make it" when it comes to code that requires a lot of system resources. UI response variability is particularly annoying; users don't know why your app is slow on the Edge network. "Sometimes it's slow; sometimes it's not. I dunno why." is a phrase to which I'm becoming perhaps too accustomed, but I strive never to hear it. Clever caching of data while remaining responsive to the user's input through concurrent programming can make an application shine, even when it isn't really doing anything more than what it did before.

iPhone devices are severely memory constrained, disk read/write speed constrained, and bandwidth constrained when compared to their bigger iron cousins. Remember that UI and data share RAM, so you might get memory warnings at seemingly strange times. You'll notice some CPU and memory monitoring code in some of the example code.

You can use it in your application to anticipate memory resource shortages and modify your application's behavior. Once you do get a memory warning, you receive a short warning, and then the system kills your app without prejudice or allowing you to save precious user data, so be prepared to strip down your views and your data at a moment's notice. There is rudimentary handling of that in the `BigViewThing` example. Look at the method called `memoryWentBoom`.

Summary

In this chapter, you learned how to make asynchronous or data-intensive applications seem more responsive to the user. With the `AAPLot` and `StockPlot` applications, you learned how to persist data between runs and for offline use, and you learned to ship your application with some placeholder information so users can get an idea of how your application will act when adding their own stuff. You also learned how to use `NSOperationQueue` as a means to put processing of downloaded data into the background, and you got a look at the `Core Plot` library to plot data. With `BigViewThing`, you learned the ins and outs of zooming in a `UIScrollView` and a method to sharpen your drawing when the user zooms in. You also peered into the future with the `Plausible Blocks` library, and you learned how to make an application that deals with massive images as responsive as possible by dropping operations into a background thread using blocks.

Dealing with large amounts of data on a limited device like the iPhone presents a challenge. It's a challenge for which we happily carry plenty of tools to solve but for which there is rarely a simple, singular answer. I hope this chapter has added a wrench or two to that toolbox.

Joe Pezzillo

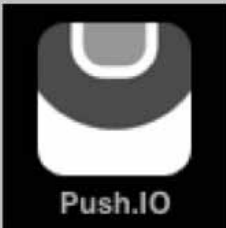


Company: Push IO LLC

Location: Boulder, Colorado

Former Life As a Developer: Joe has been programming Cocoa/Obj-C on Mac OS X since 2001, and was a lead researcher at the Apple Electronic Media Lab for the three years of its existence in the mid-90s. After that, Joe started one of the earliest and most popular Internet Radio companies in 1996 and still does an occasional volunteer afternoon music DJ shift on KGNU, the AM/FM community radio station in Boulder/Denver. Joe has been exploring computers since he was 9 years old, sneaking into the campus computer lab to play games on the terminals, and his first home computer was an Apple][+.

Life as an iPhone Developer: Joe is the co-founder of Push.IO, a startup building "smart infrastructure for smart phone developers." For the last year he's been doing enterprise iPhone development, building native iPhone applications for a Fortune 100 financial services company.



What's in This Chapter: This chapter walks the user through the implementation of Apple's Push Notification Service for their iPhone applications. We look at the client methods, the certificate and provisioning process, and finally the server component required to make it all work, wrapping it all up into a sample project that includes both client and server code.

Key Technologies:

- *Apple Push Notification Service (APNS)*
- *CocoaTouch/Objective-C methods for implementing APNS*
- *The iPhone Developer Program Portal process for creating Push certificates*
- *The PHP Server Code needed to send notifications*

Demystifying Apple's Push Notification Service

Like so many things, my involvement with this technology all goes back to a quote frequently attributed to Mark Twain: “When everyone is looking for gold, it’s a good time to be in the pick and shovel business.”

I first heard this expression while I was working for Apple in the mid-90s. The Web 1.0 gold rush was on, and the R&D group I was part of was one of the few groups in the company doing anything Internet related at the time.

We were lucky to be under the direction of a senior R&D executive, who in his first meeting with our group laid out Mark Twain’s wisdom perfectly: “We will make picks and shovels.”

The lightbulb went on.

One could certainly argue that not only has Apple since gone on to make picks and shovels (how many web sites are developed on Macs?), it has also struck gold more than once, too. In fact, it has struck so much gold, it’s practically a bank!

That easy yet insightful advice has stuck with me ever since, and when it became obvious that the iPhone had become its own gold rush, I couldn’t help but wonder, “What is the picks and shovels play?”

At the first Satellite iPhoneDevCamp Colorado in 2008, I met Dan Burcaw, and we discovered we were both asking this question and went around for a couple months with ideas.

Finally, Dan was in the room in Cupertino when Apple announced iPhone OS 3.0, and we realized that the new features Apple was providing were going to need people who could provide the back-end server support—the picks and shovels, if you will. Dan’s background in servers (he cofounded the company that created the Yellow Dog Linux distribution) and mine in Internet broadcast (see U.S. patent 6434621), combined with both our prior experiences working at Apple, made us think this could be the perfect entry point.

By the time you've finished this chapter, you'll have a working implementation of a simple Apple Push Notification Service client on your iPhone that talks to a remote server and allows you to send notifications to all users of this particular app.

What Is the Apple Push Notification Service?

Simply, the Apple Push Notification Service (APNS) is a way to send text alerts, custom sounds, and badge counts to your application on users' devices encouraging them to use your app, even if your app isn't running at the moment.

APNS has a number of advantages:

- Free—no SMS charges, free to use, free to develop
- Can invoke your app
- Can make your app play a sound or show a badge
- Doesn't require background processing
- Easy to add to your app

What You'll Need

You'll need to be a full member of the Apple Developer Program, with the ability to generate certificates (in other words, a Team Agent). You'll also need an iPhone or iPod touch capable of receiving notifications. I'll also presume you have a remote server where you will run the back end for this application.

Although this chapter will walk you through the entire process from end to end, I'd still say that it's basically mandatory that you read Apple's *Remote Notification Programming Guide*. It is the official documentation, after all.

Also, access to the forums will be invaluable as you implement this. Not only are there dozens of source code samples for a variety of languages, you'll also see what issues other developers are encountering and how they resolve them.

I'm also going to presume that you've already developed at least one iPhone application, even if it's just a "Hello World" app, so that you know how to use Xcode, create a project, deploy your app to your device, and have some familiarity with the iPhone Developer Program Portal.

Step 1: Create the Client

Open Xcode, and make a new project using the View-based Application template; let's call it Push2. I wanted the final product on the phone to be called 2Push2, so I went to the Target Build properties of my new project and changed the Product Name field to 2Push2 for all configurations. There are two primary parts of this app: the part that interacts with APNS and the part that interacts with the user. The meat of push notifications is going to take place in the application delegate. For the user interface, I'll also show how to make a view controller that allows you to see and send messages.

The Application Delegate

In the `Push2AppDelegate.m` file, you'll add what are basically the three required methods. I'll get to the actual code in the next section, but to give you an idea of how little work you have to do to add APNS to your application, these are the only three methods you have to add to your existing code to get started:

```
- (BOOL)application:(UIApplication *)application
didFinishLaunchingWithOptions:(NSDictionary *)launchOptions

- (void)application:(UIApplication *)application
didRegisterForRemoteNotificationsWithDeviceToken:(NSData *)deviceToken

- (void)application:(UIApplication *)application
didFailToRegisterForRemoteNotificationsWithError:(NSError *)error
```

Part of the appeal of push notifications is that they can help bring users back into your application even if it is not running, but if you want to also be able to handle notifications while your app is open, then you'll want to add this fourth method:

```
- (void)application:(UIApplication *)application
didReceiveRemoteNotification:(NSDictionary *)userInfo
```

Registration

The first step in the process is to tell the iPhone OS what kinds of notifications you want to receive, which can be any combination of badges, sounds, and alerts. So, the first of the three methods you need to tackle are these:

```
(BOOL)application:(UIApplication *)application
didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
```

Open the `Push2AppDelegate.m` file, and add the code for this new method:

```
- (BOOL)application:(UIApplication *)application
didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    // other setup tasks here...
    [[UIApplication sharedApplication]
registerForRemoteNotificationTypes:(UIRemoteNotificationTypeAlert |
UIRemoteNotificationTypeSound)];

    // [self updateWithRemoteData]; // freshen your app!

    // RESET THE BADGE COUNT
    application.applicationIconBadgeNumber = 0;

    // ...
    // call the original applicationDidFinishLaunching method to handle the basic view
    setup tasks
    [self applicationDidFinishLaunching:application];

    return YES;
}
```

`didFinishLaunchingWithOptions` is a new method in iPhone OS 3.0, intended to replace `applicationDidFinishLaunching` and recommended by Apple as a replacement for that old method since it handles both the delivery of the push notification payload to your application and the case that your application is opened by a custom URL protocol handler. It's important to note that when you use this new method, your old `applicationDidFinishLaunching` will *not* get called. Since the Xcode template already includes a placeholder `applicationDidFinishLaunching` that handles presenting the view, you'll note that you still call that as the last step in this new method.

It's in this method that you call `UIApplication`'s `registerForRemoteNotificationTypes` to set up the types of notifications you're interested in receiving.

This registers the application with the system to get notifications. Call the `registerForRemoteNotificationTypes` method with the set of options you want to support—alerts (text dialogs), sounds, and badges. Use the “or” command (the pipe) to combine these values:

```
UIRemoteNotificationTypeBadge,  
UIRemoteNotificationTypeSound,  
UIRemoteNotificationTypeAlert,
```

NOTE: You can see that in this example I'm registering to receive alert and sound notifications.

You'll also notice that you might start loading the freshest data from your remote server here, and you're updating the badge count to 0, removing any badge from the application's icon on the home screen.

When `registerForRemoteNotificationTypes` returns successfully, the app will now be able to receive notifications even when it's not running! You should make sure that you call this method every time your application launches because there is a chance that the device token will change, for example, if the user has restored their device from a backup. This also means that if the user has restored from a backup, they must run your application to start receiving notifications again. There's no harm in calling this method every time, since if the device token hasn't changed, the OS knows and will simply return you to the still-current device token.

Device Token Acquisition

Once the system has successfully registered your app, you will get called back (asynchronously) and given a token specific to both this application and this device in the second method you have to implement:

```
- (void)application:(UIApplication *)app  
didRegisterForRemoteNotificationsWithDeviceToken:(NSData *)devToken {  
    [self sendDeviceTokenToRemote:devToken]; // send the token to your server  
}
```

From here, you need to pass this token to your server. I'll show an example of how to do this in the demo application using its simple server script, but for now, make a note that

you'll need to implement your own version of the hypothetical `sendDeviceTokenToRemote:` method that's shown here.

Check for Errors

There's also an error callback method you should choose to implement to handle the case when you do not get a token, which is especially useful during debugging:

```
- (void)application:(UIApplication *)app
didFailToRegisterForRemoteNotificationsWithError:(NSError *)err {
    NSLog(@"Failed to register, error: %@", err);
}
```

So, after implementing these three methods, you're now set up to handle the case that your application is called as the result of the user selecting the View button in a text alert or unlocking their phone immediately after receiving a text alert.

But what about when your application receives a notification while it is already running?

To handle this case, you also need to implement the following method in the `AppDelegate:`

```
- (void)application:(UIApplication *)application
didReceiveRemoteNotification:(NSDictionary *)userInfo
```

You'll use this to update the user interface if a new notification comes in while the app is running.

Handling Incoming Notifications

When the payload finally arrives in your application ready for use, it comes in the form of a dictionary. It's so easy! When either one of the following two methods gets called, you've got a notification. Either the following gets called as a result of launching the app after a notification comes in, and then the `launchOptions` `UIApplicationLaunchOptionsRemoteNotificationKey` will have the notification value in the `aps` key:

```
- (BOOL)application:(UIApplication *)application
didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
```

or the following gets called as a result of a notification arriving while the app is running, in which case the `userInfo` `NSDictionary` object will have the notification value in the `aps` key:

```
- (void)application:(UIApplication *)application
didReceiveRemoteNotification:(NSDictionary *)userInfo
```

The notification is itself a dictionary with the included components, so the following gives you the text of the notification alert in the alert variable:

```
NSDictionary *aps = [userInfo valueForKey:@"aps"];
NSString *alert = [aps valueForKey:@"alert"];
```

You can also send your own custom data in the notification, as I'll demonstrate later in the chapter.

Sounds

As you've already seen, one of the notification types you can make use of is a custom sound, and even better, it's very easy to implement!

Since it's the system that plays these sounds, they have to be in one of the standard system sound formats (linear PCM, μ Law/aLaw, or MA4) and file types (.aiff, .wav, .caf).

The Apple Push Notification Programming Guide also shows how to use the `afconvert` command-line tool to prepare standard system sounds for use, too. For example, here's how they show converting the system sound Submarine for use:

```
afconvert /System/Library/Sounds/Submarine.aiff ~/Desktop/sub.caf -d ima4 -f  
caff -v
```

Once your audio is prepared, simply add it to the Xcode Resources group, and it will be built into your application bundle and be available to APNS.

Then, with the sound file in your app's bundle, you simply reference that sound file in the JSON payload you send from your server to APNS, which I'll cover in the "Step 3: Set Up the Server" section of this chapter.

Note that if the user has their phone muted, then it will vibrate in lieu of playing the sound.

Although sounds can be an incredibly useful way to notify your users, do consider the experience your users might have if either your application is causing sounds to play constantly or your application is just one of many that are causing sounds to play. Remember that the user can turn off notifications in the system Settings application, and you can pretty much presume they will if they become too annoying.

Build and Go! Er, Not So Fast...

Now, with all of this done (and I haven't even gotten to the interface yet), you might be tempted to build and run the application, which is a fine thing to do, except you'll probably encounter one of the first requirements. You must be able to deploy to a device, because APNS does not work on the simulator:

```
2009-07-26 19:45:38.880 2Push2[12444:20b]  
didFailToRegisterForRemoteNotificationsWithError:Error Domain=NSCocoaErrorDomain  
Code=3010 UserInfo=0xd2a170 "remote notifications are not supported in the simulator"
```

Switching over to the device trying to build will then lead you to one of two errors. In Xcode while building, you'll get this error:

```
Code Sign error: a valid provisioning profile matching the application's Identifier  
'com.yourcompany.2push2' could not be found
```

Or in the console when running, you'll get this error:

```
2009-07-26 19:52:55.415 2Push2[3046:207]
didFailToRegisterForRemoteNotificationsWithError:Error Domain=NSCocoaErrorDomain
Code=3000 UserInfo=0x12faa0 "no valid 'aps-environment' entitlement string found for
application"
```

Both of which will be addressed by creating the certificate in the next section.

Step 2: Create the Certificate

In this section, I'll walk through all the steps necessary to generate the required server-side development SSL certificate and mobile provisioning files for use with your push notification application.

If you have already made distribution certificates for yourself for ad hoc or App Store distribution, this will build on your experience. If you haven't already made certificates for yourself, although this isn't particularly difficult, it's a lengthy, involved process, so I recommend starting with just an ad hoc certificate before leaping right in to APNS.

There are a couple of key things to keep in mind about the setup. First, there's a secret certificate file that lives on your server that talks to Apple's APNS. This certificate identifies you and your application so that Apple can both trust you as the originator of the notification and know that the device has asked to receive the notification.

Second, there's also going to be a custom mobileprovision file that identifies your application to the system to receive its notifications.

In addition, you will need to use a custom, unique application ID in the program portal. You cannot use a wildcard application ID for this, because each application needs to be addressed individually.

Once you have created this certificate, it will be used to authenticate and encrypt your connection to Apple's push notification servers using SSL.

Because the topic is encryption, which is a genuinely tough subject and because there are so many steps (that can go wrong), creating certificates is one of the most dreaded tasks in iPhone development. If you're new to this, follow along the first time through, which will demystify the process along the way, and just get it done.

A Walk-Through of the Program Portal Process

In your web browser, go to this location:

<http://developer.apple.com/iphone/>

Log in using your Team Agent account.

From here, on the right side near the top, you'll see the iPhone Developer Program Portal link.

NOTE: If you don't see the program portal option, then it's possible that either you are not signed in as the Team Agent or you are not yet fully enrolled in the program. You'll need to be enrolled and signed in as Team Agent to complete this section of the process.

Click the link to go to the program portal.

The program portal is where you manage all your certificate-related details, from devices to app IDs to provisioning profiles. I'll presume you already have your basic development certificate set up and that you've previously deployed code to the device at a minimum and ideally for distribution.

The first step here is to create an app ID (Figure 10-1).

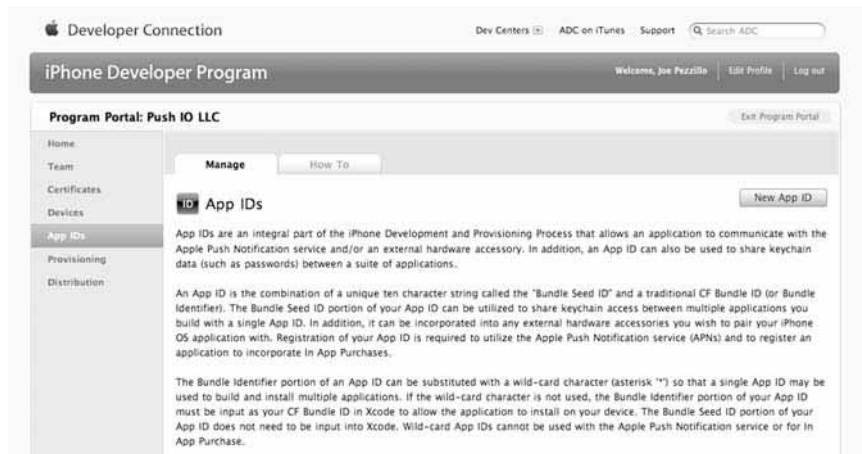


Figure 10-1. The program portal screen for creating a new app ID

I'm going to use `com.pushio.2push2` for my app ID.

Enter the information required, enter a brief description, let it generate a new bundle seed ID, and enter the reverse-domain notation app bundle ID (that's the `com.pushio.2push2` string for me), as shown in Figure 10-2.

Figure 10-2. *Entering a description and the bundle identifier*

Then, you'll see your new app ID in the list on the main manage App IDs page, as shown in Figure 10-3.

Description	Apple Push Notification service	In App Purchase	Action
4668AH5CTX.com.pushio.appid New App ID	<input checked="" type="checkbox"/> Configurable for Development <input checked="" type="checkbox"/> Configurable for Production	<input checked="" type="checkbox"/> Configurable	Configure

Figure 10-3. *Your new app ID in the list on the main manage app IDs page*

Choose Configure, and then enable your app for APNS. Select the check box next to "Enable for Apple Push Notification service," as shown in Figure 10-4. Then click the Configure button.

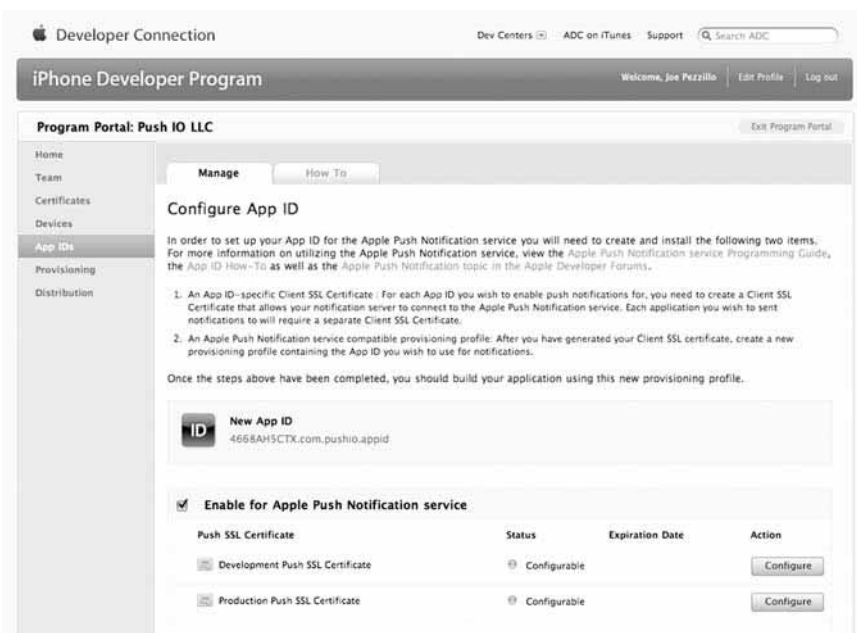


Figure 10-4. Select the “Enable for Apple Push Notification service” check box.

For this app, click the Configure button for the Development Push SSL Certificate option. You’re now prompted to launch the Keychain Access application and generate a certificate signing request (CSR), as shown in Figure 10-5, so let’s do that!



Figure 10-5. The APNS SSL Certificate Assistant walks you through the required steps.

As the assistant tells you, once in Keychain Access, select Keychain Access ► Certificate Assistant ► Request a Certificate from a Certificate Authority, as shown in Figure 10-6.

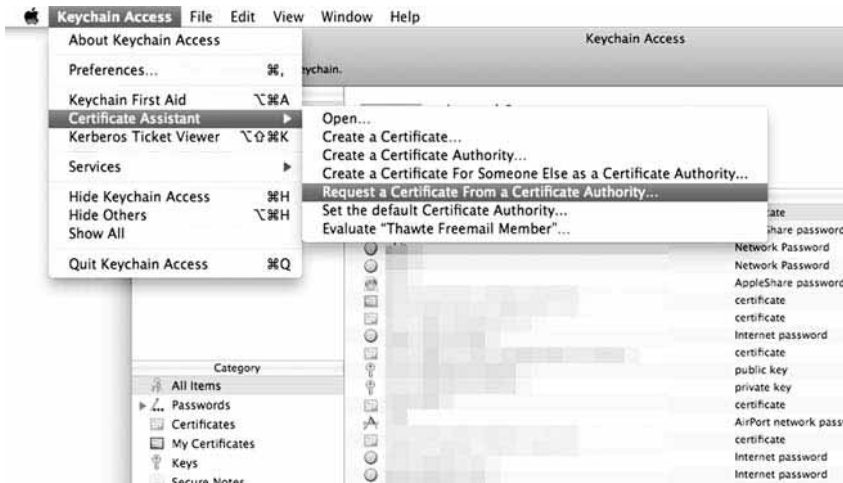


Figure 10-6. Choosing the *Request a Certificate from a Certificate Authority* menu in Keychain Access

Enter a valid e-mail address, a memorable name for this development certificate (so you can easily identify the key pair), leave the CA Email Address field blank, and choose to save it to disk, as shown in Figure 10-7. Click Continue, and it will prompt you for where to save the file. For this exercise, save it to your desktop as 2Push2-Development-APNS.certSigningRequest.



Figure 10-7. Creating the CSR for the development push SSL certificate

When done, it will tell you that your certificate request has been created on disk. Now switch back to the program portal.

Here, the assistant is waiting to choose the CSR file you just created. Select Choose File, and find the new CSR file you saved on the desktop, as shown in Figure 10-8.



Figure 10-8. Attach the CSR you just created with the Choose File button.

When that is done, it will present you with a new certificate file to download, as shown in Figure 10-9.



Figure 10-9. Download the new certificate file, and rename it to something more memorable.

Download this file, which will be saved as `aps_developer_identity.cer`.

Immediately rename it to something that will help you identify the file later, such as `2push2-aps_developer_identity.cer`. Double-click this file, which will then open in Keychain Access.

Add it to the login keychain as prompted, as shown in Figure 10-10.



Figure 10-10. Add the certificate you just downloaded to the login keychain.

One of the key reasons to name that certificate you just created with a memorable common name is because it will be showing up in the keychain access list along with other such certificates, and you need to be able to tell them apart easily, as shown in Figure 10-11.

Apple Development Push Services: X	5	P	certificate	Oct 24, 2009 1...	login
Apress Apush Dev Certificate			private key	--	login
Apple Development Push Services: X	5	A	certificate	Oct 19, 2009 1...	login
Push			private key	--	login
Apple Development Push Services: D	M:8	4	certificate	Dec 5, 2009 12...	login
2Push2 Dev Cert			private key	--	login

Figure 10-11. The certificate with the renamed private key is easier to find in Keychain Access.

Now, here's the tricky part that's most likely needed by your server, such as in this case where you'll use a remotely hosted PHP script to do the back-end processing.

Select the certificate, and choose Export. It will prompt you for a name and location to save the file. Once again, put it on the desktop for now, with a name you'll recognize later, such as 2Push2-Dev-Cert.p12. Save it in the .p12 format. When prompted for a password for the certificate, as shown in Figure 10-12, do *not* enter one (more on this later). Click OK.



Figure 10-12. For now, do not enter a password on this screen; just click OK.

Next, you are prompted for your login password to authorize the export. Enter the password you use to log in to your machine here (you do use a password to log in to your machine, yes?), as shown in Figure 10-13.



Figure 10-13. Your login password gets entered here, just to authorize the export.

Your password is not being added to the certificate; it is only being used to authorize the export of the APNS certificate, which is sensitive data and should be protected from disclosure. Click Allow to authorize this for this one time only.

You should now have this sensitive file on your desktop. It is sensitive because it contains the private key that you created earlier that uniquely identifies you to Apple to authorize notifications.

You now need to take one more step on this file to get it ready for use on the server, and that's to convert it from the .p12 format to the .pem format. (PEM stands for Privacy Enhanced Mode.)

This is not hard but does require you to make a trip into the Terminal.app file.

From the Terminal prompt, you'll need to navigate to where the .p12 file lives. You last left it on the desktop, so enter `cd ~/Desktop` to navigate to your desktop.

Then, enter the following command, as shown in Figure 10-14:

```
openssl pkcs12 -in 2Push2-Dev-Cert.p12 -out 2push2-dev-cert.pem \
-nodes -clcerts
```

When it asks you to enter the import password, enter nothing. Instead, simply hit Return.



Figure 10-14. Convert the .p12 file to a .pem file for the server using `openssl` on the command line

I named the output file `2push2-dev-cert.pem` to help keep track of what it is. You may want to shorten the name.

Note that the resulting file is extremely sensitive and should be protected from disclosure.

Keep the .pem file handy, because you'll use it again during step 3.

Back to the Portal

OK, you're not quite done yet; you need to go back to the Apple program portal and create one more file.

Click the Provisioning link in the left column, which will take you to your Development Provisioning Profiles list, as shown in Figure 10-15.

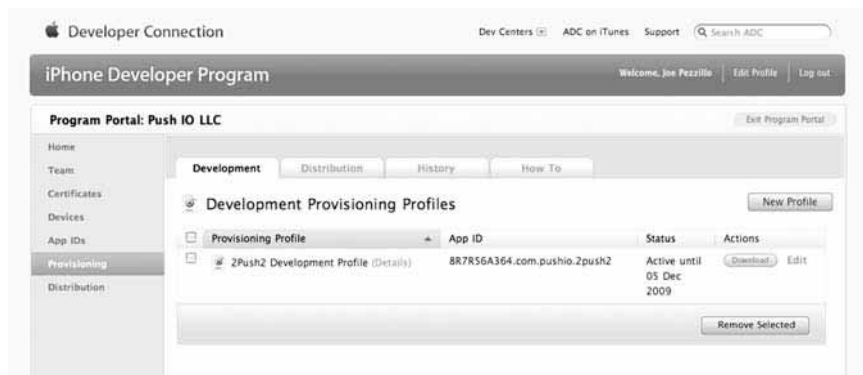


Figure 10-15. The Development Provisioning Profiles list in the program portal

Click New Profile.

Enter a useful profile name, choose your development certificate, select the app ID you created earlier, and select the devices on which you want to be able to do development of this application, as shown in Figure 10-16.

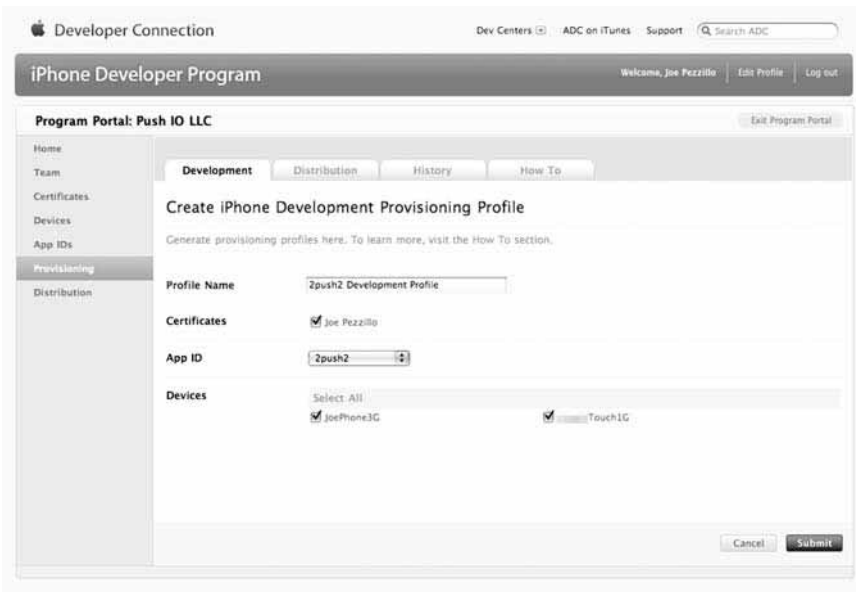


Figure 10-16. Set up the certificate, app ID, and devices for the development provisioning profile.

NOTE: Once you use a device for APNS development, it is “locked” into push development mode. Switching back to Distribution (that is, ad hoc) mode may require a restore of the device. If you search Apple’s iPhone Developer Forums about this topic, you’ll find plenty of discussion. Some have reported that you can simply run an app that you got from the App Store that is in distribution mode to switch your device. Your mileage may vary.

Submit the new provisioning profile setup, and wait for a moment while it’s generated; then download your new `mobileprovision` file. In this case, it’s called `2Push2DevAPNS.mobileprovision`.

Add the Mobile Provisioning File for Code Signing

Double-click that file. Xcode will open, and the file will appear in the `mobileprovision` files list. You may want to quit and restart Xcode immediately after this step, just to make sure that it “takes” the file and recognizes it before the next step.

Next, go back to your iPhone client application, `2push2.xcodeproj`, and open the target settings (click the target in the left list, and then click the big blue “i” button at the top of the screen).

In the inspector window that opens, choose the Build tab at the top. Select your mobileprovisioning file that you just created, as shown in Figure 10-17.

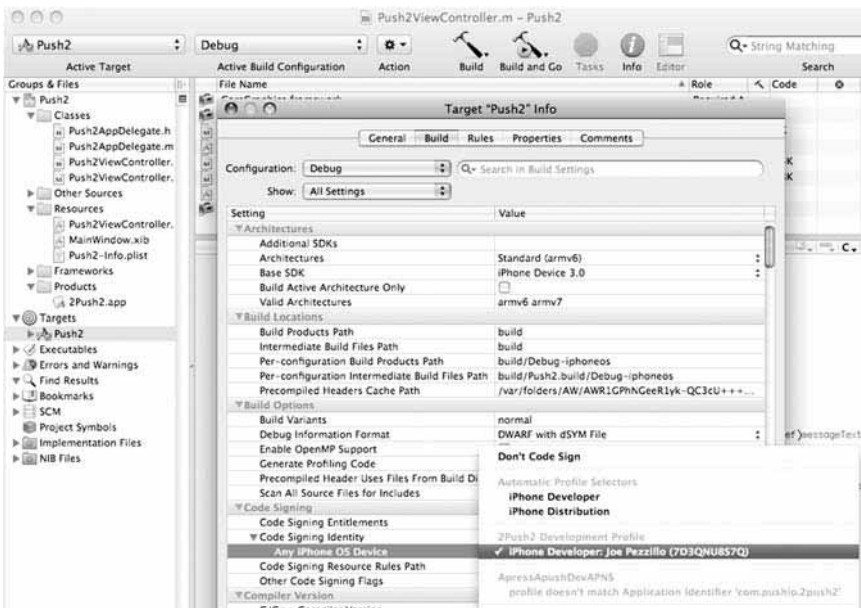


Figure 10-17. Selecting the new code-signing identity in Xcode in the **Targets > Info > Build** pane.

If it's there but not selectable, double-check to make sure that the app ID matches your bundle ID under the Properties pane of the same Target Info inspector window.

Build and run your app, and you should see two juicy bits. The first is that the device token is now appearing in your console log output:

```
2009-07-26 22:56:57.840 2Push2[3272:207]
didRegisterForRemoteNotificationsWithDeviceToken:<7c8f50b4 51ef62e1 6c068b42 b3425e47
839be4c6 5aeac1cd d0ede0f 85467304>
```

The second is that your application should have prompted the user to allow push notifications, as shown in Figure 10-18.



Figure 10-18. The iPhone OS prompt to confirm the user wants push notifications from your application.

Now, you have everything you need to set up the server.

Step 3: Set Up the Server

This, along with the user interface to it, is actually where you're going to have some fun with this application.

This is really the ultimate magic of push notifications. It's not the technology itself; it's what developers do with it. Sure, getting your AIM messages and tweets is actually already really great, and there will be so much more that people create.

I'm going to show how to do this in PHP mostly because it's accessible to a wide range of programmers even in other languages, but there are lots and lots of options on the forums for all your favorite languages.

NOTE: This PHP example code by Jerome Poichet (<http://frencaze.com>) is available on the Apple Developer Forums, used with permission. See <https://devforums.apple.com/message/50461>.

There are two places where you must make changes to this script, ENTER_DEVICE_TOKEN_HERE and CERTIFICATE_FILENAME:

```
<?php
$pass = ''; // Passphrase for the private key (.pem file)
$token64 = 'ENTER_DEVICE_TOKEN_HERE'; // base64 encoded device token

// The actual notification payload
```

```

$body = array();
$body['aps'] = array( 'alert' => 'Greetings from Joe!', 'sound' =>
'sound.aif', 'badge' => 1);

/* End of Configurable Items */

$ctx = stream_context_create();
stream_context_set_option($ctx, 'ssl', 'local_cert', 'CERTIFICATE_FILENAME.pem');
stream_context_set_option($ctx, 'ssl', 'passphrase', $pass);

$fp = stream_socket_client('ssl://gateway.sandbox.push.apple.com:2195', $err, $errstr,
60, STREAM_CLIENT_CONNECT, $ctx);
if (!$fp) {
    print "Failed to connect $err $errstr\n";
    return;
}

$payload = json_encode($body);
// Thank you to the Perl example - MODIFIED
$msg = chr(0) . chr(0) . chr(32) . pack('H*', $token64) . chr(0) . chr(strlen($payload))
. $payload;

fwrite($fp, $msg);
fclose($fp);

```

You're doing something really simple here, and that is sending a text alert notification to a single device that is hand-coded into the script.

There are a few key things to note in this script:

- There's an SSL connection made to the sandbox server, gateway.sandbox.push.apple.com on port 2195, using the SSL server certificate created in the previous section.
- The `$body['aps']` associative array has the values for the three different types of notifications that can be sent. Specifically, alert is the text for what should appear on the screen in the alert dialog box, sound is the name of the sound file you want played, and badge is an integer for the count to be displayed as a badge on your application's icon.
- This script is using the `json_encode` function to convert a PHP array into a JSON encoding.
- The `$msg` variable serializes the data being sent to the APNS servers in the required format: two null (zero) bytes, an ASCII space, the token "packed" into hexadecimal, another null byte, the payload length, and finally, the JSON payload itself.

The iPhone OS generates the device token and provides it to the application in the `didRegisterForRemoteNotificationsWithDeviceToken` method you saw earlier. Our example app shows it on-screen and prints it to the console. You'll want to copy and paste it either from the iPhone application to an e-mail or from the console directly. If

you copy it from the console, remember to remove the brackets (< >) and spaces so you have one long 64-character string that you paste into the PHP script.

Yes, this way of approaching things is a bit ugly, but otherwise the chapter would have been about web services and not APNS!

NOTE: For more information about web services on the iPhone, refer to Joachim Bondo's chapter.

A Walk-Through of What This Script Does

First, the script offers an option for a password, but you're leaving it blank (as you have throughout the process). If you want to add a password to your certificate, then you would also need to supply that password here, which if you hard-code it may create an additional security issue.

Next, it sets up the value of the device token, which you copied and pasted in from the console output or e-mail.

Then it sets up the payload for the notification. Here, you're sending only a text alert.

Once the variables are set up, the script opens an SSL socket connection to the Apple sandbox server on the appropriate port, encodes the token and message, writes it through the socket, and then closes the connection.

To try this, save this script on your server as `apns_test.php`, copy the `.pem` certificate file you created earlier into a directory outside your web-accessible directory, and adjust the path to the `CERTIFICATE_FILENAME` and path appropriately. For example, I might put it in my home directory, and then the path might be something like this:

```
/home/joe/2push2_dev_cert.pem
```

NOTE: Do not store the `.pem` file in a web-readable directory!

Now, run the PHP script, and if everything goes according to plan, it should look like Figure 10-19.



Figure 10-19. *Your first push notification! (suitable for framing)*

Congratulations, you're pushing!

Download Server File

Now, I could spend the rest of the book writing about how to build a server for this, but I'm already running long, so I'm providing sample code you can use to host the server side of this application.

You can most likely test this PHP script on your own laptop, but you'll want to make sure it's on a host that's accessible from the Internet if you want multiple users of your app to be able to have access.

Download the server script for your own use here:

<http://2push2.us/apress/server/>

Using this script will require some basic knowledge of PHP hosting and MySQL to get the script uploaded, running, and connected to a database. The SQL command to create the single needed devices table is included in the script.

Install this script (`apress.php`) and the appropriate development certificate file by uploading them to your host.

There are a few things you'll need to change in the script:

- The Server address

- The certificate file path

- Your MySQL host, username and password

If you call this script without any arguments, it will return usage information:

Apress.php

Sample server program for use with Apress Push Notification Chapter

Usage:

?token=DEVICE_TOKEN&cmd=(reg|msg)&(name|msg)=(USERNAME OR MESSAGE)

e.g.

Register a token:

?token=1d19fc527407d39bcd1d69deff7a3e7abe569d7a8c7b0c69b0b3d30269c0b8d1&cmd=reg&name=test

Send a message:

?token=1d19fc527407d39bcd1d69deff7a3e7abe569d7a8c7b0c69b0b3d30269c0b8d1&cmd=msg&msg=Hello,%20World!

Test it using the included sample arguments to make sure it's installed and working.

If registration is working, the script should return the following:

```
Write token to registrations
success registering
```

Sending is harder to test from the web page without a valid device token. It will be easier to test once you set up the client application, so let's get to that next.

If you try it, you should see "Send message" followed by at least one device token and then a "success" message. If you don't get success, you're likely to see something like this:

```
Warning: stream_socket_client() [function.stream-socket-client]: unable to connect to
ssl://gateway.sandbox.push.apple.com:2195 (Connection timed out) in
/home/content/html/apress/apress.php on line 209
Failed to connect 110 Connection timed out
```

which basically means that your server couldn't connect to the Apple server, and that's possible for one of a couple primary reasons:

- Your host doesn't support connections over port 2195.
- Your certificate is set up wrong. (Do you have the wrong file name in variable? Is it in a directory that the script can read? Is it the correct certificate for app, such as the correct app ID/bundle name, development vs. production certificate?)

NOTE: It is not good practice to leave the security certificate in the same directory as the script, such as in a web-accessible location.

The Home Stretch

With the new server script and the certificate file up on the host, you're ready to finish the application and start pushing something interesting!

You'll recall the app specification—an iPhone application that distributes push notifications from any user to all users of the app.

So, now that you've sent a notification through the system, you can add the finishing touches to your app.

The server API for this test is a simple GET call where you set the `msg` parameter to the message you want to send (it has to be URL encoded, and the server will truncate it after 140 characters).

There are two other parameters. One is the device token of the sending device so that you can authenticate that the message came from a registered user of the app, and the other is a command argument so you can tell the difference between the process of registration and sending a message.

That is, if you send this:

```
http://SERVER/DIRECTORY/apress.php?token=DEVICE_TOKEN&cmd=msg&msg=Hello%20World!
```

then in your application's console output, you should see this:

```
2009-08-07 00:22:40.624 2Push2[1142:207] didReceiveRemoteNotification:{
  aps = {
    alert = "test says: Hello, World!";
  };
  custom = test;
}
```

And here, you also see what it looks like when you attach your own custom values to the APNS payload, in this case the username string in the “custom” field.

NOTE: I'm going to make a bunch of changes in the Cocoa code for the iPhone application. You can save yourself from typing this all in by downloading the project and code files at <http://2push2.us/apress/client>.

Wiring Up the Client

You want to be able to send to the UI elements on-screen and know when the button has been pressed.

First, create the outlets and actions that you're going to connect to in `Push2ViewController.h`.

In particular, add `IBOutlet`s for a `UITextView` and `UILabel` and an `IBAction` for a button:

```
IBOutlet UITextView *messageTextView;
IBOutlet UILabel*deviceTokenField;
IBOutlet UITextField *usernameField;
```

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

Next, switch to Interface builder and add the UITextView, a UILabel, a UITextField (for the username), and a UIButton.

Then wire these all up in Interface Builder, as shown in Figure 10-20.



Figure 10-20. The objects, layout, and connections in Interface Builder for our sample application

Then in the code, you have a few details for talking back and forth between the push notification methods and the user interface, as well as the remote server.

Here's what that looks like in the ViewController code:

```
-(IBAction)handleSendButton:(id)sender
{
    NSLog( @"handleSendButton" );

    // make a get request to our script with the msg parameter set
    // msg=URLENCODEDSTRING;

    CFStringRef outString = CFURLCreateStringByAddingPercentEscapes(kCFAllocatorDefault,
(CFStringRef)messageTextView.text, NULL, NULL, kCFStringEncodingUTF8);

    NSString *urlFormatString =
@"http://2push2.us/apress/apress.php?token=%@&cmd=msg&msg=%@";

    NSURL *composedURL = [NSURL URLWithString:[NSString
stringWithFormat:urlFormatString,deviceTokenField.text,(NSString *)outString]];

    NSLog( @"composedURL:%@", composedURL );

    NSString *result = [NSString stringWithContentsOfURL:composedURL];
```

```

        NSLog( @"result:%@", result );

        CFRelease(outString);
    }

    -(void)handleSetDeviceTokenField:(NSString *)inDeviceToken
    {
        NSLog( @"handleSetDeviceTokenField:%@", inDeviceToken );

        deviceTokenField.text = inDeviceToken;
    }

    -(void)handleDidReceiveRemoteNotification:(NSDictionary *)userInfo
    {
        NSDictionary *aps = [userInfo valueForKey:@"aps"];
        NSString *alert = [aps valueForKey:@"alert"];

        messageTextView.text = alert;
    }

```

And in the App Delegate where we interact with the notifications, here are the three key methods:

```

- (BOOL)application:(UIApplication *)application
didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    // register for remote notifications
    UIRemoteNotificationType types = UIRemoteNotificationTypeBadge |
    UIRemoteNotificationTypeSound | UIRemoteNotificationTypeAlert;
    [application registerForRemoteNotificationTypes:types];

    // because we implement didFinishLaunchingWithOptions, the "old" entry method
    doesn't get called
    [self applicationDidFinishLaunching:application];

    return YES;
}

- (void)application:(UIApplication *)application
didRegisterForRemoteNotificationsWithDeviceToken:(NSData *)deviceToken
{
    NSLog( @"didRegisterForRemoteNotificationsWithDeviceToken:%@", deviceToken );

    NSString *inDeviceTokenStr = [deviceToken description];
    NSString *tokenString = [inDeviceTokenStr
stringByTrimmingCharactersInSet:[NSCharacterSet characterSetWithCharactersInString:@"<
>"]];
    tokenString = [tokenString stringByReplacingOccurrencesOfString:@" "
withString:@""];
}

```

```

// send it to the remote server
// we don't have the username yet
NSString *hostString = @"http://2push2.us/apress/apress.php";
NSString *nameString = @"2Push2User";
NSString *argsString = @"%?token=%&cmd=reg&name=%";
NSString *getString = [NSString
stringWithFormat:argsString,hostString,tokenString,nameString];
NSString *registerResult = [NSString stringWithContentsOfURL:[NSURL
URLWithString:getString]];

NSLog( @"registerResult:%@", registerResult );

// display it in the field on the view controller
[self.viewController handleSetDeviceTokenField:tokenString];
}

- (void)application:(UIApplication *)application
didReceiveRemoteNotification:(NSDictionary *)userInfo
{
    NSLog( @"didReceiveRemoteNotification:%@", userInfo );
    [self.viewController handleDidReceiveRemoteNotification:userInfo];
}

```

NOTE: As you switch between development and ad hoc distribution versions of your application, be sure to clean out (or change) your device tokens table, because you can't use development device tokens on the production service!

Now, you're ready to resume your development and test where you left off in the server section earlier.

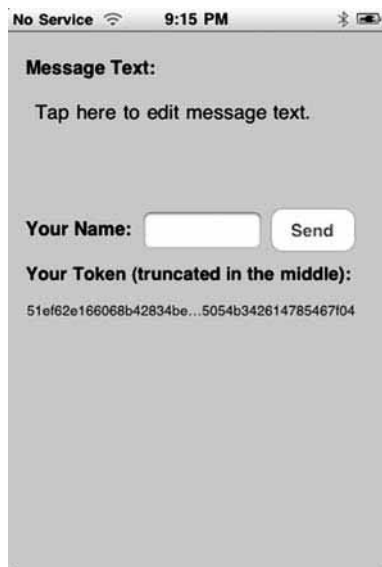


Figure 10-21. *The 2Push2 client application user interface*

Let's take a look at it in action.

Presuming you've handled the certificates, built the app, uploaded the server script, and done all the preliminary testing as suggested, you should finally be up and running, ready to test the app.

First, load the server script to your remote server and, while you're at it, the certificate (.pem) file. The server script is how the app will register its device token to receive notifications, so the server-side has to be in place before the client-side app is run.

Second, run the iPhone application. Optionally, install this on several devices right now to increase the fun!

Third, send notifications!

For convenience sake, when you now load the server script without any arguments, it will present you with the first device_token in the database instead of a placeholder, and thus you can copy and paste the "Send Message" example arguments and send a message right from your browser.

From here, there's plenty more you can do to build out this application or anything else you want to do with the Apple Push Notification Server.

I truly can't wait to see what you come up with!

Additional Considerations/Advanced Topics

Now that you have the basics handled, let's make sure to also touch on a couple more important points about using APNS.

Feedback Server

Even with everything that's already in place, there's still one last step to finish to be able to say that everything is complete.

The feedback server is an Apple-provided facility for determining whether there are any devices that have dropped from your service, usually because of uninstalling your app.

Apple requests that you periodically check the feedback server, get the device tokens of these dropped users that it provides you, and then discontinue sending to those tokens.

Jake Olefsky (<http://www.toodledo.com>) has posted a prototype feedback server script in PHP and with his permission I'm including it here. You can find the original at <https://devforums.apple.com/message/92559#92559>.

Here's how you do it:

```
<?php

function iPhoneGetUninstalledTokens() {
    global $certPassphrase;

    $ctx = stream_context_create();
    stream_context_set_option($ctx, 'ssl', 'local_cert', '/path/to/production.pem');
    stream_context_set_option($ctx, 'ssl', 'passphrase', $certPassphrase);
    $fp = stream_socket_client('ssl://feedback.push.apple.com:2196', $err, $errstr, 60,
    STREAM_CLIENT_CONNECT, $ctx); //for sandbox: feedback.sandbox.push.apple.com:2196

    if(!$fp) {
        echo "Failed to connect $err $errstr\n";
    } else {
        $contents = stream_get_contents($fp);
        if($contents){
            echo "Feedback Received";
        }else{
            echo "Failed to receive Feedback";
        }
    }

    fclose($fp);

    return $contents;
}

$data = iPhoneGetUninstalledTokens();

$tuples = strlen($data)/38;
```

```

for($i=0;$i<$tuples;$i++) {
    $offset = $i*38;

    $time = substr($data,$offset,4);
    $time = hexdec(bin2hex($time)); //unix timestamp

    $len = substr($data,$offset+4,2);
    $len = hexdec(bin2hex($len)); //always 32

    $token = substr($data,$offset+6,32);
    $token = bin2hex($token); //hex token

    echo $time." ".$len." ".$token;

    //put your removal code here
    // $q = "DELETE FROM apresschapter.apressdevices WHERE device_token = '$token'";
    // Execute MySQL Query and test result
}
?>

```

It's important to check the Feedback service regularly to update your database of invalid device tokens.

SSL Server Connections

Try to leave your connections open! Don't be opening/closing your socket to Apple frequently. You want to send as many notifications through on each connection as you can. If you get disconnected, simply reconnect and keep sending. Do not open a new connection for each notification, since the connection setup is a comparatively expensive operation and Apple may interpret your repeated connections as a type of denial-of-service attack.

Moving from Development Sandbox to Production

When you're ready to switch from development to distribution (either for ad hoc testing or for final App Store deployment), you'll repeat the same process of creating the certificate as you did in "Step 2: Create the Certificate" of this chapter, only this time, you'll create a distribution version of the SSL certificate instead of a development version.

After generating the Production Push certificate and provisioning file, the other key change you need to make is to switch from using the sandbox server to the production server at gateway.push.apple.com on the same port (2195). You'll need to keep track of your development and production certificates to make sure you're using the right version for deployment. You also need to make sure that you're not mixing device tokens from development with those for production. You'll use the production service for both Ad Hoc and App Store (or Enterprise) deployments.

Development vs. Ad Hoc

If you make the switch from development to distribution (ad hoc), and then your app immediately crashes when you launch it, you may need to restore your phone from scratch so that it can get into distribution mode. The “easiest” solution for this problem is to have multiple devices and keep them separated by mode (development or distribution); your second device can be an iPod Touch or a WiFi-only previous-generation phone that you kept after an upgrade. See the note in the “Back to the Portal” section earlier in this chapter for more information on this issue.

Mobile Provisioning Files

If you don't get notifications, try deleting your app, deleting the mobileprovision file from the device, and then reinstalling from Xcode. To delete your mobileprovision file, go to the Settings.app and then General ► Profiles.

If you get the following error:

```
2009-07-27 00:25:34.089 2Push2[98:207]
didFailToRegisterForRemoteNotificationsWithError:Error Domain=NSCocoaErrorDomain
Code=3000 UserInfo=0x12e3f0 "no valid 'aps-environment' entitlement string found for
application"
```

it probably means you don't have the correct mobileprovision file selected for development. Instead of one for an existing device, you should be using the new one created with the APNS app ID.

It's likely you'll switch between development and distribution modes during your development phase, and this can be tricky. Remember to make sure that the version of the iPhone application you are using (development or distribution) matches your remote SSL certificate and your tokens.

Debugging

The best resource for learning more about debugging APNS is session 120 from WWDC 2009, which you can get on ADC on iTunes. It's also the only way to get the inside story on the #squawk hashtag.

User Experience

This is a development chapter focused on implementing code, but do consider the user experience of push notifications. The user can easily turn off all notifications for your application in the Settings application, so be careful not to give them a reason to do so! Badges are the least intrusive option you can use; they're a simple visual indication that there is something new. Sounds can be a fantastic feedback mechanism for your users but can be easily overdone. Text alerts are possibly the most intrusive given that they need to be acted upon (dismissed or accepted) but can also convey the most targeted

information in a glance. Remember, too, that your application will be sharing your user's device with other applications that will be sending notifications.

Open Source Code

You'll find lots of open source code that you can use as the basis for your own server-side implementation; one of the best examples is PHP APNS available on Google Code:

<http://code.google.com/p/php-apns/>

This is a great example of a message queue-based daemon written in PHP using memcached and available under the LGPL license and including fully public domain code.

Hosted Solutions

You'll find quite a bit of information on the forums about hosting your own servers or using a third-party service.

If you have an enterprise-class APNS (or other server-related) project you need handled, I of course encourage you to call on Push IO. Drop me a line at joe@push.io.

Summary

I've covered a lot of material in this chapter, starting with the new methods in iPhone OS 3.0 that support push notifications, then the process of creating a server certificate to communicate with Apple's servers, server-side code in PHP for capturing device tokens and sending notifications, and, of course, a working client-side implementation on the iPhone.

Push notifications are one of the most exciting new features in iPhone OS 3.0 because they have the unique ability to keep your users informed about information they care about and encourage them to engage with your application, even while your app is not running.

Already a variety of innovative and practical services are being built around the Apple Push Notification Service, from letting you know what your friends are doing (Foursquare) to getting the latest real-time discussions (via Twitter clients) to reminding you of important events (Powerybase's NotifyMe application) to keeping you up-to-date on your favorite sports (as in the 91st PGA application).

I hope this chapter has given you everything you need to begin implementing push notifications in your own applications and that you'll send me examples of the cool things you do with this new capability.

Noel Llopis



Company: Snappy Touch

Location: San Diego, CA

Former Life As a Developer: *Cut my teeth many years ago on Z80 assembly and having to save programs on cassette tapes. Developed games professionally in just about every platform out there in the last ten years: Windows, PSX, Xbox, PS2, Xbox 360, and PS3. My main areas of expertise are game engines, computer graphics, and asset pipelines. Past games include:*

- *The Bourne Conspiracy (Xbox360, PS3) (2008)*
- *Darkwatch (PS2, Xbox) (2005)*
- *MechAssault 2 (Xbox) (2004)*
- *MechAssault (Xbox) (2002)*
- *Battleship: Surface Thunder (PC) (2000)*
- *Missile Command (PC, PSX) (1999)*

Life as an iPhone Developer:

- *Flower Garden. Games. Uses a mix of OpenGL and UIKit.*



- *Tea Time. Utilities. Part of my one-day-app-experiment.*

What's in This Chapter:

- ***OpenGL specular lighting***
- ***Introduction to environment mapping***
- ***Spherical environment mapping***
- ***Normal environment mapping***
- ***Environment map plus reflection mask to control reflection per pixel***

Key Technologies:

- ***OpenGL***
- ***Texture combiners***
- ***Multi-pass rendering***

Shine On: Environment Mapping and Reflections with OpenGL ES

The moment I first got my hands on an 8-bit computer, it was instant attraction on both sides, and it quickly blossomed into a love affair with game development that continues to this day. Since then, I've always kept up with the latest hardware at the forefront of games technology, pushing each new platform to its limits to get the most amazing graphics yet.

So, it was quite a change when I went from working on game console graphics to doing iPhone development full-time. It felt like I was trading a Formula 1 car for a scooter. The scooter was much slower than the car, but it was nimble, light, and maneuverable. It was a lot more fun to drive!

The Beginnings

Coming from the traditional AAA game console world of big-budget titles and loud explosions, I wanted to create a new experience on the iPhone. I wanted something that was creative and could be shared with friends, and I wanted something that fit the usage patterns of a mobile device and took full advantage of the device (touch input, accelerometer, Internet connection, and so on). That's how Flower Garden for the iPhone got started (see Figure 11-1).



Figure 11-1. *The final look of the flowers in Flower Garden*

The basic concept behind Flower Garden came together pretty quickly: the user could plant different types of seeds and water and care for them over time, and the seeds would grow into full plants and blossom. Then, the flowers could be cut, arranged into bouquets, and sent to anyone through e-mail and Facebook.

To get started, I learned all I could about flower morphology, created a PC prototype of the flower-growing technology, and ported everything to the iPhone. That whole process took about a month and a half. The rest was a matter of fleshing out the application: creating different seed types, coming up with an intuitive interface, cutting flowers and sending bouquets, and adding the plant-caring element.

At that point, the rendering of the flowers was very simple. I had decided not to go for a realistic look because the techniques I would need to apply just weren't possible on an iPhone without pixel shaders and powerful graphics hardware. So, instead I went for a fairly plain, almost illustrated look to the flowers using OpenGL ES 1.1, which is available in all models of the iPhone and iPod touch. The animation of the flowers swaying in the wind and reacting to the touch was very effective and really made those simple renderings come alive.

It was all coming along great, but something started bothering me: as I worked with the graphic designer on the look of the garden and the landscape in the background, the flowers started looking out of place. It's not that the flowers had gotten worse; they were still the simple, cartoony flowers they were at the beginning. But they were now surrounded by an almost photorealistic environment. It was like putting Mickey Mouse in a film noir picture, and the contrast was jarring. They just didn't belong there.

First Steps: OpenGL Lighting

This was an iPhone game, not a big console title, so I couldn't afford to waste a single day of development on something that wasn't going to have a big impact in the final program. Even worse, I didn't want to spend time on something that I wasn't sure was possible or something that wasn't going to look good in the end.

Even so, I knew that if I made those flowers look more realistic, they would not only fit much better in their surroundings but would make Flower Garden a much more graphically interesting application. It was worth a shot.

So, why did the flowers look so plain, and what could I do about it? At this point, the petals and leaves had no shading at all, just a texture used to blend between two different colors. The stem and the head of the flower, because they're more rounded and solid, were rendered with standard diffuse and ambient reflection.

Both ambient and diffuse reflection are standard lighting models in OpenGL ES 1.1. With ambient reflection, a surface is uniformly shaded depending on the light color and intensity. Diffuse reflection, on the other hand, changes the shading on the surface depending on the angle between the light direction and the surface normal. The following equation describes the color of a surface with diffuse reflection:

$$C_d = \max(0, L \cdot N) * C_l$$

C_d is the color of the diffuse reflection, L is the vector from the surface to the light, N is the surface normal, and C_l is the color of the light. Notice that the color specifically depends on the dot product between L and N , which means it's directly related to the cosine of the angle between those two vectors. When the dot product is negative (L and N are pointing away from each other), there is no diffuse reflection.

The following code creates an OpenGL light with diffuse properties and a global ambient light:

```
glEnable(GL_LIGHT0);
float diffuseColor[] = { 0.8f, 0.8f, 0.7f, 1.0f };
glLightfv(GL_LIGHT0, GL_DIFFUSE, diffuseColor);

float ambientColor[] = { 0.6f, 0.6f, 0.6f, 1.0f };
glLightModelfv(GL_LIGHT_MODEL_AMBIENT, ambientColor);
```

The sample program included with this chapter renders a 3D model of a car under different light conditions. You can build it, run it, and spin the object around to appreciate how the shading changes with the angle between the surface and the light. You can also toggle the different lighting approaches described in this chapter by pressing the top arrows. Figure 11-2 shows the program displaying a model lit with ambient and diffuse reflections.

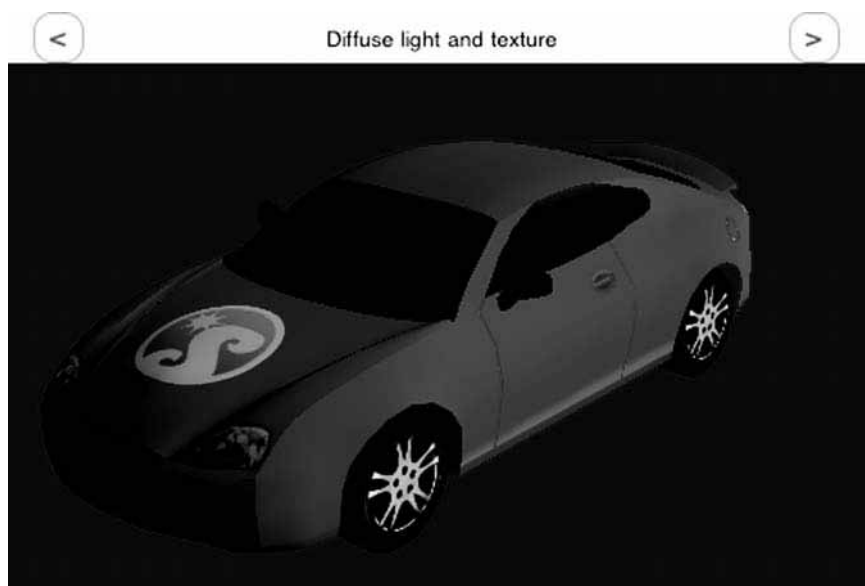


Figure 11-2. Model lit with standard, OpenGL ambient and diffuse reflection lighting

Looking at real plants and flowers, it's very obvious that there's a lot more to lighting than what I was doing. Plants are shinier, and petals are somewhat translucent, almost ethereal sometimes. Leaves almost sparkle, especially when they're wet, and they change as they move in the wind or you look at them from different angles. What I needed was some sort of specular highlights on the leaves that would really make them come alive.

Specular highlights are the bright spots that appear on shiny or wet surfaces when affected by a strong light. Look at the rippling surface of the water on a sunny day, and you'll see plenty of specular highlights.

With diffuse reflections, the shading on an object is completely determined by the cosine of the angle between the surface normal and the light. Specular lighting depends on the viewer position and adds an exponential drop-off factor, so the object is much brighter when the surface reflects the light straight into the viewer, and it drops off very quickly after that. That behavior is described in the following equation:

$$C_s = \max(0, s \cdot N)^{\text{shininess}} * C_s$$

Here, C_s is the specular color contribution, N is the surface normal, and C_s is the color of the specular light source. What's new here is s , which is an average of the vector from the surface to the viewer position and the vector from the surface to the light.

Fortunately, OpenGL ES 1.1 also includes a specular lighting model. The theory was that I should be able to turn that on and get all the sparkles I wanted. Before, when the petals were completely unlit, the vertices didn't need to contain a normal as part of their structure. Now, as you can see in the specular reflection equation, I needed that normal to compute the highlights, so I had to extend the vertex format to include them as well.

Finally, all I had left to do was to tweak the lighting parameters (primarily the shininess exponential factor) until I got the desired results.

The following code adds a specular color to an OpenGL light:

```
float specularColor[] = { 0.9f, 0.9f, 1.0f, 1.0f };  
glLightfv(GL_LIGHT0, GL_SPECULAR, specularColor);
```

The material also needs to have the specular parameters turned on, and the shininess is controlled from there:

```
float specularMatColor[] = { 1.0f, 1.0f, 1.0f, 1.0f };  
glMaterialfv(GL_FRONT, GL_SPECULAR, specularMatColor);  
float shininess[] = { 50.0f };  
glMaterialfv(GL_FRONT, GL_SHININESS, shininess);
```

Unfortunately, the standard specular lighting didn't end up looking as dazzling as I was hoping. I could even say it made things much worse. Figure 11-3 shows the sample program with specular lighting turned on.

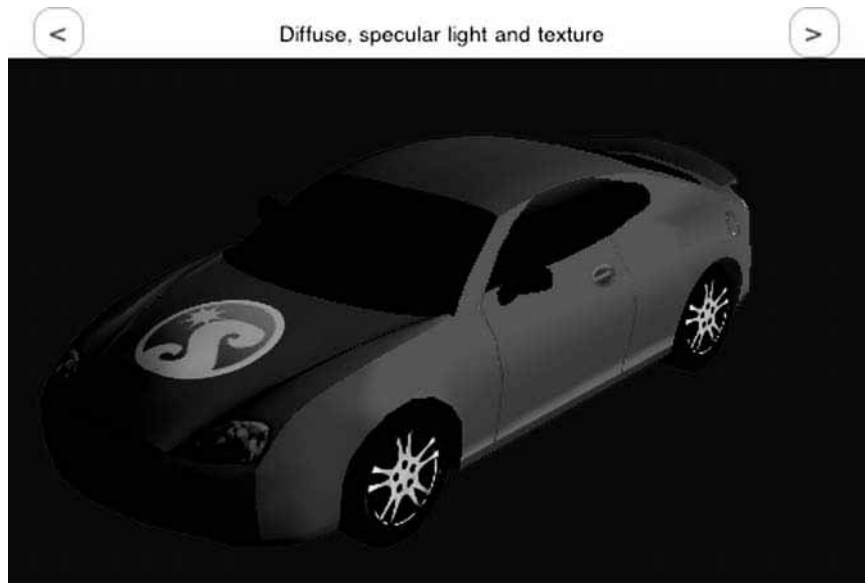


Figure 11-3. Model with per-vertex specular highlights. Not pretty!

The biggest problem with specular lighting on the iPhone is that it's computed at a vertex level. So, the highlight is calculated at each vertex and is then interpolated along the pixels between vertices. This means that if you try to make the highlight bright and sharp, you're going to end up with something that looks like smeared chalk across your mesh—that is, unless you have a mesh so dense that there's almost a vertex at each pixel, but that would be too much for the iPhone graphics hardware to handle, and performance would suffer.

Another problem is that doing the specular lighting calculation at each vertex isn't cheap. That's not a big deal because the frame rate was affected only a bit, but the

problem is that you get only one specular highlight per light source. Look at a shiny object again, and chances are you'll see multiple highlights for different light sources in the room you're in. If I wanted to add more highlights, I would need to add more lights to the scene, and that would again become prohibitively expensive.

This was clearly a dead end of a solution. I had to look for a different approach.

Turning to Environment Mapping

What I was trying to accomplish with specular lighting was to give the object some shiny spots where the light from the scene reflected off the surface. Environment mapping is a computer graphics technique I used in some of my past games to get similar shiny effects, so maybe I could use it here as well to good effect.

An *environment map* is a texture that contains information about the scene surrounding an object. This texture is then applied to a mesh in such a way as if it is reflecting the scene around it. You don't want to make the object a perfect mirror, but you can encode the bright spots from the lights around us in the environment map and then combine them somehow on the object to give the impression of shiny spots.

Environment mapping is a great technique that creates visually interesting scenes with very little overhead or extra work on the developer's part. One of the main drawbacks of environment mapping is that they capture the surrounding scene from a single location, so moving objects would have incorrect reflections, or the environment map itself would have to be recomputed in real time, which can be quite expensive. Fortunately, in the case of Flower Garden, the pot with flowers is at a fixed location. The camera can change positions, and the flowers can move in the wind, but they aren't changing positions enough to be a problem, so environment mapping seems like a perfect solution.

In many kinds of graphics hardware, this can be a really easy solution. All you have to do is provide an environment map, turn on the environment-mapping mode, and off you go. Unfortunately, there was a snag in my plan: the iPhone hardware doesn't support environment mapping.

All is not lost, though. The hardware might not do it automatically for you, but that doesn't mean you can't roll up your sleeves and, with some extra work, do it yourself. To accomplish that, you need to take a close look at the math behind environment mapping.

Capturing the scene around you on a single, flat texture is a challenging task. It's very similar to the problem of trying to create an accurate projection of the surface of the globe of the earth onto a piece of paper (except that in this case you see the scene from inside the sphere, not from the outside). No matter what approach you take, the scene is going to have some amount of distortion. That might be a problem if you're trying to render a smoothly, reflecting mirror ball, but all I was going to use it for was to add some shiny spots on the leaves, so accuracy was not a goal.

In computer graphics, there are two common techniques for mapping a scene onto a texture: spherical environment mapping and cube environment mapping. In this case, you'll map the scene surrounding you as mapped on a sphere centered at your location. In the other case, the scene will be mapped into each of the six sides of a cube centered at your location. For this implementation, I went with spherical mapping because it is a bit faster to compute, and the resulting quality is plenty for these needs.

Spherical Environment Mapping Implementation

Assume that the lights around your scene are not changing, so you can create a spherical environment map offline and use it every frame during your rendering. Figure 11-4 shows an example of a spherical environment map. Notice that the object itself is usually not part of the environment map, just the objects and lights surrounding it.



Figure 11-4. Sample spherical environment map

Ultimately, what you want is to generate a second set of texture coordinates that indexes into the right place in the environment map based on the eye position and on the surface normal. If the surface were a perfect mirror, what would you see? To find out, you could cast a ray from the eye to the surface and then bounce it off the surface taking the normal into account. This new vector is called a *reflection vector* (see Figure 11-5).

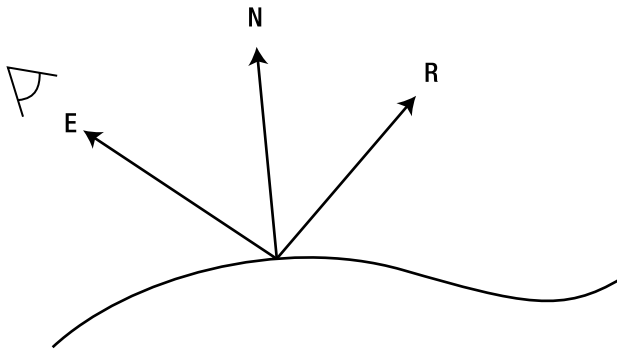


Figure 11-5. Reflection vector

Mathematically, a reflection vector is computed with the following formula:

$$\mathbf{R} = \mathbf{E} - 2(\mathbf{E} \cdot \mathbf{N}) \mathbf{N}$$

\mathbf{E} is the unit eye vector, and \mathbf{N} is the unit normal vector. The incident vector is the vector from the eye position to the point on the surface of the model. Because the incident vector changes for each point on the mesh, it needs to be recomputed for each vertex. Also, since the reflection vector depends on the eye position, it will change as the camera moves around the object (or the object changes position relative to the camera), so you need to recompute it every frame.

Once you have this reflection vector, you can apply a formula to index into your environment map. This is the standard formula for spherical environment mapping:

$$m = 2 \sqrt{rx^2 + ry^2 + (rx+1)^2}$$

$$u = \frac{rx}{m} + \frac{1}{2}$$

$$v = \frac{ry}{m} + \frac{1}{2}$$

That's quite a bit of work to do for each vertex, especially with those square roots on the iPhone CPU. Fortunately, you can make some approximations again that will let you speed things up significantly.

The key observation is that you could “fake” the reflection vector by simply using the normal vector at each vertex. The reflection wouldn't be physically accurate, but it would reflect something, which is good enough. That by itself isn't enough, though: if you use only the normal vector, the reflection wouldn't change as the camera moves around the object or the object changes position. To fix that, you need to rotate the vertex normal by the camera transform. Finally, you discard the z component of the normal and scale and bias the x and y components so they are between 0 and 1. The results are the texture coordinates corresponding to the environment mapping as seen from the correct viewpoint. This technique is sometimes referred to as *normal environment mapping*, because it uses the normals of the model instead of the true reflection vectors.

The normal environment mapping equation is shown here. F is the world to view the “forward” vector, and U is the world to view the “up” vector.

$$[rstq] = \begin{bmatrix} 0.5 * Fx & -0.5 * Ux & 0 & 0 \\ 0.5 * Fy & -0.5 * Uy & 0 & 0 \\ 0.5 * Fz & -0.5 * Uz & 0 & 0 \\ 0.5 & 0.5 & 0 & 0 \end{bmatrix}$$

That still seems like quite a bit of work. Why go to all that trouble instead of using the reflection vector? Well, now you can perform that computation on the GPU, which is much better suited than the CPU for that type of work. To accomplish that, you need to change your vertex format to include all three parameters for the normal vector, load the OpenGL texture transform with the camera transform, and apply it to the second set of texture coordinates.

The new vertex format becomes as follows:

```
struct NewVertex
{
    float x, y, z;
    float nx, ny, nz;
    float u0, v0;
    float u1, v1, t1;
};
```

The following code sets up the texture transform:

```
glMatrixMode(GL_TEXTURE);
float mat[] = {
    0.5f * worldToView[0].x, -0.5f * worldToView[1].x, 0, 0,
    0.5f * worldToView[0].y, -0.5f * worldToView[1].y, 0, 0,
    0.5f * worldToView[0].z, -0.5f * worldToView[1].z, 0, 0,
    0.5f, 0.5f, 0, 1};
glLoadMatrixf(mat);
```

Figure 11-6 shows the results of applying spherical environment mapping to the model. The results are even good enough to make it into an almost perfect mirror.

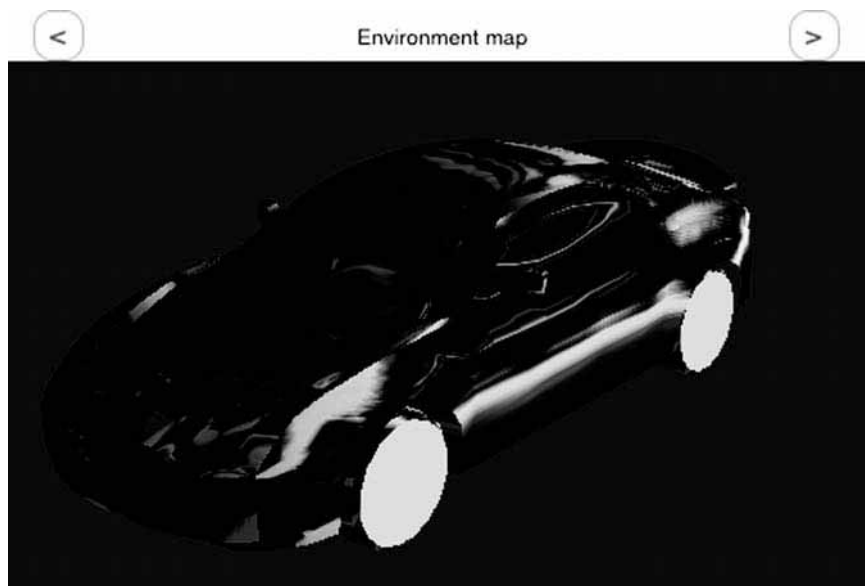


Figure 11-6. *Model with fully reflective, spherical environment mapping*

Combining Environment Mapping and Diffuse Textures

When I started out, my goal was not to make mirror-like objects but to add shiny, reflective areas to petals and leaves. To do that, you need to combine the diffuse texture with the environment mapping you computed in the previous section.

Up until now, it's possible that you just used one texture unit on the iPhone. But to add environment mapping to an object, you're going to have to load both texture units available on the iPhone and create the final result by using texture combiners.

Like most things with OpenGL, the current texture unit is determined by a state. By default that unit is set to the first one, so all texture operations apply to that unit. The following code loads up two textures, one in each texture unit:

```
glActiveTexture(GL_TEXTURE0);  
glEnable(GL_TEXTURE_2D);  
glBindTexture(GL_TEXTURE_2D, diffuseTexture);  
glActiveTexture(GL_TEXTURE1);  
glEnable(GL_TEXTURE_2D);  
glBindTexture(GL_TEXTURE_2D, environmentMap);
```

This vertex type has now two sets of texture coordinates, one for each texture. That way, you can index them independently of each other.

Finally, now that you have both textures loaded, you can decide which part of each texture to display for each vertex. But what should the final result look like? How should those textures be combined to create the final color? The answer lies in the texture combiners.

A *texture combiner* allows you to combine the parameters that contribute to the color of each pixel. The iPhone has two texture combiners, and each combiner can operate on at most three inputs. Those inputs can be the color read from a texture, a constant color you set yourself, or the results from a previous combiner. The actual operations they can perform are restricted to simple combinations of additions and multiplications. They're listed in Table 11-1.

Table 11-1. *Texture Combiner Operations*

GL_COMBINE	Function
GL_REPLACE	Arg0
GL_MODULATE	$\text{Arg0} * \text{Arg1}$
GL_ADD	$\text{Arg0} + \text{Arg1}$
GL_ADD_SIGNED	$\text{Arg0} + \text{Arg1} - 0.5$
GL_INTERPOLATE	$\text{Arg0} * (\text{Arg2}) + \text{Arg1} * (1 - \text{Arg2})$
GL_SUBTRACT	$\text{Arg0} - \text{Arg1}$
GL_DOT3_RGB	$4 * ((\text{Arg0}_r - 0.5) * (\text{Arg1}_r - 0.5) + (\text{Arg0}_g - 0.5) * (\text{Arg1}_g - 0.5) + (\text{Arg0}_b - 0.5) * (\text{Arg1}_b - 0.5))$

When working with texture combiners, I find it easier to visualize them instead of thinking of them in terms of mathematical operations. For example, Figure 11-7 shows the setup to display an object with a diffuse texture and some amount of reflection added on top.

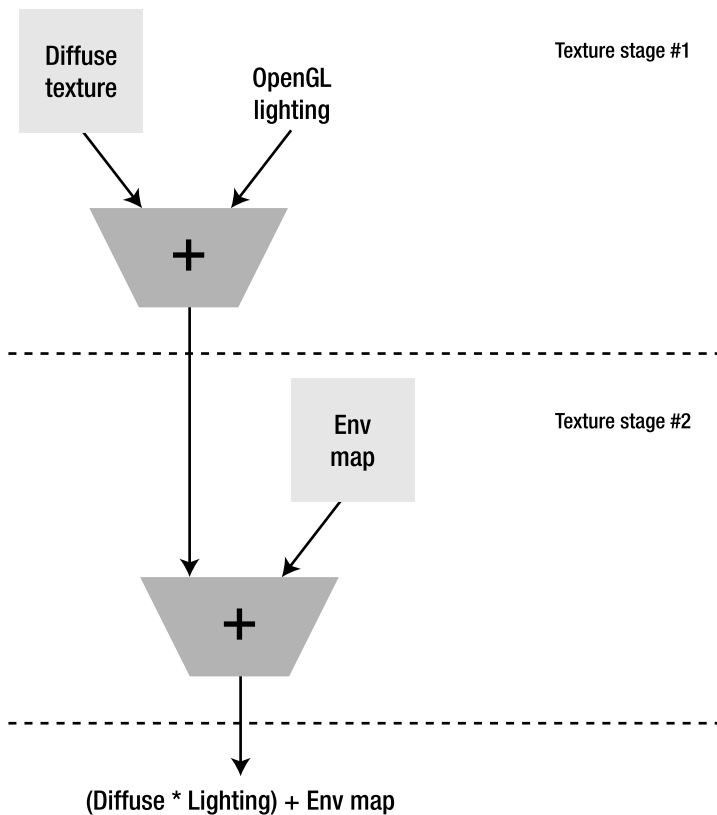


Figure 11-7. Texture combiners set up to add an environment map on top of a diffuse texture

The following is the code to set up OpenGL in that state:

```
glActiveTexture(GL_TEXTURE0);
glTexEnv(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_COMBINE);
glTexEnv(GL_TEXTURE_ENV, GL_COMBINE_RGB, GL_MODULATE);
glTexEnv(GL_TEXTURE_ENV, GL_SRC0_RGB, GL_TEXTURE);
glTexEnv(GL_TEXTURE_ENV, GL_SRC1_RGB, GL_PRIMARY_COLOR);
glTexEnv(GL_TEXTURE_ENV, GL_OPERAND0_RGB, GL_SRC_COLOR);
glTexEnv(GL_TEXTURE_ENV, GL_OPERAND1_RGB, GL_SRC_COLOR);

glActiveTexture(GL_TEXTURE1);
glTexEnv(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_COMBINE);
glTexEnv(GL_TEXTURE_ENV, GL_COMBINE_RGB, GL_ADD);
glTexEnv(GL_TEXTURE_ENV, GL_SRC0_RGB, GL_TEXTURE);
glTexEnv(GL_TEXTURE_ENV, GL_SRC1_RGB, GL_PREVIOUS);
glTexEnv(GL_TEXTURE_ENV, GL_OPERAND0_RGB, GL_SRC_COLOR);
glTexEnv(GL_TEXTURE_ENV, GL_OPERAND1_RGB, GL_SRC_COLOR);
```

Which one do you find easier to understand, the diagram or the OpenGL code? I thought so.

Figure 11-8 shows the results of adding the environment map on top of the diffuse texture.

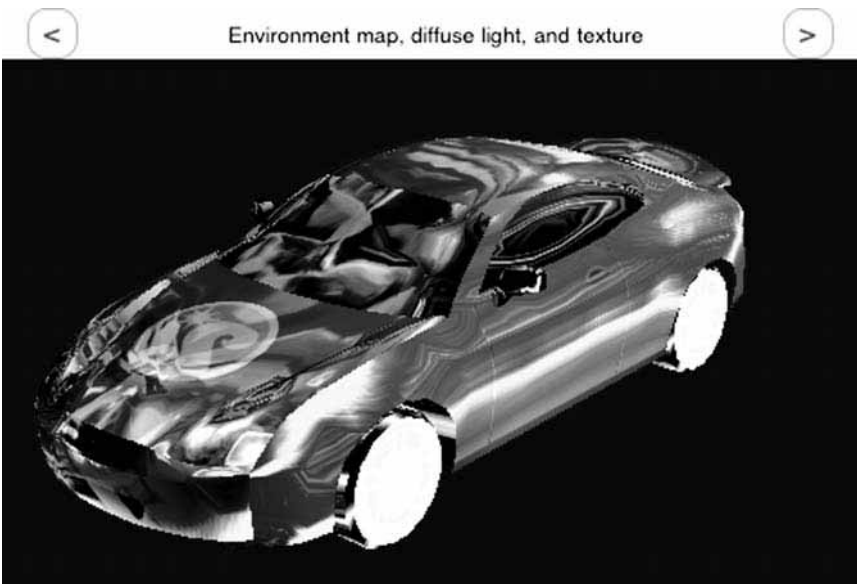


Figure 11-8. Model rendered with an environment map added to the diffuse texture

Per-Pixel Reflections

At this point, you have objects with a diffuse texture and an environment map applied on top of it. This is already a huge improvement over objects without environment maps at all. However, when I implemented this technique with the plants, the leaves were not shiny enough. They were a bit reflective, but they didn't have shiny spots that stood out, which is the effect I originally set out to find.

The reason for this is that the environment map is applied uniformly to the whole object. Some places in the environment map will be brighter than others, but the surface of the object is equally reflective everywhere. Look at the objects around you, especially if there are any wet ones, and you'll quickly see that some parts are more reflective than others. For example, dry spots are not very reflective, and the fleshy parts of a leaf are much more reflective than the stem.

Before we go any further, let's look at the equation that determines how you're rendering each pixel of a model with the environment map technique so far:

$$C = L * T + E$$

Here, C is the final pixel color, L is all the incident light colors, T is the value of the texture at that point, and E is the color contribution from the environment map reflected at that point.

You can implement different amounts of reflection with a specular mask: a grayscale texture that is white where the object is fully reflective and is black where it doesn't reflect at all. Intermediate gray values indicate how shiny it is at each spot. All you have

to do is multiply this specular mask with the environment map, and you'll get variable amounts of reflection or shininess.

The new rendering equation looks like this:

$$C = L * T + E * M$$

M is the specular mask. Simple enough? Unfortunately, there's a wrinkle in our plans. You have three textures: T, E, and M. But the iPhone 3G has only two texture units!

One common solution is to combine the specular mask into the alpha channel of the diffuse texture. That will keep the number of textures down to two so you can fit everything in a single pass. However, sometimes the alpha channel will already be in use to indicate transparency. It is also possible that a texture combiner is already in use if you're doing something more than a plain diffuse texture. In the case of Flower Garden, petals were rendered by interpolating two colors based on another texture, so a full texture combiner was used up, and I couldn't fit that plus the masked environment map.

When that happens, the solution is to break up the rendering into two passes. The first pass draws the model with the original, diffuse texture, and the second pass draws the same polygons again but adds the environment map multiplied by the specular texture. This makes the rendering significantly more expensive, but the final effect is well worth it. Also, this frees one texture unit during the first pass, so you can use it as a detail map, light map, or any other technique that requires an extra texture.

The new rendering equation is the following (each pass is in parentheses):

$$C = (L * T) + (E * M)$$

Figure 11-9 shows the new texture combiner setup.

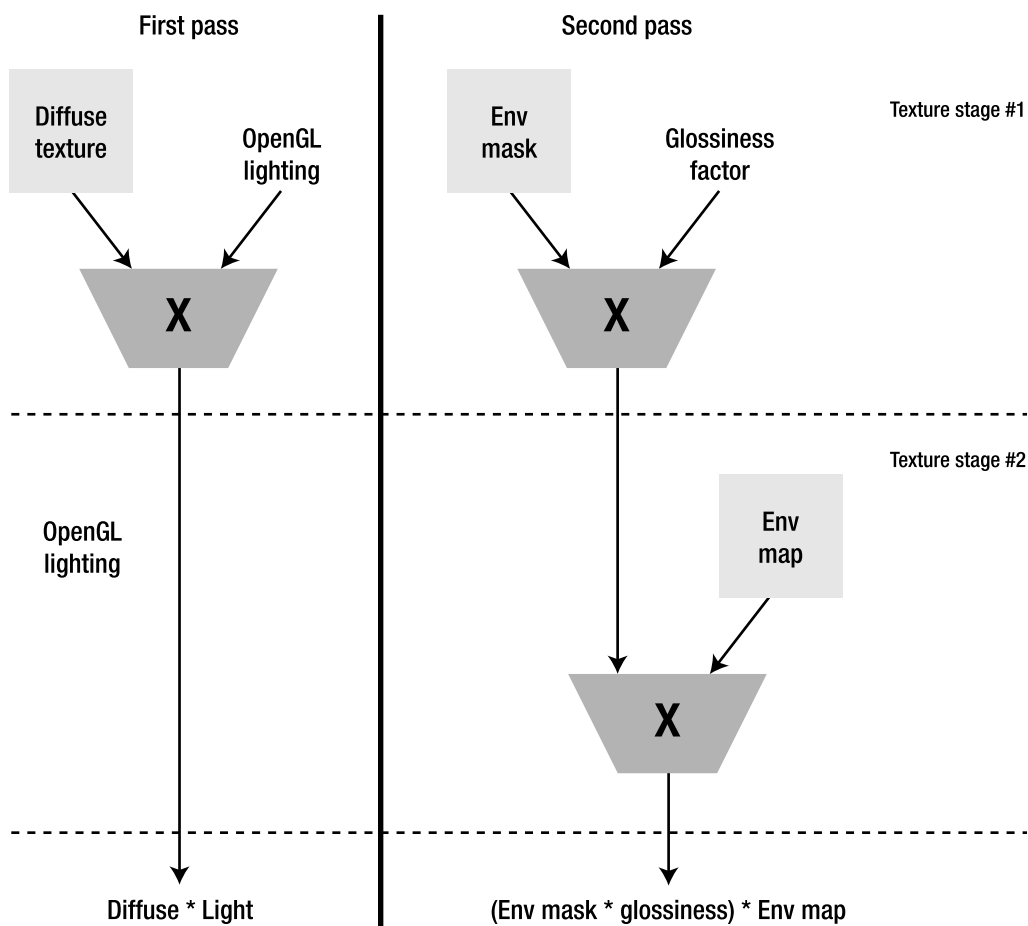


Figure 11-9. Texture combiner setup for a two-pass rendering. The first pass has lighting plus a diffuse texture, and the second pass adds an environment map modulated by a mask.

When you do any kind of two-pass rendering, you have to be careful how you draw the triangles to avoid any kind of *z-fighting* (rendering artifacts caused by rendering two polygons on top of each other). You should always render the same geometry in both passes, and the second pass should use the depth-test function `GL_EQUAL` to render only those pixels that match the depth currently on-screen.

```
glDepthFunc(GL_EQUAL);
```

Figure 11-10 shows the model rendered with this technique. Notice how some parts are more reflective than others.

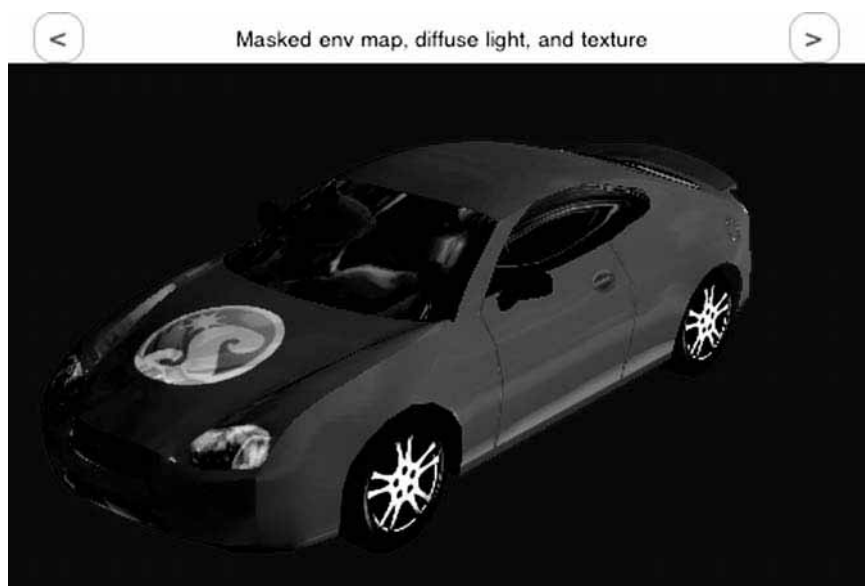


Figure 11-10. Model rendered in two passes, with lighting plus a diffuse texture in the first and an environment map with a mask on the second

iPhone 3GS

Per-pixel environment mapping is still a valid technique on the iPhone 3GS. Everything I've covered will run on the 3GS and will run faster because of its more powerful graphics hardware. But you can go beyond this and implement it in a simpler and even faster way to take full advantage of the new hardware.

To start with, the iPhone 3GS supports OpenGL ES 2.0, which means you have a fully programmable shader pipeline. That means you don't have to deal with texture combiners anymore, and you can write your pixel-rendering equation directly on a fragment shader.

It also means you can perform true spherical environment mapping, computing the correct reflection vector at each vertex (or pixel).

Finally, you can take advantage of the fact that the 3GS has eight texture units and do the whole rendering in a single pass. Because of this, per-pixel reflections on the 3GS will run much faster than on the original 3G. The following is the vertex and shader code for a per-pixel environment map implemented with OpenGL ES 2.0:

```
// Vertex shader
uniform mat4 u_mvpMatrix;
uniform mat3 u_cameraTransform;
uniform vec4 u_ambientLightColor;
uniform vec4 u_directionalLightColor;
uniform vec3 u_directionalLightDir;

attribute vec4 a_position;
```

```

attribute vec3 a_normal;
attribute vec2 a_texCoord;

varying vec2 v_texCoord0;
varying vec2 v_texCoord1;
varying vec4 v_lighting;
void main()
{
    gl_Position = u_mvMatrix * a_position;
    v_texCoord0 = a_texCoord;

    vec3 transNormal = u_cameraTransform * a_normal;
    v_texCoord1 = vec2(transNormal);

    float ndotl = max(0.0, dot(a_normal, u_directionalLightDir));
    v_lighting = u_ambientLightColor + u_directionalLightColor*ndotl;
}

// Fragment shader
precision mediump float;
uniform sampler2D u_diffuseMap;
uniform sampler2D u_envMap;
uniform sampler2D u_envMask;
uniform float u_shininess;

varying vec2 v_texCoord0;
varying vec2 v_texCoord1;
varying vec4 v_lighting;
void main()
{
    vec4 diffuseColor = texture2D(u_diffuseMap, v_texCoord0);
    vec4 envMapColor = texture2D(u_envMap, v_texCoord1);
    vec4 envMaskColor = texture2D(u_envMask, v_texCoord1);
    gl_FragColor = diffuseColor*v_lighting + (envMapColor * envMaskColor) *
u_shininess;
}

```

Summary

It was a long road all the way from having the initial vision of how I wanted the flowers to look until it was implemented and running at a good frame rate. The results show the effort I put into it are definitely worth it, and the shiny look of the flowers was definitely one of the reasons people liked Flower Garden.

Environment mapping is a very flexible technique that produces impressive results with very little extra effort. The whole scene around an object is captured in a single environment map and applied to the object depending on the viewer's position. The use of normal environment mapping uses the normal at the surface instead of the reflection vector, allowing you to move expensive computations from the CPU to the graphics

hardware, making it even faster. Finally, applying a reflection mask to the environment map creates much more realistic images by specifying the amount of reflection in different areas.

Applying environment mapping transforms any scene and makes it stand out. It's definitely a tool that graphics programmers should have in their toolboxes.

Index

A

- AAPLot, 280–292
 - caching logic, 283
 - data persistence, 284–285
 - extending, 288–292
 - placeholder data for, 286–288
 - plotting historical stock prices with, 280–282
 - storing data between runs, 283
- AAPLotViewController class, 280–281
- AccountViewController class, 225, 229, 244
- AccountViewController.h file, 224
- AccountViewController.m file, 224, 228
- Active Record, 144
 - finder methods, 171–174
 - mapping layer, writing, 164–175
- addAttribute method, 202
- address family, 257
- AddressBook framework, 230
- addVertex method, 33
- Adobe Photoshop Lightroom project, 141
- AF_INET, 257
- AF_INET6, 257
- afconvert command-line tool, 318
- ambient reflection, 349
- animate:or flag, 31
- .app file, 112
- app ID, 321
- App IDs page, 321
- APP_STORE_FINAL, 106–107
- app.yaml, 48–49
- Apple Developer Program, 314
- Apple Push Notification Server (APNS), 43, 313–344
 - advanced topics, 341–344
 - advantages of, 314
 - application delegate, 315–317
 - building and deploying, 318
 - certificate creation, 319–331
 - client application, 336–340
 - client creation, 314–319
 - debugging, 343
 - defined, 314
 - enabling application for, 321
 - feedback server, 341–342
 - handling incoming notifications, 317
 - hosted solutions, 344
 - mobile provisioning files, 343
 - moving from development to distribution, 342–343
 - open source code, 344
 - setting up server, 331–335
 - sounds, 318
 - SSL server connections and, 342
 - user experience and, 343
- application delegate (AppDelegate), 233
- application object, 52
- applicationDidFinishLaunching method, 177, 217, 316
- applicationWillResignActive method, 270
- aps_developer_identity.cer file, 325
- APYahooDataPuller, 280–285, 289
- APYahooDataPullerDelegate, 281
- AQOutputCallback, 77
- asserts
 - built-in, 105
 - custom, 105–107
- asynchronous calls, 57, 58
- asynchronous data downloading, 72
- atos command, 111–112
- audio buffer pool, 83
- audio buffers, 83
- audio data callback, 81, 87–88
- audio files, 93
- Audio File Stream Services, 81–92
- audio format property, 87
- audio playback/streaming, 65–97
 - app for, 93–96
 - audio format and, 94–95
 - AudioServicesPlaySystemSound(), 69
 - AVAudioPlayer class, 69–72
 - determining buffer size, 79–80
 - iPhone OS 3.0 and, 96
 - MPMoviePlayerController class, 66–68

- stopping, 80
- with Core Audio, 74–93
 - Audio File Stream Services, 81–92
 - Audio Queue Services, 74–80
- Audio Queue Services, 74–80
 - callback function, 79
 - class declaration using, 76
 - playback cycle, 75
 - starting audio playback, 77–79
 - stopping playback, 80
- Audio Session API, 92–93
- audio streaming. *See* audio playback/streaming
- audio streams, 95
- audioDataCallback() method, 87–89
- AudioFileClose() method, 80
- AudioFileOpenURL() method, 76
- AudioFileReadPackets(), 79–80
- AudioFileStreamOpen() method, 85
- AudioFileStreamParseBytes() method, 81, 87, 91
- AudioQueueBufferRef class, 83
- AudioQueueEnqueueBuffer() method, 90
- AudioQueueNewOutput() method, 77, 79, 87
- AudioQueueStart() method, 77
- AudioQueueStop() method, 80
- AudioServicesPlaySystemSound() method, 69
- AudioStreamBasicDescription, 95
- Aurora Feint, 7
- autodiscovery
 - Bonjour for, 271–272
 - custom solution for, in iTunes, 273–275
- automatic migrations, 194
- AVAudioPlayer class, 69–72
- AVAudioSession class, 96

B

- beforeDelete function, 171
- beforeSave callback, 169
- big endian, 259
- bind() function, 260
- bindArguments:parameters:toStatement: method, 161–162
- blocks, 293–297
- Bluetooth
 - support for, 253
 - third-party applications and, 253–254
- \$body['aps'] associative array, 332
- Bonjour, 271–272
- breakpoints, on malloc_error_break, 123–125
- BSD networking API, 264–269
- BSD sockets, 255, 261–263
- buffer overruns, 119

- buffer sizes, 77, 79–80, 83
- buffering, on mobile devices, 84
- buffers
 - audio, 83, 88–92
 - preallocating, 82
- build target, 106
- byte ordering, 259

C

- C API, creating wrapper around, 144–157
- C language, 75, 104
- C++
 - asserts in, 105
 - vs. Objective-C, 104
- C++ class, creating, 116–118
- Caches directory, 285
- call stacks, 115
- callbacks, 58
- C-based Core Foundation framework, 255
- CBR (constant bit rate), 77
- certificate signing request (CSR), 322–324
- CERTIFICATE_FILENAME, 331–333
- CFBundleURLName string, 54
- CFSHOW, 107
- CFSocket, 255, 262–263
- CFSocketConnectToAddress() function, 262
- CFSocketContext, 263
- CFSocketCreateWithNative() function, 262
- CFSocketGetNative() function, 262
- cgContext, 305–309
- CGContextRefs, 298
- closures, 293
- Cocoa, 41
- Cocoa data types, 60
- code
 - #define, 108–109
 - custom asserts, 105–107
 - custom logging, 107–108
 - debugging, 105–109, 115–116
 - separating data representation and, 56, 57
- color-changing particles, 28–35
- column names, 166–167
- column types, 154–155
- columnTypeToInt, 154
- components, communication between using
 - notifications, 272–273
- concurrency, 292–297
- connect() function, 260
- connection:didFailWithError: method, 58–59
- connection:didReceiveData: method, 58–59, 85
- connection:didReceiveResponse: method, 58–59

- connectionDidFinishLoading: method, 58–59
- connectivity issues, 212
- constant bit rate (CBR), 77
- Contact class, 232
- copyValuesFromStatement:toRow:queryInfo:columnTypes:columnNames: function, 172
- Core Animation, 3
- Core Audio, 66, 74–93
 - Audio File Stream Services, 81–92
 - Audio Queue Services, 74–80
 - on iPhone Simulator, 75
- Core Data, 179, 183–207, 213
 - application development, 185–194
 - classes, 185
 - creating reusable objects, 199–203
 - history of, 184–185
 - model migrations, 194–198
 - model, creating, 235–238
 - remote databases and, 203–206
 - stack objects, 221
 - tutorial, 186, 189, 193–198
 - working with, 218–224
- Core Foundation data types, 263
- Core Plot, 281, 292
- correspondence chess, 39–61
 - accepting invitations, 43, 54–57
 - coding, 47–60
 - Google App Engine and, 44–47
 - inviting friend to game, 42–43, 50–53
 - making moves, 43, 57–60
 - notifications, 43
 - tasks, 42–44
- crash logs
 - from rarely occurring crashes, 112
 - from testers, 109–110
 - symbolicating, 110–111
- crashes, 109–115
 - hypothesis about, 113–115
 - increasing probability of, 115
 - reproducing, 112–113
 - sudden, 122
 - testing, 114–115
- CREATE statements, 155, 158–159
- create, read, update, and delete (CRUD) operations, 174
- CSR (certificate signing request), 322–324
- cstrings, 108
- currentBufferIndex, 90–92
- custom asserts, 105–107
- custom logging, 107–108
- custom URL scheme, in Deep Green, 43

D

- Dapple, 101, 104, 123
- data
 - displaying large amounts of, 298–299
 - downloading, with `NSURLConnection`, 72–73
 - managing, with `DataManager`, 226–228
 - placeholder, 286–288
 - saving, to iPhone application sandbox, 285–286
 - sending and receiving, 260
 - separating code and, 56–57
- data access, with `Active Record`, 144
- data applications, 183. *See also* Core Data
- data migrations, 194–198
- data model, adding new entity to, 197
- data persistence, with plists, 284–285
- data sources, 143
- data storage, 143
- database connections
 - maintaining, 165
 - opening, 149–151
- databases
 - creation and initialization, 148–149
 - deleting objects, 170–171
 - lookups, 171–174
 - making simple requests, 152–157
 - migration handling, 176–179
 - preventing duplicate create statements, 158–159
 - remote, 203–206
 - saving objects, 168–170
 - updating objects, 170
- data-driven applications, 141–180
 - `Active Record` layer, writing, 164–175
 - alternative implementations, 179
 - database creation and initialization, 148–149
 - migration handling, 176–179
 - opening database connection, 149–151
 - setting up, 145–147
- datagram-oriented connections, 260
- datagrams, 256
- `DataManager` class, 225–228, 232
- `dataPullerDidFinishFetch()` method, 281
- datastores, replicating, 45
- date conversion, 57
- db module, 52
- `dealloc` function, 147, 150–151
- debug symbols, 110
- debugging, 101–137
 - APNS, 343
 - `atos` command and, 111–112
 - background knowledge for, 102–104
 - code, 115–116
 - crashes, 109–113

- custom asserts, 105–107
- custom logging, 107–108
- .dSYM files and, 110
 - with link map files, 135–137
- memory stomps, 118–137
- scientific method of, 113–115
- symbolicatecrash script for, 110–111
- techniques for, 115–137
- tools, 123–131
- turning on/off, 108
- using #define, 108–109, 114
- using variable watch, 131–135
- while writing code, 105–109

Deep Green

- correspondence chess, 39–61
 - accepting invitations, 43, 54–57
 - coding, 47–60
 - Google App Engine and, 44–47
 - inviting friend to game, 42–43, 50–53
 - making moves, 43, 57–60
 - notifications, 43
 - tasks, 42–44
- development of, 41
- home page, 41
- on Newton platform, 41
- user interface, 40–42

DEFERRED transaction type, 164

#define, 108–109, 114

degenerate triangles, 15–16

delete statement, 170

deleted methods, calling, 119–120

deleted objects

- calling method on, 120
- returning, 120–122

DeriveBufferSize() function, 77, 83

Development Provisioning Profiles list, 328

Development Push SSL Certificate option, 322

device token acquisition, 316

devices, contacting all, on network, 267–268

+dictionaryWithContentsOfURL:, 55–57

didFinishLaunchingWithOptions method, 316

didRegisterForRemoteNotificationsWithDeviceToken

- method, 332

diffuse reflection, 349–350

diffuse textures, environment mapping and, 356–363

Discover application, 272

Django templates, 60

DMG file, 254

DNS-SD protocol, 271

Documents directory, 285

documentsDirectory variable, 286

drawing, into off-screen context, 304–309

.dSYM files, 110

@dynamic directive, 237

E

EAGAIN error, 261

EAGLView, 8

e-mail, in-application, 211–245

- composing and sending messages, 230–235
- Core Data for, 218–221, 235–238
- DataManager, 226–228–230
- switching to online mode, 241–44
- threaded messages, 239
- Three20 and, 221–224, 228
- user interface, 211–218, 224–226

e-mail messages

- composing and sending, offline, 230–235
- threaded, sending, 239

emailInvocationOperation: selector, 235, 237

embedded systems programming, 309

emissionRange, 20

emit button, 23

emitCounter, 20

emitter, 11

- adding force, 25–27
- building, 13–14
- explosion, 21–23
- particle emitting, 20–21
- update loop for, 17

Enable Guard Malloc, 129–131

ENABLE_DATA_STRUCTURE_DEBUG_LOGS, 108

encryption, 206, 319

enqueueCurrentBuffer() function, 88–91

ENTER_DEVICE_TOKEN_HERE, 331

Enterprise Object Framework (EOF), 184

entities, 197, 236

EntryForm class, 52

EntryForm object, 52

environment mapping, 352–364

- diffuse textures and, 356–363
- normal, 354
- per-pixel, 359–363
- spherical, 353–356, 362

error codes, 150–152

errors

- See also* debugging
- scoping, 106

Event class, 187–190, 197

EventExtra entity, 199–200

eventsArray ivar, 186

EventViewController class, 189–191

EventViewController.m file, 202

EXCLUSIVE transaction type, 164

executeSql: method, 153, 155, 162, 170

executeSql:withParameters: method, 162, 172

explosion particle emitter, 21–23

Expressions window, 103–104

F

F_SETFL command, 261
 fcntl() system call, 261
 feedback server, 341–342
 fetchIfNeeded method, 295
 FileMagnet, 286
 fileTypeHint property, 95
 finder class methods, 165
 finder methods, 171–174
 Flower Garden
 development of, 347–349
 environment mapping, 353–363
 lighting, 349–352
 FMDB, 179, 213–214
 ForceBufferOverrun() method, 132
 Foundation framework, 234

G

game keys, 60
 Game model class, 52
 game.plist template file, 56
 GameController class, 50, 53
 GameTypes, 8
 gaming, correspondence, 39–61
 accepting invitations, 43, 54–57
 coding, 47–60
 Google App Engine and, 44–47
 inviting friend to game, 42–43, 50–53
 making moves, 43, 57–60
 notifications, 43
 tasks, 42–44
 GET requests, 50
 get() method, 50, 53, 56
 getNewMessageID, 237
 -getResultSetFromQueue method, 243
 GL_COLOR_ARRAY, 28
 GL_TRIANGLE_STRIP method, 15
 GL_TRIANGLES method, 15
 gogoDocs application, 283–284, 288
 Google App Engine (GAE), 44–47
 webapp module, 56
 Google Docs service, 283
 GoogleAppEngineLauncher, 48–49

H

handleOpenURL: method, 55–56
 handlers, 50
 Hewitt, Joe, 213
 htonl() function, 259
 htons() function, 259
 HTTPS connections, 206
 hypothesis, about crash, 113–115

I

ifreq structures, 265
 IMMEDIATE transaction type, 163
 INADDR_ANY, 260
 index.yaml, 48
 Info.plist file, 54
 init functions, 148–150
 initWithRecipients: method, 228
 InputViewController, 8
 insert, 169–170
 instance variables, 215–217
 Instapaper, 212
 interface flags, 266
 interface responsiveness
 AAPLot, 280–292
 concurrency and, 292–297
 displaying large amounts of data, 298–299
 drawing into off-screen context, 304–309
 expectations of, 280
 interfaceNamesAddresses, 268
 interpolation, 28–30
 IO controls, 264–267
 network-related, 267
 SIOCGIFCONF, 264–266
 SIOCGIFFLAGS, 266
 ioctl() function, 264
 IP addresses
 byte ordering, 259
 multicasts and, 267
 IP socket address structure, 258
 IP_MULTICAST_IF option, 267
 iPhone
 application sandbox, saving data to, 285–286
 challenges in developing for, 252–255
 development for, 104
 screen size, 252
 iPhone 3GS
 environment mapping and, 362
 Open GL ES 2.0 and, 362
 iPhone development program, 249

iPhone OS
 audio playback in, 96
 networking APIs, 255–256
 networking requirements and, 269–270
 registering URL scheme support with, 54
 iPhone OS 3.0, 96
 iPhone Simulator, 75, 256, 287
 iPhone version check, 147
 ISDatabase class, 144–157
 ISModel class, 164–175
 iTap
 autodiscovery solution in, 255, 273–275
 connection between iTap receiver and, 250
 development of, 249, 252–255
 GUI, 250–252
 multiplatform support, 254–255
 networking subsystem of, 271–275
 overview, 251
 power management and, 269–270
 querying network configuration, 264–267
 WiFi networking and, 253, 255–271
 iTap receiver, 251, 254

J

Java SDK, 45
 json_encode function, 332

K

Keychain Access application, 322, 325–326
 Key-Value Coding (KVC), 185, 199–203
 Key-Value Observing (KVO), 185
 KeyValueCollection, 192
 KeyValueCollection, 192
 kNetworkReachabilityChangedNotification notification,
 226
 KPSMTP, 213
 KRCC application, 96
 KVC protocol, 186, 199

L

lerping, 28–30
 Library directory, 285
 lighting
 OpenGL, 349–352
 specular, 350–352
 linear interpolation, 28–30

link map files, 135–137
 little endian, 259
 load balancing, 45
 local address, 258–260
 Locations project, 186, 189
 LocationsAppDelegate class, 193
 logging, custom, 107–108
 login keychain, 325
 lookups, 171–174

M

.m file, 148
 MacFUSE, 214
 mail module, 52
 Mail.app behavior, 226
 main.py file, 48
 malloc_error_break, 123–125
 malloc: *** error for object 0×XXXXXX: Non-aligned
 pointer being freed, 123, 129
 managed object context (MOC), 221
 managed object model (MOM), 221
 managedObjectContext, 244
 mapping. *See* environment mapping
 mapping layer, writing, 164–168
 material controller, 16
 MaterialController, 8
 memory constraints, 309
 Memory Browser, 103–104
 memory bugs, 105
 memory protection, in Objective-C, 105
 memory stomps, 119–123
 buffer overruns, 119
 calling a deleted method, 119–120
 debugging, 131–137
 defined, 118–121
 identifying, 122–123
 link map files and, 135–137
 returning a deleted object, 120–122
 tools to detect, 123–131
 variables and, 131–135
 MemoryBugsViewController class, 118
 memoryWentBoom method, 310
 mesh, 16
 Message class, 232, 235–237
 messageId, 237
 messageSent: delegate, 243
 mGetStomped, 132–134
 migration handling, 176–179
 migrations, managing model, 194–198
 mobile devices, buffering on, 84
 mobile provisioning files, 319, 329, 331, 343

- Model class, 237
- model classes, database connection references, 165
- model migrations, 194–198
- model objects, 165–166
 - deleting, 170–171
 - looking up, 171–174
 - mapping, 166–168
 - retrieving, 165
 - saving, 168–170
 - updating, 170
 - working directly with, 174–175
- move.plist template file, 60
- MPMoviePlayerController class, 66–68
- \$msg variable, 332
- .msi file, 254
- multicast datagrams, 256
- multicasts/multicasting, 267–269, 273
- multiplatform applications, 254
- multiple documents, downloading, 288
- multiple threads, 179
- multithreading, 292–297
- mutexes, 227

N

- naming conventions, 166
- NAND flash memory, 284
- navigation-based application, 145
- network byte order, 259
- network changes, 84
- network configuration, querying, 264–267
- network connections, 253
- network connectivity, 212
- network interfaces
 - flags, 266
 - querying names of available, 264–266
- network protocols, 258
- network state, changes in, 216, 224–226
- NetworkDiscovery class, 256–257, 272–273
- NetworkDiscoveryPeerTable, 272
- networking, 249–275
 - CFSocket and, 262–263
 - contacting all devices on network, 267–268
 - iTap subsystem, 271–275
 - multicasts, 267–269
 - socket-based, 257–263
 - WiFi, 255–271
 - detecting availability, 268–269
 - power management and, 269–270
- networking APIs, 255–256
- networkOperationQueue, 243
- NetworkStatus, 215
- Newton platform, 41
- NeXTStep, 184
- normal environment mapping, 354
- notifications, 44, 272–273
- NS_BLOCK_ASSERTIONS, 105
- NSArray arrayWithObjects: method, 162
- NSArray class, 105, 119, 152, 159
- NSArray property, 146
- NSAssert method, 105
- NSAutoreleasePool, 235
- NSBindings class, 203
- NSCAssert method, 105
- NSCoder APIs, 179
- NSCondition object, 83, 92
- NSDate object, 57
- NSDecimalNumbers, 284, 292
- NSDictionary objects, 56, 59–60, 152, 157, 284–285
- NSEntityDescription class, 238
- NSException class, 150
- NSFetchRequest class, 238
- NSHTTPURLResponse, 73
- NSInvocationOperation class, 233
- NSLog class, 107
- NSManagedObject class, 190–193, 203, 235–237
- NSMutableArray class, 105, 119
- NSMutableData instance variable, 58
- NSMutableData object, 70, 73
- NSMutableDictionary object, 155, 172, 274
- NSMutableURLRequest class, 72, 203
- NSNetService class, 271
- NSNetServiceBrowser class, 271
- NSNotification class, 216
- NSNotificationCenter, 216, 226
- NSNumbers class, 292
- NSObject class, 115, 148
- NSOperation objects, 57, 293–297, 308–309
- NSOperationQueue class, 293–297, 307–308
- NSPredicate class, 238, 309
- NSPropertyListSerialization class, 59, 203
- NSRunLoop object, 239–241
- NSSocketPort class, 256
- NSString class, 108, 153
- NSURLConnection class, 59, 72–73, 83, 85, 295
- NSURLRequest object, 72
- NSURLResponse object, 73
- NSUserDefaults class, 228
- NSZombieEnabled class, 126–128
- ntohl() function, 259
- ntohs() function, 259
- NYTimes App, 212

O

- O_NONBLOCK flag, 261
- Objective-C, 75, 164, 185
 - asserts in, 105
 - vs. C and C++, 104
 - types, 155–156
- Objective-C class, creating, 115–116
- Objective-C-based Foundation framework, 256
- object-relational mapping (ORM) framework, 184
- objects. *See also specific types*
 - deleting, 170–171
 - reusable, creating, 199–203
 - saving, 168–170
 - updating, 170
 - viewing, 187–192
- Observer pattern, 216
- observers, 273
- offline applications (OfflineMailer)
 - SMTP client
 - composing and sending messages, 230–235
 - Core Data, 218–224, 235–238
 - DataManager class, 226–228
 - planning, 212–213
 - setting up instance variables, 215–217
 - SKPSMTPMessage class, 239–241
 - switching to online mode, 241–244
 - TTMessageController class, 228–229
 - user interface, 213–218, 224–226
- OfflineMailer.xcdatamodel, 235
- OfflineMailerAppDelegate.h file, 215–217
- OfflineViewController class, 224–226
- open source, 213
- open source code, 213, 244
- open() function, 149
- OpenGL, 347–364
 - environment mapping with, 347–364
 - lighting, 349–352
 - particle systems and, 8–9
- OpenGL ES 1.1, 348–349
- OpenGL ES 2.0, iPhone 3GS and, 362
- OpenStep, 184
- openURL: method, 54–55
- os module, 56
- particle systems, 3–36
 - assigning textures, 16
 - basics of, 10–12
 - building, 12–19
 - code, 12–19, 21–24
 - color-changing particles, 28–35
 - emitting particles, 20–21
 - explosion emitter, 21–23
 - implementation, 14
 - initial conditions, 19
 - introduction to, 5–7
 - OpenGL and, 8–9
 - Particles (example)
 - basic game flow, 9–10
 - code, 12–19
 - code overview, 8–9
 - random numbers and, 19
 - update loops, 17
 - variations, 23–27
- particle textures, 10
- particles, 10
 - building, 12
 - color-changing, 28–35
 - emitting, 20–21
 - uniqueness of, 19
- peer-to-peer networking, 253
- .pem format, 327–328
- per-pixel environment mapping, 362–363
- per-pixel reflections, 359
- persistence framework, 180
- persistent object store (POS), 221
- persistent storage, Core Data and, 221
- persistent store coordinator, 221
- persistentStorageCoordinator, 221
- PersistentStoreCoordinator instance, 195
- persistentStoreCoordinator method, 221
- pl_addOperationWithBlock method, 296
- placeholder data, 286–288
- Plausible Blocks compiler, 293–295
- play method, 77, 95
- playback queue, starting, 91
- PlayQueueData, 83
- PlayQueueData structure, 83–84, 89, 91–92
- PLBlockOperation, 296–297
- plist file, 56, 179
- plistRep method, 284
- plists, 284–285
- port numbers, 259
- POSIX threads (pthreads), 227
- POST requests, 53, 203
- post() method, 53
- post-play callback function, 82, 87, 91–92
- power management, 269–270
- predicateString variable, 238

P

- parameter values, 172
- parameters, handling, in SQL, 160–162
- particle emitter. *See* emitter
- particle life cycle, 11–12

- Preferences directory, 285
- primary key column, 167, 170
- primaryKey property, 170
- PrivateMethod category, 156
- PrivateMethods category, 172
- program portal, 320
- @property declarations, 31
- property listener callback, 81, 86–87
- @property token, 186
- property values, 167
- propertyListenerCallback(), 91–92
- push notifications. *See also* Apple Push Notification Service
 - custom sounds for, 318
 - handling incoming, 317
 - prompting user to allow, 330
 - received, while application is running, 317
 - registration of, 315–316
 - script for, 331–335
- push notification service. *See* Apple Push Notification Service
- Push2AppDelegate.m file, 315
- PUT requests, 58–59
- put() method, 58–60
- Python scripting language, 44–45

Q

- Qt, 254
- Qt toolkit, 275
- queues, request, 57

R

- race conditions, 113
- raiseSQLException: method, 149–150
- random numbers, 19
- Reachability classes, 214–215
- reachabilityChanged: method, 216
- read() function, 260–261
- receivedData instance variable, 59
- rectangular emission volumes, 27
- recvfrom() function, 260
- refactoring, 162–163
- reflection vector, 353–354
- registerForRemoteNotificationTypes method, 316
- remote address, 258–260
- remote databases, 203–206
- RenderController, 8
- request handlers/handling, 50, 57
- requests
 - making, of SQLite database, 152–157
 - queues, 57
 - on the server, 59–60
 - response variable, 59–60
 - RESTful approach, 50
 - reusable objects, creating, 199–203
 - RFC 821, 218
 - RootViewController class, 187–188, 289
 - RootViewController.h file, 146
 - RootViewController.m file, 157

S

- save method, 168–170
- SceneController class, 8
- SceneObject class, 8
- SCNetworkReachability framework, 268
- scoping errors, 106
- SeeqPod, 66, 93
- SELECT statements, 152–154
- self variable, 87
- semitransparent particles, 10
- sendDeviceTokenToRemote: method, 317
- sendto() function, 260, 267
- server
 - dealing with requests on, 59–60
 - separating code and data representation on, 56–57
 - setting up, for APNS application, 331–335
- setEventsArray: method, 186
- setNeedsLayout method, 303
- setsockopt() function, 267
- Settings application, 218
- sharedDataManager method, 227
- Shark, 290–292, 297–299
- Simple Mail Transfer Proto: (SMTP), 217
- sin_addr, 259
- sin_port, 259
- singleton design pattern, 226
- SIOCGIFADDR IO control, 267
- SIOCGIFBRDADDR IO control, 267
- SIOCGIFCONF IO control, 264–267
- SIOCGIFFLAGS IO control, 266–268
- SIOCGIFNETMASK IO control, 267
- SkateDude, 4
- SKPSMTPMessage class, 239–241
- sleep() method, 303
- SMTP client, offline
 - composing and sending, 230–235
 - Core Data, 218–224, 235–238
 - DataManager class, 226–228
 - planning, 212–213

- SKPSMTPMessage class, 239–241
- switching to online mode, 241–244
- TTMessageController class, 228–229
- user authentication with, 218
- user interface, 213–218, 224–226
- NSMutableDictionary, 274
- SnowDude, 3
- SnowFerno, 3, 5
- SOCK_DGRAM sockets, 257
- SOCK_STREAM sockets, 257
- socket() function, 258
- sockets
 - address family, 257
 - BSD, 261
 - CFSocket, 262–263
 - creating, 258–262
 - introduction to, 257–258
 - IO controls and, 266
 - local and remote addresses, 258–260
 - protocol, 258
 - sending and receiving data, 260
 - types, 257
- specular lighting, 350–352
- specular mask, 359
- spherical emission volumes, 27
- spherical environment mapping, 353–356, 362
- SQL
 - advanced, 158–164
 - cleanup, 162–163
 - CREATE statements, 155, 159–159
 - executing statements, 156–157
 - grouping statements into transactions, 163–164
 - handling parameters, 160–162
 - logging messages, 152
 - refactoring, 162–163
 - SELECT statements, 152–154
- SQL database, 143
- SQLite databases, 143
 - column types, 155
 - creating and initializing, 148–149
 - deleting objects, 170–171
 - error codes, 152
 - handling parameters, 160–162
 - lookups, 171–174
 - making simple requests, 152–157
 - opening database connection, 149–151
 - preventing duplicate create statements, 158–159
 - refactoring and cleanup, 162–163
 - saving objects, 168–170
 - transaction support in, 163–164
 - updating objects, 170
- SQLITE_DONE, 153
- sqlite_master table, 159
- SQLITE_OK, 152
- SQLITE_ROW, 153
- sqlite3_column_decltype, 154
- sqlite3_errmsg16, 150
- sqlite3_step, 153
- sqlite3_stmt variable, 152
- SSL certificates, creating for APNS application, 319–331
- SSL server certificates, 332
- SSL server connections, 342
- startQueue method, 91
- StockPlot, 288–297
- Stocks application, 283
- storage layouts, for bytes, 259
- streaming audio, 81–95
- Streaming colour Studios, 101
- stringWithFormat: method, 160
- struct ifconf, 264–265
- struct ifreq, 265–266
- struct keyword, 118
- struct sockaddr, 258
- struct sockaddr_in datatype, 258, 265
- struct sockaddr_in6 datatype, 258
- symbolicatecrash script, 110–111
- symlinks command, 46
- @synchronized() method, 227
- synchronous calls, 57
- synchronous data downloading, 72
- @synthesize declarations, 31, 147–148, 186
- system crashes, 113
- SystemConfiguration framework, 241

T

- table names, retrieving, 166
- TCP protocol, 218
- template module, 56
- template_values dictionary, 56
- template.render() call, 56
- templates, 56, 60
- Terminal.app file, 327
- TestCppClass, 133
- testers, crash logs from, 109–110
- testing, crashes, 114–115
- texture atlases, 17
- texture combiner, 356–361
- threaded message sending, 239
- threads, crashing, 112
- Three20, 212, 221–224, 228–229, 232
- tmp directory, 285
- top-level data, managing with DataManager, 226–228
- toRow parameter, 172

TouchEngine, 284
 transactions, grouping SQL statements into, 163–164
 TTAddressBookDataSource class, 229
 TTMessageController class, 228–229
 TTMessageRecipientField class, 232
 typeForStatement:column: method, 154

U

UDP packets, 251
 UIApplication, 270
 UIApplicationDelegate proto:, 217–218
 UIImageView, 226
 UIKit, 304
 UINavigationController class, 186, 189, 192, 289
 UIRequiresPersistentWiFi, 270
 UIScrollView, 300–304
 UITableView, 226, 272
 UITableViewController class, 190
 UITableViewDelegate methods, 191
 UIViewController class, 189, 192, 225
 UIWebView, 299–300
 Unix, 255, 264
 update statement, 169–170
 updateDownloadStatus method, 296
 updateNetworkStatus method, 216
 updateResolution method, 301–304
 URL requests, handling, 54–55
 URL scheme, registering support with iPhone OS, 54
 usability, 252
 USB, third-party applications and, 253–254
 useNSEntityDescription class, 238
 user authentication, with SMTP servers, 218
 user experience, 261–262
 user interface, for offline SMTP client, 213–218, 224–226

V

valueForKey: method, 159

__VA_ARGS__ identifier, 107
 variable bit rate (VBR), 77
 variable watch, 131–135
 variables, memory stomps and, 131–135
 vertex data arrays, 16
 vertexCount variable, 18
 vertexIndex variable, 17, 33
 view controllers, 187–192
 ViewController class, 117
 viewDidLoad method, 118, 126, 132, 175

W

web application, creating new, 47–50
 web services, 333. *See also* Deep Green
 webapp module, 56
 WebObjects, 184
 WhatNext application, 142–143
 WiFi, 253–254
 WiFi networking, 255–271
 CFSocket and, 262–263
 detecting availability, 268–269
 power management and, 269–270
 Windows Installer XML (WIX) toolkit, 254
 write() function, 260–261

X, Y

xcdatamodel files, 193, 195–197
 Xcode, 213–214

Z

z-fighting, 361
 zoom operation, resetting resolution after, 301–304

You Need the Companion eBook

Your purchase of this book entitles you to buy the companion PDF-version eBook for only \$10. Take the weightless companion with you anywhere.

We believe this Apress title will prove so indispensable that you'll want to carry it with you everywhere, which is why we are offering the companion eBook (in PDF format) for \$10 to customers who purchase this book now. Convenient and fully searchable, the PDF version of any content-rich, page-heavy Apress book makes a valuable addition to your programming library. You can easily find and copy code—or perform examples by quickly toggling between instructions and the application. Even simultaneously tackling a donut, diet soda, and complex code becomes simplified with hands-free eBooks!

Once you purchase your book, getting the \$10 companion eBook is simple:

- 1 Visit **www.apress.com/promo/tendollars/**.
- 2 Complete a basic registration form to receive a randomly generated question about this title.
- 3 Answer the question correctly in 60 seconds, and you will receive a promotional code to redeem for the \$10.00 eBook.

Apress®
THE EXPERT'S VOICE™



233 Spring Street, New York, NY 10013

All Apress eBooks subject to copyright protection. No part 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. The purchaser may print the work in full or in part for their own noncommercial use. The purchaser may place the eBook title on any of their personal computers for their own personal reading and reference.