Erica Sadun

Companion to *The Advanced iOS 6 Developer's Cookbook*

Fourth Edition

iOS 6, new **Objective-C** features, and more!

The Core iOS 6 Developer's Cookbook

Developer's Library

FREE SAMPLE CHAPTER

SHARE WITH OTHERS

The Core iOS 6 Developer's Cookbook

This page intentionally left blank

The Core iOS 6 Developer's Cookbook

Erica Sadun

Addison-Wesley

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

ISBN-13: 978-0-321-88421-3
ISBN-10: 0-321-88421-3

Second Printing: May 2013

Editor-in-Chief
Mark Taub

Senior Acquisitions Editor
Trina MacDonald

Senior Development Editor
Chris Zahn

Managing Editor
Kristy Hart

Project Editor
Jovana San Nicolas-Shirley

Copy Editor
Keith Cline

Indexer
WordWise Publishing Services

Proofreader
Debbie Williams

Technical Editors
Duncan Champney
Oliver Drobnik
Rich Wardwell

Editorial Assistant
Olivia Basegio

Cover Designer
Chuti Prasertsith

Compositor
Nonie Ratcliff

❖

I dedicate this book with love to my husband, Alberto, who has put up with too many gadgets and too many SDKs over the years while remaining both kind and patient at the end of the day.

❖

Contents at a Glance

Table of Contents

Acknowledgments

This book would not exist without the efforts of Chuck Toporek, who was my editor and whipcracker for many years and multiple publishers. He is now at Apple and deeply missed. There'd be no Cookbook were it not for him. He balances two great skill sets: inspiring authors to do what they think they cannot, and wielding the large "reality trout" of whacking[1] to keep subject matter focused and in the real world. There's nothing like being smacked repeatedly by a large virtual fish to bring a book in on deadline and with compelling content.

Thanks go as well to Trina MacDonald (my terrific new editor), Chris Zahn (the awesomely talented development editor), and Olivia Basegio (the faithful and rocking editorial assistant who kept things rolling behind the scenes). Also, a big thank you to the entire Addison-Wesley/Pearson production team, specifically Kristy Hart, Jovana San Nicolas-Shirley, Keith Cline, Larry Sweazy, Debbie Williams, Nonie Ratcliff, and Chuti Prasertsith. Thanks also to the crew at Safari for getting my book up in Rough Cuts and for quickly fixing things when technical glitches occurred.

Thanks go as well to Neil Salkind, my agent of many years, to the tech reviewers Oliver Drobnik, Rich Wardwell, and Duncan Champney, who helped keep this book in the realm of sanity rather than wishful thinking, and to all my colleagues, both present and former, at TUAW, Ars Technica, and the Digital Media/Inside iPhone blog.

I am deeply indebted to the wide community of iOS developers, including Jon Bauer, Tim Burks, Matt Martel, Tim Isted, Joachim Bean, Aaron Basil, Roberto Gamboni, John Muchow, Scott Mikolaitis, Alex Schaefer, Nick Penree, James Cuff, Jay Freeman, Mark Montecalvo, August Joki, Max Weisel, Optimo, Kevin Brosius, Planetbeing, Pytey, Michael Brennan, Daniel Gard, Michael Jones, Roxfan, MuscleNerd, np101137, UnterPerro, Jonathan Watmough, Youssef Francis, Bryan Henry, William DeMuro, Jeremy Sinclair, Arshad Tayyeb, Jonathan Thompson, Dustin Voss, Daniel Peebles, ChronicProductions, Greg Hartstein, Emanuele Vulcano, Sean Heber, Josh Bleecher Snyder, Eric Chamberlain, Steven Troughton-Smith, Dustin Howett, Dick Applebaum, Kevin Ballard, Hamish Allan, Oliver Drobnik, Rod Strougo, Kevin McAllister, Jay Abbott, Tim Grant Davies, Maurice Sharp, Chris Samuels, Chris Greening, Jonathan Willing, Landon Fuller, Jeremy Tregunna, Wil Macaulay, Stefan Hafeneger, Scott Yelich, chrallelinder, John Varghese, Andrea Fanfani, J. Roman, jtbandes, Artissimo, Aaron Alexander, Christopher Campbell Jensen, Nico Ameghino, Jon Moody, Julián Romero, Scott Lawrence, Evan K. Stone, Kenny Chan Ching-King, Matthias Ringwald, Jeff Tentschert, Marco Fanciulli, Neil Taylor, Sjoerd van Geffen, Absentia, Nownot, Emerson Malca, Matt Brown, Chris Foresman, Aron Trimble, Paul Griffin, Paul Robichaux, Nicolas Haunold, Anatol Ulrich (hypnocode GmbH), Kristian Glass, Remy "psy" Demarest, Yanik Magnan, ashikase, Shane Zatezalo, Tito Ciuro, Mahipal Raythattha, Jonah Williams of Carbon Five, Joshua Weinberg, biappi, Eric Mock, and everyone at the iPhone developer channels at irc.saurik.com and irc.freenode.net, among many others too numerous to name individually. Their techniques, suggestions, and feedback helped make this book possible. If I have overlooked anyone who helped contribute, please accept my apologies for the oversight.

Special thanks go out to my family and friends, who supported me through month after month of new beta releases and who patiently put up with my unexplained absences and frequent howls of despair. I appreciate you all hanging in there with me. And thanks to my children for their steadfastness, even as they learned that a hunched back and the sound of clicking keys is a pale substitute for a proper mother. My kids provided invaluable assistance over the past few months by testing applications, offering suggestions, and just being awesome people. I try to remind myself on a daily basis how lucky I am that these kids are part of my life.

About the Author

Erica Sadun is the bestselling author, coauthor, and contributor to several dozen books on programming, digital video and photography, and web design, including the widely popular *The iOS 5 Developer's Cookbook*. She currently blogs at TUAW.com, and has blogged in the past at O'Reilly's Mac Devcenter, Lifehacker, and Ars Technica. In addition to being the author of dozens of iOS-native applications, Erica holds a Ph.D. in Computer Science from Georgia Tech's Graphics, Visualization and Usability Center. A geek, a programmer, and an author, she's never met a gadget she didn't love. When not writing, she and her geek husband parent three geeks-in-training, who regard their parents with restrained bemusement, when they're not busy rewiring the house or plotting global dominance.

Preface

Welcome to a new iOS Cookbook!

With iOS 6, Apple's mobile device family has reached new levels of excitement and possibility. This Cookbook is here to help you get started developing. This revision introduces all new features announced at the latest WWDC, showing you how to incorporate them into your applications.

For this edition, my publishing team has sensibly split the Cookbook material into manageable print volumes. This book, *The Core iOS 6 Developer's Cookbook,* provides solutions for the heart of day-to-day development. It covers all the classes you need for creating iOS applications using standard APIs and interface elements. It offers the recipes you need for working with graphics, touches, and views to create mobile applications.

A second volume, *The Advanced iOS 6 Developer's Cookbook,* centers on common frameworks such as Store Kit, Game Kit, and Core Location. It helps you build applications that leverage these special-purpose libraries and move beyond the basics. This volume is for those who have a strong grasp on iOS development and are looking for practical how-to for specialized areas.

Finally, there's *Learning iOS 6: A Hands-on Guide to the Fundamentals of iOS Programming*, which will cover much of the tutorial material that used to comprise the first several chapters of the Cookbook. There you'll find all the fundamental how-to you need to learn iOS 6 development from the ground up. From Objective-C to Xcode, debugging to deployment, *Learning iOS 6* teaches you how to get started with Apple's development tool suite.

As in the past, you can find sample code at GitHub. You'll find the repository for this Cookbook at https://github.com/erica/iOS-6-Cookbook, all of it refreshed for iOS 6 after WWDC 2012.

If you have suggestions, bug fixes, corrections, or any thing else you'd like to contribute to a future edition, please contact me at erica@ericasadun.com. Let me thank you all in advance. I appreciate all feedback that helps make this a better, stronger book.

—Erica Sadun, September 2012

What You'll Need

It goes without saying that, if you're planning to build iOS applications, you're going to need at least one iOS device to test out your application, preferably a new model iPhone or tablet. The following list covers the basics of what you need to begin:

- **Apple's iOS SDK—**You can download the latest version of the iOS SDK from Apple's iOS Dev Center (http://developer.apple.com/ios). If you plan to sell apps through the App Store, become a paid iOS developer. This costs $99/year for individuals and $299/year for enterprise (that is, corporate) developers. Registered developers receive certificates that allow them to "sign" and download their applications to their iPhone/iPod touch for testing and debugging and to gain early access to prerelease versions of iOS. Free-program

developers can test their software on the Mac-based simulator but cannot deploy to device or submit to the App Store.

> **University Student Program**
>
> Apple also offers a University Program for students and educators. If you are a computer science student taking classes at the university level, check with your professor to see whether your school is part of the University Program. For more information about the iPhone Developer University Program, see http://developer.apple.com/support/iphone/university.

- **A modern Mac running Mac OS X Lion (v 10.7) or, preferably, Mac OS X Mountain Lion (v 10.8)—**You need plenty of disk space for development, and your Mac should have as much RAM as you can afford to put into it.
- **An iOS device—**Although the iOS SDK includes a simulator for you to test your applications in, you really do need to own iOS hardware to develop for the platform. You can tether your unit to the computer and install the software you've built. For real-life App Store deployment, it helps to have several units on hand, representing the various hardware and firmware generations, so that you can test on the same platforms your target audience will use.
- **An Internet connection—**This connection enables you to test your programs with a live Wi-Fi connection as well as with an EDGE or 3G service.
- **Familiarity with Objective-C—**To program for the iPhone, you need to know Objective-C 2.0. The language is based on ANSI C with object-oriented extensions, which means you also need to know a bit of C too. If you have programmed with Java or C++ and are familiar with C, you should be able to make the move to Objective-C.

Your Roadmap to Mac/iOS Development

One book can't be everything to everyone. Try as I might, if we were to pack everything you need to know into this book, you wouldn't be able to pick it up. (As it stands, this book offers an excellent tool for upper-body development. Please don't sue if you strain yourself lifting it.) There is, indeed, a lot you need to know to develop for the Mac and iOS platforms. If you are just starting out and don't have any programming experience, your first course of action should be to take a college-level course in the C programming language. Although the alphabet might start with the letter A, the root of most programming languages, and certainly your path as a developer, is C.

Once you know C and how to work with a compiler (something you'll learn in that basic C course), the rest should be easy. From there, you'll hop right on to Objective-C and learn how to program with that alongside the Cocoa frameworks. The flowchart shown in Figure P-1 shows you key titles offered by Pearson Education that can help provide the training you need to become a skilled iOS developer.

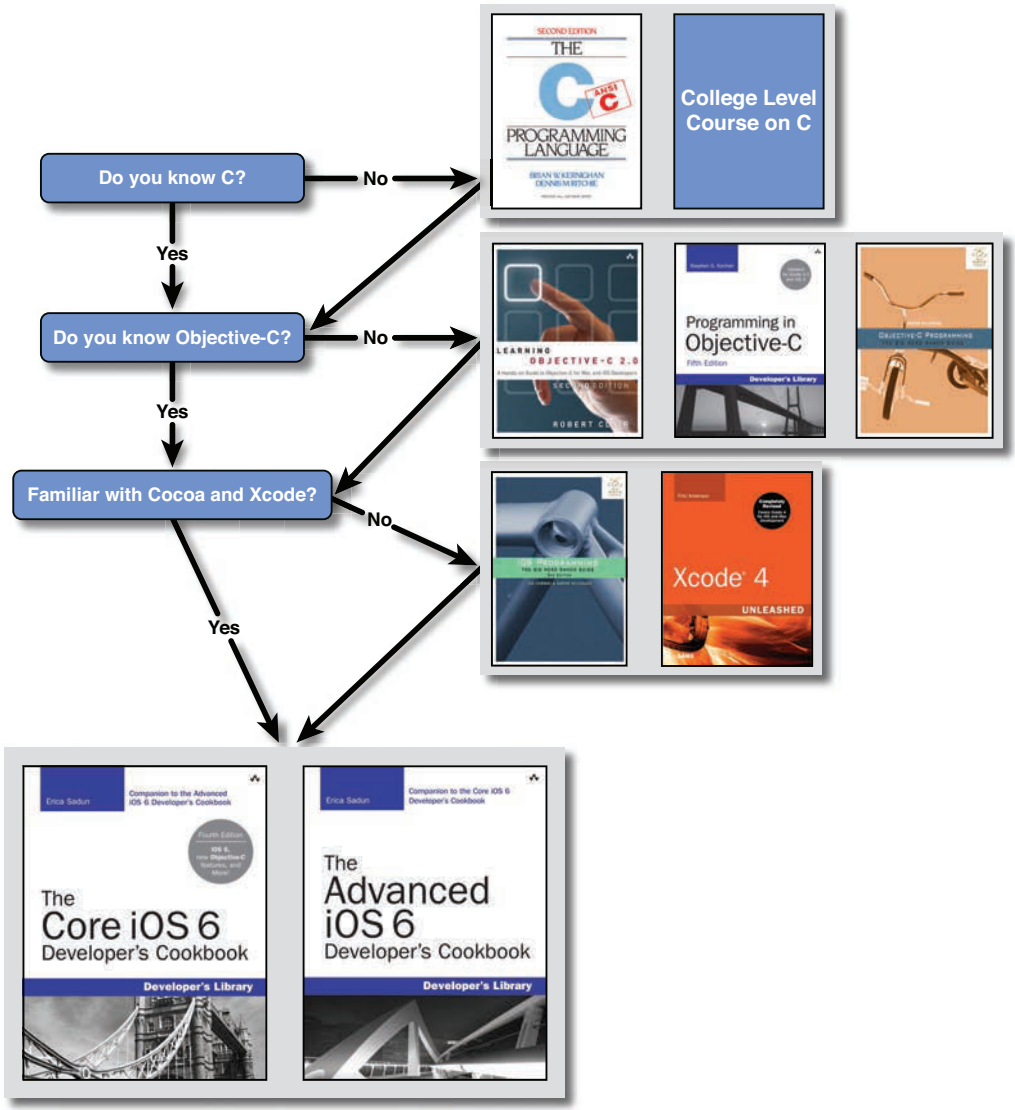

Figure P-1 A roadmap to becoming an iOS developer.

Once you know C, you've got a few options for learning how to program with Objective-C. If you want an in-depth view of the language, you can either read Apple's own documentation or pick up one of these books on Objective-C:

- *Objective-C Programming: The Big Nerd Ranch Guide,* by Aaron Hillegass (Big Nerd Ranch, 2012)

- *Learning Objective-C 2.0: A Hands-on Guide to Objective-C for Mac and iOS Developers,* by Robert Clair (Addison-Wesley, 2012)
- *Programming in Objective-C, Fifth Edition,* by Stephen Kochan (Addison-Wesley, 2012)

With the language behind you, next up is tackling Cocoa and the developer tools, otherwise known as Xcode. For that, you have a few different options. Again, you can refer to Apple's own documentation on Cocoa and Xcode,[2] or if you prefer books, you can learn from the best. Aaron Hillegass, founder of the Big Nerd Ranch in Atlanta,[3] is the coauthor of *iOS Programming: The Big Nerd Ranch Guide, Second Edition,* and author of *Cocoa Programming for Mac OS X,* soon to be in its fourth edition. Aaron's book is highly regarded in Mac developer circles and is the most-recommended book you'll see on the cocoa-dev mailing list. To learn more about Xcode, look no further than Fritz Anderson's *Xcode 4 Unleashed* from Sams Publishing.

> **Note**
>
> There are plenty of other books from other publishers on the market, including the bestselling *Beginning iPhone 4 Development*, by Dave Mark, Jack Nutting, and Jeff LaMarche (Apress, 2011). Another book that's worth picking up if you're a total newbie to programming is *Beginning Mac Programming*, by Tim Isted (Pragmatic Programmers, 2011). Don't just limit yourself to one book or publisher. Just as you can learn a lot by talking with different developers, you will learn lots of tricks and tips from other books on the market.

To truly master Mac development, you need to look at a variety of sources: books, blogs, mailing lists, Apple's own documentation, and, best of all, conferences. If you get the chance to attend WWDC, you'll know what I'm talking about. The time you spend at those conferences talking with other developers, and in the case of WWDC, talking with Apple's engineers, is well worth the expense if you are a serious developer.

How This Book Is Organized

This book offers single-task recipes for the most common issues new iOS developers face: laying out interface elements, responding to users, accessing local data sources, and connecting to the Internet. Each chapter groups together related tasks, allowing you to jump directly to the solution you're looking for without having to decide which class or framework best matches that problem.

The Core iOS 6 Developer's Cookbook offers you "cut-and-paste convenience," which means you can freely reuse the source code from recipes in this book for your own applications and then tweak the code to suit your app's needs.

Here's a rundown of what you find in this book's chapters:

- **Chapter 1, "Gestures and Touches"**—On iOS, the touch provides the most important way that users communicate their intent to an application. Touches are not limited to button presses and keyboard interaction. This chapter introduces direct manipulation

interfaces, Multi-Touch, and more. You see how to create views that users can drag around the screen and read about distinguishing and interpreting gestures, as well as how to create custom gesture recognizers.

- **Chapter 2, "Building and Using Controls"**—Take your controls to the next level. This chapter introduces everything you need to know about how controls work. You discover how to build and customize controls in a variety of ways. From the prosaic to the obscure, this chapter introduces a range of control recipes you can reuse in your programs.
- **Chapter 3, "Alerting the User"**—iOS offers many ways to provide users with a heads-up, from pop-up dialogs and progress bars to local notifications, popovers, and audio pings. Chapter 3 shows how to build these indications into your applications and expand your user-alert vocabulary. It introduces standard ways of working with these classes and offers solutions that allow you to craft linear programs without explicit callbacks.
- **Chapter 4, "Assembling Views and Animations"**—The `UIView` class and its subclasses populate the iOS device screens. This chapter introduces views from the ground up. This chapter dives into view recipes, exploring ways to retrieve, animate, and manipulate view objects. You learn how to build, inspect, and break down view hierarchies and understand how views work together. You discover the role geometry plays in creating and placing views into your interface, and you read about animating views so they move and transform onscreen.
- **Chapter 5, "View Constraints"**—The iOS 6 software development kit (SDK) revolutionized view layout. Apple's layout features are about to make your life easier and your interfaces more consistent. This is especially important when working across members of the same device family with different screen sizes—like the iPhone 4S and the iPhone 5. This chapter introduces code-level constraint development. You'll discover how to create relations between onscreen objects and specify the way iOS automatically arranges your views. The outcome is a set of robust rules that adapt to screen geometry.
- **Chapter 6, "Text Entry"**—Chapter 6 introduces text recipes that support a wide range of solutions. You'll read about controlling keyboards, making onscreen elements "text aware," scanning text, formatting text, and so forth. From text fields and text views to iOS's inline spelling checkers, this chapter introduces everything you need to know to work with iOS text in your apps.
- **Chapter 7, "Working with View Controllers"**—Discover the various view controller classes that enable you to enlarge and order the virtual spaces your users interact with, learning from how-to recipes that cover page view controllers, split view controllers, navigation controllers, and more.
- **Chapter 8, "Common Controllers"**—The iOS SDK provides a wealth of system-supplied controllers that you can use in your day-to-day development tasks. This chapter introduces some of the most popular ones. You read about selecting images from your photo library, snapping photos, and recording and editing videos. You discover how to allow users to compose e-mails and text messages, and how to post updates to social services such as Twitter and Facebook.

- **Chapter 9, "Accessibility"—**This chapter offers a brief overview of VoiceOver accessibility to extend your audience to the widest possible range of users. You read about adding accessibility labels and hints to your applications and testing those features in the simulator and on the iOS device.
- **Chapter 10, "Creating and Managing Table Views"—**Tables provide a scrolling interaction class that works particularly well both on smaller devices and as a key player on larger tablets. Many iOS apps center on tables due to their simple natural organization features. Chapter 10 introduces tables. It explains how tables work, what kinds of tables are available to you as a developer, and how you can leverage table features in your applications.
- **Chapter 11, "Collection Views"—**Collection views use many of the same concepts as tables but provide more power and more flexibility. This chapter walks you through all the basics you need to get started. Prepare to read about creating side-scrolling lists, grids, one-of-a-kind layouts like circles, and more. You'll learn about integrating visual effects through layout specifications and snapping items into place after scrolling, and you'll discover how to take advantage of built-in animation support to create the most effective interactions possible.
- **Chapter 12, "A Taste of Core Data"—**Core Data offers managed data stores that can be queried and updated from your application. It provides a Cocoa Touch–based object interface that brings relational data management out from SQL queries and into the Objective-C world of iOS development. Chapter 12 introduces Core Data. It provides just enough recipes to give you a taste of the technology, offering a jumping-off point for further Core Data learning. You learn how to design managed database stores, add and delete data, and query that data from your code and integrate it into your UIKit table views and collection views.
- **Chapter 13, "Networking Basics"—**As an Internet-connected device, iOS is particularly suited to subscribing to Web-based services. Apple has lavished the platform with a solid grounding in all kinds of network computing services and their supporting technologies. Chapter 13 surveys common techniques for network computing and offers recipes that simplify day-to-day tasks. You read about network reachability, synchronous and asynchronous downloads, using operation queues, working with the iPhone's secure keychain to meet authentication challenges, XML parsing, JSON serialization, and more.
- **Appendix, "Objective-C Literals"—**This appendix introduces new Objective-C constructs for specifying numbers, arrays, and dictionaries.

About the Sample Code

For the sake of pedagogy, this book's sample code uses a single main.m file. This is not how people normally develop iPhone or Cocoa applications, or, honestly, how they should be developing them, but it provides a great way of presenting a single big idea. It's hard to tell a story when readers must look through five or seven or nine individual files at once. Offering a single file concentrates that story, allowing access to that idea in a single chunk.

These examples are not intended as stand-alone applications. They are there to demonstrate a single recipe and a single idea. One main.m file with a central presentation reveals the implementation story in one place. Readers can study these concentrated ideas and transfer them into normal application structures, using the standard file structure and layout. The presentation in this book does not produce code in a standard day-to-day best-practices approach. Instead, it reflects a pedagogy that offers concise solutions that you can incorporate back into your work as needed.

Contrast that to Apple's standard sample code, where you must comb through many files to build up a mental model of the concepts that are being demonstrated. Those examples are built as full applications, often doing tasks that are related to but not essential to what you need to solve. Finding just those relevant portions is a lot of work. The effort may outweigh any gains.

In this book, you'll find exceptions to this one-file-with-the-story rule: the Cookbook provides standard class and header files when a class implementation is the recipe. Instead of highlighting a technique, some recipes offer these classes and categories (that is, extensions to a preexisting class rather than a new class). For those recipes, look for separate .m *and* .h files in addition to the skeletal main.m that encapsulates the rest of the story.

For the most part, the examples for this book use a single application identifier: com.sadun.helloworld. This book uses one identifier to avoid clogging up your iOS devices with dozens of examples at once. Each example replaces the previous one, ensuring that your home screen remains relatively uncluttered. If you want to install several examples simultaneously, simply edit the identifier, adding a unique suffix, such as com.sadun.helloworld.table-edits. You can also edit the custom display name to make the apps visually distinct. Your Team Provisioning Profile matches every application identifier, including com.sadun.helloworld. This allows you to install compiled code to devices without having to change the identifier; just make sure to update your signing identity in each project's build settings.

Getting the Sample Code

You'll find the source code for this book at github.com/erica/iOS-6-Cookbook on the open-source GitHub hosting site. There, you find a chapter-by-chapter collection of source code that provides working examples of the material covered in this book. Recipes are numbered as they are in the book. Recipe 6 in Chapter 5, for example, appears in the C05 folder in the 06 subfolder.

Any project numbered 00 or that has a suffix (like 05b or 02c) refers to material used to create in-text coverage and figures. For example, Chapter 10's 00 – Cell Types project helped build Figure 10-2, showing system-supplied table view cells styles. Normally I delete these extra projects. Early readers of this manuscript requested that I include them in this edition. You'll find a half dozen or so of these extra samples scattered around the repository.

Contribute!

Sample code is never a fixed target. It continues to evolve as Apple updates its SDK and the Cocoa Touch libraries. Get involved. You can pitch in by suggesting bug fixes and corrections as well as by expanding the code that's on offer. GitHub allows you to fork repositories and grow them with your own tweaks and features, and share those back to the main repository. If you come up with a new idea or approach, let me know. My team and I are happy to include great suggestions both at the repository and in the next edition of this Cookbook.

Getting Git

You can download this Cookbook's source code using the git version control system. An OS X implementation of git is available at http://code.google.com/p/git-osx-installer. OS X git implementations include both command-line and GUI solutions, so hunt around for the version that best suits your development needs.

Getting GitHub

GitHub (http://github.com) is the largest git-hosting site, with more than 150,000 public repositories. It provides both free hosting for public projects and paid options for private projects. With a custom Web interface that includes wiki hosting, issue tracking, and an emphasis on social networking of project developers, it's a great place to find new code or collaborate on existing libraries. You can sign up for a free account at their website, allowing you to copy and modify the Cookbook repository or create your own open-source iOS projects to share with others.

Contacting the Author

If you have any comments or questions about this book, please drop me an e-mail message at erica@ericasadun.com, or stop by the github repository and contact me there.

Endnotes

[1] No trouts, real or imaginary, were hurt in the development and production of this book. The same cannot be said for countless cans of Diet Coke, who selflessly surrendered their contents in the service of this manuscript.

[2] See the *Cocoa Fundamentals Guide* (http://developer.apple.com/mac/library/documentation/Cocoa/Conceptual/CocoaFundamentals/CocoaFundamentals.pdf) for a head start on Cocoa, and for Xcode, see *A Tour of Xcode* (http://developer.apple.com/mac/library/documentation/DeveloperTools/Conceptual/A_Tour_of_Xcode/A_Tour_of_Xcode.pdf).

[3] Big Nerd Ranch: www.bignerdranch.com.

Editor's Note: We Want to Hear from You!

As the reader of this book, you are our most important critic and commentator. We value your opinion and want to know what we're doing right, what we could do better, what areas you'd like to see us publish in, and any other words of wisdom you're willing to pass our way.

You can e-mail or write me directly to let me know what you did or didn't like about this book—as well as what we can do to make our books stronger.

Please note that I cannot help you with technical problems related to the topic of this book, and that due to the high volume of mail I receive, I might not be able to reply to every message.

When you write, please be sure to include this book's title and author as well as your name and phone or e-mail address. I will carefully review your comments and share them with the author and editors who worked on the book.

E-mail: trina.macdonald@pearson.com

Mail: Trina MacDonald
Senior Acquisitions Editor
Addison-Wesley/Pearson Education, Inc.
75 Arlington St., Ste. 300
Boston, MA 02116

1

Gestures and Touches

The touch represents the heart of iOS interaction; it provides the core way that users communicate their intent to an application. Touches are not limited to button presses and keyboard interaction. You can design and build applications that work directly with users' gestures in meaningful ways. This chapter introduces direct manipulation interfaces that go far beyond prebuilt controls. You see how to create views that users can drag around the screen. You also discover how to distinguish and interpret gestures, which are a high-level touch abstraction, and gesture recognizer classes, which automatically detect common interaction styles like taps, swipes, and drags. By the time you finish reading this chapter, you'll have read about many different ways you can implement gesture control in your own applications.

Touches

Cocoa Touch implements direct manipulation in the simplest way possible. It sends touch events to the view you're working with. As an iOS developer, you tell the view how to respond. Before jumping into gestures and gesture recognizers, you should gain a solid foundation in this underlying touch technology. It provides the essential components of all touch-based interaction.

Each touch conveys information: where the touch took place (both the current and previous location), what phase of the touch was used (essentially mouse down, mouse moved, mouse up in the desktop application world, corresponding to finger or touch down, moved, and up in the direct manipulation world), a tap count (for example, single-tap/double-tap), and when the touch took place (through a time stamp).

iOS uses what is called a *responder chain* to decide which objects should process touches. As their name suggests, responders are objects that respond to events and they act as a chain of possible managers for those events. When the user touches the screen, the application looks for an object to handle this interaction. The touch is passed along, from view to view, until some object takes charge and responds to that event.

At the most basic level, touches and their information are stored in `UITouch` objects, which are passed as groups in `UIEvent` objects. Each object represents a single touch event, containing single or multiple touches. This depends both on how you've set up your application to respond (that is, if you've enabled Multi-Touch interaction), and how the user touches the screen (that is, the physical number of touch points).

Your application receives touches in view or view controller classes; both implement touch handlers via inheritance from the `UIResponder` class. You decide where you process and respond to touches. Trying to implement low-level gesture control in nonresponder classes has tripped up many new iOS developers.

Handling touches in views may seem counterintuitive. You probably expect to separate the way an interface looks (its view) from the way it responds to touches (its controller). Further, using views for direct touch interaction may seem to contradict Model-View-Controller design orthogonality, but it can be necessary and help promote encapsulation.

Consider the case of working with multiple touch-responsive subviews such as game pieces on a board. Building interaction behavior directly into view classes allows you to send meaningful semantically rich feedback to your main application while hiding implementation minutia. For example, you can inform your model that a pawn has moved to Queen's Bishop 5 at the end of an interaction sequence rather than transmit a meaningless series of vector changes. By hiding the way the game pieces move in response to touches, your model code can focus on game semantics instead of view position updates.

Drawing presents another reason to work in the `UIView` class. When your application handles any kind of drawing operation in response to user touches, you need to implement touch handlers in views. Unlike views, view controllers don't implement the all-important `drawRect:` method needed for providing custom presentations.

Working at the view controller level also has its perks. Instead of pulling out primary handling behavior into a secondary class implementation, adding touch management directly to the view controller allows you to interpret standard gestures, such as tap-and-hold or swipes, where those gestures have meaning. This better centralizes your code and helps tie controller interactions directly to your application model.

In the following sections and recipes, you discover how touches work, how you can incorporate them into your apps, and how you connect what a user sees with how that user interacts with the screen.

Phases

Touches have life cycles. Each touch can pass through any of five phases that represent the progress of the touch within an interface. These phases are as follows:

- **`UITouchPhaseBegan`**—Starts when the user touches the screen.
- **`UITouchPhaseMoved`**—Means a touch has moved on the screen.

- **`UITouchPhaseStationary`**—Indicates that a touch remains on the screen surface but that there has not been any movement since the previous event.
- **`UITouchPhaseEnded`**—Gets triggered when the touch is pulled away from the screen.
- **`UITouchPhaseCancelled`**—Occurs when the iOS system stops tracking a particular touch. This usually occurs due to a system interruption, such as when the application is no longer active or the view is removed from the window.

Taken as a whole, these five phases form the interaction language for a touch event. They describe all the possible ways that a touch can progress or fail to progress within an interface and provide the basis for control for that interface. It's up to you as the developer to interpret those phases and provide reactions to them. You do that by implementing a series of responder methods.

Touches and Responder Methods

All subclasses of the `UIResponder` class, including `UIView` and `UIViewController`, respond to touches. Each class decides whether and how to respond. When choosing to do so, they implement customized behavior when a user touches one or more fingers down in a view or window.

Predefined callback methods handle the start, movement, and release of touches from the screen. Corresponding to the phases you've already seen, the methods involved are as follows. Notice that `UITouchPhaseStationary` does not generate a callback.

- **`touchesBegan:withEvent:`**—Gets called at the starting phase of the event, as the user starts touching the screen.
- **`touchesMoved:withEvent:`**—Handles the movement of the fingers over time.
- **`touchesEnded:withEvent:`**—Concludes the touch process, where the finger or fingers are released. It provides an opportune time to clean up any work that was handled during the movement sequence.
- **`touchesCancelled:WithEvent:`**—Called when Cocoa Touch must respond to a system interruption of the ongoing touch event.

Each of these is a `UIResponder` method, often implemented in a `UIView` or `UIViewController` subclass. All views inherit basic nonfunctional versions of the methods. When you want to add touch behavior to your application, you override these methods and add a custom version that provides the responses your application needs.

Your classes can implement all or just some of these methods. For real-world deployment, you will always want to add a touches-cancelled event to handle the case of a user dragging his or her finger offscreen or the case of an incoming phone call, both of which cancel an ongoing touch sequence. As a rule, you can generally redirect a canceled touch to your `touchesEnded:withEvent:` method. This allows your code to complete the touch sequence, even if the user's finger has not left the screen. Apple recommends overriding all four methods as a best practice when working with touches.

> **Note**
>
> Views have a mode called *exclusive touch* that prevents touches from being delivered to other views in the same window. When enabled, this property blocks other views from receiving touch events. The primary view handles all touch events exclusively.

Touching Views

When dealing with many onscreen views, iOS automatically decides which view the user touched and passes any touch events to the proper view for you. This helps you write concrete direct manipulation interfaces where users touch, drag, and interact with onscreen objects.

Just because a touch is physically on top of a view doesn't mean that a view has to respond. Each view can use a "hit test" to choose whether to handle a touch or to let that touch fall through to views beneath it. As you see in the recipes that follow, you can use clever response strategies to decide when your view should respond, particularly when you're using irregular art with partial transparency.

With touch events, the first view that passes the hit test opts to handle or deny the touch. If it passes, the touch continues to the view's superview and then works its way up the responder chain until it is handled or until it reaches the window that owns the views. If the window does not process it, the touch moves to the application instance, where it is either processed or discarded.

> **Note**
>
> Touches are limited to `UIViews` and their subclasses. This includes windows. When developing apps targeted at both iPhone 4S-and-earlier and iPhone 5 platforms, make sure your window extends across the entire screen. Problems are most commonly seen when the iPhone 5 is in portrait mode and apps fail to respond to touches at the bottom of the screen. This occurs when the application's key window is sized to a 3.5" screen but used on the larger 4" model. Without a backing window view, touches will not be recognized. Make sure your apps support both 3.5" and 4" screens by extending their key `UIWindow` to the screen's full vertical extent.

Multi-Touch

iOS supports both single- and Multi-Touch interfaces. Single-touch GUIs handle just one touch at any time. This relieves you of any responsibility to determine which touch you were tracking. The one touch you receive is the only one you need to work with. You look at its data, respond to it, and wait for the next event.

When working with Multi-Touch—that is, when you respond to multiple onscreen touches at once—you receive an entire set of touches. It is up to you to order and respond to that set. You can, however, track each touch separately and see how it changes over time, providing a richer set of possible user interaction. Recipes for both single-touch and Multi-Touch interaction follow in this chapter.

Gesture Recognizers

With gesture recognizers, Apple added a powerful way to detect specific gestures in your interface. Gesture recognizers simplify touch design. They encapsulate touch methods, so you don't have to implement these yourself, and provide a target-action feedback mechanism that hides implementation details. They also standardize how certain movements are categorized, as drags or swipes, and so forth.

With gesture recognizer classes, you can trigger callbacks when iOS perceives that the user has tapped, pinched, rotated, swiped, panned, or used a long press. Although their software development kit (SDK) implementations remain imperfect, these detection capabilities simplify development of touch-based interfaces. You can code your own for improved reliability, but a majority of developers will find that the recognizers, as-shipped, are robust enough for many application needs. You'll find several recognizer-based recipes in this chapter. Because recognizers all basically work in the same fashion, you can easily extend these recipes to your specific gesture recognition requirements.

Here is a rundown of the kinds of gestures built in to recent versions of the iOS SDK:

- **Taps**—Taps correspond to single or multiple finger taps onscreen. Users can tap with one or more fingers; you specify how many fingers you require as a gesture recognizer property and how many taps you want to detect. You can create a tap recognizer that works with single finger taps, or more nuanced recognizers that look for, for example, two-fingered triple-taps.
- **Swipes**—Swipes are short, single- or Multi-Touch gestures that move in a single cardinal direction: up, down, left, or right. They cannot move too far off course from that primary direction. You set the direction you want your recognizer to work with. The recognizer returns the detected direction as a property.
- **Pinches**—To pinch or unpinch, a user must move two fingers together or apart in a single movement. The recognizer returns a scale factor indicating the degree of pinching.
- **Rotations**—To rotate, a user moves two fingers at once either in a clockwise or counterclockwise direction, producing an angular rotation as the main returned property.
- **Pan**—Pans occur when users drag their fingers across the screen. The recognizer determines the change in translation produced by that drag.
- **Long press**—To create a long press, the user touches the screen and holds his or her finger (or fingers) there for a specified period of time. You can specify how many fingers must be used before the recognizer triggers.

Recipe: Adding a Simple Direct Manipulation Interface

Your design focus moves from the `UIViewController` to the `UIView` when you work with direct manipulation. The view, or more precisely the `UIResponder`, forms the heart of direct

manipulation development. You create touch-based interfaces by customizing methods that derive from the `UIResponder` class.

Recipe 1-1 centers on touches in action. This example creates a child of `UIImageView` called `DragView` and adds touch responsiveness to the class. Being an image view, it's important to enable user interaction (that is, set `setUserInteractionEnabled` to `YES`). This property affects all the view's children as well as the view itself. User interaction is generally enabled for most views, but `UIImageView` is the one exception that stumps most beginners; Apple apparently didn't think people would generally manipulate them.

The recipe works by updating a view's center to match the movement of an onscreen touch. When a user first touches any `DragView`, the object stores the start location as an offset from the view's origin. As the user drags, the view moves along with the finger—always maintaining the same origin offset so that the movement feels natural. Movement occurs by updating the object's center. Recipe 1-1 calculates x and y offsets and adjusts the view center by those offsets after each touch movement.

Upon being touched, the view pops to the front. That's due to a call in the `touchesBegan:withEvent:` method. The code tells the superview that owns the `DragView` to bring that view to the front. This allows the active element to always appear foremost in the interface.

This recipe does not implement touches-ended or touches-cancelled methods. Its interests lie only in the movement of onscreen objects. When the user stops interacting with the screen, the class has no further work to do.

Recipe 1-1 **Creating a Draggable View**

```
@interface DragView : UIImageView
{
    CGPoint startLocation;
}
@end

@implementation DragView
- (id) initWithImage: (UIImage *) anImage
{
    if (self = [super initWithImage:anImage])
        self.userInteractionEnabled = YES;
    return self;
}

- (void) touchesBegan:(NSSet*)touches withEvent:(UIEvent*)event
{
    // Calculate and store offset, and pop view into front if needed
    startLocation = [[touches anyObject] locationInView:self];
    [self.superview bringSubviewToFront:self];
}
```

```
- (void) touchesMoved:(NSSet*)touches withEvent:(UIEvent*)event
{
    // Calculate offset
    CGPoint pt = [[touches anyObject] locationInView:self];
    float dx = pt.x - startLocation.x;
    float dy = pt.y - startLocation.y;
    CGPoint newcenter = CGPointMake(
        self.center.x + dx,
        self.center.y + dy);

    // Set new location
    self.center = newcenter;
}
@end
```

> **Get This Recipe's Code**
>
> To find this recipe's full sample project, point your browser to https://github.com/erica/iOS-6-Cookbook and go to the folder for Chapter 1.

Recipe: Adding Pan Gesture Recognizers

With gesture recognizers, you can achieve the same kind of interaction shown in Recipe 1-1 without working quite so directly with touch handlers. Pan gesture recognizers detect dragging gestures. They allow you to assign a callback that triggers whenever iOS senses panning.

Recipe 1-2 mimics Recipe 1-1's behavior by adding a recognizer to the view when it is first initialized. As iOS detects the user dragging on a `DragView` instance, the `handlePan:` callback updates the view's center to match the distance dragged.

This code uses what might seem like an odd way of calculating distance. It stores the original view location in an instance variable (`previousLocation`) and then calculates the offset from that point each time the view updates with a pan detection callback. This allows you to use affine transforms or apply the `setTranslation:inView:` method; you normally do not move view centers, as done here. This recipe creates a dx/dy offset pair and applies that offset to the view's center, changing the view's actual frame.

Unlike simple offsets, affine transforms allow you to meaningfully work with rotation, scaling, and translation all at once. To support transforms, gesture recognizers provide their coordinate changes in absolute terms rather than relative ones. Instead of issuing iterative offset vectors, the `UIPanGestureRecognizer` returns a single vector representing a translation in terms of some view's coordinate system, typically the coordinate system of the manipulated view's superview. This vector translation lends itself to simple affine transform calculations and can be mathematically combined with other changes to produce a unified transform representing all changes applied simultaneously.

Here's what the `handlePan:` method looks like using straight transforms and no stored state:

```
- (void) handlePan: (UIPanGestureRecognizer *) uigr
{
    if (uigr.state == UIGestureRecognizerStateEnded)
    {
        CGPoint newCenter = CGPointMake(
            self.center.x + self.transform.tx,
            self.center.y + self.transform.ty);
        self.center = newCenter;

        CGAffineTransform theTransform = self.transform;
        theTransform.tx = 0.0f;
        theTransform.ty = 0.0f;
        self.transform = theTransform;

        return;
    }

    CGPoint translation = [uigr translationInView:self.superview];
    CGAffineTransform theTransform = self.transform;
    theTransform.tx = translation.x;
    theTransform.ty = translation.y;
    self.transform = theTransform;
}
```

Notice how the recognizer checks for the end of interaction and then updates the view's position and resets the transform's translation. This adaptation requires no local storage and would eliminate the need for a `touchesBegan:withEvent:` method. Without these modifications, Recipe 1-2 has to store previous state.

Recipe 1-2 **Using a Pan Gesture Recognizer to Drag Views**

```
@interface DragView : UIImageView
{
    CGPoint previousLocation;
}
@end

@implementation DragView
- (id) initWithImage: (UIImage *) anImage
{
    if (self = [super initWithImage:anImage])
    {
        self.userInteractionEnabled = YES;
        UIPanGestureRecognizer *panRecognizer =
            [[UIPanGestureRecognizer alloc]
```

```
                initWithTarget:self action:@selector(handlePan:)];
        self.gestureRecognizers = @[panRecognizer];
    }
    return self;
}

- (void) touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event
{
    // Promote the touched view
    [self.superview bringSubviewToFront:self];

    // Remember original location
    previousLocation = self.center;
}

- (void) handlePan: (UIPanGestureRecognizer *) uigr
{
    CGPoint translation = [uigr translationInView:self.superview];
    self.center = CGPointMake(previousLocation.x + translation.x,
        previousLocation.y + translation.y);
}
@end
```

> **Get This Recipe's Code**
>
> To find this recipe's full sample project, point your browser to https://github.com/erica/iOS-6-Cookbook and go to the folder for Chapter 1.

Recipe: Using Multiple Gesture Recognizers Simultaneously

Recipe 1-3 builds off the ideas presented in Recipe 1-2, but with several differences. First, it introduces multiple recognizers that work in parallel. To achieve this, the code uses three separate recognizers—rotation, pinch, and pan—and adds them all to the `DragView`'s `gestureRecognizers` property. It assigns the `DragView` as the delegate for each recognizer. This allows the `DragView` to implement the `gestureRecognizer:shouldRecognizeSimultaneouslyWithGestureRecognizer:` delegate method, enabling these recognizers to work simultaneously. Until this method is added to return `YES` as its value, only one recognizer will take charge at a time. Using parallel recognizers allows you to, for example, both zoom and rotate in response to a user's pinch gesture.

> **Note**
>
> `UITouch` objects store an array of gesture recognizers. The items in this array represent each recognizer that receives the touch object in question. When a view is created without gesture recognizers, its responder methods will be passed touches with empty recognizer arrays.

Recipe 1-3 extends the view's state to include scale and rotation instance variables. These items keep track of previous transformation values and permit the code to build compound affine transforms. These compound transforms, which are established in Recipe 1-3's `updateTransformWithOffset:` method, combine translation, rotation, and scaling into a single result. Unlike the previous recipe, this recipe uses transforms uniformly to apply changes to its objects, which is the standard practice for recognizers.

Finally, this recipe introduces a hybrid approach to gesture recognition. Instead of adding a `UITapGestureRecognizer` to the view's recognizer array, Recipe 1-3 demonstrates how you can add the kind of basic touch method used in Recipe 1-1 to catch a triple-tap. In this example, a triple-tap resets the view back to the identity transform. This undoes any manipulation previously applied to the view and reverts it to its original position, orientation, and size. As you can see, the touches began, moved, ended, and cancelled methods work seamlessly alongside the gesture recognizer callbacks, which is the point of including this extra detail in this recipe. Adding a tap recognizer would have worked just as well.

This recipe demonstrates the conciseness of using gesture recognizers to interact with touches.

Recipe 1-3 **Recognizing Gestures in Parallel**

```
@interface DragView : UIImageView <UIGestureRecognizerDelegate>
{
    CGFloat tx; // x translation
    CGFloat ty; // y translation
    CGFloat scale; // zoom scale
    CGFloat theta; // rotation angle
}
@end

@implementation DragView
- (void) touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event
{
    // Promote the touched view
    [self.superview bringSubviewToFront:self];

    // initialize translation offsets
    tx = self.transform.tx;
    ty = self.transform.ty;
}
```

```
- (void) touchesEnded:(NSSet *)touches withEvent:(UIEvent *)event
{
    UITouch *touch = [touches anyObject];
    if (touch.tapCount == 3)
    {
        // Reset geometry upon triple-tap
        self.transform = CGAffineTransformIdentity;
        tx = 0.0f; ty = 0.0f; scale = 1.0f; theta = 0.0f;
    }
}

- (void) touchesCancelled:(NSSet *)touches withEvent:(UIEvent *)event
{
    [self touchesEnded:touches withEvent:event];
}

- (void) updateTransformWithOffset: (CGPoint) translation
{
    // Create a blended transform representing translation,
    // rotation, and scaling
    self.transform = CGAffineTransformMakeTranslation(
        translation.x + tx, translation.y + ty);
    self.transform = CGAffineTransformRotate(self.transform, theta);
    self.transform = CGAffineTransformScale(self.transform, scale, scale);
}

- (void) handlePan: (UIPanGestureRecognizer *) uigr
{
    CGPoint translation = [uigr translationInView:self.superview];
    [self updateTransformWithOffset:translation];
}

- (void) handleRotation: (UIRotationGestureRecognizer *) uigr
{
    theta = uigr.rotation;
    [self updateTransformWithOffset:CGPointZero];
}

- (void) handlePinch: (UIPinchGestureRecognizer *) uigr
{
    scale = uigr.scale;
    [self updateTransformWithOffset:CGPointZero];
}

- (BOOL)gestureRecognizer:(UIGestureRecognizer *)gestureRecognizer
    shouldRecognizeSimultaneouslyWithGestureRecognizer:
        (UIGestureRecognizer *)otherGestureRecognizer
```

```
{
    return YES;
}

- (id) initWithImage:(UIImage *)image
{
    // Initialize and set as touchable
    if (!(self = [super initWithImage:image])) return nil;

    self.userInteractionEnabled = YES;

    // Reset geometry to identities
    self.transform = CGAffineTransformIdentity;
    tx = 0.0f; ty = 0.0f; scale = 1.0f; theta = 0.0f;

    // Add gesture recognizer suite
    UIRotationGestureRecognizer *rot = [[UIRotationGestureRecognizer alloc]
        initWithTarget:self action:@selector(handleRotation:)];
    UIPinchGestureRecognizer *pinch = [[UIPinchGestureRecognizer alloc]
        initWithTarget:self action:@selector(handlePinch:)];
    UIPanGestureRecognizer *pan = [[UIPanGestureRecognizer alloc]
        initWithTarget:self action:@selector(handlePan:)];
    self.gestureRecognizers = @[rot, pinch, pan];
    for (UIGestureRecognizer *recognizer in self.gestureRecognizers)
        recognizer.delegate = self;

    return self;
}
@end
```

Resolving Gesture Conflicts

Gesture conflicts may arise when you need to recognize several types of gestures at the same time. For example, what happens when you need to recognize both single- and double-taps? Should the single-tap recognizer fire at the first tap, even when the user intends to enter a double-tap? Or should you wait and respond only after it's clear that the user isn't about to add a second tap? The iOS SDK allows you to take these conflicts into account in your code.

Your classes can specify that one gesture must fail in order for another to succeed. Accomplish this by calling `requireGestureRecognizerToFail:`. This is a gesture method that takes one argument, another gesture recognizer. This call creates a dependency between the object receiving this message and another gesture object. What it means is this: For the first gesture to trigger, the second gesture must fail. If the second gesture is recognized, the first gesture will not be.

In real life, this typically means that the recognizer adds a delay until it can be sure that the dependent recognizer has failed. It waits until the second gesture is no longer possible. Only then does the first recognizer complete. If you recognize both single- and double-taps, the application waits a little longer after the first tap. If no second tap happens, the single-tap fires. Otherwise, the double-tap fires, but not both.

Your GUI responses will slow down to accommodate this change. Your single-tap responses become slightly laggy. That's because there's no way to tell if a second tap is coming until time elapses. You should never use both kinds of recognizers where instant responsiveness is critical to your user experience. Try, instead, to design around situations where that tap means "do something *now*" and avoid requiring both gestures for those modes.

Don't forget that you can add, remove, and disable gesture recognizers on-the-fly. A single-tap may take your interface to a place where it then makes sense to further distinguish between single- and double-taps. When leaving that mode, you could disable or remove the double-tap recognizer to regain better single-tap recognition. Tweaks like this limit interface slowdowns to where they're absolutely needed.

> **Get This Recipe's Code**
>
> To find this recipe's full sample project, point your browser to https://github.com/erica/iOS-6-Cookbook and go to the folder for Chapter 1.

Recipe: Constraining Movement

One problem with the simple approach of the earlier recipes in this chapter is that it's entirely possible to drag a view offscreen to the point where the user cannot see or easily recover it. Those recipes use unconstrained movement. There is no check to test whether the object remains in view and is touchable. Recipe 1-4 fixes this problem by constraining a view's movement to within its parent.

It achieves this by limiting movement in each direction, splitting its checks into separate x and y constraints. This two-check approach allows the view to continue to move even when one direction has passed its maximum. If the view has hit the rightmost edge of its parent, for example, it can still move up and down.

Figure 1-1 shows a sample interface. The subviews (flowers) are constrained into the black rectangle in the center of the interface and cannot be dragged off-view. Recipe 1-4's code is general and can adapt to parent bounds and child views of any size.

Recipe 1-4 **Bounded Movement**

```
- (void) handlePan: (UIPanGestureRecognizer *) uigr
{
    CGPoint translation = [uigr translationInView:self.superview];
    CGPoint newcenter = CGPointMake(
```

```
        previousLocation.x + translation.x,
        previousLocation.y + translation.y);

    // Restrict movement into parent bounds
    float halfx = CGRectGetMidX(self.bounds);
    newcenter.x = MAX(halfx, newcenter.x);
    newcenter.x = MIN(self.superview.bounds.size.width - halfx,
        newcenter.x);

    float halfy = CGRectGetMidY(self.bounds);
    newcenter.y = MAX(halfy, newcenter.y);
    newcenter.y = MIN(self.superview.bounds.size.height - halfy,
        newcenter.y);

    // Set new location
    self.center = newcenter;
}
```

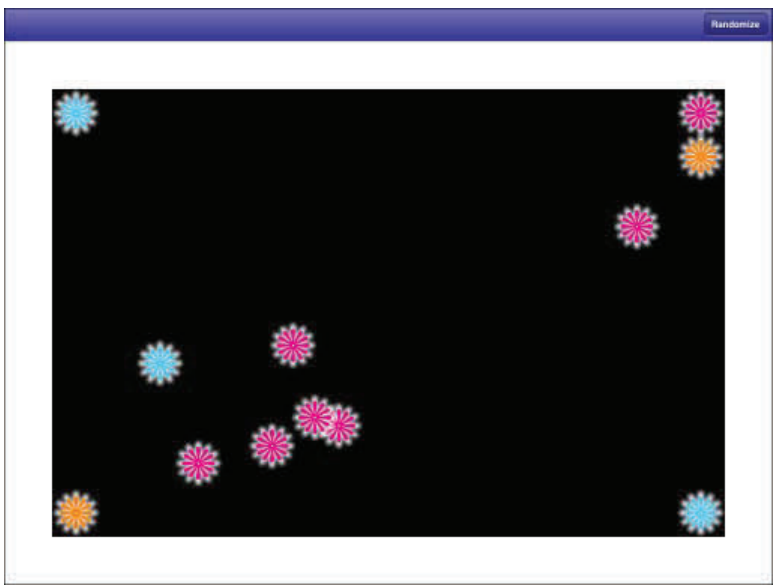

Figure 1-1 The movement of these flowers is bounded into the black rectangle.

Get This Recipe's Code

To find this recipe's full sample project, point your browser to https://github.com/erica/iOS-6-Cookbook and go to the folder for Chapter 1.

Recipe: Testing Touches

Most onscreen view elements for direct manipulation interfaces are not rectangular. This complicates touch detection because parts of the actual view rectangle may not correspond to actual touch points. Figure 1-2 shows the problem in action. The screenshot on the right shows the interface with its touch-based subviews. The shot on the left shows the actual view bounds for each subview. The light gray areas around each onscreen circle fall within the bounds, but touches to those areas should not "hit" the view in question.

iOS senses user taps throughout the entire view frame. This includes the undrawn area, such as the corners of the frame outside the actual circles of Figure 1-2, just as much as the primary presentation. That means that unless you add some sort of hit test, users may attempt to tap through to a view that's "obscured" by the clear portion of the `UIView` frame.

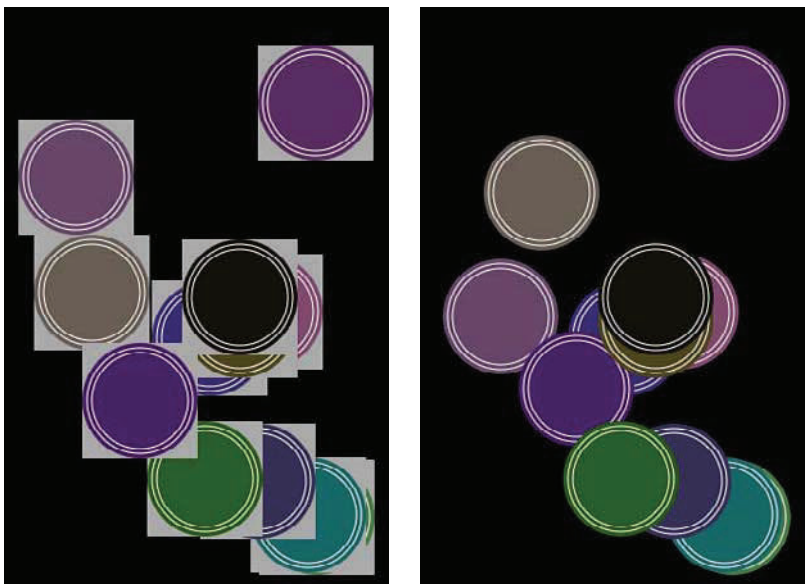

Figure 1-2 The application should ignore touches to the gray areas that surround each circle (left). The actual interface (right) uses a clear background (zero alpha values) to hide the parts of the view that are not used.

Visualize your actual view bounds by setting its background color, for example:

```
dragger.backgroundColor = [UIColor lightGrayColor];
```

This adds the backsplashes shown in Figure 1-2 (left) without affecting the actual onscreen art. In this case, the art consists of a centered circle with a transparent background. Unless you add some sort of test, all taps to any portion of this frame are captured by the view in question. Enabling background colors offers a convenient debugging aid to visualize the true extent of

each view; don't forget to comment out the background color assignment in production code. Alternatively, you can set a view layer's border width or style.

Recipe 1-5 adds a simple hit test to the views, determining whether touches fall within the circle. This test overrides the standard `UIView`'s `pointInside:withEvent:` method. This method returns either `YES` (the point falls inside the view) or `NO` (it does not). The test here uses basic geometry, checking whether the touch lies within the circle's radius. You can provide any test that works with your onscreen views. As you see in Recipe 1-6, which follows in the next section, that test can be expanded for much finer control.

Be aware that the math for touch detection on Retina display devices remains the same as that for older units. The extra onboard pixels do not affect your gesture-handling math. Your view's coordinate system remains floating point with subpixel accuracy. The number of pixels the device uses to draw to the screen does not affect `UIView` bounds and `UITouch` coordinates. It simply provides a way to provide higher detail graphics within that coordinate system.

> **Note**
>
> Do not confuse the point inside test, which checks whether a point falls inside a view, with the similar-sounding `hitTest:withEvent:`. The hit test returns the topmost view (closest to the user/screen) in a view hierarchy that contains a specific point. It works by calling `pointInside:withEvent:` on each view. If the point inside method returns `YES`, the search continues down that hierarchy.

Recipe 1-5 **Providing a Circular Hit Test**

```
- (BOOL) pointInside:(CGPoint)point withEvent:(UIEvent *)event
{
    CGPoint pt;
    float HALFSIDE = SIDELENGTH / 2.0f;

    // normalize with centered origin
    pt.x = (point.x - HALFSIDE) / HALFSIDE;
    pt.y = (point.y - HALFSIDE) / HALFSIDE;

    // x^2 + y^2 = radius^2
    float xsquared = pt.x * pt.x;
    float ysquared = pt.y * pt.y;

    // If the radius <= 1, the point is within the clipped circle
    if ((xsquared + ysquared) <= 1.0) return YES;
    return NO;
}
```

> **Get This Recipe's Code**
>
> To find this recipe's full sample project, point your browser to https://github.com/erica/iOS-6-Cookbook and go to the folder for Chapter 1.

Recipe: Testing Against a Bitmap

Unfortunately, most views don't fall into the simple geometries that make the hit test from Recipe 1-5 so straightforward. The flowers shown in Figure 1-1, for example, offer irregular boundaries and varied transparencies. For complicated art, it helps to test touches against a bitmap. Bitmaps provide byte-by-byte information about the contents of an image-based view, allowing you to test whether a touch hits a solid portion of the image or should pass through to any views below.

Recipe 1-6 extracts an image bitmap from a `UIImageView`. It assumes that the image used provides a pixel-by-pixel representation of the view in question. When you distort that view (normally by resizing a frame or applying a transform), update the math accordingly. `CGPoints` can be transformed via `CGPointApplyAffineTransform()` to handle scaling and rotation changes. Keeping the art at a 1:1 proportion to the actual view pixels simplifies lookup and avoids any messy math. You can recover the pixel in question, test its alpha level, and determine whether the touch has hit a solid portion of the view.

This example uses a cutoff of 85. That corresponds to a minimum alpha level of 33% (that is, 85 / 255). This custom `pointInside:` method considers any pixel with an alpha level below 33% to be transparent. This is arbitrary. Use any level (or other test for that matter) that works with the demands of your actual GUI.

> **Note**
>
> Unless you need pixel-perfect touch detection, you can probably scale down the bitmap so that it uses less memory and adjust the detection math accordingly.

Recipe 1-6 **Testing Touches Against Bitmap Alpha Levels**

```
// Return the offset for the alpha pixel at (x,y) for RGBA
// 4-bytes-per-pixel bitmap data
static NSUInteger alphaOffset(NSUInteger x, NSUInteger y, NSUInteger w)
    {return y * w * 4 + x * 4;}

// Return the bitmap from a provided image
NSData *getBitmapFromImage(UIImage *image)
{
    CGColorSpaceRef colorSpace = CGColorSpaceCreateDeviceRGB();
    if (colorSpace == NULL)
    {
```

```
        fprintf(stderr, "Error allocating color space\n");
        return NULL;
    }

    CGSize size = image.size;
    unsigned char *bitmapData = calloc(size.width * size.height * 4, 1);
    if (bitmapData == NULL)
    {
        fprintf (stderr, "Error: Memory not allocated!");
        CGColorSpaceRelease(colorSpace);
        return NULL;
    }

    CGContextRef context = CGBitmapContextCreate (bitmapData,
        size.width, size.height, 8, size.width * 4, colorSpace,
        kCGImageAlphaPremultipliedFirst);
    CGColorSpaceRelease(colorSpace );
    if (context == NULL)
    {
        fprintf (stderr, "Error: Context not created!");
        free (bitmapData);
        return NULL;
    }

    CGRect rect = CGRectMake(0.0f, 0.0f, size.width, size.height);
    CGContextDrawImage(context, rect, image.CGImage);
    unsigned char *data = CGBitmapContextGetData(context);
    CGContextRelease(context);

    NSData *bytes = [NSData dataWithBytes:data length:size.width * size.height * 4];
    free(bitmapData);

    return bytes;
}

// Store the bitmap data into an NSData instance variable
- (id) initWithImage: (UIImage *) anImage
{
    if (self = [super initWithImage:anImage])
    {
        self.userInteractionEnabled = YES;
        data = getBitmapFromImage(anImage);
    }
    return self;
}
```

```
// Does the point hit the view?
- (BOOL) pointInside:(CGPoint)point withEvent:(UIEvent *)event
{
    if (!CGRectContainsPoint(self.bounds, point)) return NO;
    Byte *bytes = (Byte *)data.bytes;
    uint offset = alphaOffset(point.x, point.y, self.image.size.width);
    return (bytes[offset] > 85);
}
```

> **Get This Recipe's Code**
>
> To find this recipe's full sample project, point your browser to https://github.com/erica/iOS-6-Cookbook and go to the folder for Chapter 1.

Recipe: Drawing Touches Onscreen

`UIView` hosts the realm of direct onscreen drawing. Its `drawRect:` method offers a low-level way to draw content directly, letting you create and display arbitrary elements using Quartz 2D calls. Touch plus drawing join together to build concrete, manipulatable interfaces.

Recipe 1-7 combines gestures with `drawRect` to introduce touch-based painting. As a user touches the screen, the `TouchTrackerView` class builds a Bezier path that follows the user's finger. To paint the progress as the touch proceeds, the `touchesMoved:withEvent:` method calls `setNeedsDisplay`. This, in turn, triggers a call to `drawRect:`, where the view strokes the accumulated Bezier path. Figure 1-3 shows the interface with a path created in this way.

Figure 1-3 A simple painting tool for iOS requires little more than collecting touches along a path and painting that path with UIKit/Quartz 2D calls.

Although you could adapt this recipe to use gesture recognizers, there's really no point to it. The touches are essentially meaningless, only provided to create a pleasing tracing. The basic responder methods (namely touches began, moved, and so on) are perfectly capable of handling path creation and management tasks.

This example is meant for creating continuous traces. It does not respond to any touch event without a move. If you want to expand this recipe to add a simple dot or mark, you'll have to add that behavior yourself.

Recipe 1-7 **Touch-Based Painting in a `UIView`**

```
@interface TouchTrackerView : UIView
{
    UIBezierPath *path;
}
@end

@implementation TouchTrackerView
- (void) touchesBegan:(NSSet *) touches withEvent:(UIEvent *) event
{
    // Initialize a new path for the user gesture
    self.path = [UIBezierPath bezierPath];
    path.lineWidth = 4.0f;

    UITouch *touch = [touches anyObject];
    [path moveToPoint:[touch locationInView:self]];
}

- (void) touchesMoved:(NSSet *) touches withEvent:(UIEvent *) event
{
    // Add new points to the path
    UITouch *touch = [touches anyObject];
    [self.path addLineToPoint:[touch locationInView:self]];
    [self setNeedsDisplay];
}

- (void) touchesEnded:(NSSet *)touches withEvent:(UIEvent *)event
{
        UITouch *touch = [touches anyObject];
        [path addLineToPoint:[touch locationInView:self]];
        [self setNeedsDisplay];
}

- (void) touchesCancelled:(NSSet *)touches withEvent:(UIEvent *)event
{
        [self touchesEnded:touches withEvent:event];
}
```

```
- (void) drawRect:(CGRect)rect
{
    // Draw the path
    [path stroke];
}

- (id) initWithFrame:(CGRect)frame
{
    if (self = [super initWithFrame:frame])
        self.multipleTouchEnabled = NO;
    return self;
}
@end
```

> **Get This Recipe's Code**
>
> To find this recipe's full sample project, point your browser to https://github.com/erica/iOS-6-Cookbook and go to the folder for Chapter 1.

Recipe: Smoothing Drawings

Depending on the device in use and the amount of simultaneous processing involved, capturing user gestures may produce results that are rougher than desired. Touch events are often limited by CPU demands as well as by shaking hands. A smoothing algorithm can offset those limitations by interpolating between points. Figure 1-4 demonstrates the kind of angularity that derives from granular input and the smoothing that can be applied instead.

Catmull-Rom splines create continuous curves between key points. This algorithm ensures that each initial point you provide remains part of the final curve. The resulting path retains the original path's shape. You choose the number of interpolation points between each pair of reference points. The trade-off lies between processing power and greater smoothing. The more points you add, the more CPU resources you'll consume. As you can see when using the sample code that accompanies this chapter, a little smoothing goes a long way, even on newer devices. The latest iPad is so responsive that it's hard to draw a particularly jaggy line in the first place.

Recipe 1-8 demonstrates how to extract points from an existing Bezier path and then apply splining to create a smoothed result. Catmull-Rom uses four points at a time to calculate intermediate values between the second and third points, using a granularity you specify between those points.

Recipe 1-8 provides an example of just one kind of real-time geometric processing you might add to your applications. Many other algorithms out there in the world of computational geometry can be applied in a similar manner.

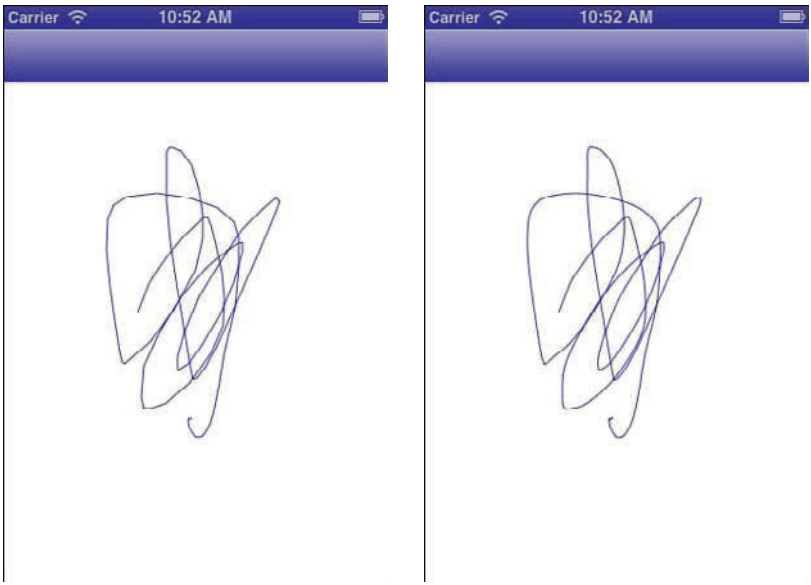

Figure 1-4 Catmull-Rom smoothing can be applied in real time to improve arcs between touch events. The images shown here are based on an identical gesture input, with and without smoothing applied.

Recipe 1-8 **Creating Smoothed Bezier Paths Using Catmull-Rom Splining**

```
#define VALUE(_INDEX_) [NSValue valueWithCGPoint:points[_INDEX_]]

@implementation UIBezierPath (Points)
void getPointsFromBezier(void *info, const CGPathElement *element)
{
    NSMutableArray *bezierPoints = (__bridge NSMutableArray *)info;

    // Retrieve the path element type and its points
    CGPathElementType type = element->type;
    CGPoint *points = element->points;

    // Add the points if they're available (per type)
    if (type != kCGPathElementCloseSubpath)
    {
        [bezierPoints addObject:VALUE(0)];
        if ((type != kCGPathElementAddLineToPoint) &&
            (type != kCGPathElementMoveToPoint))
            [bezierPoints addObject:VALUE(1)];
    }
```

```
    if (type == kCGPathElementAddCurveToPoint)
        [bezierPoints addObject:VALUE(2)];
}

- (NSArray *)points
{
    NSMutableArray *points = [NSMutableArray array];
    CGPathApply(self.CGPath, (__bridge void *)points, getPointsFromBezier);
    return points;
}
@end

#define POINT(_INDEX_) \
    [(NSValue *)[points objectAtIndex:_INDEX_] CGPointValue]

@implementation UIBezierPath (Smoothing)
- (UIBezierPath *) smoothedPath: (int) granularity
{
    NSMutableArray *points = [self.points mutableCopy];
    if (points.count < 4) return [self copy];

    // Add control points to make the math make sense
    // Via Josh Weinberg
    [points insertObject:[points objectAtIndex:0] atIndex:0];
    [points addObject:[points lastObject]];

    UIBezierPath *smoothedPath = [UIBezierPath bezierPath];

    // Copy traits
    smoothedPath.lineWidth = self.lineWidth;

    // Draw out the first 3 points (0..2)
    [smoothedPath moveToPoint:POINT(0)];

    for (int index = 1; index < 3; index++)
        [smoothedPath addLineToPoint:POINT(index)];

    for (int index = 4; index < points.count; index++)
    {
        CGPoint p0 = POINT(index - 3);
        CGPoint p1 = POINT(index - 2);
        CGPoint p2 = POINT(index - 1);
        CGPoint p3 = POINT(index);

        // now add n points starting at p1 + dx/dy up
        // until p2 using Catmull-Rom splines
        for (int i = 1; i < granularity; i++)
```

```
        {
            float t = (float) i * (1.0f / (float) granularity);
            float tt = t * t;
            float ttt = tt * t;

            CGPoint pi; // intermediate point
            pi.x = 0.5 * (2*p1.x+(p2.x-p0.x)*t +
               (2*p0.x-5*p1.x+4*p2.x-p3.x)*tt + (3*p1.x-p0.x-3*p2.x+p3.x)*ttt);
            pi.y = 0.5 * (2*p1.y+(p2.y-p0.y)*t +
               (2*p0.y-5*p1.y+4*p2.y-p3.y)*tt + (3*p1.y-p0.y-3*p2.y+p3.y)*ttt);
            [smoothedPath addLineToPoint:pi];
        }

        // Now add p2
        [smoothedPath addLineToPoint:p2];
    }

    // finish by adding the last point
    [smoothedPath addLineToPoint:POINT(points.count - 1)];

    return smoothedPath;
}

@end

// Example usage:
// Replace the path with a smoothed version after drawing completes
- (void) touchesEnded:(NSSet *)touches withEvent:(UIEvent *)event
{
    UITouch *touch = [touches anyObject];
    [path addLineToPoint:[touch locationInView:self]];
    path = [path smoothedPath:4];
    [self setNeedsDisplay];
}
```

> **Get This Recipe's Code**
>
> To find this recipe's full sample project, point your browser to https://github.com/erica/iOS-6-Cookbook and go to the folder for Chapter 1.

Recipe: Using Multi-Touch Interaction

Enabling Multi-Touch interaction in `UIView` instances lets iOS recover and respond to more than one finger touch at a time. Set the `UIView` property `multipleTouchEnabled` to `YES` or override `isMultipleTouchEnabled` for your view. When enabled, each touch callback returns

an entire set of touches. When that set's count exceeds 1, you know you're dealing with Multi-Touch.

In theory, iOS supports an arbitrary number of touches. You can explore that limit by running the following recipe on an iPad, using as many fingers as possible at once. The practical upper limit has changed over time; this recipe modestly demurs from offering a specific number.

When Multi-Touch was first explored on the iPhone, developers did not dream of the freedom and flexibility that Multi-Touch combined with multiple users offered. Adding Multi-Touch to your games and other applications opens up not just expanded gestures but also new ways of creating profoundly exciting multiuser experiences, especially on larger screens like the iPad. I encourage you to include Multi-Touch support in your applications wherever it is practical and meaningful.

Multi-Touch touches are not grouped. If you touch the screen with two fingers from each hand, for example, there's no way to determine which touches belong to which hand. The touch order is also arbitrary. Although grouped touches retain the same finger order (or, more specifically, the same memory address) for the lifetime of a single touch event, from touch down through movement to release, the correspondence between touches and fingers may and likely will change the next time your user touches the screen. When you need to distinguish touches from each other, build a touch dictionary indexed by the touch objects, as shown in this recipe.

Perhaps it's a comfort to know that if you need it, the extra finger support has been built in. Unfortunately, when you are using three or more touches at a time, the screen has a pronounced tendency to lose track of one or more of those fingers. It's hard to programmatically track smooth gestures when you go beyond two finger touches. So instead of focusing on gesture interpretation, think of the Multi-Touch experience more as a series of time-limited independent interactions. You can treat each touch as a distinct item and process it independently of its fellows.

Recipe 1-9 adds Multi-Touch to a `UIView` by setting its `multipleTouchEnabled` property and tracing the lines that each finger draws. It does this by keeping track of each touch's physical address in memory but without pointing to or retaining the touch per Apple's recommendations.

This is, obviously, an oddball approach, but it has worked reliably throughout the history of the SDK. That's because each `UITouch` object persists at a single address throughout the touch-move-release life cycle. Apple recommends against retaining `UITouch` instances, which is why the integer values of these objects are used as keys in this recipe. By using the physical address as a key, you can distinguish each touch, even as new touches are added or old touches are removed from the screen.

Be aware that new touches can start their life cycle via `touchesBegan:withEvent:` independently of others as they move, end, or cancel. Your code should reflect that reality.

This recipe expands from Recipe 1-7. Each touch grows a separate Bezier path, which is painted in the view's `drawRect` method. Recipe 1-7 essentially started a new drawing at the end of each

touch cycle. That worked well for application bookkeeping but failed when it came to creating a standard drawing application, where you expect to iteratively add elements to a picture.

Recipe 1-9 continues adding traces into a composite picture without erasing old items. Touches collect into an ever-growing mutable array, which can be cleared on user demand. This recipe draws in-progress tracing in a slightly lighter color, to distinguish it from paths that have already been stored to the drawing's stroke array.

Recipe 1-9 **Accumulating User Tracings for a Composite Drawing**

```
@interface TouchTrackerView : UIView
{
    NSMutableArray *strokes;
    NSMutableDictionary *touchPaths;
}
- (void) clear;
@end

@implementation TouchTrackerView

// Establish new views with storage initialized for drawing
- (id) initWithFrame:(CGRect)frame
{
    if (self = [super initWithFrame:frame])
    {
        self.multipleTouchEnabled = YES;
        strokes = [NSMutableArray array];
        touchPaths = [NSMutableDictionary dictionary];
    }

    return self;
}

// On clear remove all existing strokes, but not in-progress drawing
- (void) clear
{
    [strokes removeAllObjects];
    [self setNeedsDisplay];
}

// Start touches by adding new paths to the touchPath dictionary
- (void) touchesBegan:(NSSet *) touches withEvent:(UIEvent *) event
{
    for (UITouch *touch in touches)
    {
        NSString *key = [NSString stringWithFormat:@"%d", (int) touch];
        CGPoint pt = [touch locationInView:self];
```

```
        UIBezierPath *path = [UIBezierPath bezierPath];
        path.lineWidth = IS_IPAD? 8: 4;
        path.lineCapStyle = kCGLineCapRound;
        [path moveToPoint:pt];

        [touchPaths setObject:path forKey:key];
    }
}
// Trace touch movement by growing and stroking the path
- (void) touchesMoved:(NSSet *) touches withEvent:(UIEvent *) event
{
    for (UITouch *touch in touches)
    {
        NSString *key =
            [NSString stringWithFormat:@"%d", (int) touch];
        UIBezierPath *path = [touchPaths objectForKey:key];
        if (!path) break;

        CGPoint pt = [touch locationInView:self];
        [path addLineToPoint:pt];
    }

    [self setNeedsDisplay];
}

// On ending a touch, move the path to the strokes array
- (void) touchesEnded:(NSSet *)touches withEvent:(UIEvent *)event
{
    for (UITouch *touch in touches)
    {
        NSString *key = [NSString stringWithFormat:@"%d", (int) touch];
        UIBezierPath *path = [touchPaths objectForKey:key];
        if (path) [strokes addObject:path];
        [touchPaths removeObjectForKey:key];
    }

    [self setNeedsDisplay];
}

- (void) touchesCancelled:(NSSet *)touches withEvent:(UIEvent *)event
{
    [self touchesEnded:touches withEvent:event];
}

// Draw existing strokes in dark purple, in-progress ones in light
- (void) drawRect:(CGRect)rect
{
```

```
    [COOKBOOK_PURPLE_COLOR set];
    for (UIBezierPath *path in strokes)
        [path stroke];

    [[COOKBOOK_PURPLE_COLOR colorWithAlphaComponent:0.5f] set];
    for (UIBezierPath *path in [touchPaths allValues])
        [path stroke];
}
@end
```

> **Get This Recipe's Code**
>
> To find this recipe's full sample project, point your browser to https://github.com/erica/iOS-6-Cookbook and go to the folder for Chapter 1.

> **Note**
>
> Apple provides many Core Graphics/Quartz 2D resources on its developer website. Although many of these forums, mailing lists, and source code examples are not iOS specific, they offer an invaluable resource for expanding your iOS Core Graphics knowledge.

Recipe: Detecting Circles

In a direct manipulation interface like iOS, you'd imagine that most people could get by just pointing to items onscreen. And yet, circle detection remains one of the most requested gestures. Developers like having people circle items onscreen with their fingers. In the spirit of providing solutions that readers have requested, Recipe 1-10 offers a relatively simple circle detector, which is shown in Figure 1-5.

In this implementation, detection uses a multistep test. A time test checks that the stroke was not lingering. A circle gesture should be quickly drawn. There's an inflection test checking that the touch did not change directions too often. A proper circle includes four direction changes. This test allows for five. There's a convergence test. The circle must start and end close enough together that the points are somehow related. A fair amount of leeway is needed because when you don't provide direct visual feedback, users tend to undershoot or overshoot where they began. The pixel distance used here is generous, approximately a third of the view size.

The final test looks at movement around a central point. It adds up the arcs traveled, which should equal 360 degrees in a perfect circle. This example allows any movement that falls within 45 degrees for not-quite-finished circles and 180 degrees for circles that continue on a bit wider, allowing the finger to travel more naturally.

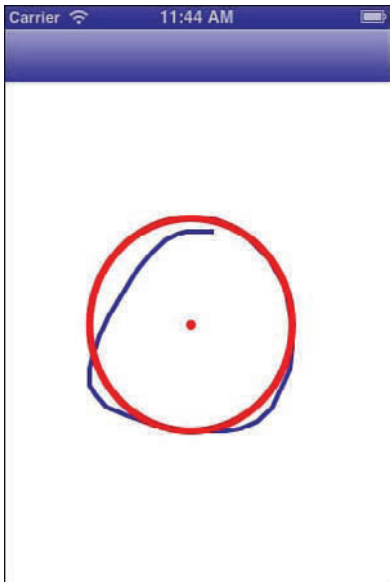

Figure 1-5 The dot and the outer ellipse show the key features of the detected circle.

Upon these tests being passed, the algorithm produces a least bounding rectangle and centers that rectangle on the geometric mean of the points from the original gesture. This result is assigned to the circle instance variable. It's not a perfect detection system (you can try to fool it when testing the sample code), but it's robust enough to provide reasonably good circle checks for many iOS applications.

Recipe 1-10 **Detecting Circles**

```
// Retrieve center of rectangle
CGPoint GEORectGetCenter(CGRect rect)
{
    return CGPointMake(CGRectGetMidX(rect), CGRectGetMidY(rect));
}

// Build rectangle around a given center
CGRect GEORectAroundCenter(CGPoint center, float dx, float dy)
{
    return CGRectMake(center.x - dx, center.y - dy, dx * 2, dy * 2);
}

// Center one rect inside another
CGRect GEORectCenteredInRect(CGRect rect, CGRect mainRect)
```

```
{
    CGFloat dx = CGRectGetMidX(mainRect)-CGRectGetMidX(rect);
    CGFloat dy = CGRectGetMidY(mainRect)-CGRectGetMidY(rect);
    return CGRectOffset(rect, dx, dy);
}

// Return dot product of two vectors normalized
CGFloat dotproduct (CGPoint v1, CGPoint v2)
{
    CGFloat dot = (v1.x * v2.x) + (v1.y * v2.y);
    CGFloat a = ABS(sqrt(v1.x * v1.x + v1.y * v1.y));
    CGFloat b = ABS(sqrt(v2.x * v2.x + v2.y * v2.y));
    dot /= (a * b);

    return dot;
}

// Return distance between two points
CGFloat distance (CGPoint p1, CGPoint p2)
{
    CGFloat dx = p2.x - p1.x;
    CGFloat dy = p2.y - p1.y;

    return sqrt(dx*dx + dy*dy);
}

// Offset in X
CGFloat dx(CGPoint p1, CGPoint p2)
{
    return p2.x - p1.x;
}

// Offset in Y
CGFloat dy(CGPoint p1, CGPoint p2)
{
    return p2.y - p1.y;
}

// Sign of a number
NSInteger sign(CGFloat x)
{
    return (x < 0.0f) ? (-1) : 1;
}

// Return a point with respect to a given origin
CGPoint pointWithOrigin(CGPoint pt, CGPoint origin)
{
```

```
    return CGPointMake(pt.x - origin.x, pt.y - origin.y);
}

// Calculate and return least bounding rectangle
#define POINT(_INDEX_) [(NSValue *)[points \
    objectAtIndex:_INDEX_] CGPointValue]

CGRect boundingRect(NSArray *points)
{
    CGRect rect = CGRectZero;
    CGRect ptRect;

    for (int i = 0; i < points.count; i++)
    {
        CGPoint pt = POINT(i);
        ptRect = CGRectMake(pt.x, pt.y, 0.0f, 0.0f);
        rect = (CGRectEqualToRect(rect, CGRectZero)) ?
            ptRect : CGRectUnion(rect, ptRect);
    }

    return rect;
}

CGRect testForCircle(NSArray *points, NSDate *firstTouchDate)
{
    if (points.count < 2)
    {
        NSLog(@"Too few points (2) for circle");
        return CGRectZero;
    }

    // Test 1: duration tolerance
    float duration = [[NSDate date]
        timeIntervalSinceDate:firstTouchDate];
    NSLog(@"Transit duration: %0.2f", duration);

    float maxDuration = 2.0f;
    if (duration > maxDuration)
    {
        NSLog(@"Excessive duration");
        return CGRectZero;
    }

    // Test 2: Direction changes should be limited to near 4
    int inflections = 0;
    for (int i = 2; i < (points.count - 1); i++)
    {
```

```
        float deltx = dx(POINT(i), POINT(i-1));
        float delty = dy(POINT(i), POINT(i-1));
        float px = dx(POINT(i-1), POINT(i-2));
        float py = dy(POINT(i-1), POINT(i-2));

        if ((sign(deltx) != sign(px)) ||
            (sign(delty) != sign(py)))
            inflections++;
    }

    if (inflections > 5)
    {
        NSLog(@"Excessive inflections");
        return CGRectZero;
    }

    // Test 3: Start and end points near each other
    float tolerance = [[[UIApplication sharedApplication]
        keyWindow] bounds].size.width / 3.0f;
    if (distance(POINT(0), POINT(points.count - 1)) > tolerance)
    {
        NSLog(@"Start too far from end");
        return CGRectZero;
    }

    // Test 4: Count the distance traveled in degrees.
    CGRect circle = boundingRect(points);
    CGPoint center = GEORectGetCenter(circle);
    float distance = ABS(acos(dotproduct(
        pointWithOrigin(POINT(0), center),
        pointWithOrigin(POINT(1), center))));
    for (int i = 1; i < (points.count - 1); i++)
        distance += ABS(acos(dotproduct(
            pointWithOrigin(POINT(i), center),
            pointWithOrigin(POINT(i+1), center))));

    float transitTolerance = distance - 2 * M_PI;

    if (transitTolerance < 0.0f) // fell short of 2 PI
    {
        if (transitTolerance < - (M_PI / 4.0f)) // under 45
        {
            NSLog(@"Transit too short");
            return CGRectZero;
        }
    }
```

```
    if (transitTolerance > M_PI) // additional 180 degrees
    {
        NSLog(@"Transit too long ");
        return CGRectZero;
    }

    return circle;
}
@end
```

> **Get This Recipe's Code**
>
> To find this recipe's full sample project, point your browser to https://github.com/erica/iOS-6-Cookbook and go to the folder for Chapter 1.

Recipe: Creating a Custom Gesture Recognizer

It takes little work to transform the code shown in Recipe 1-10 into a custom recognizer, as introduced in Recipe 1-11. Subclassing `UIGestureRecognizer` enables you to build your own circle recognizer that you can add to views in your applications.

Start by importing `UIGestureRecognizerSubclass.h` into your new class. The file declares everything you need your recognizer subclass to override or customize. For each method you override, make sure to call the original version of the method by calling the superclass method before invoking your new code.

Gestures fall into two types: continuous and discrete. The circle recognizer is discrete. It either recognizes a circle or fails. Continuous gestures include pinches and pans, where recognizers send updates throughout their life cycle. Your recognizer generates updates by setting its `state` property.

Recognizers are basically state machines for fingertips. All recognizers start in the possible state (`UIGestureRecognizerStatePossible`), and then for continuous gestures pass through a series of changed states (`UIGestureRecognizerStateChanged`). Discrete recognizers either succeed in recognizing a gesture (`UIGestureRecognizerStateRecognized`) or fail (`UIGestureRecognizerStateFailed`), as demonstrated in Recipe 1-11. The recognizer sends actions to its target each time you update state *except* when the state is set to possible or failed.

The rather long comments you see in Recipe 1-11 belong to Apple, courtesy of the subclass header file. I've included them here because they help explain the roles of the key methods that override their superclass. The `reset` method returns the recognizer back to its quiescent state, allowing it to prepare itself for its next recognition challenge.

The touches began (and so on) methods are called at similar points as their `UIResponder` analogs, enabling you to perform your tests at the same touch life cycle points. This example

waits to check for success or failure until the touches ended callback, and uses the same `testForCircle` method defined in Recipe 1-10.

> **Note**
>
> As an overriding philosophy, gesture recognizers should fail as soon as possible. When they succeed, you should store information about the gesture in local properties. The circle gesture should save any detected circle so users know where the gesture occurred.

Recipe 1-11 **Creating a Gesture Recognizer Subclass**

```
#import <UIKit/UIGestureRecognizerSubclass.h>
@implementation CircleRecognizer

// called automatically by the runtime after the gesture state has
// been set to UIGestureRecognizerStateEnded any internal state
// should be reset to prepare for a new attempt to recognize the gesture
// after this is received all remaining active touches will be ignored
// (no further updates will be received for touches that had already
// begun but haven't ended)
- (void)reset
{
    [super reset];

    points = nil;
    firstTouchDate = nil;
    self.state = UIGestureRecognizerStatePossible;
}

// mirror of the touch-delivery methods on UIResponder
// UIGestureRecognizers aren't in the responder chain, but observe
// touches hit-tested to their view and their view's subviews
// UIGestureRecognizers receive touches before the view to which
// the touch was hit-tested
- (void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event
{
    [super touchesBegan:touches withEvent:event];

    if (touches.count > 1)
    {
        self.state = UIGestureRecognizerStateFailed;
        return;
    }

    points = [NSMutableArray array];
    firstTouchDate = [NSDate date];
```

```
    UITouch *touch = [touches anyObject];
    [points addObject: [NSValue valueWithCGPoint:
        [touch locationInView:self.view]]];
}

- (void)touchesMoved:(NSSet *)touches withEvent:(UIEvent *)event
{
    [super touchesMoved:touches withEvent:event];
    UITouch *touch = [touches anyObject];
    [points addObject: [NSValue valueWithCGPoint:
        [touch locationInView:self.view]]];
}

- (void) touchesEnded:(NSSet *)touches withEvent:(UIEvent *)event
{
    [super touchesEnded:touches withEvent: event];
    BOOL detectionSuccess = !CGRectEqualToRect(CGRectZero,
        testForCircle(points, firstTouchDate));
    if (detectionSuccess)
        self.state = UIGestureRecognizerStateRecognized;
    else
        self.state = UIGestureRecognizerStateFailed;
}
@end
```

> **Get This Recipe's Code**
>
> To find this recipe's full sample project, point your browser to https://github.com/erica/iOS-6-Cookbook and go to the folder for Chapter 1.

Recipe: Dragging from a Scroll View

iOS's rich set of gesture recognizers doesn't always accomplish exactly what you're looking for. Here's an example. Imagine a horizontal scrolling view filled with image views, one next to another, so you can scroll left and right to see the entire collection. Now, imagine that you want to be able to drag items out of that view and add them to a space directly below the scrolling area. To do this, you need to recognize downward touches on those child views (that is, orthogonal to the scrolling direction).

This was the puzzle I encountered while trying to help developer Alex Hosgrove, who was trying to build an application roughly equivalent to a set of refrigerator magnet letters. Users could drag those letters down into a workspace and then play with and arrange the items they'd chosen. There were two challenges with this scenario. First, who owned each touch? Second, what happened after the downward touch was recognized?

Both the scroll view and its children own an interest in each touch. A downward gesture should generate new objects; a sideways gesture should pan the scroll view. Touches have to be shared to allow both the scroll view and its children to respond to user interactions. This problem can be solved using gesture delegates.

Gesture delegates allow you to add simultaneous recognition, so that two recognizers can operate at the same time. You add this behavior by declaring a protocol (`UIGestureRecognizerDelegate`) and adding a simple delegate method:

```
- (BOOL)gestureRecognizer:(UIGestureRecognizer *)gestureRecognizer
    shouldRecognizeSimultaneouslyWithGestureRecognizer:
        (UIGestureRecognizer *)otherGestureRecognizer
{
    return YES;
}
```

You cannot reassign gesture delegates for scroll views, so you must add this delegate override to the implementation for the scroll view's children.

The second question, converting a swipe into a drag, is addressed by thinking about the entire touch lifetime. Each touch that creates a new object starts as a directional drag but ends up as a pan once the new view is created. A pan recognizer works better here than a swipe recognizer, whose lifetime ends at the point of recognition.

To make this happen, Recipe 1-12 manually adds that directional-movement detection, outside of the built-in gesture detection. In the end, that working-outside-the-box approach provides a major coding win. That's because once the swipe has been detected, the underlying pan gesture recognizer continues to operate. This allows the user to keep moving the swiped object without having to raise his or her finger and retouch the object in question.

This implementation detects swipes that move down at least 16 vertical pixels without straying more than 8 pixels to either side. When this code detects a downward swipe, it adds a new `DragView` (the same class used earlier in this chapter) to the screen and allows it to follow the touch for the remainder of the pan gesture interaction.

At the point of recognition, the class marks itself as having handled the swipe (`gestureWasHandled`) and disables the scroll view for the duration of the panning event. This allows the child complete control over the ongoing pan gesture without the scroll view reacting to further touch movement.

Recipe 1-12 **Dragging Items Out of Scroll Views**

```
@implementation DragView

#define DX(p1, p2)    (p2.x - p1.x)
#define DY(p1, p2)    (p2.y - p1.y)

#define SWIPE_DRAG_MIN 16
#define DRAGLIMIT_MAX 8
```

```
// Categorize swipe types
typedef enum {
    TouchUnknown,
    TouchSwipeLeft,
    TouchSwipeRight,
    TouchSwipeUp,
    TouchSwipeDown,
} SwipeTypes;

@implementation PullView
// Create a new view with an embedded pan gesture recognizer
- (id) initWithImage: (UIImage *) anImage
{
    if (self = [super initWithImage:anImage])
    {
        self.userInteractionEnabled = YES;
        UIPanGestureRecognizer *pan =
            [[[UIPanGestureRecognizer alloc] initWithTarget:self
                action:@selector(handlePan:)] autorelease];
        pan.delegate = self;
        self.gestureRecognizers = @[pan];
}

// Allow simultaneous recognition
- (BOOL)gestureRecognizer:(UIGestureRecognizer *)gestureRecognizer
    shouldRecognizeSimultaneouslyWithGestureRecognizer:
        (UIGestureRecognizer *)otherGestureRecognizer
{
    return YES;
}

// Handle pans by detecting swipes
- (void) handlePan: (UISwipeGestureRecognizer *) uigr
{
    // Only deal with scroll view superviews
    if (![self.superview isKindOfClass:[UIScrollView class]]) return;

    // Extract superviews
    UIView *supersuper = self.superview.superview;
    UIScrollView *scrollView = (UIScrollView *) self.superview;

    // Calculate location of touch
    CGPoint touchLocation = [uigr locationInView:supersuper];

    // Handle touch based on recognizer state
```

```
if(uigr.state == UIGestureRecognizerStateBegan)
{
    // Initialize recognizer
    gestureWasHandled = NO;
    pointCount = 1;
    startPoint = touchLocation;
}

if(uigr.state == UIGestureRecognizerStateChanged)
{
    pointCount++;

    // Calculate whether a swipe has occurred
    float dx = DX(touchLocation, startPoint);
    float dy = DY(touchLocation, startPoint);

    BOOL finished = YES;
    if ((dx > SWIPE_DRAG_MIN) && (ABS(dy) < DRAGLIMIT_MAX))
        touchtype = TouchSwipeLeft;
    else if ((-dx > SWIPE_DRAG_MIN) && (ABS(dy) < DRAGLIMIT_MAX))
        touchtype = TouchSwipeRight;
    else if ((dy > SWIPE_DRAG_MIN) && (ABS(dx) < DRAGLIMIT_MAX))
        touchtype = TouchSwipeUp;
    else if ((-dy > SWIPE_DRAG_MIN) && (ABS(dx) < DRAGLIMIT_MAX))
        touchtype = TouchSwipeDown;
    else
        finished = NO;

    // If unhandled and a downward swipe, produce a new draggable view
    if (!gestureWasHandled && finished &&
        (touchtype == TouchSwipeDown))
    {
        dv = [[DragView alloc] initWithImage:self.image];
        dv.center = touchLocation;
        [supersuper addSubview:dv];
        scrollView.scrollEnabled = NO;
        gestureWasHandled = YES;
    }
    else if (gestureWasHandled)
    {
        // allow continued dragging after detection
        dv.center = touchLocation;
    }
}

if(uigr.state == UIGestureRecognizerStateEnded)
{
```

```
        // ensure that the scroll view returns to scrollable
        if (gestureWasHandled)
            scrollView.scrollEnabled = YES;
    }
}
@end
```

> **Get This Recipe's Code**
>
> To find this recipe's full sample project, point your browser to https://github.com/erica/iOS-6-Cookbook and go to the folder for Chapter 1.

Recipe: Live Touch Feedback

Have you ever needed to record a demo for an iOS app? There's always compromise involved. Either you use an overhead camera and struggle with reflections and the user's hand blocking the screen or you use a tool like Reflection (http://reflectionapp.com) but you only get to see what's directly on the iOS device screen. These app recordings lack any indication of the user's touch and visual focus.

Recipe 1-13 offers a simple set of classes (called *TOUCHkit*) that provide a live touch feedback layer for demonstration use. With it, you can see both the screen that you're recording as well as the touches that create the interactions you're trying to present. It provides a way to compile your app for both normal and demonstration deployment. You don't change your core application to use it. It's designed to work as a single toggle, providing builds for each use.

To demonstrate this, the code shown in Recipe 1-13 is bundled in the sample code repository with a standard Apple demo. This shows how you can roll the kit into nearly any standard application.

Enabling Touch Feedback

You add touch feedback by switching on the TOUCHkit feature, without otherwise affecting your normal code. To enable TOUCHkit, you set a single flag, compile and use that build for demonstration, complete with touch overlay. For App Store deployment, you disable the flag. The application reverts to its normal behavior, and there are no App Store unsafe calls to worry about:

```
#define USES_TOUCHkit    1
```

This recipe assumes that you're using a standard application with a single primary window. When compiled in, the kit replaces that window with a custom class that captures and duplicates all touches, allowing your application to show the user's touch bubble feedback.

There is one key code-level change you must make, but it's a very small one. In your application delegate class, you define a `WINDOW_CLASS` to use when building your iOS screen:

```
#if USES_TOUCHkit
#import "TOUCHkitView.h"
#import "TOUCHOverlayWindow.h"
#define WINDOW_CLASS TOUCHOverlayWindow
#else
#define WINDOW_CLASS UIWindow
#endif
```

Then instead of declaring a `UIWindow`, you use whichever class has been set by the toggle:

```
WINDOW_CLASS *window;
window = [[WINDOW_CLASS alloc]
    initWithFrame:[[UIScreen mainScreen] bounds]];
```

From here, you can set the window's `rootViewController` as normal.

Intercepting and Forwarding Touch Events

The key to this overlay lies in intercepting touch events, creating a floating presentation above your normal interface, and then forwarding those events on to your application. A TOUCHkit view lies on top of your interface. The custom window class grabs user touch events and presents them as circles in the TOUCHkit view. It then forwards them as if the user were interacting with a normal `UIWindow`. To accomplish this, this recipe uses event forwarding.

Event forwarding is achieved by calling a secondary event handler. The `TOUCHOverlayWindow` class overrides `UIWindow`'s `sendEvent:` method to force touch drawing and then invokes its superclass implementation to return control to the normal responder chain.

The following implementation is drawn from Apple's Event Handling Guide for iOS. It collects all the touches associated with the current event, allowing Multi-Touch as well as single touch interactions, dispatches them to TOUCHkit view layer, and then redirects them to the window via the normal `UIWindow sendEvent:` implementation:

```
@implementation TOUCHOverlayWindow
- (void) sendEvent:(UIEvent *)event
{
    // Collect touches
    NSSet *touches = [event allTouches];
    NSMutableSet *began = nil;
    NSMutableSet *moved = nil;
    NSMutableSet *ended = nil;
    NSMutableSet *cancelled = nil;

    // Sort the touches by phase for event dispatch
    for(UITouch *touch in touches) {
        switch ([touch phase]) {
```

```
            case UITouchPhaseBegan:
                if (!began) began = [NSMutableSet set];
                [began addObject:touch];
                break;
            case UITouchPhaseMoved:
                if (!moved) moved = [NSMutableSet set];
                [moved addObject:touch];
                break;
            case UITouchPhaseEnded:
                if (!ended) ended = [NSMutableSet set];
                [ended addObject:touch];
                break;
            case UITouchPhaseCancelled:
                if (!cancelled) cancelled = [NSMutableSet set];
                [cancelled addObject:touch];
                break;
            default:
                break;
        }
    }

    // Create pseudo-event dispatch
    if (began)
        [[TOUCHkitView sharedInstance]
            touchesBegan:began withEvent:event];
    if (moved)
        [[TOUCHkitView sharedInstance]
            touchesMoved:moved withEvent:event];
    if (ended)
        [[TOUCHkitView sharedInstance]
            touchesEnded:ended withEvent:event];
    if (cancelled)
        [[TOUCHkitView sharedInstance]
            touchesCancelled:cancelled withEvent:event];

    // Call normal handler for default responder chain
    [super sendEvent: event];
}
@end
```

Implementing the TOUCHkit Overlay View

The TOUCHkit overlay is a single clear `UIView` singleton. It's created the first time the application requests its shared instance, and the call adds it to the application's key window. The overlay's user interaction flag is disabled, allowing touches to continue on through the responder

chain, even after processing those touches through the standard began/moved/ended/cancelled event callbacks.

The touch processing events draw a circle at each touch point, creating a strong pointer to the touches until that drawing is complete. Recipe 1-13 details the callback and drawing methods that handle that functionality.

Recipe 1-13 **Creating a Touch Feedback Overlay View**

```
+ (id) sharedInstance
{
    // Create shared instance if it does not yet exist
    if(!sharedInstance)
    {
        sharedInstance = [[self alloc] initWithFrame:CGRectZero];
    }

    // Parent it to the key window
    if (!sharedInstance.superview)
    {
        UIWindow *keyWindow= [UIApplication sharedApplication].keyWindow;
        sharedInstance.frame = keyWindow.bounds;
        [keyWindow addSubview:sharedInstance];
    }

    return sharedInstance;
}

// You can override the default touchColor if you want
- (id) initWithFrame:(CGRect)frame
{
    if (self = [super initWithFrame:frame])
    {
        self.backgroundColor = [UIColor clearColor];
        self.userInteractionEnabled = NO;
        self.multipleTouchEnabled = YES;
        touchColor =
            [[UIColor whiteColor] colorWithAlphaComponent:0.5f];
        touches = nil;
    }

    return self;
}

// Basic Touches processing
- (void) touchesBegan:(NSSet *)theTouches withEvent:(UIEvent *)event
{
```

```
    touches = theTouches;
    [self setNeedsDisplay];
}

- (void) touchesMoved:(NSSet *)theTouches withEvent:(UIEvent *)event
{
    touches = theTouches;
    [self setNeedsDisplay];
}

- (void) touchesEnded:(NSSet *)theTouches withEvent:(UIEvent *)event
{
    touches = nil;
    [self setNeedsDisplay];
}

// Draw touches interactively
- (void) drawRect: (CGRect) rect
{
    // Clear
    CGContextRef context = UIGraphicsGetCurrentContext();
    CGContextClearRect(context, self.bounds);

    // Fill see-through
    [[UIColor clearColor] set];
    CGContextFillRect(context, self.bounds);

    float size = 25.0f; // based on 44.0f standard touch point

    for (UITouch *touch in touches)
    {
        // Create a backing frame
        [[[UIColor darkGrayColor] colorWithAlphaComponent:0.5f] set];
        CGPoint aPoint = [touch locationInView:self];
        CGContextAddEllipseInRect(context,
            CGRectMake(aPoint.x - size, aPoint.y - size, 2 * size, 2 * size));
        CGContextFillPath(context);

        // Draw the foreground touch
        float dsize = 1.0f;
        [touchColor set];
        aPoint = [touch locationInView:self];
        CGContextAddEllipseInRect(context,
            CGRectMake(aPoint.x - size - dsize, aPoint.y - size - dsize,
                2 * (size - dsize), 2 * (size - dsize)));
        CGContextFillPath(context);
    }
```

```
    // Reset touches after use
    touches = nil;
}
```

> **Get This Recipe's Code**
>
> To find this recipe's full sample project, point your browser to https://github.com/erica/iOS-6-Cookbook and go to the folder for Chapter 1.

Recipe: Adding Menus to Views

The `UIMenuController` class allows you to add pop-up menus to any item that acts as a first responder. Normally menus are used with text views and text fields, enabling users to select, copy, and paste. Menus also provide a way to add actions to interactive elements like the small drag views used throughout this chapter. Figure 1-6 shows a customized menu. In Recipe 1-14, this menu is presented after long-tapping a flower. The actions will zoom, rotate, or hide the associated drag view.

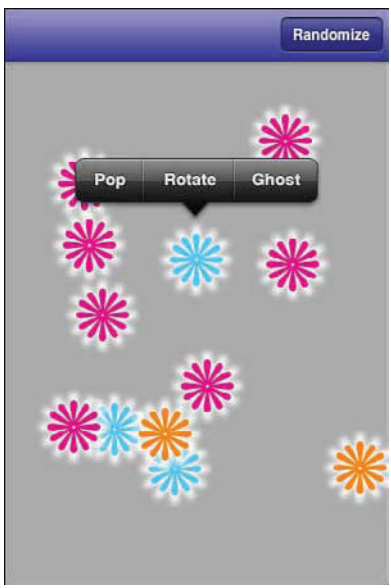

Figure 1-6 Contextual pop-up menus allow you to add interactive actions to first responder views.

This recipe demonstrates how to retrieve the shared menu controller and assign items to it. Set the menu's target rectangle (typically the bounds of the view that presents it), adjust the menu's arrow direction, and update the menu with your changes. The menu can now be set visible.

Menu items work with standard target-action callbacks, but you do not assign the target directly. Their target is always the first responder view. This recipe omits a `canPerformAction:withSender:` responder check, but you'll want to add that if some views support certain actions and other views do not. With menus, that support is often tied to state. For example, you don't want to offer a copy command if the view has no content to copy.

Recipe 1-14 **Adding Menus to Interactive Views**

```
- (BOOL) canBecomeFirstResponder
{
    // Menus only work with first responders
    return YES;
}

- (void) pressed: (UILongPressGestureRecognizer *) recognizer
{
    if (![self becomeFirstResponder])
    {
        NSLog(@"Could not become first responder");
        return;
    }

    UIMenuController *menu = [UIMenuController sharedMenuController];
    UIMenuItem *pop = [[UIMenuItem alloc]
        initWithTitle:@"Pop" action:@selector(popSelf)];
    UIMenuItem *rotate = [[UIMenuItem alloc]
        initWithTitle:@"Rotate" action:@selector(rotateSelf)];
    UIMenuItem *ghost = [[UIMenuItem alloc]
        initWithTitle:@"Ghost" action:@selector(ghostSelf)];
    [menu setMenuItems:@[pop, rotate, ghost]];

    [menu setTargetRect:self.bounds inView:self];
    menu.arrowDirection = UIMenuControllerArrowDown;
    [menu update];

    [menu setMenuVisible:YES];
}
```

```
- (id) initWithImage: (UIImage *) anImage
{
    if (!(self = [super initWithImage:anImage])) return nil;

    self.userInteractionEnabled = YES;
    UILongPressGestureRecognizer *pressRecognizer =
        [[UILongPressGestureRecognizer alloc] initWithTarget:self
            action:@selector(pressed:)];
    [self addGestureRecognizer:pressRecognizer];

    return self;
}
```

> **Get This Recipe's Code**
>
> To find this recipe's full sample project, point your browser to https://github.com/erica/iOS-6-Cookbook and go to the folder for Chapter 1.

Summary

`UIViews` and their underlying layers provide the onscreen components your users see. Touch input lets users interact directly with views via the `UITouch` class and gesture recognizers. As this chapter has shown, even in their most basic form, touch-based interfaces offer easy-to-implement flexibility and power. You discovered how to move views around the screen and how to bound that movement. You read about testing touches to see whether views should or should not respond to them. You saw how to "paint" on a view and how to attach recognizers to views to interpret and respond to gestures. Here's a collection of thoughts about the recipes in this chapter that you might want to ponder before moving on:

- Be concrete. iOS devices have perfectly good touch screens. Why not let your users drag items around the screen or trace lines with their fingers? It adds to the reality and the platform's interactive nature.
- Users typically have five fingers per hand. iPads, in particular, offer a lot of screen real estate. Don't limit yourself to a one-finger interface when it makes sense to expand your interaction into Multi-Touch territory for one or more users, screen space allowing.
- A solid grounding in Quartz graphics and Core Animation will be your friend. Using `drawRect:`, you can build any kind of custom `UIView` presentation you want, including text, Bezier curves, scribbles, and so forth.
- If Cocoa Touch doesn't provide the kind of specialized gesture recognizer you're looking for, write your own. It's not that hard, although it helps to be as thorough as possible when considering the states your custom touch might pass through.

- Use Multi-Touch whenever possible, especially when you can expand your application to invite more than one user to touch the screen at a time. Don't limit yourself to one-person, one-touch interactions when a little extra programming will open doors of opportunity for multiuser use.
- Explore! This chapter only touched lightly on the ways you can use direct manipulation in your applications. Use this material as a jumping-off point to explore the full vocabulary of the `UITouch` class.

This page intentionally left blank

Index

SYMBOLS

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z