www.it-ebooks.info

Building Apps That
Make the iPad Shine

Beginning iPad Development for iPhone Developers

Mastering the iPad SDK

Jack Nutting | Dave Wooldridge | David Mark

Apress®

www.it-ebooks.info

www.it-ebooks.info

Beginning iPad Development for iPhone Developers

Mastering the iPad SDK

Jack Nutting
Dave Woolridge
David Mark

Apress®

www.it-ebooks.info

Beginning iPad Development for iPhone Developers: Mastering the iPad SDK

Copyright © 2010 by Jack Nutting, Dave Wooldridge, David Mark

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

ISBN-13 (pbk): 978-1-4302-3021-2

ISBN-13 (electronic): 978-1-4302-3022-9

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

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

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

President and Publisher: Paul Manning
Lead Editor: Clay Andres
Development Editor: Douglas Pundick
Technical Reviewer: Mark Dalrymple
Editorial Board: Clay Andres, Steve Anglin, Mark Beckner, Ewan Buckingham, Gary Cornell, Jonathan Gennick, Jonathan Hassell, Michelle Lowman, Matthew Moodie, Duncan Parkes, Jeffrey Pepper, Frank Pohlmann, Douglas Pundick, Ben Renow-Clarke, Dominic Shakeshaft, Matt Wade, Tom Welsh
Coordinating Editor: Kelly Moritz
Copy Editor: Marilyn Smith
Compositor: MacPS, LLC
Indexer: Kevin Broccoli
Artist: April Milne
Cover Designer: Anna Ishchenko

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

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

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

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

The source code for this book is available to readers at www.apress.com. You will need to answer questions pertaining to this book in order to successfully download the code.

www.it-ebooks.info

To Weronica, for believing in me

—Jack Nutting

To my wonderful wife and soul mate, Madeline, for her amazing love and support

—Dave Wooldridge

To my best friend and partner in life, Deneen

LFU4FREIH

—David Mark

www.it-ebooks.info

Contents at a Glance

www.it-ebooks.info

Contents

www.it-ebooks.info

www.it-ebooks.info

www.it-ebooks.info

www.it-ebooks.info

About the Authors

Jack Nutting has been using Cocoa since the olden days, long before it was even called Cocoa. He has used Cocoa and its predecessors to develop software for a wide range of industries and applications, including gaming, graphic design, online digital distribution, telecommunications, finance, publishing, and travel. When he isn't working on Mac, iPhone, or iPad projects, he is developing web applications with Ruby on Rails. Jack is a passionate proponent of Objective-C and the Cocoa frameworks. At the drop of a hat, he will speak at length on the virtues of dynamic dispatch and runtime class manipulations to anyone who will listen (and even to some who won't). Jack is the principal author of *Learn Cocoa on the Mac* (Apress, 2010). He blogs from time to time at `www.nuthole.com`, and you can follow his more frequent random musings at `twitter.com/jacknutting`.

As the founder of Electric Butterfly, **Dave Wooldridge** has been developing award-winning web sites and software for 15 years. When he is not creating Mac and iOS apps, he can be found writing. Dave is the author of *The Business of iPhone App Development: Making and Marketing Apps that Succeed* (Apress, 2010). He also has written numerous articles for leading tech publications, including a monthly software marketing column for *MacTech*. Follow Dave at `twitter.com/ebutterfly`.

Dave Mark is a longtime Mac developer and author. His books include *Beginning iPhone 3 Development* (Apress, 2009), *Learn C on the Mac* (Apress, 2009), *The Macintosh Programming Primer* series (Addison-Wesley, 1992), and *Ultimate Mac Programming* (Wiley, 1995). Dave loves the water and spends as much time as possible on it, in it, or near it. He lives with his wife and three children in Virginia.

www.it-ebooks.info

About the Technical Reviewer

Mark Dalrymple is a longtime Mac and Unix programmer, working on cross-platform toolkits, Internet publishing tools, high-performance web servers, and end-user desktop applications. He is the principal author of *Advanced Mac OS X Programming* (Big Nerd Ranch, 2005) and *Learn Objective-C on the Mac* (Apress, 2009). In his spare time, he plays trombone and bassoon and makes balloon animals.

www.it-ebooks.info

Acknowledgments

The authors wish to thank the editorial staff at Apress who helped make this book possible: Clay Andres, who assembled the team and got the ball rolling; Kelly Moritz, whose vigilance in pulling all of our strings at the appropriate times has been crucial in moving the book forward along its ambitious schedule; Douglas Pundick, who helped to bring coherence and flow to each chapter; and Marilyn Smith, whose expert wordsmithing has helped make us seem like better writers on nearly every page of the book. We are also eternally indebted to Mark Dalrymple, whose keen eye and rapier wit have helped us avoid many technical missteps.

www.it-ebooks.info

Preface

The world has changed. Since work on this book began, the iPad was released (selling three million units in the first 80 days), and the iPhone OS was renamed to iOS, which debuted as iOS 4 in the new iPhone 4 (which was preordered by 600,000 people the first day). Meanwhile, more than 11,000 apps were released for the iPad—a mix of iPad-only apps and universal apps, which can run on both the iPhone and the iPad. By any measure, the iPad is a runaway hit. It is already inspiring many creative uses far beyond the sort of ultimate media-consumption device that Apple began touting it as back in January.

By the time this book goes to print and reaches your hands, Xcode 4 may be available (throwing a monkey-wrench into our careful descriptions of using Xcode and Interface Builder), Apple will have sold one or two million more iPads, and iOS 4 may even be available for iPad. Apple is keeping iPhone and iPad developers on their toes, and authors are no exception! We've kept all of this in mind while writing this book, and have worked to make a book that will stand the test of time, regardless of OS versions and release dates.

At the end of the day, iPhone and iPad are inherently two different beasts, with different form factors and capabilities that encourage different usage patterns, despite the similarities in their underlying OSes; Beginning iPad Development for iPhone Developers is meant to highlight those differences, helping you build upon your iPhone development knowledge with new tools and techniques to let you create great iPad apps!

www.it-ebooks.info

Chapter 1

Welcome to the Paradigm Shift

Unless you've been living under a rock, you're well aware that the new darling in Apple's product line is the iPad, a thin, touchscreen tablet that aims to revolutionize mobile computing and portable media consumption. The iPad was undoubtedly one of the most heavily rumored, hyped, and anticipated product launches in recent memory… at least since Apple's introduction of the original iPhone in 2007.

One major difference here is that the first iPhone model did not include an App Store. It wasn't until a little more than a year later that Apple launched the iTunes App Store, with only 500 native apps from third-party developers. Fast-forward to 2010, and you'll find more than 200,000 apps in the App Store. With the iPad's ability to run most of those existing apps without any modifications, users will have access to a vast catalog of software immediately upon powering up their brand-new iPads. But iPhone apps pale in comparison to the sheer beauty and flexibility of native iPad apps!

With the new iPad-only features and user interface elements offered in iPhone SDK 3.2 and the powerful graphics and processing engine under the hood, the iPad represents a much greater business opportunity for developers than even the early days of the iPhone. Unlike the iPhone, with its limited memory constraints and small screen, the iPad offers developers a unique mobile platform for creating truly sophisticated, desktop-quality apps!

But to take advantage of this exciting new opportunity and develop apps that consumers want, it's important to understand who the iPad was designed for.

Reinventing the Tablet

So why a tablet? To carve out a new category that sits between the laptop and the smartphone, the device must satisfy a need that is currently not being delivered by those other products. For the past few years, netbooks have tried to bridge that gap. But as Steve Jobs has famously remarked, netbooks are nothing but cheap laptops in a

www.it-ebooks.info

small form factor. Running Microsoft Windows or a Linux-based operating system, netbooks don't make computing any easier than laptops. Their only advantage is that they're more affordable.

A thin tablet device is much more intimate than a laptop or netbook, and it can be easily held from almost any angle—on a couch or in bed, for example. And I'm willing to bet that some of you even take your iPad into the bathroom for a little quality time, surfing the Web and reading e-books.

The iPad is certainly not the first computer tablet to hit the market, and it won't be the last. Since the early 1990s, countless companies have attempted to lure consumers with feature-packed tablet models, but none of them were ever successful. Why? Because the software was either limited in functionality or too difficult to use.

In 1993, Apple launched the Newton MessagePad, its first stab at a tablet. With its monochrome screen and limited selection of software, it was largely perceived as a big PDA, rather than a true mobile computer. Since the product never seemed to grow beyond a small, yet loyal, cult following, Apple discontinued Newton development in 1998.

Most of the other hardware companies that followed with their own tablets tried a different approach. Running on various flavors of Windows or Linux, those tablets were powerful computers, but were ultimately not the right mobile solution for most consumers. Like Apple's Newton, many of them required the use of a stylus pen in order to accurately tap the tiny on-screen buttons and menu options. The underlying problem was that those desktop operating systems were never designed for a finger-driven touchscreen. From a usability standpoint, a cursor-based desktop operating system is a very cumbersome interface for a mobile tablet device that's typically operated with one hand.

It's All About the Software

When rumors first started circulating back in 2009 about the development of a mysterious Apple tablet, the big question was which operating system it would run. With a larger screen, it could certainly handle Mac OS X, and I must admit, a small part of me had secretly hoped that Apple would announce a Mac-based tablet, so that I could run my favorite Mac apps on it. But as a developer, I'm not the average consumer. If Apple had released the iPad as a Mac OS X-powered device, it would surely have met the same lukewarm reception as the countless tablets that came before it.

But Apple is smarter than that. To succeed, Apple knew this new class of mobile device had to be easier to use than a laptop, yet more powerful than a smartphone. To achieve this, the tablet needed an operating system that was engineered from the ground up for multitouch finger gestures and efficient touchscreen navigation. One of Apple's greatest strengths as a technology company is that it controls the design of both the hardware and the software, always striving for a seamless marriage between the two. The iPad is a perfect example of that ideology.

Powered by a tablet-enhanced version of iOS, the iPad avoids the usual trappings of adapting mouse-driven desktop software to a touchscreen environment. With millions of iPhone and iPod touch users already familiar with the iOS interface, there's an

www.it-ebooks.info

immediate comfort level with the iPad. It looks easy to use because it is. When Apple first announced the iPad, the tablet's emphasis on simplicity seemed to underwhelm some critics, but that is the very element that will make it a game-changer in the world of mobile computing.

The iPad Is *Not* a Big iPod touch

The naysayers who are skeptical of the iPad's future—merely passing it off as a big iPod touch—are the ones who don't get it. They don't see the big picture here. The instant-on iOS proved ideal for a small smartphone device, and it will prove even more effective for the iPad's larger screen. But don't be fooled by its deceptive exterior. Sure, it may *look* like a super-sized iPod touch, but under the hood, the iPad boasts a powerful graphics engine and Apple's speedy, battery-optimized A4 processor.

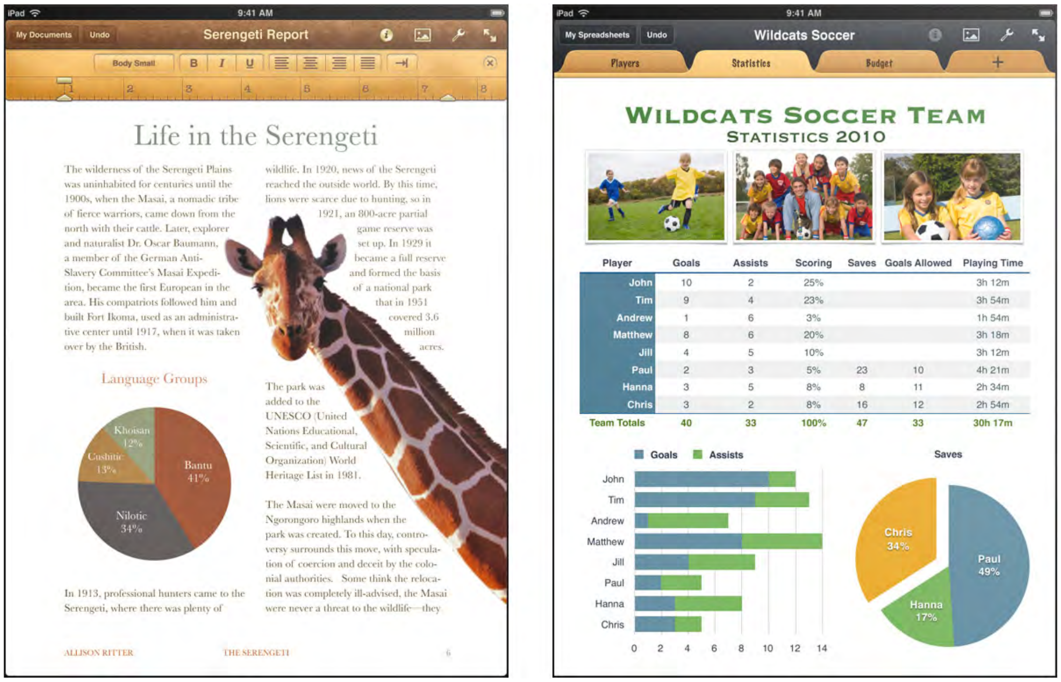
The iPad is not just about games. Even though games have made the iPod touch a dominant force in portable gaming, and many of those titles will be optimized for Apple's tablet, I believe the iPad will become a popular platform for productivity apps—even more so than the iPhone.

Beyond the convenience of the larger display and full-size touchscreen keyboard, iPhone SDK 3.2 includes iPad support for Core Text and several other exciting new frameworks and user interface niceties that make it much easier to develop feature-rich productivity apps. Apple has set the stage for the iPad to become the portable computer of choice for not only the general public, but also doctors, teachers, students, salespeople, scientists, pilots, engineers, and countless other markets.

Apple hinted as much with its new iWork suite for the iPad (see Figure 1–1). By delivering sophisticated iPad versions of its Mac counterparts (Keynote, Pages, and Numbers), iWork is Apple's shot across the bow at critics, proving its new tablet is so much more than just a glorified iPod.

When a laptop is too unwieldy or too heavy to carry around all day, the iPad is a much more practical form factor, capable of running state-of-the-art, desktop-caliber applications. And with the simplicity of the iOS interface, this new class of mobile apps will increase productivity and provide a much more intimate and immersive experience that is easily accessible to even the most nontechnical neophytes.

www.it-ebooks.info



Life in the Serengeti

The wilderness of the Serengeti Plains was uninhabited for centuries until the 1900s, when the Masai, a nomadic tribe of fierce warriors, came down from the north with their cattle. Later, explorer and naturalist Dr. Oscar Baumann, a member of the German Anti-Slavery Committee's Masai Expedition, became the first European in the area. His compatriots followed him and built Fort Ikoma, used as an administrative center until 1917, when it was taken over by the British.

Language Groups

In 1913, professional hunters came to the Serengeti, where there was plenty of wildlife. In 1920, news of the Serengeti reached the outside world. By this time, lions were scarce due to hunting, so in 1921, an 800-acre partial game reserve was set up. In 1929 it became a full reserve and formed the basis of a national park that in 1951 covered 3.6 million acres.

The park was added to the UNESCO (United Nations Educational, Scientific, and Cultural Organization) World Heritage List in 1981.

The Masai were moved to the Ngorongoro highlands when the park was created. To this day, controversy surrounds this move, with speculation of coercion and deceit by the colonial authorities. Some think the relocation was completely ill-advised, the Masai were never a threat to the wildlife—they

ALLISON RITTER THE SERENGETI 6

WILDCATS SOCCER TEAM
STATISTICS 2010

Player	Goals	Assists	Scoring	Saves	Goals Allowed	Playing Time
John	10	2	25%			3h 12m
Tim	9	4	23%			3h 54m
Andrew	1	6	3%			1h 54m
Matthew	8	6	20%			3h 18m
Jill	4	5	10%			3h 12m
Paul	2	3	5%	23	10	4h 21m
Hanna	3	5	8%	8	11	2h 34m
Chris	3	2	8%	16	12	2h 54m
Team Totals	40	33	100%	47	33	30h 17m

Figure 1–1. *Pages (left) and Numbers (right) are part of Apple's iWork suite for the iPad, a perfect showcase of how sophisticated, desktop-quality apps can be designed for ease of use on a mobile touchscreen.*

Inspired by the iPad's potential, the prominent iPhone development firm, Agile Web Solutions, began working on iPad apps as soon as Apple released the first SDK betas. David Chartier, Chief Media Producer of Agile Web Solutions, says this about the iPad's potential:

> *Some write off the iPad as a "big iPod touch," but that's shortsighted. I think the iOS on a larger screen will allow for a much more engaging multi-touch experience. Just look at all the features Apple included in its iWork [for iPad]. The iPad offers much more powerful hardware and more features for developers. This really could become a secondary computer, or even a primary one for a lot of users, and that's really compelling.*

www.it-ebooks.info

Personal Computing for the Masses

When exploring the new iPad-centric SDK features in later chapters, you'll immediately see that Apple has provided extensive tools for creating very robust apps. With this newfound power at your fingertips, many of you may be inclined to build apps that mimic traditional desktop interface models, derived from years of programming for Mac OS X, Microsoft Windows, or Linux. Even though the iPad platform removes many of the user interface design restraints and memory limitations that developers grappled with on the iPhone, this would be entirely the wrong approach.

There's a reason Apple used iOS instead of Mac OS X as the iPad's operating system. Beyond the fact that it was designed for a finger-based touchscreen, iOS also serves another valuable role. Unlike traditional desktop operating systems, iOS hides the filesystem from users, placing the focus instead on content.

Although many see the iPad as a mere laptop replacement, I believe Apple's new tablet was designed with a more ambitious goal: to reinvent mainstream personal computing, much like the original Mac did back in 1984. Apple feels that computers have become far too complicated for the average consumer and aims to simplify the experience. Although desktop operating systems like Mac OS X will continue to thrive for years to come, the iPad presents a major paradigm shift in computing.

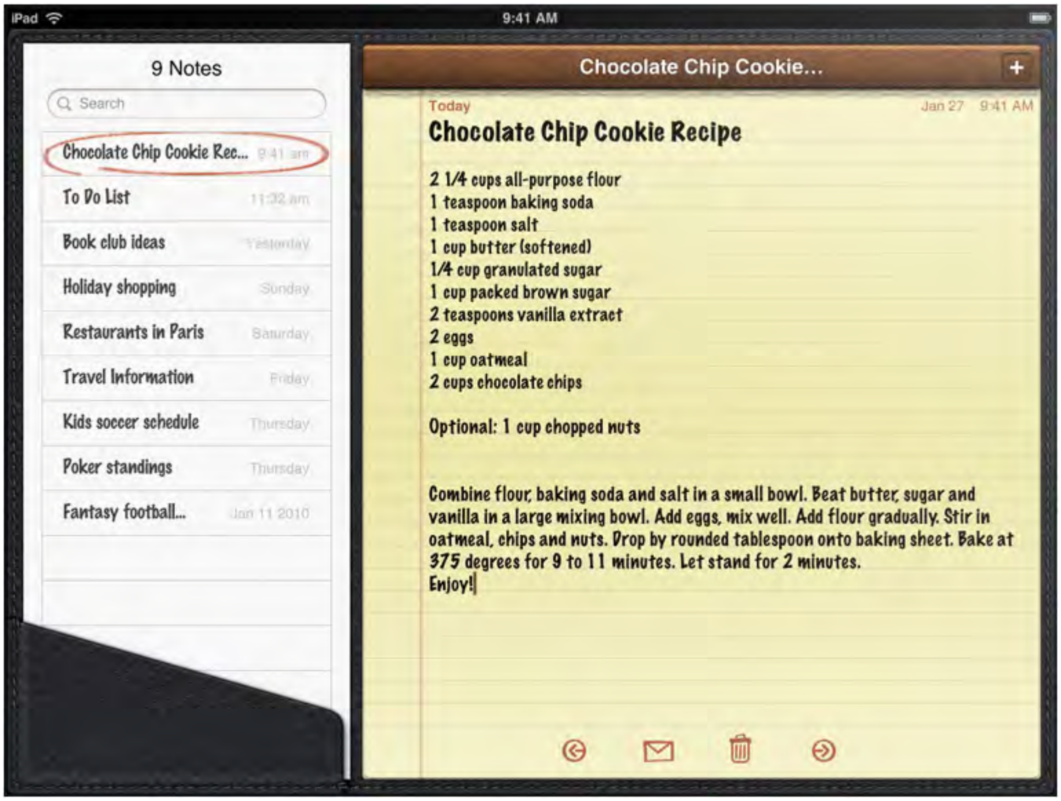
The iPad was designed for people who don't like using computers. And yet it's packed with enough engineering muscle to easily accommodate the needs of power users. In a nutshell, the iPad is the portable, personal computer for *everyone*.

The genius of the iPad is in its sheer simplicity. I know I've mentioned this a few times already, but it is the single most important factor to remember when developing apps for this new device. A major component in Apple's strategy to deliver a more organic and intuitive user experience is incorporating real-world metaphors into the interface design process. A good example of this is Apple's preinstalled Notes app (see Figure 1–2).

A computer newbie could launch the Notes app for the first time and immediately figure out how to use it, without the need for any instructions or prior computer knowledge. Your grandparents could use this app! To make the experience fun, Apple even added realistic graphical flourishes, such as the stitched-leather binder that holds both the yellow, lined notepad and the white, card-based notes list. And marking the currently selected note with a "hand-drawn" red circle is a nice touch!

Apple encourages developers to embrace this sea change in their own app interface designs as well. Obviously, integrating real-world objects, imagery, and textures to communicate functionality won't be practical in every scenario, but the primary objective is clear: keep it simple. Remember this fundamental rule as you mull over potential app ideas to develop. We'll be exploring additional user interface design considerations in Chapter 3.

www.it-ebooks.info


iPad
9:41 AM
9 Notes
Search
Chocolate Chip Cookie Rec... 9:41 am
To Do List 11:32 am
Book club ideas Yesterday
Holiday shopping Sunday
Restaurants in Paris Saturday
Travel Information Friday
Kids soccer schedule Thursday
Poker standings Thursday
Fantasy football... Jan 11 2010
Chocolate Chip Cookie...
+
Today
Jan 27 9:41 AM

Chocolate Chip Cookie Recipe

2 1/4 cups all-purpose flour
1 teaspoon baking soda
1 teaspoon salt
1 cup butter (softened)
1/4 cup granulated sugar
1 cup packed brown sugar
2 teaspoons vanilla extract
2 eggs
1 cup oatmeal
2 cups chocolate chips

Optional: 1 cup chopped nuts

Combine flour, baking soda and salt in a small bowl. Beat butter, sugar and vanilla in a large mixing bowl. Add eggs, mix well. Add flour gradually. Stir in oatmeal, chips and nuts. Drop by rounded tablespoon onto baking sheet. Bake at 375 degrees for 9 to 11 minutes. Let stand for 2 minutes.
Enjoy!

Figure 1–2. *Apple designed the user interface of its Notes app to emulate a physical notepad, eliminating the learning curve for first-time users.*

The continued success of the iPad rests solely on the software that powers it. Apple's revolutionary tablet provides developers with a new opportunity to create apps for people who want a simplified computer experience without having to sacrifice on features. The iPad apps that succeed will be the ones that are packed with functionality while remaining highly intuitive and easy to use.

Like most developers, I constantly find myself serving the role of tech support for family and friends. Most of the time, the problems they encounter involve locating misplaced files and e-mail attachments, deciphering endless configuration options, grappling with arcane software installers, and so on. If they all had iPads—downloading and using apps easily with only a few finger taps—I can guarantee I would be spending a lot more quality time with my family and fewer precious hours troubleshooting their archaic desktop machines. At some point in the near future, everyone will begin to recognize the iPad for what it really is: the next evolutionary step in personal computing for the masses.

www.it-ebooks.info

Developing Apps for the iPad

Even though most of the 200,000 apps in the App Store will run "as is" on the iPad, the small 320-by-480 pixel dimensions of an iPhone app are less than half the size of the iPad's large 768-by-1024 pixel screen. Although the iPad includes backward-compatibility support for iPhone apps, the end result leaves much to be desired. Apple provides only two options for running iPhone apps on the iPad: displayed at normal size in the center of the screen (the rest of the unused area is left black) or magnified two times to fill the screen. The iPad's scaling algorithm seems to work fairly well, but full-screen iPhone apps still appear rather pixelized. After becoming spoiled by beautiful, high-resolution iPad apps, users will find magnified iPhone apps on the same large screen very crude and unattractive.

When Your iPhone App Is No Longer Good Enough

Although your existing iPhone app may run fine on the iPad, don't settle for an inferior user experience. iPhone apps were designed for the iPhone. The iPad should be treated as an entirely new platform, with its own set of design requirements. Consumers will certainly grow weary of running pixelized iPhone apps on the iPad, especially if iPad-enhanced alternatives are available in the App Store. With this in mind, it's never too soon to begin developing iPad versions of your apps.

Apple is encouraging this new breed of iPad-optimized apps by showcasing them in a special iPad section of the App Store. Obviously, it's in Apple's best interests to champion iPad app development, since an extensive selection of iPad apps will help sell more iPads. And this, in turn, will ultimately help you sell more apps, as the number of new iPad owners increase, all flocking to the App Store to download new software.

As proof of this development push, none of Apple's preinstalled apps were left untouched. Apple took the time to redesign all of them—such as Mail, Calendar, Contacts, Photos, Safari, and even Notes (see Figure 1–2)—to utilize the expanded screen space and new interface capabilities of the iPad platform. And most app developers seem to agree that this is the right direction to take to properly meet user expectations. Here's what David Chartier of Agile Web Solutions had to say on the subject:

> *Sure, there's that 2x button for running existing iPhone apps in a full-screen mode on the iPad. But I think that once iPad customers see what's capable with the iPad's increased screen space and hardware and software advantages over the iPhone and iPod touch, you will find that the "2x" mode quickly becomes the Mac OS Classic on the iPad. To succeed on the iPad, there's no question in my mind that developers will need to incorporate the new features and interface tools to provide the best user experience. If you don't, users won't hesitate to check out your competition.*

www.it-ebooks.info

Knowing that iPad users won't be content with running blown-up, pixelized iPhone apps, developers are racing to port their existing iPhone apps into new, enhanced iPad versions. Their efforts go far beyond simply scaling the interface to accommodate the larger screen real estate. Major changes to app navigation and user interface architecture are being implemented to take advantage of the iPad's unique software and hardware features.

There are several important design methodologies and recommended interface guidelines to consider when developing apps for the iPad, all of which will be discussed in great detail in Chapter 3. For now, it's time for a little inspiration to get the creative juices flowing. Let's take a look at how several iPhone developers are retooling their apps for the iPad.

Exploring the Possibilities

A handful of well-known developers were kind enough to share their insights about developing apps for the iPad platform. In taking a closer look at these apps, several iPad-specific user interface elements and concepts are mentioned. If you're unfamiliar with any of them, don't worry. All of the new iPad frameworks and user interface controls available in iPhone SDK 3.2 will be thoroughly explained throughout the rest of the book, starting with Chapter 3.

Brushes

Steve Sprang's acclaimed iPhone app, Brushes (`http://brushesapp.com/`), is a painting program designed exclusively for the mobile screen. With a deceptively simple interface, Brushes is packed with features, such as an advanced color picker, several realistic brushes, multiple layers, extreme zooming, and even undo/redo options. It is a powerful tool for painting on the iPhone, which has spawned a vast community of mobile digital artists.

Choosing a new color or a different brush requires moving to a new screen view. Due to the iPhone's small size, this is a necessary design strategy to keep the user interface uncluttered and easy to use. Once a selection is made, the artist can then return to the main canvas screen (see Figure 1–3).

www.it-ebooks.info

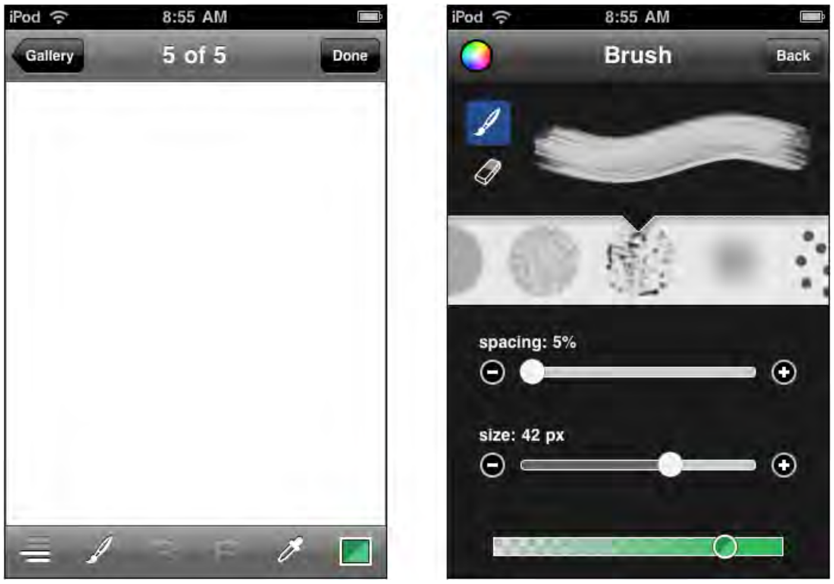


Figure 1–3. *Brushes assigns color picker and brush palettes to separate screen views on the iPhone. This requires users to navigate between various screens, but on such a small device, it's a necessary design to keep the interface uncluttered and easy to use.*

Many people first heard of Brushes when the June 1, 2009, issue of *The New Yorker* featured a beautiful cover by artist Jorge Colombo, created entirely in Brushes on the iPhone. Then in January 2010, Brushes returned to the media spotlight as Steve Sprang was invited to unveil his forthcoming iPad version of Brushes during Apple's iPad keynote announcement. Beyond showcasing the extended drawing space on the iPad's large screen, he also demonstrated how those separate color picker and brush palettes could be made easily accessible from within the main canvas screen by using the new popover controller (see Figure 1–4).

Popovers empower the iPad version of Brushes to behave more like a traditional desktop application, alleviating the need to move back and forth between various screen views, such as on a small iPhone or iPod touch. This is just one of many new user interface features that allowed Sprang to provide a more powerful and simplified Brushes experience on the iPad.

Knowing that Steve Sprang was one of the first developers outside Apple to work with the iPad SDK frameworks and user interface additions, I was curious to learn more about his experience programming for the iPad. I was fortunate enough to steal him away from his busy schedule for a brief interview.

www.it-ebooks.info

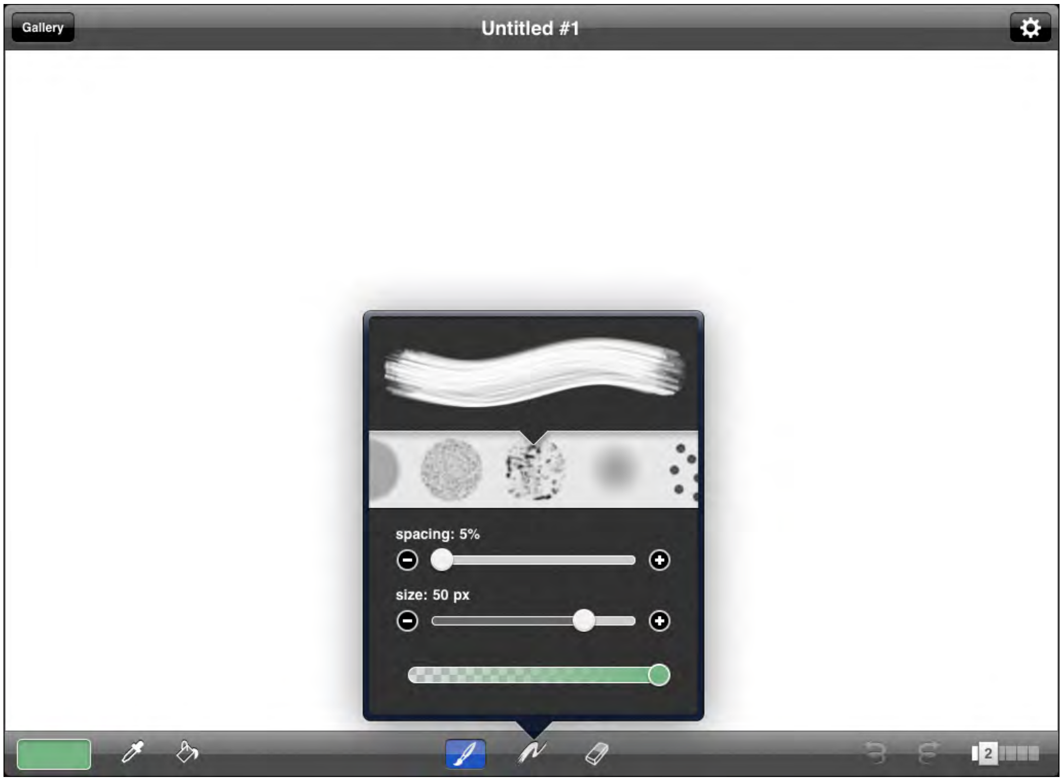


Figure 1–4. *Popovers enabled developer Steve Sprang to integrate the color picker and brush views within the main canvas screen in the iPad version of Brushes.*

Beyond the larger screen size for the Brushes "canvas," what have the new iPad features in the SDK allowed you to do to simplify and enhance the user experience that wasn't possible in the iPhone version?

The larger screen makes it easier to deal with multiple orientations. For example, on the iPhone, the color panel in Brushes would require an alternate layout to work well in landscape mode, but on the iPad, the popover works equally well in any orientation. Popovers are also a big win in terms of workflow, allowing quick access to many controls while still keeping them tucked away when not in use.

In porting Brushes to the iPad, can you share your experience working with the new SDK?

Most of my effort was spent redesigning the interface to work well on the iPad. The gallery view is completely new, as well as the in-app playback feature. Some interface elements from the iPhone were easily reused. For example, the original gallery view from the iPhone now appears as a thumbnail popover in the iPad gallery (for quicker navigation). The painting engine is basically the same, but some optimizations were necessary to deal with the increased number of bits being pushed around on the screen.

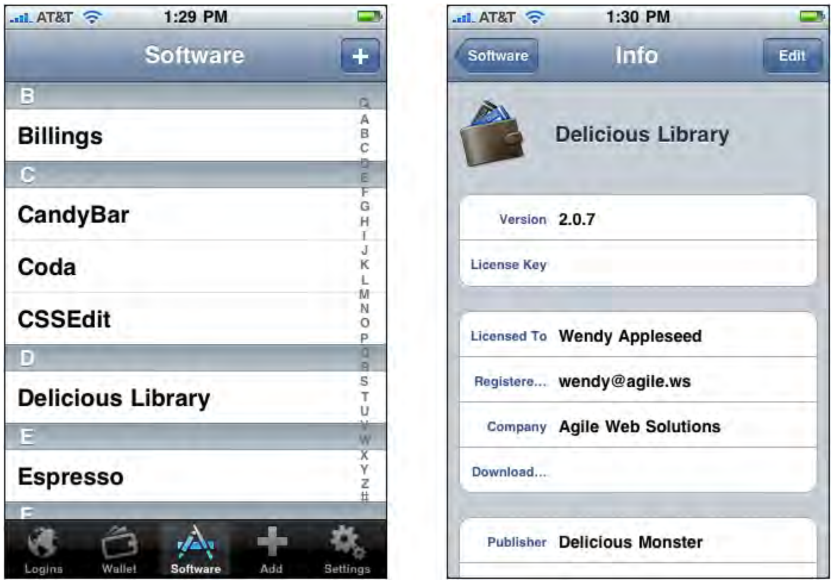
www.it-ebooks.info

Any useful tips or words of wisdom for developers looking to port their own iPhone apps to the iPad platform?

I think it's easy to underestimate the amount of work involved in redesigning an iPhone app to work well on the iPad. In many ways, it's an entirely new design problem. On the iPhone, you could get away with pushing a view controller onto a navigation controller, but on the iPad, you'll likely need a custom transition if you want things to feel right. It's going to take more effort than just scaling up your old interface.

1Password Pro

The best-selling iPhone app, 1Password Pro (`http://agile.ws/`), securely stores your important information, software licenses, and passwords, and can automatically log you in to web sites with a single tap. Limited by the small screen of the iPhone, Agile Web Solutions employed a navigation controller and tab bar controller in the user interface design, so that users could easily organize and access their stored entries. The goal was to avoid cluttering the small screen with too many elements, but like most iPhone apps, this required navigating back and forth between different screen views (see Figure 1–5).



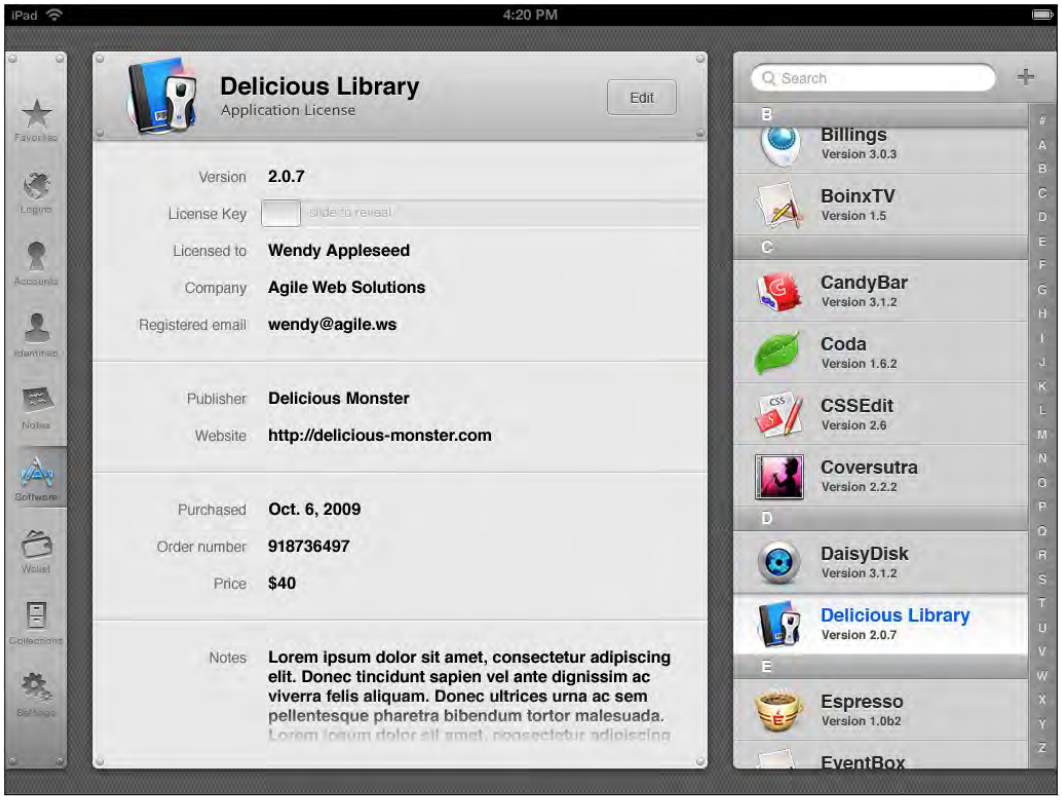
Version	2.0.7
License Key	
Licensed To	Wendy Appleseed
Registere...	wendy@agile.ws
Company	Agile Web Solutions
Download...	
Publisher	Delicious Monster

Figure 1–5. *On the small iPhone, Agile Web Solutions used tab bar and navigation controllers to maintain a streamlined 1Password Pro interface across multiple screens.*

Although the iPhone app interface for 1Password Pro was very intuitive and easy to use, Agile Web Solutions developers were eager to take advantage of the iPad's expanded screen size, enabling them to consolidate those primary views into one, multipane interface for the iPad version of 1Password Pro (see Figure 1–6). David Chartier explains the design:

www.it-ebooks.info

In the big picture, the larger screen space allowed us to design a more cohesive 1Password experience for our users. But really, it's about the little details. We can display a few more of the essential 1Password sections (passwords, secure notes, software licenses, etc.) in a wider toolbar, and present controls in a popover instead of making users tap between multiple screens to create new items. Instead of tapping into a new screen to view an item's details, we can display them in-line in the item list, which can feature web site and application icons to help users pick out the one they need more quickly.



Delicious Library
Application License

Version	2.0.7
License Key	
Licensed to	Wendy Appleseed
Company	Agile Web Solutions
Registered email	wendy@agile.ws
Publisher	Delicious Monster
Website	http://delicious-monster.com
Purchased	Oct. 6, 2009
Order number	918736497
Price	$40
Notes	Lorem ipsum dolor sit amet, consectetur adipiscing elit. Donec tincidunt sapien vel ante dignissim ac viverra felis aliquam. Donec ultrices urna ac sem pellentesque pharetra bibendum tortor malesuada. Lorem ipsum dolor sit amet, consectetur adipiscing

Billings
Version 3.0.3
BoinxTV
Version 1.5
CandyBar
Version 3.1.2
Coda
Version 1.6.2
CSSEdit
Version 2.6
Coversutra
Version 2.2.2
DaisyDisk
Version 3.1.2
Delicious Library
Version 2.0.7
Espresso
Version 1.0b2
EventBox

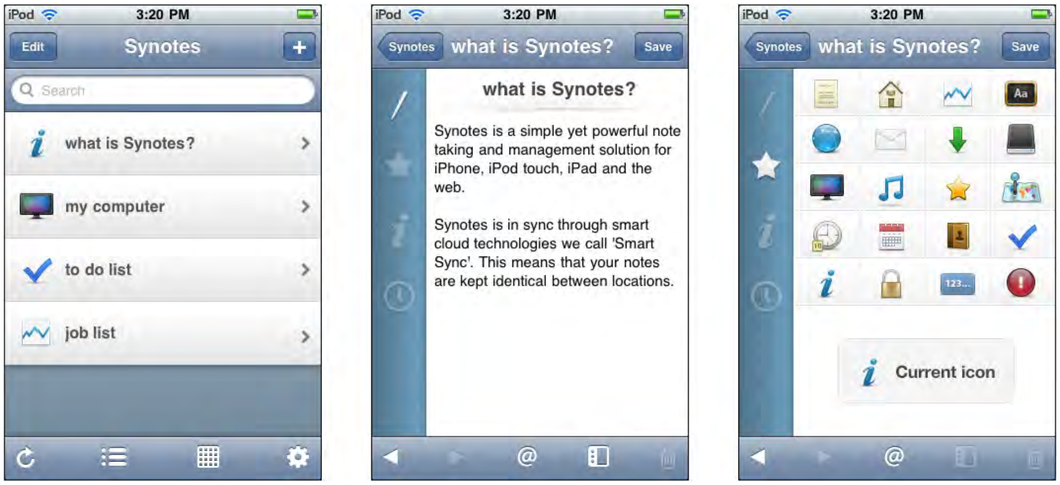
Figure 1–6. *Empowered by the iPad's large screen, Agile Web Solutions consolidated several screen views from the 1Password Pro iPhone app into one, multipane interface for the iPad version.*

Synotes

When Syncode set out to create Synotes (`http://www.syncode.com.au/`), a note-taking iPhone app that effortlessly "cloud" synchronizes saved notes across multiple devices and the Web, the goal was to provide a stylish and user-friendly interface that was easy to use. As with Brushes and 1Password Pro, this required navigating between several screen views to maintain an uncluttered interface on the small iPhone and iPod touch. A

www.it-ebooks.info

navigation controller manages movement from the main notes list to a selected note, and within a detailed note view, a custom vertical toolbar provides access to additional options, such as assigning an icon to the currently selected note (see Figure 1–7).



what is Synotes?

Synotes is a simple yet powerful note taking and management solution for iPhone, iPod touch, iPad and the web.

Synotes is in sync through smart cloud technologies we call 'Smart Sync'. This means that your notes are kept identical between locations.

Figure 1–7. *In the iPhone version of Synotes, several navigation screens are required to preserve an effective and user-friendly experience.*

In redesigning Synotes for the iPad, the larger screen gave the developers the freedom to consolidate those multiple screens into a more unified interface (see Figure 1–8).

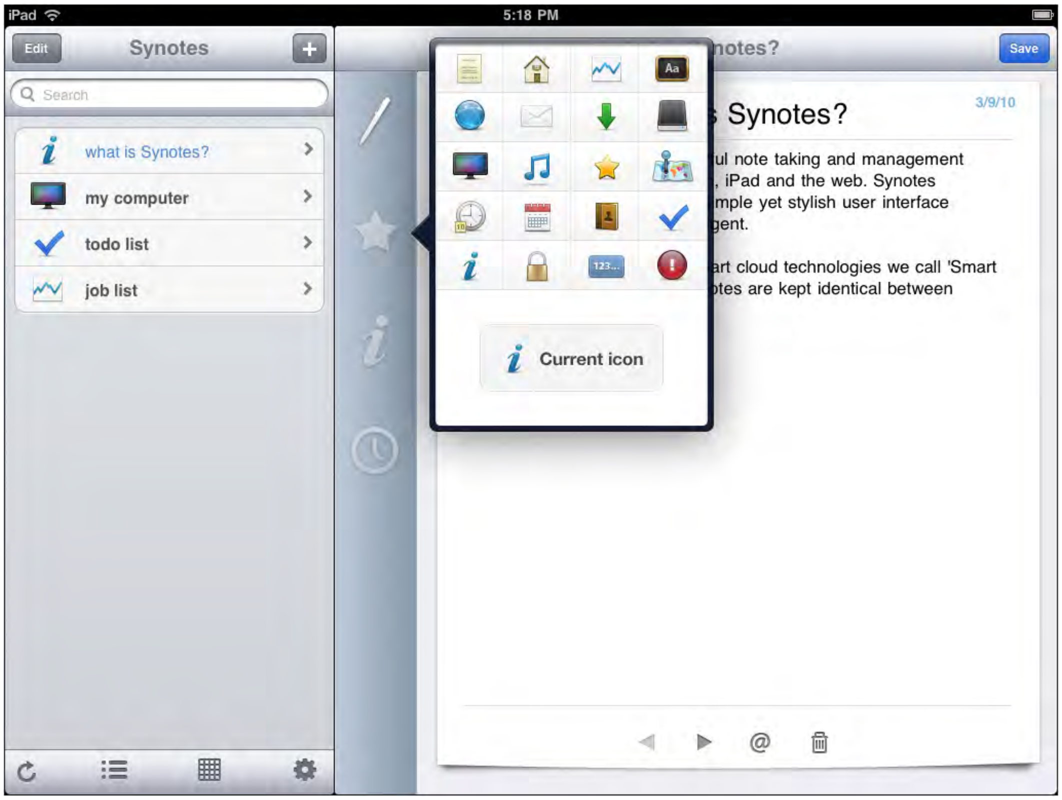
Matthew Lesh, cofounder of Syncode, describes their approach:

> *Syncode has found the challenge of porting Synotes to the iPad both exciting and rewarding. The apparent difference between platforms is space, so the question becomes how to most logically utilize the extra screen real estate. Synotes for iPad utilizes three key iPad-specific SDK features. Firstly, `UISplitViewController`, a key element to Synotes that enables us to follow Apple's "any orientation" style guides and in the process, perfectly fitting the list and content nature of Synotes. Secondly, the `UIPopoverController` has been vital to display information that doesn't require the entire screen, such as the icon selection screen. Thirdly, `UIModalViewControllers` have enabled the display of further views that would have traditionally been the third step in a navigation controller, such as settings or history items.*

As evident in the iPad version (Figure 1–8), assigning an icon to the currently selected note is now accomplished with a popover, whereas that feature once required navigating to a separate screen view on the iPhone. Although the landscape orientation is shown here with the notes list displayed in a split-view column, rotating the iPad to portrait mode automatically puts that notes list in a popover view. That way, the narrower portrait view allows more room for the selected note, while the main notes list

www.it-ebooks.info

always remains accessible from the top navigation bar. The beauty here is that the split view controller handles all of this for you, the developer!



s Synotes?

ul note taking and management
, iPad and the web. Synotes
mple yet stylish user interface
gent.

art cloud technologies we call 'Smart
otes are kept identical between

Figure 1–8. *The iPad's new user interface elements and larger screen enabled Syncode to redesign Synotes, so that one optimized screen could accomplish what once required multiple screens on the iPhone.*

ScribattlePad

The tablet's expanded screen size doesn't just benefit productivity apps. It's also a boon for game developers. Coauthor Jack Nutting couldn't wait to start building iPad-optimized versions of his Rebisoft games (`http://www.rebisoft.com/`).

Remember the stick-figure war games you used to play as a kid with a pencil and some graph paper? Jack has meticulously emulated the authentic look and feel of those paper-based drawings in his fast-paced iPhone game, Scribattle. In preparing a new iPad-optimized version, affectionately named ScribattlePad, he discovered the freedom to include features that had previously proven difficult on the iPhone's smaller screen (see Figure 1–9). He describes the overhaul as follows:

> *The larger screen provides for some interesting new interactions. The main innovation here will be the existence of a new two-player option,*

www.it-ebooks.info

with each player operating one end of the device, in either a co-op or competitive mode. It will also allow for more strategic play. Each player will have opportunities to move and regroup their guys. If you put them in groups, they'll fire their weapons and activate shields simultaneously, with the challenge of presenting an easier target for enemies to fire upon. This basically brings the game quite a bit closer to my original vision, based on my recollections of childhood play, where we did similar things on paper, but with a whole lot more thrown in as well.

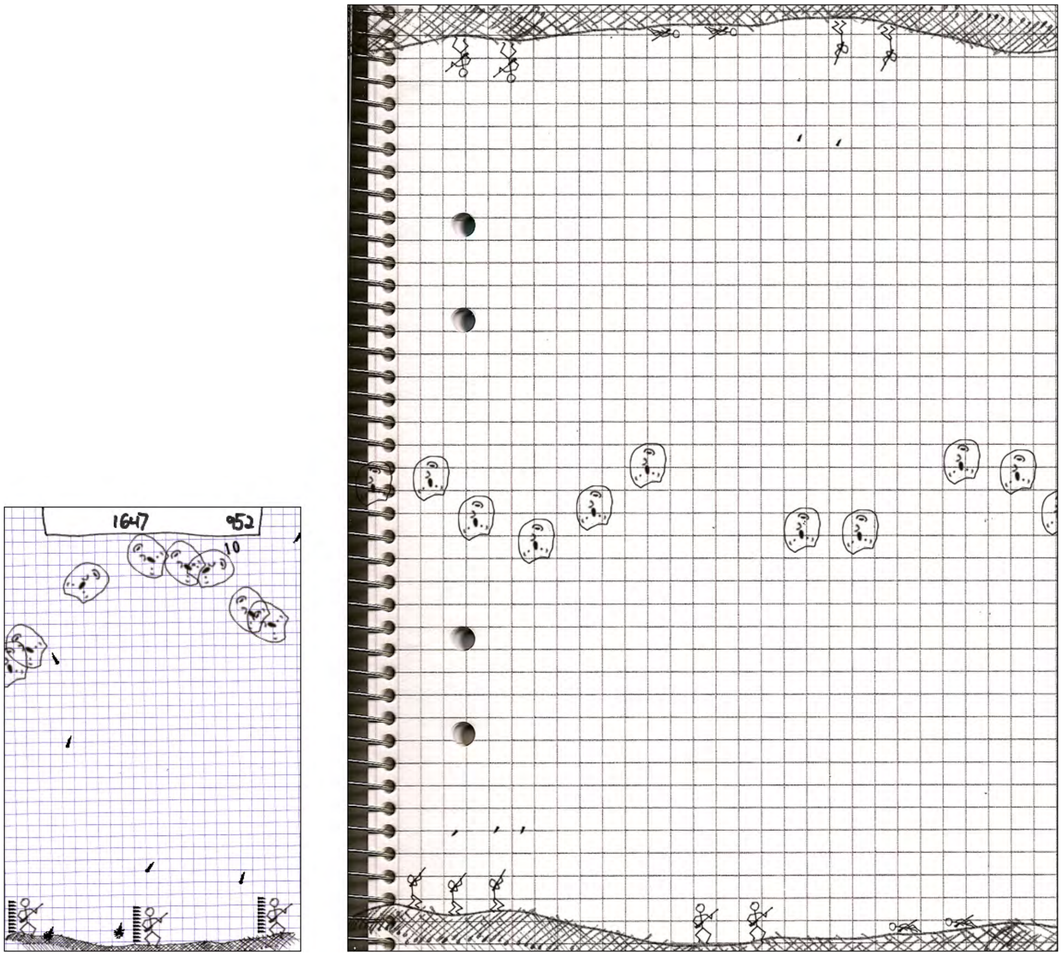


Figure 1–9. *Comparing the iPhone's Scribattle (left) and the iPad's enhanced ScribattlePad (right), it's obvious the iPad's extra screen real estate can make a huge difference in the amount of game play and interaction featured on the screen.*

www.it-ebooks.info

Zen Bound 2

Secret Exit's Zen Bound (`http://zenbound.com/`) is a meditative puzzle game that involves wrapping wooden sculptures with rope. Unlike most games, a high score is not the primary goal here. Instead, the intention is to enjoy the process at your own relaxed pace. Many people consider this game to be one of the most beautifully rendered apps currently available on the iPhone.

In planning the sequel, Secret Exit chose to develop Zen Bound 2 exclusively for the iPad, taking advantage of the tablet's powerful graphics engine. The larger screen and superior graphics capabilities enabled the developers to surpass the stunning imagery of the original iPhone game. The result is nothing short of breathtaking (see Figure 1–10). This is definitely an important factor to keep in mind when choosing a platform for your next game!

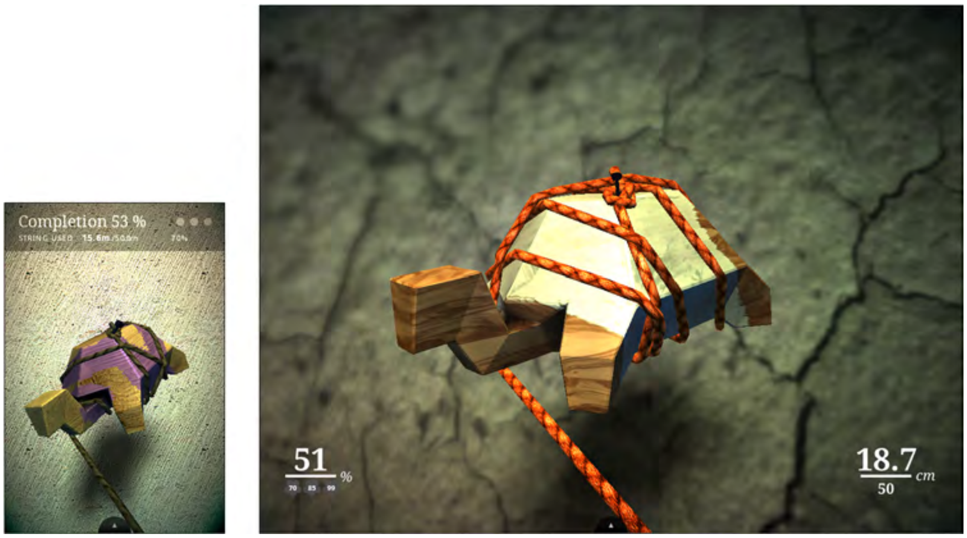


Figure 1–10. *Taking advantage of the iPad's powerful graphics engine, the stunning Zen Bound 2 (right) far surpasses the imagery in the iPhone's Zen Bound (left).*

Opportunity Awaits

As noted by Syncode's Matthew Lesh, "Apple has provided developers with some powerful and unique tools to create stylish applications for the iPad. The challenge now is to create them."

After exploring the iPad's target market and previewing some of the beautiful apps that developers are building specifically for Apple's new tablet, you're probably pretty fired up to start writing code. Feeling inspired? Good, because you won't want to miss out on another "gold rush" opportunity as new iPad owners flock to the App Store looking to download iPad apps for their devices. It's time to dive into the exciting world of iPad programming!

www.it-ebooks.info

For those of you interested in a quick refresher course on developing apps with Xcode, Interface Builder, and Apple's iPhone SDK, you'll find Chapter 2 to be a welcome primer before jumping into the rest of the book. If you're an experienced iPhone app developer and have already installed iPhone SDK 3.2, feel free to flip ahead to Chapter 3 to begin your iPad development journey.

www.it-ebooks.info

www.it-ebooks.info

Chapter 2

Getting Started with iPad Development

Before you begin working with the new iPad features and frameworks, it's important to have the required tools and preliminary training in place, so that you start your iPad development journey on the right footing. If you've already installed iPhone SDK 3.2 and consider yourself an advanced iPhone developer—perhaps you even have a few apps in the App Store—you may want to skip ahead to Chapter 3. But if you feel your skill set is a little rusty, then take a few minutes to read through this quick refresher course on developing apps with Xcode, Interface Builder, and Cocoa Touch.

Acquiring the Tools of the Trade

As an iPhone developer, you're undoubtedly a frequent visitor to Apple's iPhone Dev Center at `http://developer.apple.com/iphone/` and have already downloaded previous versions of the iPhone SDK to build your iPhone apps. Although access to the iPhone SDK, code samples, tutorials, and documentation are free to registered developers, if you eventually plan to submit your apps to the App Store, you'll need to enroll in Apple's iPhone Developer Program.

Enrolling in the iPhone Developer Program

Don't let the name fool you. The iPhone Developer Program encompasses everything related to iOS, so even if you're building only iPad apps, this is the program you want. Enrollment costs an annual $99 fee for individual developers or a small development team. Many newcomers balk at that admission price, but if you're serious about developing iPad and iPhone apps for the lucrative App Store, this will prove to be the easiest $99 you've ever spent in your programming career.

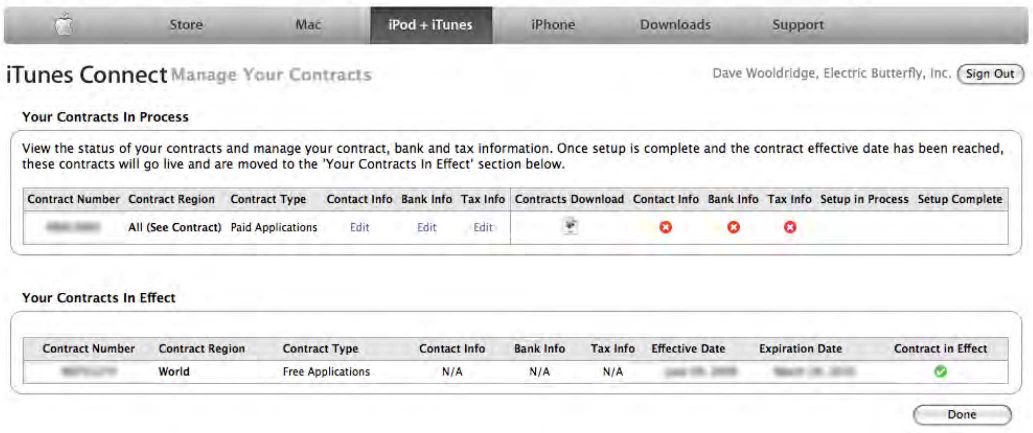
Beyond submitting apps to the App Store, membership also grants you the ability to create provisioning profiles for testing apps on an actual iPhone, iPod touch, and iPad device. The program also provides additional support resources from Apple and enables

www.it-ebooks.info

you to set up ad hoc distribution for beta testing apps. For details, visit `http://developer.apple.com/programs/iphone`.

Do not wait until your iPad app is ready to be submitted to the App Store, since it can take weeks to receive acceptance into the iPhone Developer Program, which would delay your progress unnecessarily. After being accepted, pay the $99 fee to complete your registration. After your payment has been processed, when you log in to the iPhone Dev Center, you'll see an iPhone Developer Program column on the right side of the browser screen. Click the iTunes Connect button there.

On the main page of iTunes Connect, be sure to visit the Contracts, Tax, & Banking Information section to view the contracts you currently have in effect. By default, you should have the Free Applications contract already activated, which allows you to submit free apps to the App Store. But if you want to submit paid apps to the App Store, you'll need to request a Paid Applications contract. Apple needs your bank and tax information so that it can pay you when you've accrued revenue from app sales. Since Apple transfers money via secure electronic deposits, make sure your bank supports electronic transactions with third-party vendors. You'll need to provide your bank's ABA routing number, name, address, and your account number (along with your bank's SWIFT code for receiving payments from international App Stores). Until you complete the required steps (see Figure 2–1), Apple will hold any money it owes you in trust. And since this can also be a fairly lengthy process, I highly recommend completing the Paid Applications contract long before submitting your iPad app to the App Store.



iTunes Connect Manage Your Contracts

Dave Wooldridge, Electric Butterfly, Inc. Sign Out

Your Contracts In Process

View the status of your contracts and manage your contract, bank and tax information. Once setup is complete and the contract effective date has been reached, these contracts will go live and are moved to the 'Your Contracts In Effect' section below.

Contract Number	Contract Region	Contract Type	Contact Info	Bank Info	Tax Info	Contracts Download	Contact Info	Bank Info	Tax Info	Setup in Process	Setup Complete
[redacted]	All (See Contract)	Paid Applications	Edit	Edit	Edit		✗	✗	✗		

Your Contracts In Effect

Contract Number	Contract Region	Contract Type	Contact Info	Bank Info	Tax Info	Effective Date	Expiration Date	Contract in Effect
[redacted]	World	Free Applications	N/A	N/A	N/A	[redacted]	[redacted]	✓

Done

Figure 2–1. *In order to get paid for your App Store sales, make sure you complete Apple's required Paid Applications contract in the iTunes Connect online portal.*

Installing iPhone SDK 3.2

If you haven't already installed the iPhone SDK 3.2, download it now from Apple's iPhone Dev Center (`http://developer.apple.com/iphone/`).

The iPhone SDK 3.2 requires an Intel-based Mac running Mac OS X Snow Leopard 10.6.2 or later. The SDK includes Apple's complete developer tool set, such as Xcode,

www.it-ebooks.info

Interface Builder, and the iPhone Simulator. The installer provides both the iPhone and iPad frameworks, so you can continue to develop iPhone apps while you work on your new iPad app within the same version of Xcode. You can even test your iPad apps in the iPhone Simulator, which also emulates the iPad environment.

NOTE: During the installation process, be sure to choose the Custom Install option, which will allow you to choose the specific iPhone SDK versions you want to install. Remember that the iPhone SDK 3.2 supports only iPad development. If you also need to work on iPhone apps, also select iPhone SDK 3.1.3 in the Custom Install list. If your iPhone apps need to support older versions of iOS, such as 3.1 and 3.0, select those as well.

Working with Beta Versions of the SDK

With Apple frequently releasing beta versions of forthcoming SDKs, you'll be eager to test and integrate those shiny new features into your apps in anticipation of future iOS releases. Obviously, when compiling your apps for the App Store, you'll need to keep the current, official SDK as well. And beyond that, the beta developer tools may not be stable enough yet for commercial use. In that case, you don't want the beta SDK installer to replace your existing developer tools.

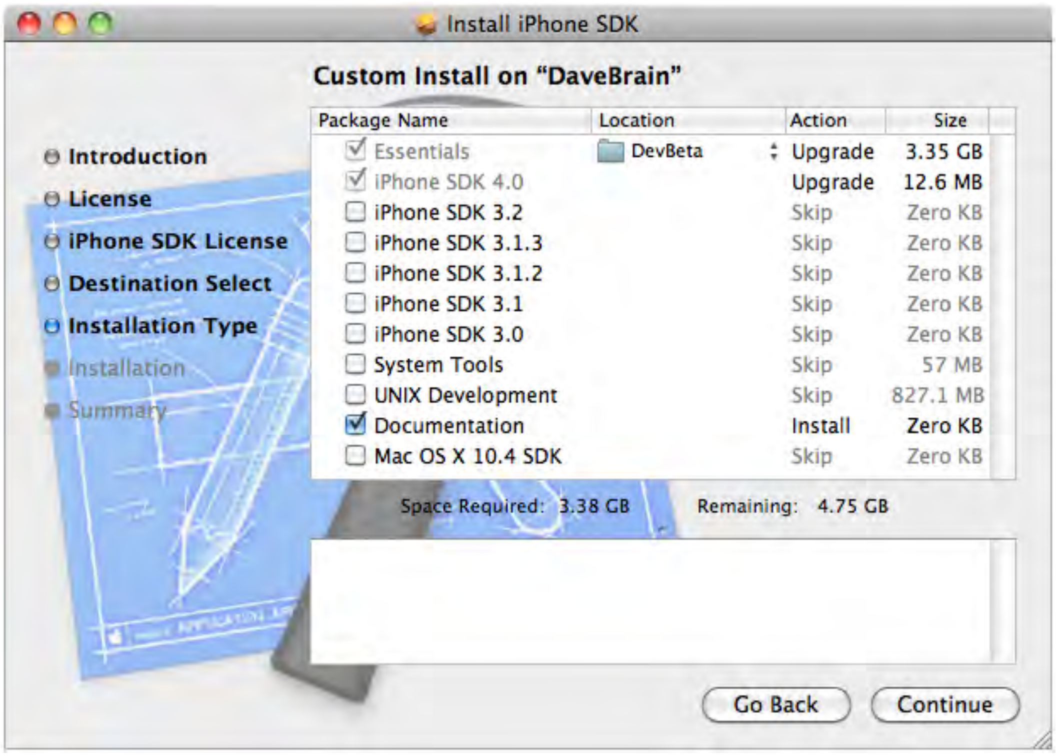
Luckily, there's an easy way around this dilemma—as long as you have plenty of hard drive space to spare. The optimal solution is to maintain two separate sets of Apple's developer tools on your Mac. The primary set is the latest, official SDK and Xcode tools. The second set consists of the beta SDK and Xcode tools that you want to begin experimenting with.

After installing the iPhone SDK 3.2, your primary drive's root directory now includes a new Developer folder. If you attempt to install the latest beta SDK with the default installation settings, the existing SDK 3.2 applications and files in the Developer folder will be overwritten with the new beta tools. To prevent that from happening, you need to direct the installer to place the new beta SDK tools in a different location by following these simple steps:

1. Download the beta SDK from Apple's iPhone Dev Center (an iPhone Developer Program membership is required to download betas).
2. Open/mount the downloaded disk image (.dmg), and then double-click its installer package to launch the installer program.
3. To install the beta developer tools in a directory other than the default Developer folder, choose the Custom Install option. At the top of the Custom Install list, click the Developer folder icon in the Essentials Location column. From the pop-up menu that appears, select Other… and choose a different location.

www.it-ebooks.info

Your new beta tools folder must be located at the root directory of your primary drive, just like the existing Developer folder. For example, I created a new folder named DevBeta that resides at the same directory level as the Developer folder (see Figure 2–2).



Custom Install on "DaveBrain"

Package Name	Location	Action	Size
✓ Essentials	DevBeta	Upgrade	3.35 GB
✓ iPhone SDK 4.0		Upgrade	12.6 MB
☐ iPhone SDK 3.2		Skip	Zero KB
☐ iPhone SDK 3.1.3		Skip	Zero KB
☐ iPhone SDK 3.1.2		Skip	Zero KB
☐ iPhone SDK 3.1		Skip	Zero KB
☐ iPhone SDK 3.0		Skip	Zero KB
☐ System Tools		Skip	57 MB
☐ UNIX Development		Skip	827.1 MB
✓ Documentation		Install	Zero KB
☐ Mac OS X 10.4 SDK		Skip	Zero KB

Space Required: 3.38 GB Remaining: 4.75 GB

Figure 2–2. *To preserve the previously installed developer tools, use the Custom Install option to install the beta SDK in a different location.*

If the apps you plan to build with the new beta SDK require backward-compatibility with older SDK versions, you can elect to install those as well within that Custom Install list. If you don't need any of those older SDKs, then deselecting them will help help conserve valuable hard drive space.

Note: Only one version of the System Tools and UNIX Development packages can be installed on your Mac. Even if you choose a new location for the beta installation, leaving System Tools and UNIX Development selected will replace your existing System Tools and UNIX Development packages with the latest beta versions, which is probably not what you want. To preserve your current System Tools and UNIX Development sets, make sure those items are left unchecked during the custom installation (see Figure 2–2).

If you later need to install multiple beta versions on your hard drive, simply follow the same custom installation process, giving a unique name to each new developer tools

www.it-ebooks.info

folder you create at the root directory. When installing multiple beta releases, it's helpful to include the version number in the directory name for easy reference. For example, instead of the generic DevBeta, you could adopt a naming convention of Developer_4_b1, Developer_4_b2, and so on. Just remember that each installation of developer tools clocks in at around 2GB to 5GB (depending on which components are installed), so multiple sets can quickly consume a lot of hard drive space.

New to Objective-C and Cocoa Touch?

Since this book was designed specifically for iPhone developers, it is assumed that you are already familiar with the Objective-C programming language and the iS frameworks that make up Cocoa Touch. If you're new to iPhone app development, your first step is to acquire that basic foundation before attempting iPad development, which builds on top of the iPhone development core skill set.

Obviously, there's more to learning Objective-C and Cocoa Touch than can be squeezed into a single chapter. Thankfully, quite a few excellent online resources and books will arm you with the necessary knowledge. Apple's iPhone Dev Center offers various guides, including the following:

- *The iPhone OS Reference Library*, which provides comprehensive documentation on Objective-C and Cocoa Touch is available at: `http://developer.apple.com/iphone/library/navigation/`
- *The Objective-C Programming Language* reference guide can be downloaded as a PDF from: `http://developer.apple.com/iphone/library/documentation/Cocoa/Conceptual/ObjectiveC/ObjC.pdf`.

If you're finding it difficult to wade through Apple's dense sea of documentation, you may find it easier to learn Objective-C and the iPhone SDK from the proven, step-by-step approaches found in these best-selling Apress books:

- *Learn Objective-C on the Mac* by Mark Dalrymple and Scott Knaster (`http://www.apress.com/book/view/9781430218159`)
- *Beginning iPhone 3 Development: Exploring the iPhone SDK* by Jeff LaMarche and Dave Mark (`http://www.apress.com/book/view/9781430224594`)
- *More iPhone 3 Development: Tackling the iPhone SDK 3* by Jeff LaMarche and Dave Mark (`http://www.apress.com/book/view/9781430225058`)

These references will serve you well as you apply your iPhone knowledge to developing apps for the iPad. That's the nice thing about the iPhone SDK. Sure, the iPad has additional, exclusive APIs (covered extensively in this book), but there are also hundreds of frameworks that work the same on both the iPhone and iPad platforms.

www.it-ebooks.info

Embracing the Model-View-Controller Concept

After programming in Xcode and arranging user interface (UI) elements in Interface Builder, it becomes apparent that Cocoa Touch was carefully structured to utilize the Model-View-Controller (MVC) design pattern. This approach neatly separates your Xcode project into three distinct pieces of functionality:

- *Model*: This is your application's data, such as the data model object classes in your project. The model also includes any database architecture employed, such as Core Data or working directly with SQLite files.
- *View*: As the name implies, this is your app's visual interface that users see. This encompasses the various UI components constructed in Interface Builder.
- *Controller*: This is the logic that ties the model and view elements together, processing user inputs and UI interactions. Subclasses of `UIKit` components such as the `UINavigationController` and `UITabBarController` first come to mind, but this concept also extends to the application delegate and custom subclasses of `NSObject`.

Although there will be plenty of interaction between the three MVC elements in your Xcode project (see Figure 2–3), the code and objects you create should be easily defined as belonging to only one of them. Sure, it's easy enough to generate your UI purely within code, or store all your data model methods within the controller classes, but if your source code isn't structured properly, that could drastically blur the line between the model, view, and controller.

You may be thinking, "If the app's performance is fast and works as intended, then why would it matter how the project's infrastructure is crafted?" Besides the fact that it's poor programming form, here's the short answer: *reusability*!

Before the advent of the iPad, your app's structure may not have mattered much at all, especially if you were not planning to reuse any of that code in other projects. At the time, you were developing your app for only one form factor: the iPhone's small 320-by-480 screen. But now you want to port that app to the iPad, taking advantage of the tablet's new features and expanded screen size. If your iPhone app doesn't adhere to the MVC design pattern, porting your Xcode project to the iPad suddenly becomes a daunting task, requiring you to rewrite a lot of code in order to produce an iPad-enhanced version.

www.it-ebooks.info

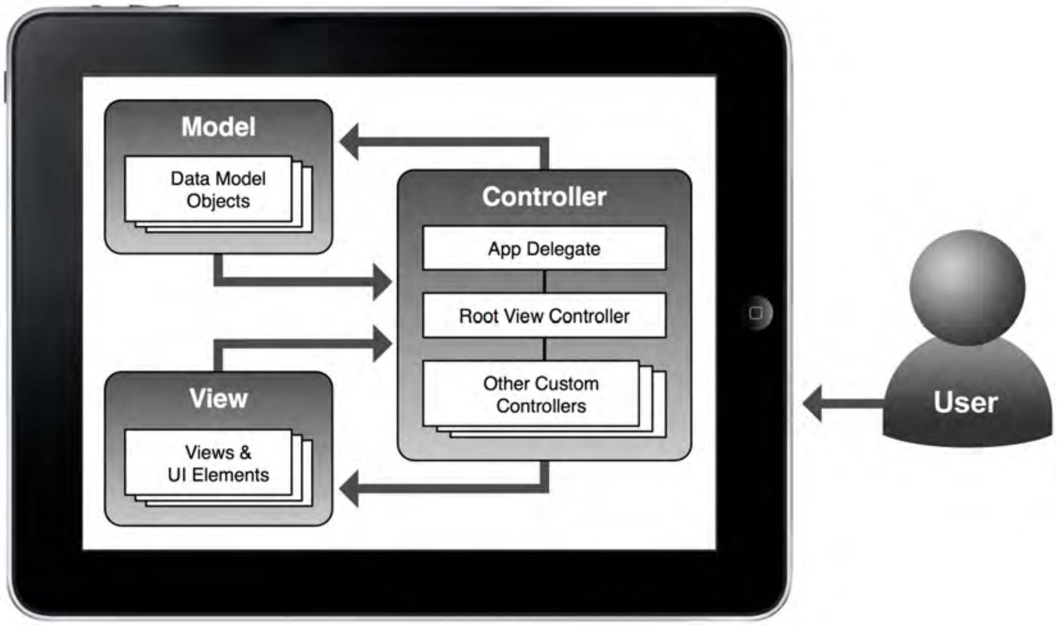


Figure 2–3. *Adhering to Xcode's Model-View-Controller design pattern will greatly simplify the process of converting an iPhone app into an enhanced iPad version.*

For example, let's say your root view controller classes contain all the code for not only retrieving database records through Core Data, but also dynamically generating a `UINavigationController` and a nested `UITableView` for displaying those records. That may work fine on the iPhone, but in moving to the iPad, you would want to use a `UISplitViewController` to display those database records. Yikes! Now you're saddled with the laborious task of manually ripping out all of that `UINavigationController` code, so that you can add in the new `UISplitViewController` functionality.

If you had kept your data classes (model) separate from your interface elements (view) and controller objects (controller), then porting the project to the iPad would have been a much simpler, streamlined process.

Reusability in Xcode

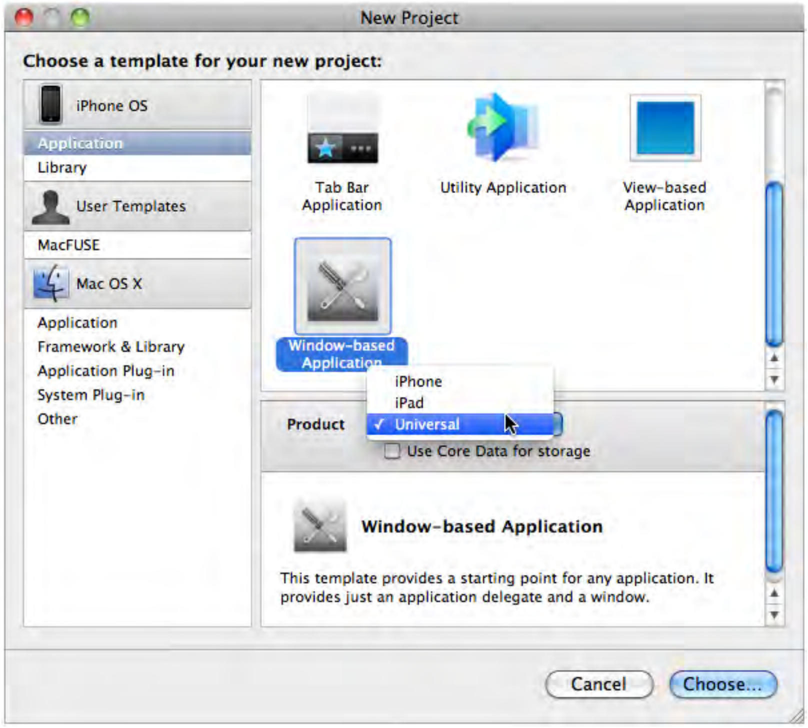
The majority of the work you'll be doing when porting an existing iPhone app to the iPad platform entails redesigning your app's UI to utilize new iPad UI components. Following the MVC design pattern from the very beginning enables you to focus most of your development time on converting the UI to the iPad, rather than losing countless hours reengineering your entire codebase. But the importance of MVC doesn't end there. Ah, yes, the plot thickens…

The iPhone SDK 3.2 introduces a new universal app format. This provides developers with a convenient path for distributing a single application package that contains both iPhone and iPad versions—hence the appropriate *universal* name. As you might expect,

www.it-ebooks.info

if the app is downloaded on the iPad, the iPad version will run; if downloaded on the iPhone, the iPhone version will run. Obviously, you can opt to compile your app as only a stand-alone iPad app or iPhone app as well. Which format should you choose? There are unique business and marketing advantages to both scenarios, which will be touched upon in Chapter 3.

For now, let's say you decide to build your application as a universal app. As a basic example, go ahead and create a new project in Xcode by choosing Window-based Application from the iPhone OS Application templates and selecting **Universal** from the related product menu (see Figure 2–4).



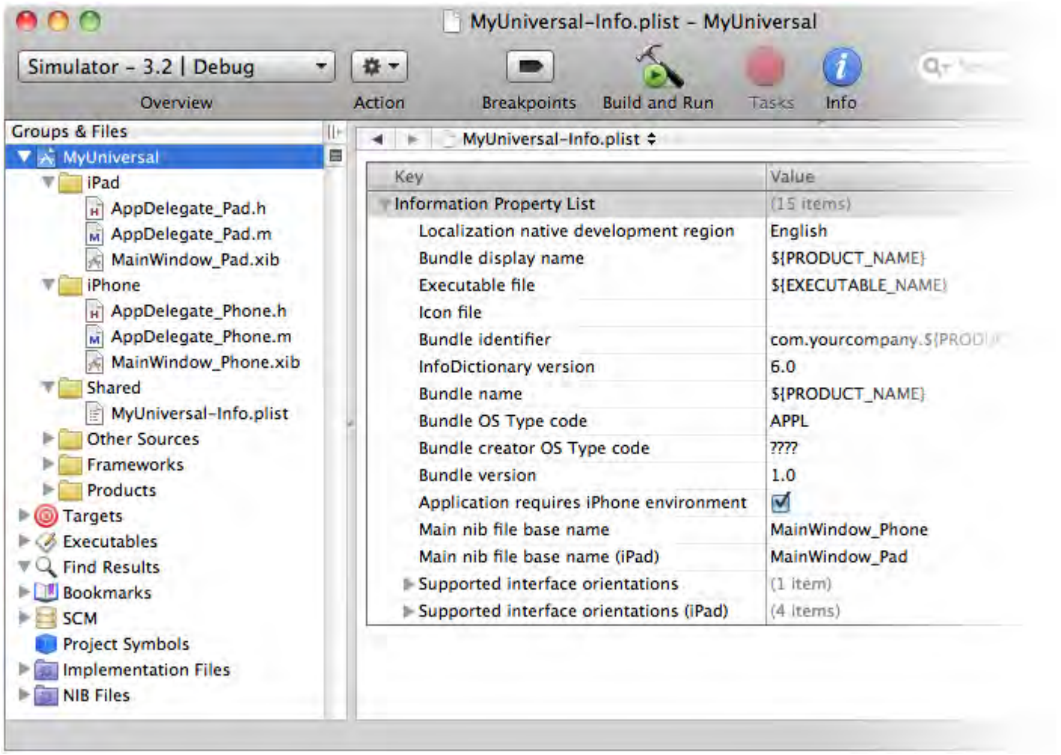
Window-based Application

This template provides a starting point for any application. It provides just an application delegate and a window.

Figure 2–4. *To create a new universal app project, choose the Window-based Application template and select Universal from the product menu.*

Once you've given your project a name, the main Xcode project window that appears is where you'll spend most of your development time. As you may already know, the Xcode integrated development environment (IDE) is the central application in Apple's developer tools arsenal. Here, you manage your project's files and resources, as well as debug and test your app via the iPhone Simulator or a connected device.

In the Universal version of the Window-based Application template, you'll notice that the default project that's generated organizes the source files into distinct folders. In the Groups & Files list, iPad-specific files are located in an iPad folder, and iPhone-specific files are located in an iPhone folder (see Figure 2–5). So far, this doesn't look any different from maintaining two different codebases within the same project, but wait! See that Shared folder? Beyond sharing a common *.plist* file, a Universal project can also share common classes, databases, resources, and select controllers and UI views!



Key	Value
Information Property List	(15 items)
Localization native development region	English
Bundle display name	${PRODUCT_NAME}
Executable file	${EXECUTABLE_NAME}
Icon file	
Bundle identifier	com.yourcompany.${PROD
InfoDictionary version	6.0
Bundle name	${PRODUCT_NAME}
Bundle OS Type code	APPL
Bundle creator OS Type code	????
Bundle version	1.0
Application requires iPhone environment	✓
Main nib file base name	MainWindow_Phone
Main nib file base name (iPad)	MainWindow_Pad
Supported interface orientations	(1 item)
Supported interface orientations (iPad)	(4 items)

Figure 2–5. *Sharing common source files and resources for both iPhone and iPad platforms within a single Universal project is yet another reason why utilizing the MVC design pattern is so important.*

By sharing common classes between both iPhone and iPad versions, you'll not only remove redundant code from your project, but moving forward, your codebase will also be much easier to maintain. This becomes extremely useful when adding a new feature that needs to be made available to both platforms. And what if Apple decides to someday extend iOS to yet another hardware device configuration, which might require adding a third platform to your Universal project? By adhering to the MVC approach, your codebase will be much easier to adapt to whatever the future may hold.

You will learn more about creating universal apps in Chapters 3 and Chapter 11.

NOTE: If you're interested in improving your knowledge of Apple's Xcode tools beyond what's offered in the embedded help, check out the Apress book *Learn Xcode Tools for Mac OS X and iPhone Development* by Ian Piper (`http://www.apress.com/book/view/9781430272212`).

www.it-ebooks.info

Designing in Interface Builder

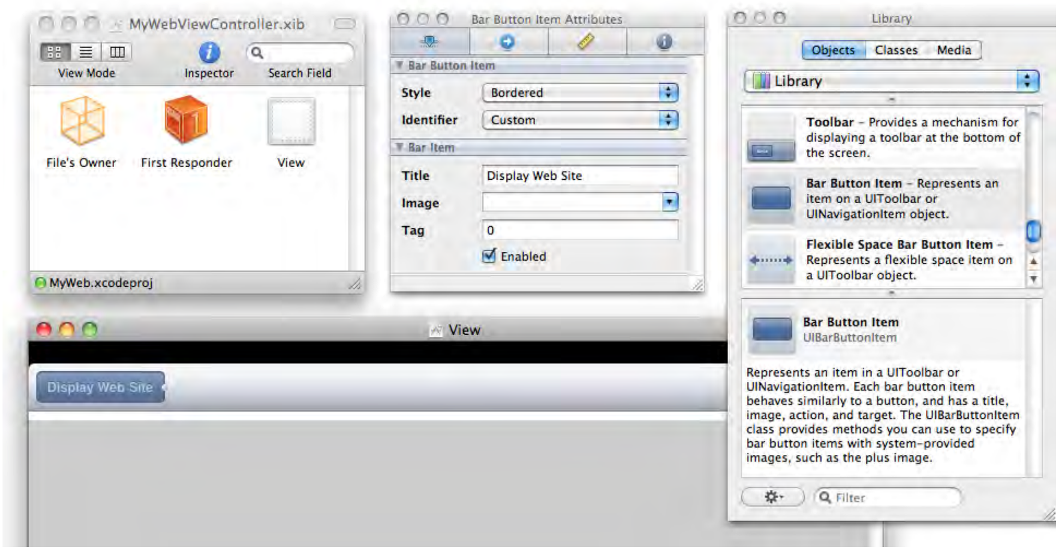
Interface Builder provides an easy way to quickly create your app's UI by customizing the various view controllers, views, and UI components. As part of this brief refresher on using Apple's developer tools, let's create a new project, so that we can explore the power of Interface Builder.

In Xcode, choose **File ➤ New Project**. From the New Project window, choose the View-based Application template from the iPhone OS list. Select **iPad** from the related product menu, and name the project `MyWeb`.

In the main Xcode project window that appears, the Groups & Files pane lists all of your project's source files and resources. The template generates some basic class files in the Classes folder and the corresponding user interface *.xib* files in the Resources folder.

For this example, we'll build a very simple iPad app with a button that loads the Apress.com web site into a `UIWebView`. The complete project can be downloaded along with the rest of the examples in this book from `http://www.apress.com/book/view/9781430230212`.

Double-click the *MyWebViewController.xib* file, and that UI view will open in Interface Builder. Drag a `UIToolbar` from the Library window to the top of the View window. Select the default `UIBarButtonItem` that is included in the toolbar. In the attribute inspector window, rename the button's title to **Display Web Site** (see Figure 2–6).



Bar Button Item
Style: Bordered
Identifier: Custom
Bar Item
Title: Display Web Site
Image:
Tag: 0
☑ Enabled

Toolbar – Provides a mechanism for displaying a toolbar at the bottom of the screen.

Bar Button Item – Represents an item on a UIToolbar or UINavigationItem object.

Flexible Space Bar Button Item – Represents a flexible space item on a UIToolbar object.

Bar Button Item
UIBarButtonItem

Represents an item in a UIToolbar or UINavigationItem. Each bar button item behaves similarly to a button, and has a title, image, action, and target. The UIBarButtonItem class provides methods you can use to specify bar button items with system-provided images, such as the plus image.

Figure 2–6. *Create a UI in Interface Builder by dragging components from the Library onto the View window. Selecting the button enables you to customize its properties in the attribute inspector.*

Why use a toolbar? Since the rest of the screen will hold the `UIWebView`, encapsulating the button in a toolbar will look much nicer than a lonely `UIButton` in the corner of the screen. So you've probably guessed what's next—it's time to drag a `UIWebView` from the

www.it-ebooks.info

Library onto the View window. Grab the selection points on the ends of the `UIWebView` and make sure it covers the remaining screen space below the toolbar.

Now that the UI has been designed, save the *MyWebViewController.xib* file, exit from Interface Builder, and return to Xcode.

Right now, the new UI has no connections to the project's source code. If you run the app in the iPhone Simulator, you'll be able to tap the button in the toolbar, but nothing will happen. So now it's time to add some interaction between the model, view, and controller pieces of our project.

Within Xcode, open the *MyWebViewController.h* header file and add the following new lines of code (highlighted in bold):

```
// MyWebViewController.h
#import <UIKit/UIKit.h>

@interface MyWebViewController : UIViewController {
    UIWebView *mywebView;
    UIBarButtonItem *urlButton;
}

@property (nonatomic, retain) IBOutlet UIWebView *mywebView;

-(IBAction)urlbuttonTapped;

@end
```

Notice that we added a `mywebView` reference to the `UIWebView`, as well as an `IBOutlet` for this object. There are also references to an `urlButton` as the `UIBarButtonItem` and `urlbuttonTapped` as an `IBAction`. If the purposes of `IBOutlet` and `IBAction` are a little fuzzy to you, don't worry—I'll explain how outlets and actions work in a moment.

After saving the *MyWebViewController.h* file, open the corresponding *MyWebViewController.m* implementation file and add the following bold code (with the exception of the `viewDidLoad` event, which was simply uncommented):

```
// MyWebViewController.m
#import "MyWebViewController.h"

@implementation MyWebViewController

@synthesize mywebView;

// Implement viewDidLoad for additional setup after loading the view.
- (void)viewDidLoad {
    [super viewDidLoad];
}

- (IBAction)urlbuttonTapped {
    // The button was tapped, so display the specified web site.
    NSURL *url = [NSURL URLWithString:@"http://www.apress.com/"];
    NSURLRequest *request = [NSURLRequest requestWithURL:url];
    [self.mywebView loadRequest:request];
}
```

www.it-ebooks.info

```
// Override to allow orientations other than the default portrait orientation.
- (BOOL)shouldAutorotateToInterfaceOrientation:(UIInterfaceOrientation)
interfaceOrientation {
    return YES;
}

- (void)didReceiveMemoryWarning {
    // Releases the view if it doesn't have a superview.
    [super didReceiveMemoryWarning];
    // Release any cached data, images, etc that aren't in use.
}

- (void)dealloc {
    [mywebView release];
    [super dealloc];
}

@end
```

We added the urlbuttonTapped method with code for loading the Apress.com URL in the UIWebView. Also, since we've instantiated the mywebView reference in memory, once our code is finished with it, we should properly release the reference in the dealloc event, freeing up that precious memory for other uses.

Even after saving the *MyWebViewController.m* file, we're not quite finished yet. True, we've added the necessary code to power our app, but our UI is still not aware of that functionality. In order to bind the UI with the relevant items in our source code, we must connect the IBAction and IBOutlet to the appropriate UI components in Interface Builder.

Wiring Actions

Double-click the *MyWebViewController.xib* file to open it again in Interface Builder. In the View window, control-click (or right-click) the UIBarButtonItem, and a dark-gray contextual menu will appear with a list of items. Click the selector's empty dot (listed under **Sent Actions**), and while holding down the mouse button, drag the cursor over to the File's Owner icon in the main window. You'll see a blue line connecting the button to your cursor's location (see Figure 2–7). After you release the mouse button, a dark-gray menu will appear above File's Owner. Select urlbuttonTapped from that hovering menu to bind the UIBarButtonItem to the urlbuttonTapped method. This "wires" the button's action, so that if a user taps that button, the urlbuttonTapped method will be called.

www.it-ebooks.info

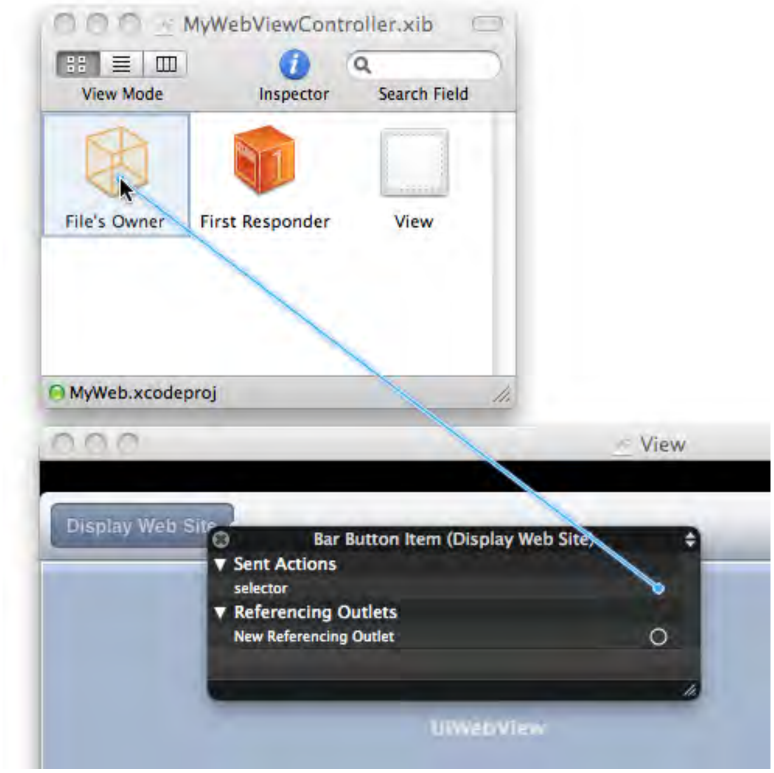


Figure 2–7. *To wire the button's action, drag a connector from the button's selector (sent action) to the File Owner's urlbuttonTapped (received action).*

Although it may seem like it took several steps within Xcode and Interface Builder to assign only a single action to a button, this approach provides a very flexible connection between your data and your UI that can be easily modified. If you decide to redesign your app in the future, replacing the existing interface with a completely new set of UI elements, you can easily control-click the `UIBarButtonItem`, remove that wired binding to the `urlbuttonTapped` method in the contextual menu, and then assign that action to a different button.

Wiring Outlets

Now that tapping the button successfully calls `urlbuttonTapped`, that method aims to load the Apress.com web site into the `UIWebView`. In order to send this URL request to the `UIWebView`, the `MyWebViewController` class needs to connect an outlet to it. Similar to how you wired the button's action, the outlet runs in the opposite direction.

In the main window, control-click (or right-click) the File's Owner icon, and a dark-gray contextual menu will appear. Click the `mywebView`'s empty dot (listed under **Outlets**) and, while holding down the mouse button, drag the cursor over to the `UIWebView` in the View

www.it-ebooks.info

window. Just as when you are wiring an action, you'll see a blue line flowing from the File's Owner icon to your cursor's location (see Figure 2–8). Release the mouse button above the `UIWebView` to complete the outlet connection. And last, but not least, save the *MyWebViewController.xib* file before returning to Xcode.

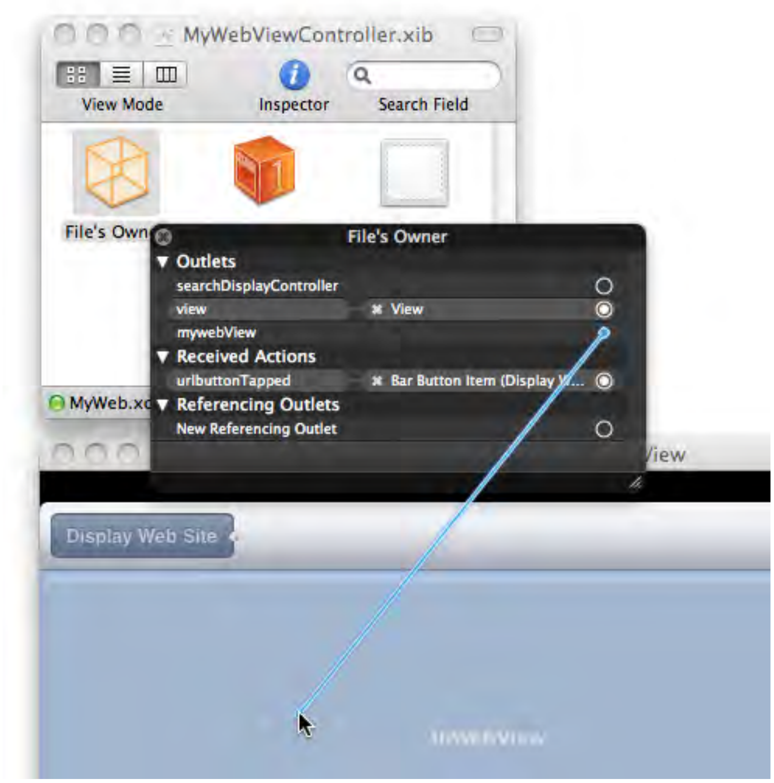


Figure 2–8. *To wire the web view's outlet, drag a connector from the File Owner's mywebView (outlet) to the UIWebView on the View window.*

With all of the functionality connected through the appropriate actions and outlets, your app is now ready to rock! To test it, ensure the **Overview** pull-down menu (in the top-right corner of the Xcode project window) is set to **Simulator - 3.2 | Debug**, and then click Build and Run to launch the iPad app in the iPhone Simulator. In the simulator's iPad window, tap the Display Web Site button, and the Apress.com home page should load into the web view (see Figure 2-9).

www.it-ebooks.info

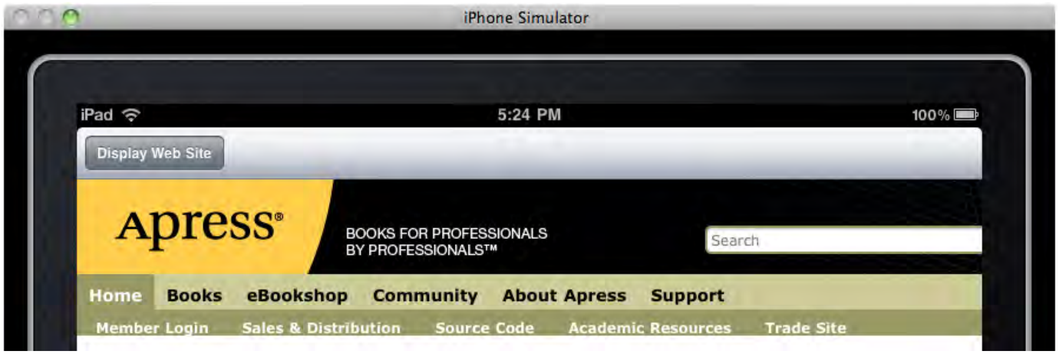


Figure 2–9. *With your project's actions and outlets wired to the UI, the MyWeb iPad app now works as intended in the iPhone Simulator.*

The Importance of Delegates

Cocoa Touch relies heavily on delegates, so it's vital that you're comfortable using them. Delegates allow one object to receive messages or modify the behavior or another object, without needing to inherit or subclass it. This is an extremely handy design pattern that helps alleviate a lot of extra coding.

A very simple example of delegation is the `UIWebView` component that we implemented in the `MyWeb` project. Besides the fact that Apple recommends not to subclass `UIWebView`, the easiest way to communicate with `UIWebView` directly is via delegation.

Let's say we want the `MyWeb` app to notify the user when the Apress.com home page finishes loading into the web view. We will designate the `MyWebViewController` class as the `UIWebViewDelegate` for its `mywebView` instance so that it can receive events from the web view and act accordingly.

Open the *MyWebViewController.h* header file and add `<UIWebViewDelegate>` to the end of the `@interface` line (see the bold code):

```
// MyWebViewController.h
#import <UIKit/UIKit.h>

@interface MyWebViewController : UIViewController <UIWebViewDelegate> {
    UIBarButtonItem *urlButton;
    UIWebView *mywebView;
}

@property (nonatomic, retain) IBOutlet UIWebView *mywebView;

-(IBAction)urlbuttonTapped;

@end
```

Save this file, and then open the *MyWebViewController.m* implementation file. Add the following (again, new code is shown in bold):

```
// MyWebViewController.m
```

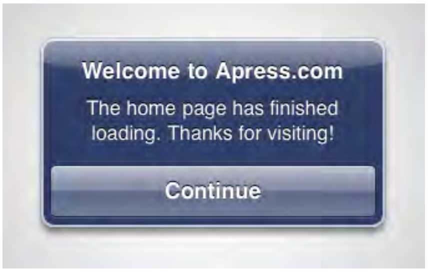
www.it-ebooks.info

```
#import "MyWebViewController.h"

@implementation MyWebViewController

@synthesize mywebView;

// Implement viewDidLoad for additional setup after loading the view.
- (void)viewDidLoad {
    self.mywebView.delegate = self;
    [super viewDidLoad];
}

- (IBAction)urlbuttonTapped {
    // The button was tapped, so display the specified web site.
    NSURL *url = [NSURL URLWithString:@"http://www.apress.com/"];
    NSURLRequest *request = [NSURLRequest requestWithURL:url];
    [self.mywebView loadRequest:request];
}

#pragma mark -
#pragma mark UIWebViewDelegate

- (void)webViewDidFinishLoad:(UIWebView *)webView {
    // Web view finished loading, so notify the user.
    UIAlertView *buttonAlert = [[UIAlertView alloc] initWithTitle:@"Welcome to
Apress.com" message:@"The home page has finished loading. Thanks for visiting!"
delegate:nil cancelButtonTitle:@"Continue" otherButtonTitles:nil];
    [buttonAlert show];
    [buttonAlert release];
}

// Override to allow orientations other than the default portrait orientation.
- (BOOL)shouldAutorotateToInterfaceOrientation:(UIInterfaceOrientation)
interfaceOrientation {
    return YES;
}

- (void)didReceiveMemoryWarning {
    // Releases the view if it doesn't have a superview.
    [super didReceiveMemoryWarning];
    // Release any cached data, images, etc that aren't in use.
}

- (void)dealloc {
    mywebView.delegate = nil;
    [mywebView release];
    [super dealloc];
}

@end
```

In the `viewDidLoad` event, the `mywebView.delegate` is assigned to the `MyWebViewController` class (`self`). And when finished, `mywebView.delegate` is set to `nil` in the `dealloc` event. The `UIWebView` includes several events, but the one we're interested in is `webViewDidFinishLoad`. Since this class is the designated delegate, it can add its own custom behavior when receiving that event. Simply add that

www.it-ebooks.info

webViewDidFinishLoad receiver to the *MyWebViewController.m* file. Any custom code added to that receiver will be called when that event fires. In this case, we're notifying the user that the web page finished loading via a UIAlertView (see Figure 2-10).


Welcome to Apress.com

The home page has finished loading. Thanks for visiting!

Figure 2–10. *Designating MyWebViewController as a UIWebViewDelegate enables the class to receive UIWebView events and add custom behavior, such as notifying the user with a UIAlertView when a web page has finished loading.*

NOTE: Delegation isn't confined to the existing Cocoa Touch framework. You can modify your own custom classes to offer a delegate protocol, so that other objects can become delegates. For details, read the section "Delegates and Data Sources" in Apple's *Cocoa Fundamentals Guide*, which can be downloaded as a PDF from http://developer.apple.com/iphone/library/documentation/Cocoa/Conceptual/CocoaFundamentals/CocoaFundamentals.pdf.

Improving App Usability with UIKit

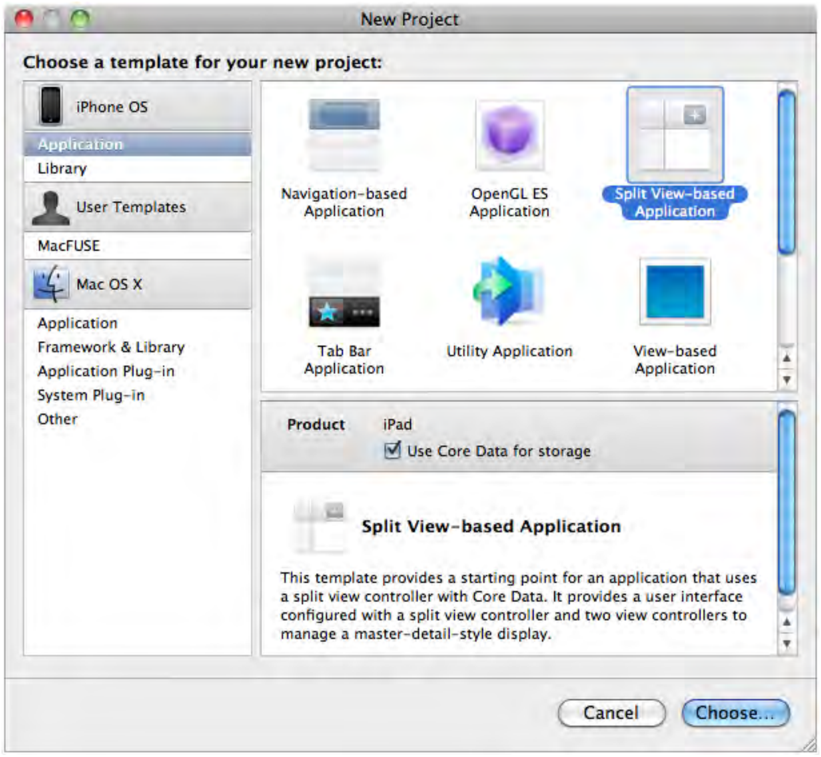
Apple has supplied a vast library of ready-made UI components and controllers in Cocoa Touch's UIKit. Since users are already familiar with how these common UI elements work, employing them in your own iPhone apps not only enhances usability, but also helps save you valuable time during the development process. For example, on the iPhone's small screen, utilizing Apple's UINavigationController or UITabBarController is an efficient method for moving between different compact views within an app.

On the iPad's much larger 768-by-1024 screen, you may be tempted to be a little more creative in your interface design choices. It's true that the iPad offers a much more robust platform for creating sophisticated, desktop-caliber apps, but it would be a huge mistake to attempt to mimic traditional desktop interface models. Just because you have a larger screen to work with doesn't mean that you can forget why the iPad is such a powerful mobile device: simplicity. Regardless of the expanded real estate, you're still dealing with a touchscreen and multifinger gestures. Let the interface breathe with plenty of space for trouble-free finger tapping. Less is more!

As with its efforts for the iPhone, Apple has provided some wonderful new iPad-centric interface elements in UIKit. Along with new UI components that have been added to

www.it-ebooks.info

Interface Builder's Library, Xcode also provides new iPad project templates, such as the Split View-based Application template (see Figure 2–11).



Product iPad
☑ Use Core Data for storage

Split View-based Application

This template provides a starting point for an application that uses a split view controller with Core Data. It provides a user interface configured with a split view controller and two view controllers to manage a master-detail-style display.

Figure 2–11. *Xcode offers new iPad project templates, such as Split View-based Application, to aid in generating an initial codebase for your new iPad app, which you can then further customize.*

The new iPad-exclusive `UISplitViewController` is employed in dozens of popular iPad apps, such as The Iconfactory's Twitterriffic and Apple's Mail. Like most of the Xcode templates, the Split View-based Application template generates a working project that you can further customize. The template offers a prebuilt split view-based app interface, configured with a `UITableView` in the master pane and a `UIView` in the detail pane. And if you select the Use Core Data for storage check box, the new project will even include sample code for populating the `UITableView` with Core Data entries. You can learn a lot about programming for the iPad by examining the code generated by these handy project templates! If this new UI controller interests you, then don't miss Chapter 8, which provides extensive coverage of using `UISplitViewController` in your own iPad apps.

With so many great interface items available in `UIKit`, why reinvent the wheel with your own UI experiments that may feel foreign to new users? Yes, you want to give your iPad app a unique interface, but if you stray too far from familiar user interactions, you run the risk of diminishing your app's immediate usability. Unless you're developing an app that requires a completely custom UI, such as a game, it's in your best interest to utilize the `UIKit` when appropriate.

www.it-ebooks.info

The beauty of the ready-made project templates and UI components is that they are fully customizable. Modify their existing attributes or subclass them, and tweak to your heart's content to give your app its own personalized polish. Your users will find your iPad app much easier to operate with an interface that is already familiar to them.

Primed for Programming

This chapter presented a quick refresher on iPhone app development. If any of this was new to you, I highly recommend reading the books and online resources listed in the "New to Objective-C and Cocoa Touch?" section before continuing. Since this book was designed for iPhone app developers, subsequent chapters assume a working knowledge of common iPhone development tasks, such as how to build and populate the rows of a `UITableView`. Having that basic foundation under your belt will help you quickly grasp and enjoy your iPad development journey!

Next up in Chapter 3, I'll introduce you to all of the new iPad frameworks and UI elements, how they relate to existing iOS features, and the proper context for using them within your own iPad apps.

www.it-ebooks.info

www.it-ebooks.info

Chapter 3

Exploring the iPhone SDK's New iPad Features

iOS 3.2 includes many new features that are currently supported on only the iPad. Because of these special iPad-exclusive features, apps compiled for 3.2 will not run on the iPhone or iPod touch. This enables developers to produce apps specifically designed for the iPad, taking advantage of the tablet's unique form factor and capabilities. This chapter provides an introduction to the new iPad offerings in iPhone SDK 3.2, as well as how to best utilize them within your apps.

Optimizing Apps for the iPad

Before diving into the new software features available to developers, let's take a look at the hardware environment that powers the operating system and installed apps. You already know about the iPad's amazing 10-hour battery life and other much-publicized selling points, so I won't bore you by listing all of the iPad's hardware specifications. Here, we'll review some of the key factors that may impact your development efforts.

Examining the Tablet

The first thing you'll notice is the iPad's brilliant display. With a 9.7-inch (diagonal) backlit in-plane switching (IPS) display, the beautiful screen boasts a 1024-by-768 pixel resolution at 132 pixels per inch. By comparison, the iPhone has a 320-by-480 pixel screen at 163 pixels per inch.

Like the display of the iPhone and iPod touch, the iPad's display is multitouch, but the big difference here is that the iPad's screen is fully capacitive, with a much greater number of touch sensors, supporting several fingers at once. With the larger screen and the almost full-size virtual keyboard (in landscape orientation), this is a significant milestone.

www.it-ebooks.info

Whereas you may typically operate an iPhone with one hand or only two thumbs, the iPad is much more immersive and often invites you to use two hands while the device rests in your lap. In a few informal tests, some independent developers have reported that the iPad's multitouch display has successfully recorded more than ten simultaneous finger taps. This is important, not only for building complex two-player games with users playing head-to-head on both ends of the tablet, but also for the ability to more efficiently track fast keyboard typing and multifinger gestures for a vast array of touch commands.

As you would expect, the iPad also includes the accelerometer, an embedded microphone, a headphone jack, and a built-in speaker. Unlike the iPhone's tiny speaker, the iPad's enhanced speaker is actually quite decent, so games, videos, and music are enjoyable even without headphones.

The iPad features the same 30-pin dock connector and wireless Bluetooth support as the iPhone, which should be welcome news for developers who utilize the iPhone SDK's existing Accessory APIs to communicate with add-on accessories.

NOTE: If you're interested in building apps that work with external hardware, check out the Apress book *Building iPhone OS Accessories: Use the iPhone Accessories API to Control and Monitor Devices* by Ken Maskrey (`http://www.apress.com/book/view/9781430229315`).

Although the 3G models support assisted GPS, keep in mind that the most popular iPad is the 16GB Wi-Fi only model. If you're developing an app that requires GPS capabilities, that particular functionality may not work quite as well on those Wi-Fi only models.

Last, but not least, the iPad is fast! Apple's custom-designed A4 processor chip provides a lightning-quick, high-performance experience that's surprisingly battery-efficient.

The iPad is so much faster than the iPhone 3GS that you might assume it also packs more RAM, but that may not be the case. Even though Apple has not published the amount of RAM in the iPad, early benchmarks from industry experts report that the iPad sports the same GPU and 256MB of RAM as the iPhone 3GS. This is important to remember when designing memory-intensive apps, such as graphics-heavy games with full-screen animation.

Managing Memory

It's true that Apple's new A4 chip is blazingly quick, and games do run much faster on the iPad, but it's not yet the Holy Grail of hardware that game developers might have been hoping for. Without any significant boost in RAM or GPU speed, creating apps for the iPad requires the same attention to memory management as previous efforts on the iPhone 3GS and iPod touch. In some regards, memory optimization is even more important on the iPad than on smaller iOS devices.

www.it-ebooks.info

With its larger screen, the iPad's GPU is forced to push around a lot more pixels, especially when running full-screen animations in games. In converting iPhone games into enhanced iPad versions, some developers have encountered noticeable animation frame rate issues, requiring them to make additional optimizations to avoid dropped frames during game play.

Obviously, the amazing graphics in Firemint's Real Racing HD and other best-selling games are proof that the iPad is a stellar game platform. The fast A4 chip definitely helps in this regard, but proactive memory management is key when programming your iPad project. Even if you plan to build productivity apps that don't include graphics-intensive animations, optimizing your app to maintain a small memory footprint is still very important.

Unlike Objective-C 2.0 for Mac OS X development, Cocoa Touch's Objective-C does not include built-in garbage collection. You need to keep track of your app's memory usage in iOS, paying close attention to your code to ensure instantiated objects are released after being used. While memory management is certainly important when optimizing apps for the iPad's limited RAM, it will become especially critical in the future when iOS 4's multitasking functionality eventually comes to the iPad, enabling multiple apps to run in the background.

Testing Your Apps on an iPad

While it's always helpful to know the iPad's hardware specifications, never assume your app will perform well on the device, just because it runs flawlessly in Xcode's iPhone Simulator. The Simulator does not support several features, such as the accelerometer, multitouch gestures, and In App Purchase. But even if you don't use any of those elements in your app, you should always, always, always test it on an actual iPad as well.

NOTE: Don't yet have an iPad? Since you're reading this book, it's a safe assumption that you're serious about iPad app development, so you really should own an iPad or have direct access to an iPad for testing. If iPads are not sold where you live, you can easily purchase one from an online retailer that can ship it to your location.

Running your app in the iPhone Simulator is great for general debugging, but as a Mac-based software emulator, it's not a true test of how your app will perform on an actual device. Beyond testing the many features that are not supported in the iPhone Simulator, running your app on an iPad will also reveal any issues that arise from the constraints of the device's fixed memory and processing power.

Yes, I know the process of creating and installing provisioning profiles and development certificates is frustrating and tedious, but it's well worth the effort in the long run, especially if your goal is to eventually release your iPad app in the iTunes App Store. You'll want to discover and squash as many bugs and performance problems as possible to help ensure that your app is well received by customers.

www.it-ebooks.info

As an iPhone developer, you should already have experience configuring a test device with your development certificate and a new provisioning profile. The process is the same for an iPad. If this is all new to you, then log in to Apple's iPhone Dev Center and read the documentation in the iPhone Provisioning Portal, at `http://developer.apple.com/iphone/manage/overview/index.action`. The iPhone Provisioning Portal even includes a handy online Provisioning Assistant that can guide you through the process.

NOTE: The Apress book *The Business of iPhone App Development: Making and Marketing Apps that Succeed* by Dave Wooldridge with Michael Schneider (`http://www.apress.com/book/view/9781430227335`) features an extensive chapter on testing, with easy step-by-step instructions on how to set up a development device with provisioning profiles. It also explains how to configure and compile your app for beta testing via ad hoc distribution.

What's New in iPhone SDK 3.2 for the iPad

Now for the fun part! This section will walk you through the new iPad features in iPhone SDK 3.2, as well as touch on how to best use them within your own apps. Subsequent chapters will drill deeper into each subject, showing you step by step how to implement each of these new features in your Xcode projects with extensive code examples.

Shape Drawing

The new `UIBezierPath` class may not be one of the most talked about or publicized new features in iPhone SDK 3.2, but if you do any kind of 2D drawing in your app, its inclusion is actually a pretty big deal. Similar to the vector-based drawing tools found in Adobe Illustrator and Photoshop, the `UIBezierPath` class enables you to draw straight lines, circles, rectangles, and curved shapes with complete control over the line's stroke color and thickness, as well as the fill color of enclosed objects.

The process of constructing a shape is relatively simple. After creating a new `UIBezierPath` object, you set the starting point via the `moveToPoint` method, and then use the `addLineToPoint` method for each additional connected line you wish to add to your shape. Calling the `closePath` method closes the shape, drawing a final line between the first point and last point. True to its name, the `UIBezierPath` class is also capable of creating Bézier curves. You can pass control points to the `addCurveToPoint` method to set the angle of the line's curve.

The aforementioned methods define the shape of your `UIBezierPath`. In order to render the object to your current graphics context, you call the `fill` and `stroke` methods. Before doing so, you'll want to assign a unique `UIColor` to `setFill` and `setStroke`, and adjust the thickness of the line stroke by designating an integer to your path's `lineWidth` property. To avoid having your fill path overlap the stroke path, you'll want to draw the

www.it-ebooks.info

fill color before drawing the stroke outline. This is as easy as calling the `fill` method before calling the `stroke` method.

I've outlined the basics of `UIBezierPath`, but you're probably itching to see how all of this works in code. In Chapter 4, you'll learn how to draw several types of objects using the `UIBezierPath` class in the process of creating a fun drawing app project called Dudel. Figure 3–1 is a preview of drawing with Dudel.

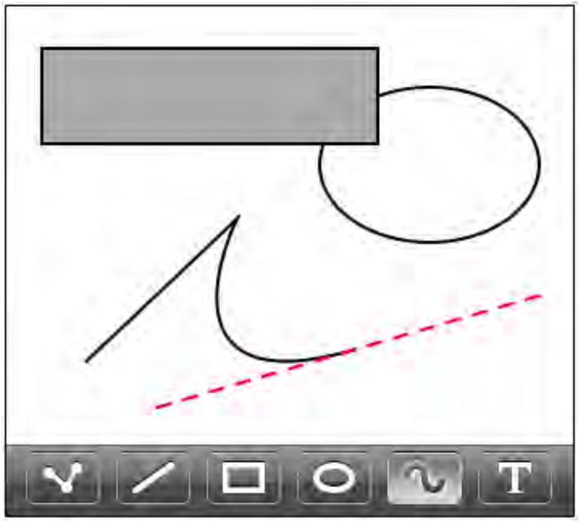


Figure 3–1. *Beyond standard objects like lines, ovals, rectangles, and custom shapes, the UIBezierPath class also enables you to draw curved lines by setting arc control points.*

Why is this graphics functionality so important? Even if you have no aspirations to develop a drawing app like Dudel, there are other practical uses for this graphics class. If you need a simple shape drawn on the screen, utilizing `UIBezierPath` requires much less memory than a PNG resource image of the same shape. With a vector-based object, only the instructions on how to draw that shape are needed. In contrast, a bitmap image file may consume several kilobytes (or more) when loaded into memory.

For example, let's say you're building a task management app. To visually indicate the priority status, your interface design places a colored dot next to each task name. A red dot represents a high priority, an orange dot indicates medium priority, and a yellow dot shows low priority. Using bitmap images, the three different dots would need to be stored in your project's Resources folder as either three separate PNG files or consolidated within one large PNG file. Changing the priority color of a task would require your app to load a new image resource into memory. But if you used `UIBezierPath` instead, you could create a very simple method that draws the colored dot. Need a different color? Just pass the new color to your custom method, which redraws the dot with the requested color.

Since conserving memory is the name of the game, this is a very economical approach to displaying simple 2D shapes on the screen. Limiting the number of bitmap image resources needed is one of many ways to help reduce the memory overhead of your app.

www.it-ebooks.info

PDF Files

If you're developing an app that creates content, then you'll find the new PDF-creation feature to be a very welcome addition to the iPad arsenal. iOS 3.2 enables developers to generate and save PDFs within their apps—all natively supported within the `UIKit` framework. Apple has done a great job of making this process very straightforward and elegant.

First, you create a PDF graphics context by calling one of two available functions. `UIGraphicsBeginPDFContextToData` stores the PDF content in an `NSMutableData` object. The more commonly used function is `UIGraphicsBeginPDFContextToFile`, which saves the PDF content as a PDF file (using your requested filename parameter) to your app's sandboxed files directory.

Unlike on-screen views, which can scroll for miles if needed, PDFs are structured as pages with a set width and height. After establishing the PDF graphics context, you must then create a new PDF page, so that you can draw content into that defined area. If you wish to create a new page using the previous default page size, then call `UIGraphicsBeginPDFPage`. But if you prefer to customize the page's size and various attributes, you should call the `UIGraphicsBeginPDFPageWithInfo` function instead.

The beauty of this new API is that all of the content you pass to the PDF graphics context is automatically translated into PDF data. After creating a new page, anything you can draw into a custom view can be drawn into your PDF, including text, bitmap images, and even vector-based shapes. For content that might not fit within the bounding box of a single PDF page, such as a large amount of text, you can call `UIGraphicsBeginPDFPage` or `UIGraphicsBeginPDFPageWithInfo` every time you need to close the current page and start a new page.

When you're finished drawing your content into the PDF graphics context, you call `UIGraphicsEndPDFContext`, which closes the current page and saves the PDF content to either an `NSMutableData` object or a PDF file, depending on whether you originally created the PDF graphics context via `UIGraphicsBeginPDFContextToData` or `UIGraphicsBeginPDFContextToFile`. Once those tasks have been completed, the `UIGraphicsEndPDFContext` function also performs a little housecleaning by automatically clearing your PDF data in memory from the graphics context stack.

You'll learn more about generating PDFs in Chapter 4. Building on the Dudel app example that showcases the new `UIBezierPath` class, you'll follow step-by-step instructions to add the ability to produce and save drawings as PDF files.

If your app needs to distribute only a single image, then exporting it as a PNG or JPEG may be the obvious path. The same holds true for plain text that's much easier to edit when saved as an ASCII text document. But what if your app needs to export a rendered web page or a sales report full of visual graphs and charts? For more complex layouts that include multiple images, tables, and styled text, saving the data as a multipage PDF file is a great solution.

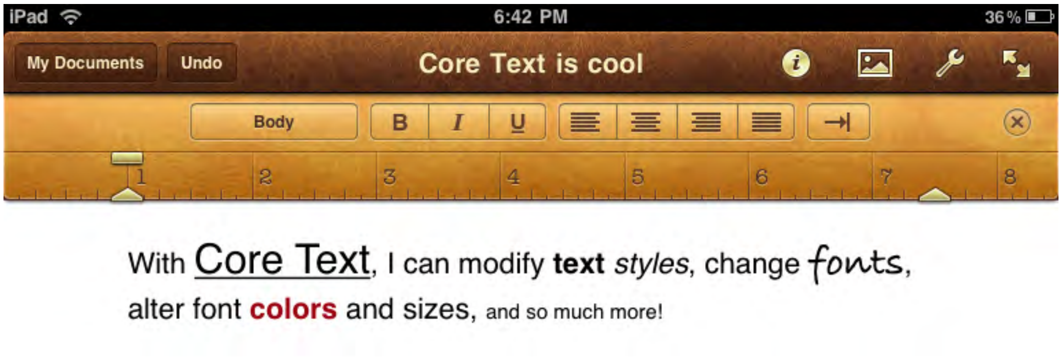
That's right, I mentioned styled text! You're not dreaming, and it's not a typo. Keep reading!

www.it-ebooks.info

Core Text

As an iPhone developer, the lack of any easy-to-use text styling functionality has probably annoyed you on countless occasions. Sure, you can display styled text as HTML in a `UIWebView`, but what about editing that styled text? For years, you've been jealous of the wonderful Core Text APIs that were available only to Mac OS X developers, wishing you could tap into that same functionality within Cocoa Touch. As of iOS 3.2, your wish has finally been granted!

Even though Apple has never officially confirmed that it utilizes Core Text in its stunning word processor app, Pages for iPad (see Figure 3–2), the arrival of Core Text in iPhone SDK 3.2 enables you to add similar sophisticated styled text features to your own iPad apps.



With Core Text, I can modify text styles, change fonts, alter font colors and sizes, and so much more!

Figure 3–2. *With Core Text, you can build styled text features into your app, similar to Apple's Pages for iPad.*

Although all of us would love to use a word processor interface like Pages in our own apps, `UIKit` does not include a ready-made word processing control for easily editing text. To emulate such a beast, you'll need to build your own from scratch. Rendering portions of a text string with different styles, fonts, sizes, or colors will require quite a bit of work on your part, but the result is well worth the effort.

Using Core Text, you draw styled text into a graphics context. To assign custom font styles to specific segments of your text, you collect the text with this associated style metadata in a special attributed string, appropriately named `NSAttributedString`. To add that text information via Core Text, you then create a `CTFramesetter` by passing that attributed string to the function, `CTFramesetterCreateWithAttributedString`. Next, you construct a `CTFrame` object by passing your `CTFramesetter` (the styled text) and a `CGPath` (a bounding rectangle area) to the `CTFramesetterCreateFrame` function. Lastly, call the `CTFrameDraw` function to draw the styled text into the designated graphics context.

Of course, I've oversimplified the steps here in order to give you a general idea of how Core Text is structured. Working with Core Text can be rather complicated, so I wouldn't recommend utilizing it for trivial text-input fields. But if you're determined to build the next great mobile word processing powerhouse for the iPad, then Core Text is your answer.

www.it-ebooks.info

Of all the new features in iPhone SDK 3.2, the Core Text classes are probably the most difficult to grasp. That's where Jack Nutting swings to the rescue! My esteemed coauthor breaks it all down in Chapter 5 by showing you how to add a Core Text-driven text tool to the Dudel drawing app. Complete with code examples and expert guidance, that chapter provides the basic building blocks needed to begin using Core Text in your own iPad apps.

Popovers

With iPhone apps, the small screen real estate could display only a very limited amount of controls and content. To keep the interface clean and easy to use, access to additional settings and elements were presented in separate views. This required shuffling back and forth between various screens.

Even though the iPad's larger screen size gives developers room to include more functionality into a single, consolidated screen, the design objective still remains the same: keep it simple. Rather than clutter the screen with an overly complex interface, your goal should be to minimize the interface where ever possible, allowing users to focus on your app's primary purpose and content. To solve this problem, popovers were introduced in iPhone SDK 3.2. Exclusive to the iPad, popovers display a secondary view on top of the main view. Typically, this subview contains user-selectable settings or additional contents that do not require the full screen.

Remember the iPad apps showcased in Chapter 1? On the iPad, both Brushes (Figure 1–4) and Synotes (Figure 1–8) utilize popovers to display views that previously required navigating between separate iPhone screens. A popover controller can contain almost any kind of view you want. Although popovers are most commonly displayed when users tap toolbar buttons, you can program a popover to appear when tapping other types of objects, such as an image, a map item, or a custom interface element. In The Iconfactory's Twitterrific for iPad, tapping on a Twitter user's avatar icon conveniently presents a popover view of that user's Twitter profile information.

The iPad places an increased importance on toolbars. Unlike the iPhone, where toolbars are limited to the bottom of the screen, iPad apps support toolbar placement on both the top and bottom of your interface. In fact, since the split view controller (introduced later in this chapter) relies on a top toolbar layout, Apple recommends placing your toolbars at the top. In many aspects, this actually brings iPad interface design much closer to a traditional desktop application layout than that of an iPhone app.

In Apple's Pages for iPad, the toolbar's buttons present popovers for choosing various document styles and settings. In Figure 3–3, the Tools popover shows a `UITableView` with several options. Some of the items even include user-selectable controls.

www.it-ebooks.info

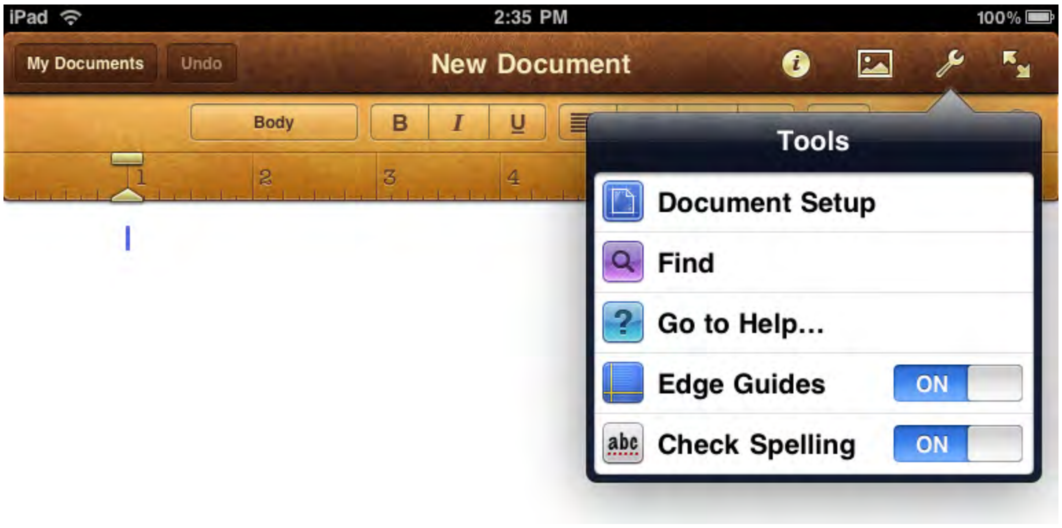


Figure 3–3. *Popovers are a great way to display user-selectable options that don't require a modal view. An effective use of popovers can be seen in Apple's Pages for iPad.*

If your app's main toolbar (or navigation bar) is configured with a default color, then a toolbar within a popover will inherit the popover's native dark-blue outline. If you assign a custom color to the toolbar, that custom color is shown instead, with the popover's dark-blue outline surrounding the view. With that in mind, if you insist on using a custom toolbar color, make sure it's a color that complements the popover outline coloring.

For best results, I recommend sticking with a default color for your app's main toolbar, unless you modify your popover code to enforce a default color for its own popover toolbar. For example, even though Pages uses a custom brown color for its main toolbar, Apple decided not to implement that custom color in its popover toolbars. This also allows a popover to visually contrast with the interface behind it, making its hovering box easily distinguishable from its parent view. If your popovers don't contain their own toolbars, then this won't be an issue for you.

Think strategically when designing your app's interface with popovers. Is your app overflowing with features? Instead of piling several buttons into a toolbar, with each one displaying a separate popover, try to consolidate all your subviews into only a few popovers. This can be done within a popover view by adding a segmented control to a toolbar. Each segmented tab loads a different view into the same popover. The feature-rich Pages for iPad effectively utilizes this concept, as shown in the example in Figure 3–4.

www.it-ebooks.info

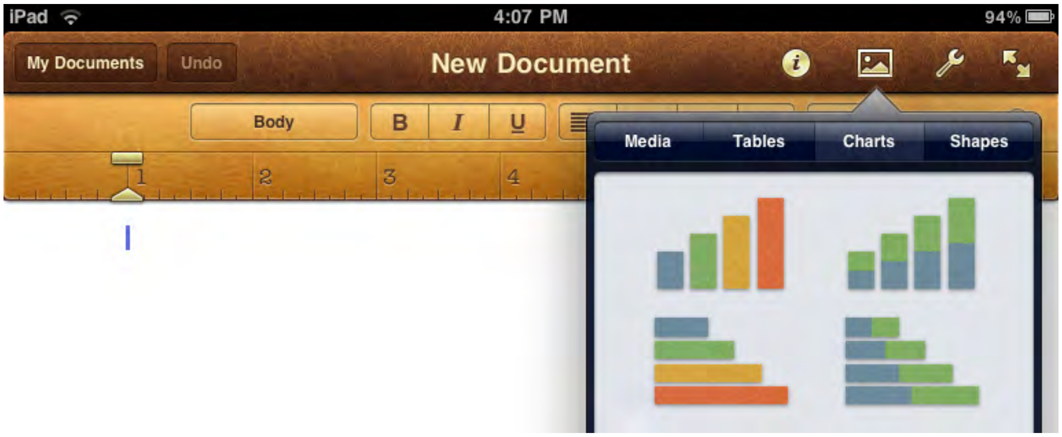


Figure 3–4. *Simplify your interface design! Within a popover, use a segmented control in a toolbar to consolidate multiple, related subviews.*

Beyond displaying custom views, popovers are also handy for presenting only a few options. Instead of showing an alert sheet, a popover is the more appropriate method on the iPad for presenting those options. A good example of this is tapping the Add Bookmark button in Mobile Safari. On the iPhone, an alert sheet is called. But on the iPad, alert sheets are displayed as popovers, as shown in Figure 3–5.

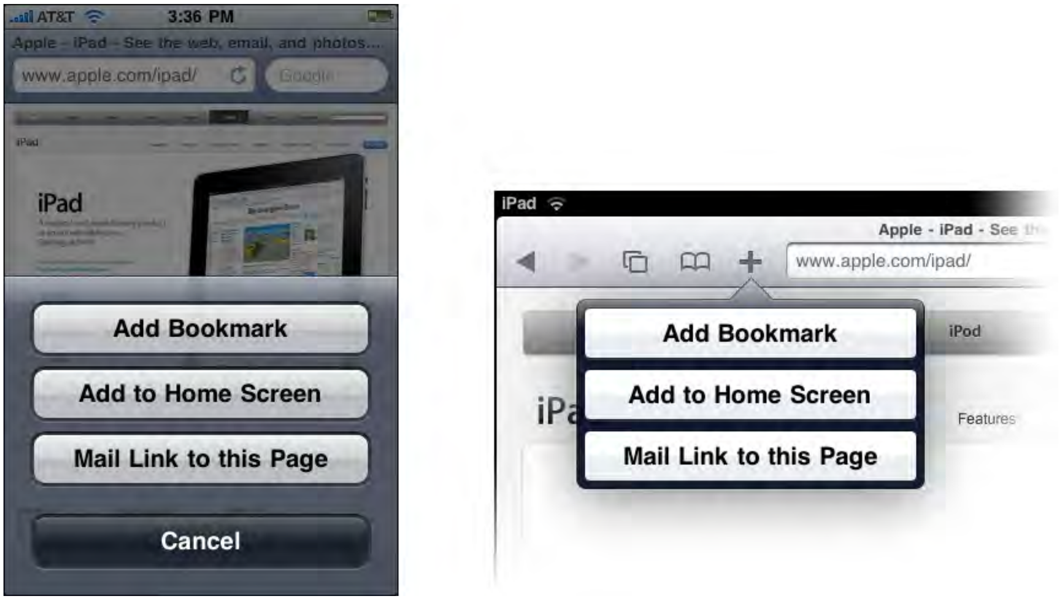


Figure 3–5. *Although an alert sheet is a good choice for displaying a few options on the iPhone (left), presenting those options as a popover is a better solution on the iPad (right).*

Unlike an alert sheet, a popover should never include a Cancel (or Close) button. If a user taps outside a popover, the popover will disappear. But any selections made within

www.it-ebooks.info

the popover will not automatically dismiss the popover, requiring you to programmatically close the popover yourself. Since Apple recommends using popovers for user-selectable options, you may wonder why this is the case. There is actually a good reason for this design. Since popovers generally include not only user-selectable items, but also other tappable elements like segmented controls (as shown in Figure 3–4), you don't want the popover disappearing after just any finger tap. With control over the closing of the popover, you can designate exactly how and when the popover is dismissed based on a user's selection.

If you need users to make a specific choice before allowing them to return to the main view, you can force the popover to be modal (dimming the screen area behind it), but depending on your needs, a popover may not be the ideal solution for that use case. For many situations where that behavior is needed, your best bet may be to present a modal view instead, as discussed in the "Modal Presentation Styles" section later in this chapter.

Displaying a popover in your code is actually quite easy. In a nutshell, you create a new instance of `UIPopoverController` and pass a custom view controller to it (which will be loaded into the popover view). The parent view should be assigned to the popover's delegate, so that communication can take place between the two. To show the popover when a user taps a toolbar button, you call the `presentPopoverFromBarButtonItem` method. If the popover is being displayed when a user taps another interface element such as an image, you should call `presentPopoverFromRect` instead.

The default size of a popover is 320 pixels wide and 1100 pixels tall, but you can easily customize the width and height with the `popoverContentSize` property. But you may find it interesting that the default 320-pixel width is the same size as the iPhone's portrait mode width. That's no coincidence! With that default width, it's much easier to convert most existing iPhone app views into popovers when creating an iPad version—you won't need to redesign much (if any) of the view's original layout.

There are a few additional configuration options and considerations when using popovers, which are covered at length in Chapter 6. You'll walk through the creation of several popovers as you continue to develop the Dudel drawing app, so that chapter is a must-read.

Popovers might just be one of the most important new features of iPhone SDK 3.2. Certainly, this new interface component will prove to be a very useful new weapon in your iPad development arsenal.

Video Playback and Display Options

As of iOS 3.2, Apple has changed the way the `MPMoviePlayerController` class works. In previous versions, videos were always played in a full-screen player interface. The iPad now offers an enhanced movie player that can be displayed in either full-screen mode or embedded within your app's views.

www.it-ebooks.info

The YouTube app that's included on the iPad is a perfect example of this new video functionality. In landscape orientation, videos play full-screen as usual, but in portrait orientation, videos play within the app's interface, as shown in Figure 3–6.

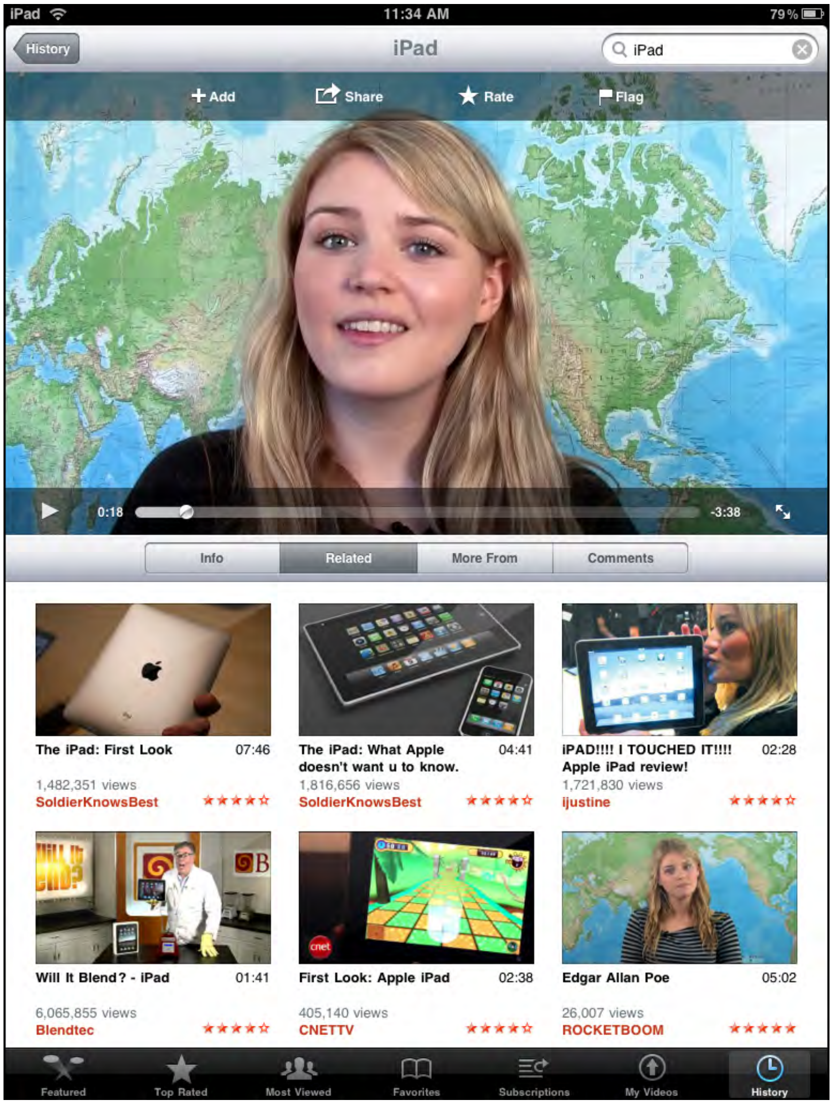


Figure 3–6. *The iPad's YouTube app showcases the MPMoviePlayerController's new embedded video playback functionality.*

One of the many advantages of this embedded player feature is that it gives you the option to allow users to interact with other elements in the app while the selected video plays. The enhanced movie controller also enables developers to change videos without initiating new controllers, overlay additional views on top of the current movie, generate thumbnail images from video frames, control the playback options via code, and much more.

www.it-ebooks.info

It's important to note that in order to provide these new capabilities and the improved playback interface, some of `MPMoviePlayerController`'s previous API has been deprecated, replaced with new methods and properties. This was necessary in order to provide developers with more granular control over the presentation of the movie. For example, instead of the movie player controller handling the video's presentation on the screen, it now provides a view object that acts as a container for your video content, giving you much more control over the movie's overall display and playback within your app. If you're porting existing iPhone movie player code to an iPad app, you'll need to modify that code to ensure that it works properly in iOS 3.2.

Beyond the iPad's display, there's also new support for presenting content on an external monitor or projector when connected to an iPad. Using the `screens` method of the `UIScreen` class, you can program your iPad app to detect if an external display device is connected via a compatible cable. `UIScreen` also includes methods and properties for not only accessing the external screen's resolution, but also for configuring your app's content for proper viewing on the connected device. But this feature isn't limited to mirroring your iPad's screen. You can also project any additional view onto the external display by assigning it to that screen object. This will prove to be a very valuable feature for iPad developers building business and media apps that need to present content on a desktop computer monitor, a projector screen, or even a TV.

In Chapter 7, you'll learn how to program your iPad apps to utilize the enhanced `MPMoviePlayerController` class, as well as how to communicate with external display devices connected to the iPad using `UIScreen`.

Split View Controller

After popovers, the new split view controller is the second most distinctive feature that distinguishes iPad apps from their iPhone siblings. Navigating back and forth between various views is a good solution for the small iPhone screen, but on the much larger iPad display, that interface mechanism is no longer necessary. To make efficient use of the iPad's extra screen real estate, while also helping developers migrate existing iPhone navigation systems to the tablet, Apple introduced a new view controller called the `UISplitViewController`.

True to its name, a split view controller contains two panes: master and detail. The master pane typically holds the navigation or primary table view for the app. Within the master pane, users can make selections. If a chosen item requires a display, then its data is loaded into the detail pane. For example, in a note-taking app, the master pane would list all of the user's saved notes. Selecting a note would open it in the detail pane, where it could be read and edited by the user.

The master pane is fixed at 320 pixels wide, while the detail pane consumes the remaining width of the window. Notice the recurring 320-pixel width? Just like the default width of popovers, this was a strategic design decision by Apple to make the conversion of iPhone apps into iPad apps as painless as possible. A navigation bar from an iPhone app could be repurposed for use within the master pane of a split view-based iPad app.

www.it-ebooks.info

As discussed in Chapter 2, the Xcode Split View-based Application project template provides a convenient starting point. Although that template populates the master pane with a table view list, you can just as easily add a navigation bar controller, if you need to provide the ability to drill down through a few levels of content within the master pane before displaying a selection in the detail pane.

In landscape orientation, the master pane is located on the left side, and the detail pane is on the right side of the screen, as shown in Figure 3–7. See how the detail pane includes a toolbar at the top of the view to match the toolbar or navigation bar in the master pane? Not only does the consistency provide a visually pleasing and balanced interface design, but this layout also reinforces Apple's push for consolidating an iPad app's primary buttons into a top-aligned toolbar.

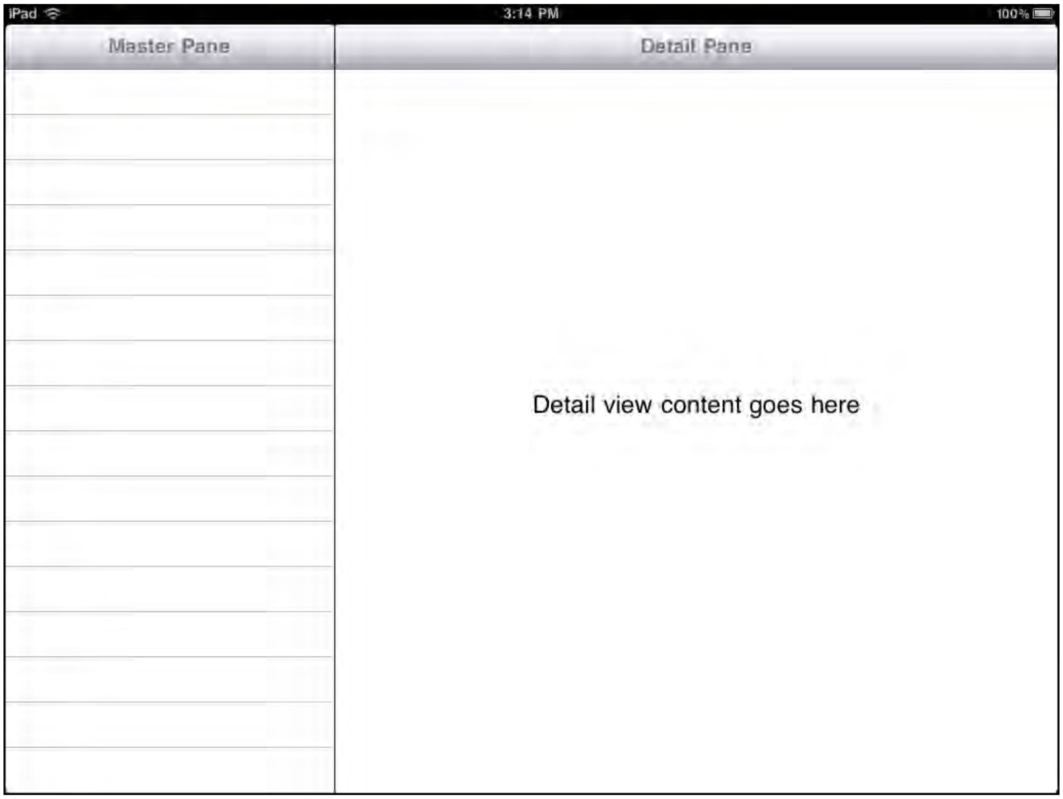


Figure 3–7. *The two-pane layout of a split view controller in landscape orientation*

Beyond aesthetics, the detail pane's toolbar serves another important purpose for the split view controller. To help preserve your interface design within the detail pane, rotating the tablet to the portrait orientation allows the detail pane to use the entire screen. In order to keep the master pane accessible to users, the split view controller automatically adds a `UIBarButtonItem` to the left side of the detail pane's toolbar. Tap that button, and a popover displays the master pane's view, as shown in Figure 3–8.

www.it-ebooks.info

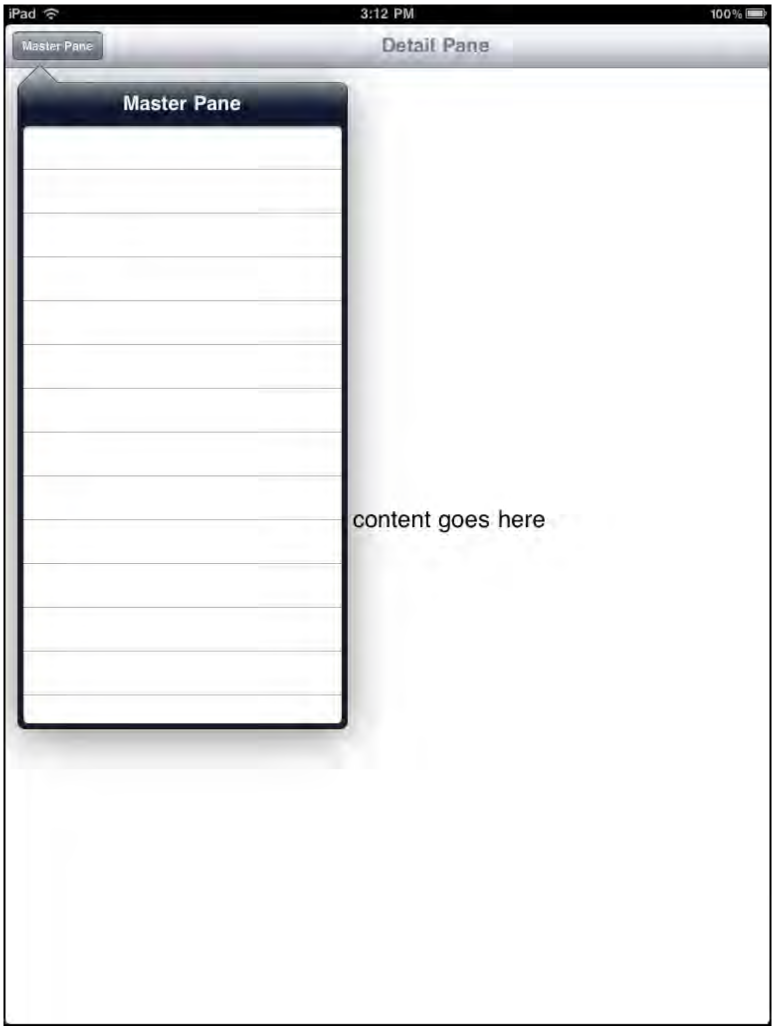


Figure 3–8. *When a split view-based app is in portrait orientation, the contents of the master pane are accessible by tapping its toolbar button to reveal a popover.*

Since the detail pane usually represents the detailed data of the item selected, the master pane should reflect the current selection. So if it's a table view row, then your code should ensure the selection remains persistent. In a simple notes app, for example, if the user is viewing a specific note in the detail pane, the master pane could visually indicate the current selection by maintaining a highlighted or checked table view row of that listed note.

Although the split view controller handles much of its functionality for you, there are some essential implementation details worth learning in order to customize it for use with your own interface needs. Going beyond the basic Split View-based Application template, Chapter 8 walks you through the steps of manually adding a `UISplitViewController` to your Xcode project.

www.it-ebooks.info

Modal Presentation Styles

As an iPhone developer, you're already familiar with how to make a `UIViewController` modal, which prevents the user from returning to the parent window until the modal view is closed. A modal view is a great solution when you need to present a much more sophisticated layout than what's possible in a limited `UIAlertView`.

On the iPhone, a modal view fills the entire screen, which is perfectly fine with only 320 by 480 pixels. But on an iPad, there's considerably more display space, so you may not always want a modal view that stretches the full 1024 by 768 pixels. To accommodate the larger surface area, Apple has introduced four new modal style options, which can be assigned to a new `UIViewController` class property called `modalPresentationStyle`. As on the iPhone, you still call a modal view via `presentModalViewController`, but before doing so, you simply assign one of the new style options to the view controller's `modalPresentationStyle` property.

For example, let's say your code already has an instance of `UIViewController` named `myController`. You could assign a modal style to it before presenting it on the screen, like this:

```
myController.modalPresentationStyle = UIModalPresentationFormSheet;
[self presentModalViewController:myController animated:YES];
```

As you can see from that code snippet, one of the new style options is `UIModalPresentationFormSheet`, which has a fixed size of 540 pixels wide by 620 pixels tall. Being smaller than the iPad's window, it is displayed in the center of the screen, with the parent view dimmed gray behind it, as shown in Figure 3–9.

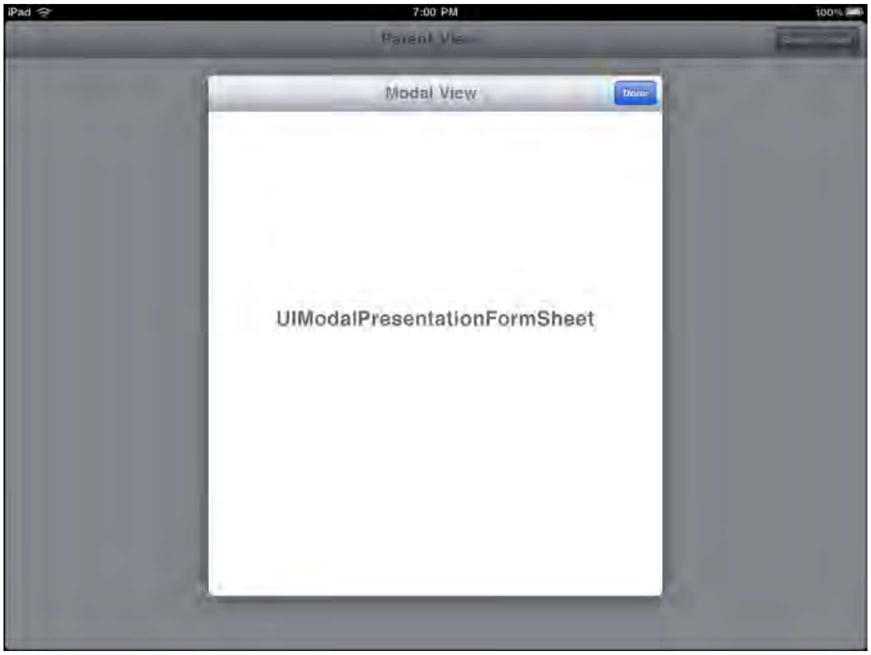


Figure 3–9. *The UIModalPresentationFormSheet style is centered on the screen with a fixed 540-by- 620 size.*

www.it-ebooks.info

The next option is `UIModalPresentationPageSheet`, which assumes the current height of the screen and a fixed width of 768 pixels. This means that in portrait orientation, it appears to fill the entire screen, but in landscape orientation, the dimmed gray parent view can be seen in the background on both sides, as shown in Figure 3–10.

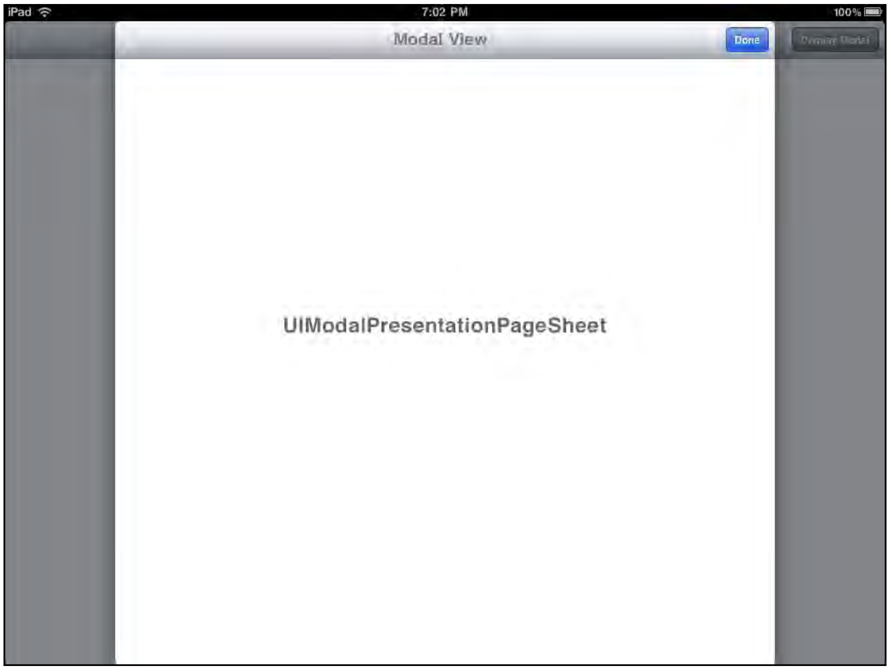


Figure 3–10. *UIModalPresentationPageSheet has a fixed 768-pixel width, but spans the full screen height.*

If you do need the modal view to utilize the entire screen, you can set the `modalPresentationStyle` property to `UIModalPresentationFullScreen`. But what do you do when the rare need arises to display a modal view within a popover or one of the split view panes? That's where `UIModalPresentationCurrentContext` comes to the rescue, presenting the modal view in the same size as the parent view that called it. For example, a `UIModalPresentationCurrentContext`-assigned modal view shown within a popover would use the same width and height dimensions as the popover.

Even though tapping outside a popover will automatically dismiss it, that won't work with modal views. Just like its counterpart on the iPhone, a modal view needs to be programmatically closed on the iPad. This can be achieved by including a Done button (as shown in Figures 3-9 and 3-10) or by designating this task to some other user interaction within the modal view.

In Chapter 8, you will learn how to put these new modal view styles to good use in your own iPad apps.

www.it-ebooks.info

Advanced Input Methods

iOS 3.2 also includes a new set of custom input methods that developers can use in their apps: edit menu actions, keyboard layouts, and gesture recognizers.

Edit Menu Actions

Depending on the object you tap and hold your finger on, the small, black edit menu that appears on the screen will display one or more of the default menu actions, such as **Copy**, **Cut**, **Paste**, **Select**, **Select All**, and **Delete**. Now, with access to the `UIMenuController`, you can insert your own custom actions into the edit menu for a specific object.

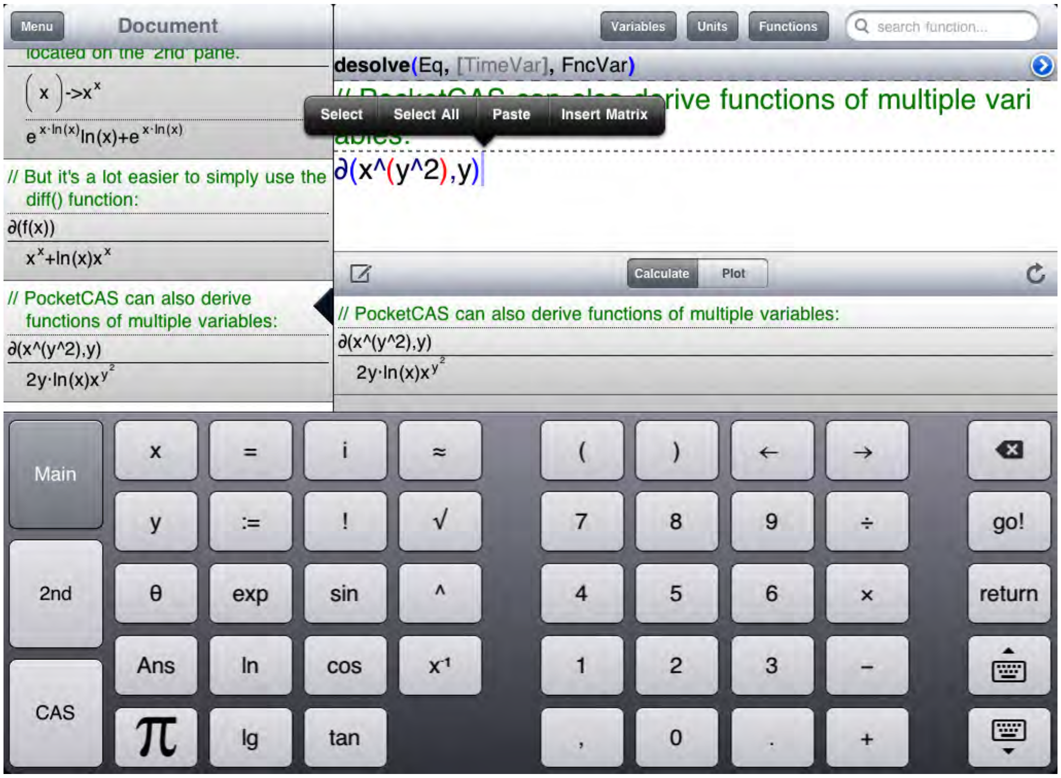
A custom menu action consists of a `UIMenuItem` with a title property and an action selector. You then assign your `UIMenuItem` to the `UIMenuController` of the appropriate object type. In order to facilitate the target action behavior, you also need to identify a target for your new menu item by setting the applicable view as the first responder for that action. The last step is to write the actual action method for handling that task if the user selects it.

For example, if you wanted to add a custom menu item for a thesaurus when a text word is selected, you could create a new `UIMenuItem` instance with the title Thesaurus that points to an action selector `thesaurusLookup`. Add that `UIMenuItem` to the `UIMenuController` assigned to that text object, and your custom **Thesaurus** menu item will appear along with the default **Copy**, **Cut**, and **Paste** options in the edit menu. The assigned target would be the parent view of that text object. The parent view controller's source code would need to include your action method, `thesaurusLookup`, so that when a user selects that menu item, your app knows how to respond.

When you add custom items, keep their menu titles short, with no more than one or two words per item. To prevent users from being overwhelmed, try not to add too many additional items to an edit menu. If you need to provide several options to the user, you should consider presenting them in a popover action sheet instead.

Eager to add custom menu items (such as the one shown in Figure 3–11) to your own iPad apps? In Chapter 9, you'll learn how to accomplish this with only a handful of code lines.

www.it-ebooks.info



located on the "2nd" pane.

(x)->x^x

e^{x·ln(x)}ln(x)+e^{x·ln(x)}

// But it's a lot easier to simply use the diff() function:

∂(f(x))

x^x+ln(x)x^x

// PocketCAS can also derive functions of multiple variables:

∂(x^(y^2),y)

2y·ln(x)x^{y^2}

desolve(Eq, [TimeVar], FncVar)

// PocketCAS can also derive functions of multiple variables:

∂(x^(y^2),y)

// PocketCAS can also derive functions of multiple variables:

∂(x^(y^2),y)

2y·ln(x)x^{y^2}

Figure 3–11. *Daniel Alm and Thomas Osthege used custom edit menu items and keyboard layouts in their PocketCAS graphics calculator iPad app. Alongside the standard edit menu commands is a custom Insert Matrix menu item.*

Keyboard Layouts

Did you notice the unique keyboard in Figure 3–11? No, it's not an interface trick. No longer limited to only the standard keyboard, you now have a way to present your own keyboard layout, which is nothing more than a custom view. To replace the system keyboard, you assign the view to the `inputView` property of a `UITextField`, a `UITextView`, or any compatible responder object.

Since users are already familiar with the system keyboard, don't stray too far from the default design when creating your own custom keyboard. Make sure the buttons appear tappable and automatically size to display well in both portrait and landscape orientations. The background of your view should extend to the full width of the screen. The height is flexible, but Apple recommends sticking with the same height as the system keyboard for consistency when possible. As an example, the PocketCAS app shown in Figure 3–11 does a nice job of emulating the look and feel of Apple's virtual keyboard design, which reduces the learning curve for first-time users.

If you need to insert only a few additional buttons to the existing system keyboard, another option is to add a keyboard extension, called an input accessory view. If you've

www.it-ebooks.info

ever filled out a web form in Mobile Safari, you may have noticed the extra translucent black toolbar that runs across the top of the keyboard, as shown in Figure 3–12. As with a custom keyboard, this is accomplished by creating a view that contains the additional interface elements you want displayed above the keyboard. Whereas a keyboard replacement uses `inputView`, your input accessory view should be assigned to the `inputAccessoryView` property instead.

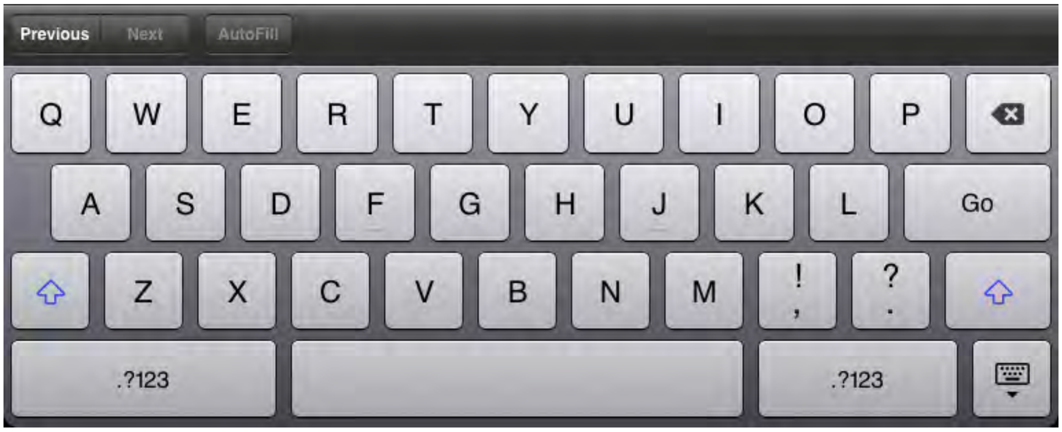


Figure 3–12. *When filling out a web form in Mobile Safari, the keyboard includes an input accessory view, which is a toolbar-like interface added to the top of the keyboard.*

Programming your app to utilize a custom keyboard layout or input accessory view can be somewhat complicated, but Chapter 9 takes you step by step through the process with helpful sample projects.

Gesture Recognizers

The user interface components in the `UIKit` framework include handling for basic touch events, such as tapping a `UIButton`, but what if you need to add event handling for specific touch behavior to a custom view or object? To help simplify what was previously a laborious task, Apple has provided iPad developers with a new `UIGestureRecognizer` class for easily detecting touch gestures. `UIKit` includes six common gesture recognizers, which are subclasses of `UIGestureRecognizer`:

- `UITapGestureRecognizer`: Finger taps.
- `UILongPressGestureRecognizer`: Holding a finger down on one spot.
- `UIPinchGestureRegnizer`: Pinching fingers in and out.
- `UIPanGestureRecognizer`: Dragging a finger.
- `UISwipeGestureRecognizer`: A quick finger swipe.
- `UIRotationGestureRecognizer`: Rotating two fingers in opposite directions.

www.it-ebooks.info

To add a gesture recognizer to a view, you first create a new instance of one of the six `UIGestureRecognizer` subclasses. Like a custom edit menu, an action selector is assigned to the gesture recognizer instance. This informs the view which method to call when the user performs that gesture. Some gesture recognizers have configurable attributes, such as `numberOfTapsRequired`, which sets the number of taps for a `UITapGestureRecognizer`. In order to give the gesture recognizer a target, it needs to be attached to the view by calling the `addGestureRecognizer` method.

If you do use one of the standard six gesture recognizers in your app, it's important to use it for an action that users associate with that gesture. For example, people know that finger pinching is typically used for zooming in and out of an image. If your app uses that `UIPinchGestureRecognizer` for deleting files, the unorthodox use of that gesture will only lead to confusion (and possibly even rejection from the App Store).

If you need a unique gesture recognizer, you can create your own subclass of `UIGestureRecognizer` and override all of its methods (such as `touchesBegan`, `touchesMoved`, `touchesEnded`, `touchesCancelled`, and `reset`) with your desired functionality. The drawback to implementing support for custom gestures is that they are unknown touch commands. It becomes your app's responsibility to properly educate users on how to use the new gestures. Unless you have a compelling reason to go this route, sticking with the well-known, common gestures is usually the best approach.

In Chapter 9, you'll add undo support to the Dudel drawing app by implementing a gesture recognizer.

Document Support

The iOS does a good job of hiding the underlying filesystem, so that users can focus on creating and consuming content. But the iPad's larger screen encourages greater productivity, so situations arise where users will want to control how some files are opened and shared between apps.

In the past, this was always tricky due to how each app was limited to its own "sandbox" directory, but with iOS 3.2, Apple introduced a new file-handling mechanism called Document Support. Apple's built-in Mail app is a good example of this new feature. If an e-mail contains a file attachment, it's displayed as a file icon at the bottom of the message. If you currently have an installed app that has registered itself with iOS as the "owner" of that file format, the e-mail attachment's file icon will reflect that app's icon. For example, if the e-mail attachment is a Microsoft Word document and you have Apple's Pages installed on your iPad, the file icon may look like the Pages app icon. As expected, if you tap the file, it will open in Pages. Since both Mail and Pages are Apple apps running on an Apple tablet, this comes as no surprise.

But the real beauty of Document Support is the power it provides to developers. It also allows a user to open that e-mail attachment in any other registered app that supports that file type! If you hold your finger on the e-mail attachment icon, a popover will appear with a few options. One option, of course, is to open the file in its owner app. In the case

www.it-ebooks.info

of the Word document, that option might be Open in Pages. But one of the other options is **Open In…**. Selecting **Open In…** replaces that popover with a new popover, listing all of the registered apps that can open Word documents. I happen to have the excellent app, GoodReader, installed on my iPad, which also supports Word files, so it's listed alongside Pages in that Open In… popover, as shown in Figure 3–13.

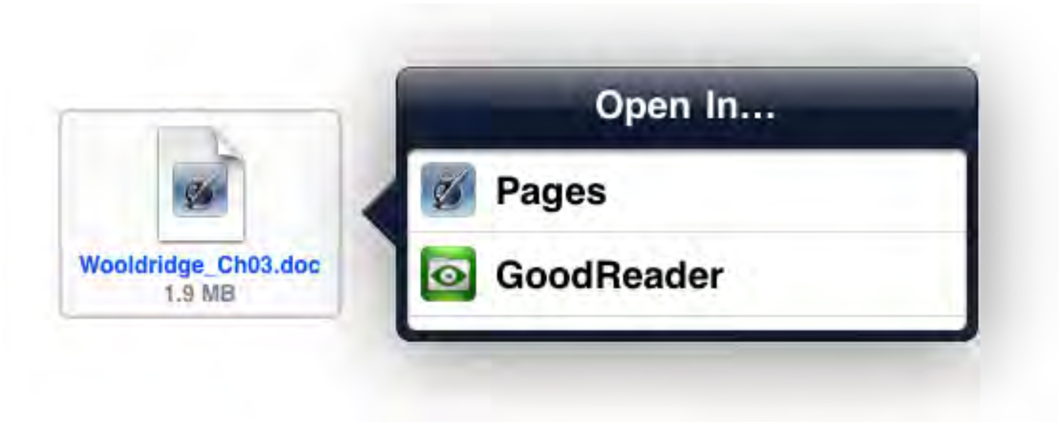


Figure 3–13. *Document Support enables the Mail app to suggest opening an e-mail attachment in other registered apps that are capable of opening that file type.*

If I select GoodReader from the list, the Word document not only opens in GoodReader, but a copy of the file is also stored in GoodReader's file directory, accessible to me any time I run GoodReader. This is a safe and sanctioned way to transfer a file from one app to another, without sacrificing the security of an app's sandbox.

So how does this all work? There are actually two factors that make this functionality possible: the sender app and the receiver app, both of which require different development steps. In Figure 3–13, the Mail app is the sender app, and Pages and GoodReader are receiver apps.

In order for your app to send a file to another app (as Mail does), you need to use the `UIDocumentInteractionController` class. A document interaction controller communicates with the iOS to see if the selected file can be previewed by the system and if any other installed apps are registered to open that file format (the Open In… popover list).

If you want your app to act as a receiver, it needs to notify the iOS registry of the specific file types that it can open. This is done by including each supported file type in the `CFBundleDocumentTypes` key of your app's *Info.plist* file. Each file type declaration consists of four attributes: name, the related uniform type identifier (UTI), handler rank, and file image icon. The handler rank informs the system whether your app is the owner of the file type (such as your own proprietary file format) or is simply capable of opening that kind of file. The image icon is optional for file type owners (which will be discussed later in the section "Required Project Resources for iPad Apps").

www.it-ebooks.info

If your app is registered with iOS as supporting a particular file type, then it will need to be able to field requests to open related files upon app launch. If another app, such as Mail, uses a document interaction controller for that file type and a user selects your app from the Open In… list, your app needs to be ready to handle that request, which is delivered to the `application:didFinishLaunchingWithOptions` method in your application delegate. The request arrives with an options dictionary that includes important information about the file your app needs to open, such as the file's location, the sender app's bundle identifier, and an annotations property list object that contains additional data about the file.

If your iPad app opens and saves files, you really should take advantage of the new Document Support feature. It provides greater flexibility for your app's offerings and better interoperability with other installed apps. To learn more about utilizing Document Support, be sure to read Chapter 10. Your users will thank you for it!

Universal Applications

Even though most iPhone apps will run on the iPad, their smaller dimensions appear rather pixelized and inferior to native iPad apps. But some developers may not want to maintain two separate Xcode projects for essentially the same product in order to properly support both platforms. To solve this problem, iPhone SDK 3.2 introduced a new universal application format that runs on both iPhone and iPad devices. Depending on the device running the universal application, the appropriate version of the app is launched. This way, you can maintain one Xcode project with shared source code, but design separate user interfaces specifically tailored for each platform. For example, your iPhone app may use a navigation controller for organizing content, yet on an iPad, you would most likely want to display a split view controller instead. Both versions use the same data, but present it in different ways that best suit the chosen device.

For developers targeting both platforms, Apple highly recommends building universal applications. Managing and updating only one application in the App Store makes it much easier for customers who use your app on both their iPhone and iPad. But if your iPad version is radically different from your iPhone app, with dozens of new features that require a fairly hefty code rewrite, a universal application may not be the ideal choice. If the two versions don't share much in the way of code, it may make more sense to build them as two stand-alone products: one for the iPhone and one for the iPad. There are also business and marketing factors that come into play as well, which we'll explore in the upcoming "To Be or Not to Be Universal" section. If you do make the decision to distribute your product as a universal application, it requires some extra planning and effort on your part.

Universal App Requirements

The first minor hurdle for universal app development is orientation. With the iPhone's small screen, providing an efficient user interface design often requires a dedicated orientation, such as a portrait-only app. That's perfectly acceptable on the iPhone, but the iPad's larger display allows more of your interface elements to be consolidated into

www.it-ebooks.info

a single window, providing enough screen space in both portrait and landscape views. In fact, Apple insists that iPad apps should support all orientations. If your iPhone app is locked into a single orientation, you'll need to configure your universal app project to accommodate multiple orientations when displaying your iPad version's user interface.

So how does the app differentiate between your iPhone code and your iPad code? This is where we encounter the next hurdle in developing universal applications. Not only does your app need to detect the current platform and run the appropriate code, but when compiling your project in Xcode, you'll need to properly "wrap" your iPad code to prevent compiler errors. Remember that in producing a universal application, the iPad features exclusive to iPhone SDK 3.2 will cause compiler errors when Xcode is building the iPhone portion of the app, so you'll need to use conditional coding to prevent that from happening.

Although you may be tempted to simply check the user's device type or operating system version, with Apple constantly releasing new devices and iOS versions, that's not the way to go. A better approach is to test for the availability of exclusive iPad classes using `NSClassFromString`. If you pass an iPad-only class name, such as `UISplitViewController` to `NSClassString` and a valid object is returned, you'll know the user's device is an iPad. If `nil` is returned, then that iPad class doesn't exist, so the user's device is an iPhone or iPod touch.

For new iPad functions that have been added to existing frameworks, checking the class name's existence will not be effective. In those cases, you can compare a specific function name with `NULL`. If an iPad-only function is equal to `NULL`, then the user's device is not an iPad.

You'll also need to perform similar code checks if your app utilizes hardware features that are available on only one of the platforms, such as the iPhone's camera. And don't forget your *.xib* interface files and images that are uniquely designed for a specific screen size. Since your iPad app may require a different interface and graphics than your iPhone app, your view controller classes will need to be programmed to load the correct resources.

All of this talk about conditional coding may sound like a lot of work, and truthfully, it is fairly time-consuming to implement at first. But your efforts will prove worthwhile when working on future updates of your app. Adding new features to a single Xcode project that shares common code between the two platforms is much more time-efficient than needing to add the same code to separate projects.

Chapter 11 covers the process of creating a universal application, complete with expert tips and conditional coding examples. You'll also learn how to convert an existing iPhone project into a universal app using Xcode's new Upgrade Current Target for iPad feature.

www.it-ebooks.info

To Be or Not to Be Universal

From a development standpoint, there are many advantages to creating a universal application, but is it the right choice for you? If your app is free, then your goal is to provide the most convenient, user-friendly access to it. A universal application makes it easy for users to download your app across all of their Apple mobile devices. But paid apps are a different story. Putting aside the technical benefits for a moment, let's look at the business factors involved.

If your iPad app represents an enhanced edition, offering dozens of exclusive new features that are not available in your iPhone version, it may make more sense to release it as a separate, stand-alone iPad app. As a universal application, existing owners of your iPhone app will be able to access the iPad version for free, since there's no official upgrade mechanism supported in universal applications. By selling the iPad app as a separate product, you have the opportunity to recoup your development costs. And if it provides additional value above and beyond your iPhone edition, then most customers won't have a problem with paying for it, even after they've already purchased the iPhone version. I say "most" because there will always be a select few users protesting that they should receive all app versions for all applicable Apple devices for free. Ironically, the loudest complaints usually come from people who paid only 99 cents for your original iPhone app. But don't cut off a potential revenue stream that could help support your continued development just because you're worried about keeping everyone happy. Here's a little secret: It's not possible to please everyone. Just build the best features and user experience possible. If you provide your customers with additional value, most of them will be more than happy to pay for the enhanced iPad version.

On the other hand, if your iPad app does not offer anything new beyond an iPad-optimized interface slapped on top of the same iPhone feature set, you may want to consider a universal application. If you can't justify the iPad app price with additional iPad-exclusive functionality, selling it as a separate product will definitely attract an angry mob of customers, wielding pitchforks and writing negative App Store reviews! And Apple may just agree with them. Apple has been known to reject stand-alone iPad apps that don't add any significant value beyond what's available from their iPhone counterparts. In these situations, Apple usually advises the developer to convert it into a universal application before resubmitting it to the App Store.

Another major issue to consider is the file size of your app. A universal application combines the incremental code and separate *.xib* files and image resources for both the iPhone and iPad versions into one package, which means it can often be nearly double the file size of a single platform app. Although Apple recently raised the cellular 3G download limit from 10MB to 20MB to help accommodate universal applications, some content-heavy apps such as games may still exceed that file size. If your universal application is larger than 20MB, that drastically reduces your app's potential audience to only people within Wi-Fi range. Whether your app is free or a paid product, this factor alone may force you to release separate iPhone and iPad versions to ensure that your app can be downloaded by both Wi-Fi and cellular 3G connections.

www.it-ebooks.info

Required Project Images for iPad Apps

For an iPhone app compiled for iOS 3.1.3 or an earlier version, you were required to include a 57-by-57-pixel app icon and a default launch image sized for portrait orientation in your Xcode project. With iOS 3.2 and the iPad's support for multiple orientations, the project images that an iPad app or a universal application requires are much different.

App Icon Images

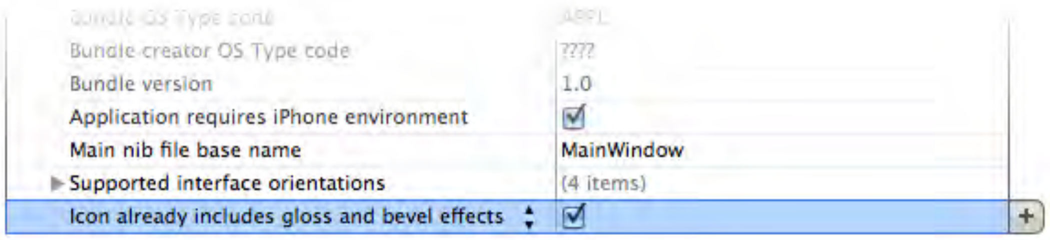
Beyond the usual 512-by-512-pixel app icon that's required by the App Store, you'll need to add the following iPad icon files to your Xcode project's Resources folder:

- *72-by-72-pixel PNG image*: The app icon displayed on the iPad's home screen.
- *50-by-50-pixel PNG image*: The app icon shown if your app name is listed in iPad Spotlight Search results. It's important to note that iOS crops 1 pixel from all sides of this icon, so only the inner 48 by 48 pixels are displayed.
- *29-by-29-pixel PNG image*: This app icon is required only if your application places settings options in iOS Settings app.

Don't worry about the rounded edges and glossy beveled look that iPad app icons typically have. iOS and the App Store automatically add those elements to the icon for you. Although you can't do anything about the dynamically added rounded edges, you do have the option to disable the default beveled gloss effect from your app icon if your app icon looks better without it.

After you've added your 72-by-72-pixel icon PNG file to the Resources folder of your iPad app project in Xcode, you'll need to open your project's *plist* file. Once you've added the icon's filename to the Icon property, click the gray plus symbol (+) button on the bottom-right side of the list to add a new entry to the *plist* file. In the new, blank row, click the tiny arrows in the left Key column to display a contextual menu of additional properties. Select **Icon already includes gloss and bevel effects** from that menu. Setting its value to True will disable the default beveled gloss effect from your app icon, as shown in Figure 3–14. When displaying your 72-by-72-pixel app icon on the iPad or the 512-by-512-pixel icon in the App Store, Apple checks your app's *plist* file first, so your preferred setting is always honored.

www.it-ebooks.info



Bundle OS Type code	APPL
Bundle creator OS Type code	????
Bundle version	1.0
Application requires iPhone environment	✓
Main nib file base name	MainWindow
▶ Supported interface orientations	(4 items)
Icon already includes gloss and bevel effects	✓

Figure 3–14. *Adding a new plist property, Icon already includes gloss and bevel effects, and setting it to True will disable the default beveled gloss effect from your iPad app icon.*

Document Type Icon Images

If your iPad app utilizes the new Document Support feature and registers a custom file type with the iOS registry, you should assign a custom document icon to it, so that users can visually identify that file type as belonging to your app. Remember the Pages document icon shown in Figure 3–13? If you don't assign a custom icon to your app's file type, iOS will display your app icon inside a white document (with a top-right corner page curl). Most apps simply rely on that system default, but if you would prefer to design your own document icon, you'll need to save your custom icon in two sizes: a 64-by-64-pixel PNG image and a 320-by-320-pixel PNG image.

Do not design your icon to emulate a document with a page curl. iOS automatically adds the document border, drop shadow, and top-right corner page curl. Due to these system-supplied graphics effects, you must take special care to place your icon's main imagery within the "safe zone" area. For the 64-by-64-pixel icon, stay within 1 pixel from the top, 4 pixels from the bottom, and 10 pixels from both sides. For the 320-by-320-pixel icon, stay within 5 pixels from the top, 20 pixels from the bottom, and 50 pixels from both sides.

Default Launch Images

A launch image is briefly displayed on the screen when an app is first loading. This means the image should represent only the basic user interface elements of the app's initial window. For example, if your app's first screen shows a split view controller, then your default launch image should reflect the same controller layout, without any text or any content.

On the iPhone, your app needed to provide only a single portrait orientation image named *Default.png*. But on the iPad, Apple wants your app to support multiple orientations. Since a user could launch your iPad app in any orientation, your Xcode project will need to include multiple default launch images. Even though the iPad screen is 768 pixels by 1024 pixels, the launch images should not include the status bar, which accounts for the top 20 pixels.

The *Default* filename is still employed, but the orientation label is appended to the name with a hyphen, as follows:

www.it-ebooks.info

- *Default-Portrait.png*: This 768-by-1004-pixel image represents the first view in portrait orientation.
- *Default-PortraitUpsideDown.png*: Unless your initial portrait window is different if viewed upside down, this image is not needed. In its absence, *Default-Portrait.png* will be shown.
- *Default-Landscape.png*: This 1024-by-748-pixel image represents the first view in landscape orientation.
- *Default-LandscapeLeft.png* and *Default-LandscapeRight.png*: Unless your initial landscape window is different depending on whether the device is rotated left or right, these images are not needed. In the absence of either (or both) of these images, *Default-Landscape.png* will be shown.

If you're building a universal application, you'll need to designate unique prefixes to your launch image filenames, so that your Xcode project can properly identify which PNG files to use for the iPhone and the iPad, respectively. This is configured via the `UILaunchImageFile` key in your app's *plist* file. In order to differentiate between the two platforms, attach a device-specific value to the `UILaunchImageFile` key title, connected with a tilde character (no spaces). The `UILaunchImageFile~iphone` key's string is for the iPhone, so it would remain as *Default*. You would use a different iPad-related name for the `UILaunchImageFile~ipad` key's string, such as *iPadDefault*. In the universal application's *plist* file, the key-string syntax would look like this:

```
<key>UILaunchImageFile~iphone</key>
<string>Default</string>
<key>UILaunchImageFile~ipad</key>
<string>iPadDefault</string>
```

With these *plist* keys in place, you then want your launch image filenames to adhere to that assigned prefix. For example, your iPad launch images would be named *iPadDefault-Portrait.png*, *iPadDefault-Landscape.png*, and so on. Your iPhone launch image would remain *Default.png*.

Drilling Deeper

Don't worry if you're feeling a bit overwhelmed. We certainly covered a lot of ground in this chapter! This was merely an introduction to the vast array of new iPad classes and functions in iPhone SDK 3.2. The rest of the book covers the major iPad features in detail. You can read each chapter at your own pace, easily absorbing the step-by-step explanations on how to develop cool apps with these new features. You'll be an iPad code master in no time!

All of the code examples listed in this book, along with the full source code of the iPad drawing app, Dudel, can be downloaded from `http://www.apress.com/book/view/9781430230212`.

www.it-ebooks.info

Chapter 4

New Graphics Functionality

Starting with version 3.2, iOS includes some compelling new graphics capabilities. Besides the larger screen in the iPad, the software has also been updated with some new features that will help developers make their apps even better. This chapter covers two of these new features: the `UIBezierPath` class, which can be used to draw and fill shapes of all kinds, and the ability to render directly to PDF format—anything that you can draw on the screen, you can also send straight to a PDF file!

One bit of graphics functionality that apparently hasn't changed is OpenGL. The iPad uses the same graphics hardware, and the same OpenGL ES 2.0, that the iPhone 3GS uses. Therefore, any OpenGL code you've written for iPhone in the past should work with little or no changes on the iPad (for that matter, any Quartz/CoreGraphics drawing code should also be functionally equivalent).

Bezier Paths

One great new feature of iOS 3.2 is the inclusion of the `UIBezierPath` class, which gives you the ability to draw paths of arbitrary complexity. Anyone who has used a vector-drawing program such as Adobe Illustrator is probably familiar with the path tool, which lets you define a curve by clicking on points. You define a start and end point for the curve, along with two control points, giving you a smooth curve from one point to the other. The curve you create in this way is actually several Bézier paths linked together. However, a Bézier path can also define straight lines, rectangles, and basically any other 2D shape you have in mind. Figure 4–1 shows some examples of paths drawn using `UIBezierPath`.

www.it-ebooks.info

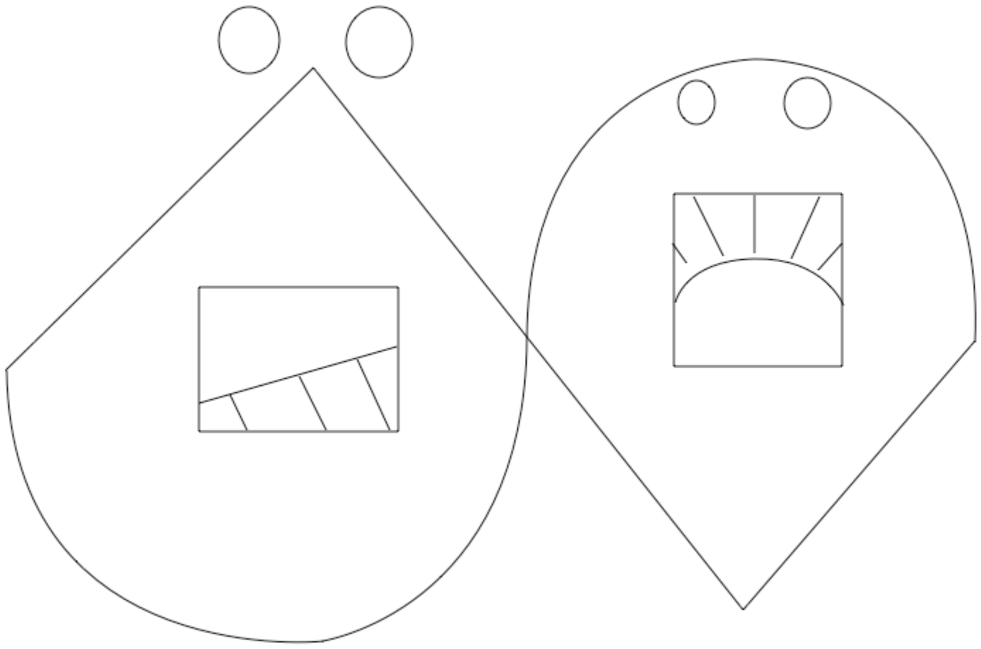


Figure 4–1. *Various paths constructed from Bézier curves*

In this chapter, we're going to build an example that demonstrates various uses of `UIBezierPath`. You'll learn how to create complex paths, define drawing characteristics for the paths you create, and use different colors to draw a path's outline (with the stroke color) and its interior (with the fill color).

Introducing Dudel

The sample iPad app we'll create will let the user create some on-screen graphics using a handful of tools, similar to what you might see in a vector-drawing application like Adobe Illustrator. We'll call it Dudel. See Figure 4–2 for a glimpse of Dudel in action.

Apart from showing the use of Bézier paths, this application will serve as the foundation for demonstrating other technologies throughout the book. We'll add one piece at a time as we work through the book, evolving and improving the app as we introduce new features. This means that in this chapter, we'll need to do a bit of project setup before we get to the actual Bézier paths. Please bear with me—it's going to be worth the wait!

www.it-ebooks.info

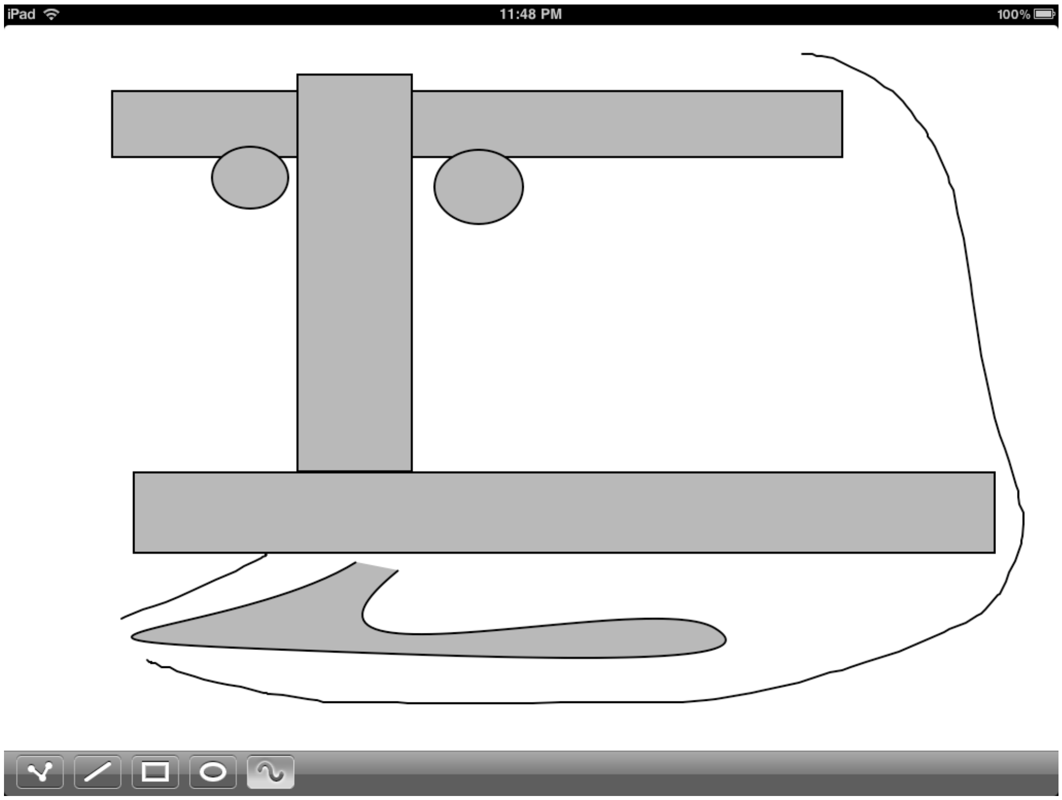


Figure 4–2. *Some Dudel action*

Creating the Dudel Project

Launch Xcode, and use the menu or Xcode's friendly startup panel to create a new project. Select iPhone OS Application in the upper left, which will bring up the familiar set of application templates in the main section. Here, you'll see a few changes compared to older SDKs. First is the addition of the new Split View-based Application template, which we'll cover in Chapter 8.

Click around to explore the various application templates. You'll see that each shows a product type in the center of the window. Some project types are only for the iPhone or only for the iPad; for those, the product type (iPhone or iPad) is displayed in a label. Other project types can apply to either platform; for those, you get a popup menu that lets you choose iPhone, iPad, or Universal (to support both). Of course, our focus is iPad apps, but in Chapter 11, you'll learn how to have your app support both the iPhone and the iPad.

For our project, pick the View-based Application template, select iPad from the product menu, and click the Choose… button. Tell the familiar save panel where you want to save this new project, and name it `Dudel`.

www.it-ebooks.info

Xcode will make a new project for you, containing *.h* and *.m* files for the `DudelAppDelegate` and `DudelViewController` classes. These contain the exact same sort of boilerplate code that you would typically find in an iPhone project. You'll also see that the project has a Resources-iPad directory, which contains the same kind of Interface Builder files you're used to seeing in iPhone projects: *MainWindow.xib* and *DudelViewController.xib*. The main difference is that these files are set up for iPad, with windows and views that are already iPad-sized.

The default *DudelViewController.xib* file contains a top-level `UIView` instance, but we're going to make our own view subclass, capable of drawing all the shapes a user creates in the app. In Xcode, add a new class to your project by right-clicking the folder where the new class should be added (the Classes folder is the classic choice) and choosing **Add ➤ New File** from the context menu. In the assistant that appears, choose Cocoa Touch Class from the iPhone section, use the pop-up menu to make it a subclass of `UIView`, and then click Next. Name the file *DudelView.m*, hit Finish, and a basic view class will be created for you.

Now, this view that does all the drawing is sure to have some complexity. Obviously, it will need to have some sort of interaction with the controller class. So let's do this the standard Cocoa way, and define a `delegate` outlet for connecting to the controller class. We'll also take the step of defining a protocol for this delegate. The protocol won't have any methods yet. We will add those after we figure out what sort of things we need to delegate! For now, make sure your *DudelView.h* looks like this (the lines in bold are the ones you need to add to the template-generated header file):

```
// DudelView.h
#import <UIKit/UIKit.h>

@protocol DudelViewDelegate
@end

@interface DudelView : UIView {
  IBOutlet id <DudelViewDelegate> delegate;
}
@end
```

That defines just enough for us to be able to hook it up in Interface Builder. Now we will continue to pull together the rest of the pieces for the nib-based portion of the app. Later on, we'll go back and implement the view itself.

The source code archive accompanying this book includes a set of buttons meant for use in Dudel. If you don't have the archive at hand, use your favorite graphics editor (I'm partial to GIMP) to create buttons similar to what you see in Table 4-1. They don't need to be pixel-perfect, but should be roughly similar so that your version of the app looks and feels about the same as mine. The button images shown here are 46 by 32, and you should try to stick to a similar size.

www.it-ebooks.info

Table 4-1. *Buttons for the Main Dudel View*

Filename	Image
button_bezier.png	∿
button_bezier_selected.png	∿
button_cdots.png	∨
button_cdots_selected.png	∨
button_ellipse.png	⬭
button_ellipse_selected.png	⬭
button_line.png	╱
button_line_selected.png	╱
button_rectangle.png	▭
button_rectangle_selected.png	▭

www.it-ebooks.info

NOTE: If you're making your own button images, keep in mind the way that `UIToolbar` renders its button images. Rather than drawing their content directly, it uses the brightness as a sort of transparency mask. White areas are completely transparent (letting the `UIToolbar` itself show through), black areas show up as a solid color that contrasts well against the `UIToolbar`'s background color (e.g., black on a light-gray background or white on a dark-gray background), and all gray values are treated somewhere in between. In the graphics supplied for this example, the "normal" images for each button are mostly completely transparent, with just a border and the contained symbol, and the "selected" images have a gradient background to make them stand out clearly.

Drag all of the button image files into the Resources-iPad folder in your Xcode project. Be sure to check the Copy items into destination group's folder check box before clicking the Add button. Once those images are in place, they'll be ready to use within your application code and nib files.

Next, we need to define the interface for our controller. For now, we just want to set up enough to allow us to hook up the few components we need in the nib file.

As you saw in Figure 4–2, Dudel will contain a row of buttons that let the user select a drawing tool. These buttons are actually instances of `UIBarButtonItem`, which will be placed on a `UIToolbar`. We'll need to have an outlet for each button so that we can control its appearance, and we'll need an action for each button to trigger, all of which will be set up in Interface Builder. We'll also create an outlet to point at a `DudelView` instance, which will be initialized when the nib file is loaded. Last but not least, we'll declare our class to conform with the `DudelViewDelegate` protocol, so that Interface Builder will let us hook it up.

Open *DudelViewController.h* and add the code shown in bold.

```
// DudelViewController.h
#import <UIKit/UIKit.h>
#import "DudelView.h"

@interface DudelViewController : UIViewController <DudelViewDelegate> {
  IBOutlet DudelView *dudelView;
  IBOutlet UIBarButtonItem *freehandButton;
  IBOutlet UIBarButtonItem *ellipseButton;
  IBOutlet UIBarButtonItem *rectangleButton;
  IBOutlet UIBarButtonItem *lineButton;
  IBOutlet UIBarButtonItem *pencilButton;
}
- (IBAction)touchFreehandItem:(id)sender;
- (IBAction)touchEllipseItem:(id)sender;
- (IBAction)touchRectangleItem:(id)sender;
- (IBAction)touchLineItem:(id)sender;
- (IBAction)touchPencilItem:(id)sender;
@end
```

www.it-ebooks.info

For the sake of having an app that we're able to build without errors at any time, let's go ahead and add some minimal implementations of those action methods to *DudelViewController.m*. They won't have any functionality yet, but their presence will let the compiler compile this class without complaint.

```
// DudelViewController.m
#import "DudelViewController.h"

@implementation DudelViewController

- (IBAction)touchFreehandItem:(id)sender {}
- (IBAction)touchEllipseItem:(id)sender {}
- (IBAction)touchRectangleItem:(id)sender {}
- (IBAction)touchLineItem:(id)sender {}
- (IBAction)touchPencilItem:(id)sender {}

// skipping the boilerplate code that's part of the template
// [...]

@end
```

We'll add more to this class later on, but this is all we need to create and hook up the GUI in Interface Builder.

Adding a Simple GUI

Now let's move to Interface Builder and construct a simple GUI. Double-click *DudelViewController.xib* to open it in Interface Builder.

First, let's make sure that the nib file uses our `DudelView` class instead of just a plain-old `UIView`. Select the `UIView` object and open the identity inspector (⌘4). At the top of the panel, click the Class combo box and change the selection from `UIView` to `DudelView`. While we're still looking at it, let's create the connections between this view and the controller. Start by control-dragging from the Dudel View icon to the File's Owner icon, which represents the `DudelViewController` that loads this nib. Release the mouse button, and in the context menu that appears, pick **delegate**. Now control-drag from the File's Owner icon back to DudelView, and select **dudelView** from the resulting context menu.

With that out of the way, we can focus on those all-important buttons at the bottom of the screen. Start by making sure you can see the DudelView layout window. If not, double-click the Dudel View icon in the main *.nib* window so that the layout window appears. If your screen isn't large enough to display the entire window, scroll down so that you can see the bottom. Now find `UIToolbar` in the Library, and drag one out to the DudelView layout window, placing it at the bottom of the window. Open the attribute inspector (⌘1) and change the toolbar's style to Black Translucent.

The toolbar you just created includes a single `UIBarButtonItem`, which we'll use as a starting point for all our toolbar buttons. Use the attribute inspector to set the button item's Style to Plain (since the images we're using have a border of their own), and then press ⌘D to duplicate the item. And duplicate it again and again and again. We want to

www.it-ebooks.info

have five of these items, and we want them all to be Plain, so this is quicker than dragging a button item out from the Library five times and setting the style each time.

These button items need to be assigned some imagery. Click the left-hand button and use the attribute inspector to set its Image to *button_cdots_selected.png*. This first button is going to be selected by default, so we'll start it off with the selected image. Now go through the remaining buttons and set their images to *button_line.png*, *button_rectangle.png*, *button_ellipse.png*, and *button_bezier.png*. These are the normal (unselected) versions of the images. You should now see something like Figure 4–3.

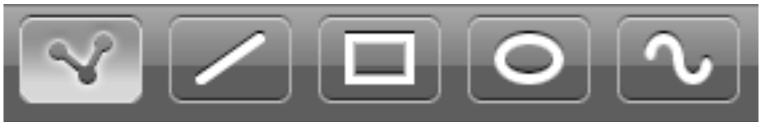


Figure 4–3. *The buttons are all in place.*

Now all that's left is to connect each of those buttons to an action method in the controller, and connect an outlet from the controller back to each button. Start with the button item on the left. Control-drag from it to the File's Owner icon, and select the `touchPencilItem:` action. Then control-drag from the File's Owner icon back to that button item, and select the `pencilButton` outlet. Go to the next button (the one with the straight line icon), control-drag from it to the File's Owner icon and select the `touchLineItem:` action, and then control-drag from File's Owner back to that button item and select the `lineButton` outlet. Do the same for the three remaining button items, so that each is set to trigger the appropriate action method in the controller, and each of the controller's outlets is connected to the correct button item.

With all this in place, you should now be able to build and run your app in the iPhone Simulator. If you get a big, blank canvas with a row of buttons at the bottom that don't do anything when you touch them, then you've achieved the goal for this portion of our project. Congratulations!

The Basic Drawing Architecture

Now it's time to talk about how Dudel is going to draw. Instead of dealing with an underlying pixel-buffer canvas on which all operations are performed, we're going to maintain a list of drawing operations, defined by the user's actions. Each object that the user draws will be added to an array, and each of the objects in the array will be drawn when necessary.

For maximum modularity, let's decide that each drawing operation should know how to draw itself. Then all our `DudelView` class really needs to do is hang onto an array and pass along a draw request whenever it's time to redraw. We'll try to maintain some sort of order here by defining a protocol called `Drawable`, which contains a single method called `draw`. Any object that represents a drawing operation should conform to this protocol. Also, there will be times—such as while the user is creating a drawing operation by dragging a finger around the screen—that some temporary drawing will need to be done. The view class won't be responsible for this, however. We'll pass that

www.it-ebooks.info

to our delegate object, the DudelViewController. So, we'll add a method to the DudelViewDelegate method, giving the controller a chance to do some temporary, context-based drawing.

Create a new protocol header file (one of the file types found in the Cocoa Touch Class section of the New File Assistant) in your project, name it *Drawable.h*, and give it the following content:

```
//  Drawable.h

@protocol Drawable
- (void)draw;
@end
```

Now we're ready to flesh out the DudelView class. First, add a few lines to *DudelView.h*:

```
//  DudelView.h
#import <UIKit/UIKit.h>

@protocol DudelViewDelegate
- (void)drawTemporary;
@end

@interface DudelView : UIView {
  NSMutableArray *drawables;
  IBOutlet id <DudelViewDelegate> delegate;
}
@property (retain, nonatomic) NSMutableArray *drawables;
@end
```

Then define the *DudelView.m* file as follows:

```
// DudelView.m
#import "DudelView.h"
#import "Drawable.h"

@implementation DudelView
@synthesize drawables;
- (id)initWithFrame:(CGRect)frame {
  if ((self = [super initWithFrame:frame])) {
    drawables = [[NSMutableArray alloc] initWithCapacity:100];
  }
  return self;
}
- (id)initWithCoder:(NSCoder *)aDecoder {
  if ((self = [super initWithCoder:aDecoder])) {
    drawables = [[NSMutableArray alloc] initWithCapacity:100];
  }
  return self;
}
- (void)drawRect:(CGRect)rect {
  for (<Drawable> d in drawables) {
    [d draw];
  }
  [delegate drawTemporary];
}
```

www.it-ebooks.info

```
- (void)dealloc {
  [drawables release];
  [super dealloc];
}
@end
```

You'll notice that we define two different initialization methods. The first is normally called within code; the second is called when an object is being instantiated from a nib file. In our current implementation, only the nib file version is being used, but we may as well cover the other possibility as well.

Apart from that, this code is quite straightforward. Other objects can directly add drawable display operations to the view's array. Whenever `drawRect:` is called (as a result of someone, somewhere, calling `setNeedsDisplay` on the view), the view just calls `draw` everywhere else.

We Are All Tool Users

At this point, it's time to deal with an interesting aspect of our application's architecture: the representation and use of tools corresponding to the buttons we put in the GUI. We'll define a number of tool classes, each conforming to a particular protocol so that our controller can talk to them all. Our controller will keep a pointer to the active tool, based on the user's selection, and will pass all touch events along to the active tool, which will deal with the events in whatever way is appropriate for it. It will also pass along the `drawTemporary` message from the view, so that the tool can draw a representation of the work in progress if it's a multistage creation action. Our controller doesn't need to know any specifics about how a tool interprets events, defines drawing operations, or anything else.

In addition, `DudelViewController` will maintain state information about other potential user-selected values, such as the current fill color (used to fill the inside of the shape that's being drawn) and the current stroke color (used to draw the edge of the shape). That information will be available for the active tool to access when it's doing its temporary drawing and when it's creating a completed drawing operation to give to the view. For now, we're not going to provide any mechanism for setting the fill and stroke colors. We'll just give them predefined values. Later, in Chapter 6, we'll demonstrate a nice way to let the user specify these colors using the latest version of the SDK.

The final piece of functionality that `DudelView` will include is responding to presses of the `UIBarButtonItems` in the toolbar, which will result in a new tool being set as the active tool. To get started, add a few lines to *DudelViewController.h*:

```
//  DudelViewController.h
#import <UIKit/UIKit.h>
#import "Tool.h"
#import "DudelView.h"

@interface DudelViewController : UIViewController <ToolDelegate, DudelViewDelegate> {
  id <Tool> currentTool;
  IBOutlet DudelView *dudelView;
```

www.it-ebooks.info

```
  IBOutlet UIBarButtonItem *freehandButton;
  IBOutlet UIBarButtonItem *ellipseButton;
  IBOutlet UIBarButtonItem *rectangleButton;
  IBOutlet UIBarButtonItem *lineButton;
  IBOutlet UIBarButtonItem *pencilButton;
  UIColor *strokeColor;
  UIColor *fillColor;
  CGFloat strokeWidth;
}

@property (retain, nonatomic) id <Tool> currentTool;
@property (retain, nonatomic) UIColor *strokeColor;
@property (retain, nonatomic) UIColor *fillColor;
@property (assign, nonatomic) CGFloat strokeWidth;
- (IBAction)touchFreehandItem:(id)sender;
- (IBAction)touchEllipseItem:(id)sender;
- (IBAction)touchRectangleItem:(id)sender;
- (IBAction)touchLineItem:(id)sender;
- (IBAction)touchPencilItem:(id)sender;
@end
```

Next, move to *DudelViewController.m*. This is where the real changes take place. Among other things, this file contains implementations for all the action methods we defined in the header file. These are all empty for now, but will be filled in later as we cover each new tool. It also defines a few utility methods for internal use, for taking care of some repetitive tasks that we'll need to do for each of those action methods.

```
//  DudelViewController.m
#import "DudelViewController.h"

#import "DudelView.h"

@implementation DudelViewController

@synthesize currentTool, fillColor, strokeColor, strokeWidth;

- (void)deselectAllToolButtons {
  [textButton setImage:[UIImage imageNamed:@"button_text.png"]];
  [freehandButton setImage:[UIImage imageNamed:@"button_bezier.png"]];
  [ellipseButton setImage:[UIImage imageNamed:@"button_ellipse.png"]];
  [rectangleButton setImage:[UIImage imageNamed:@"button_rectangle.png"]];
  [lineButton setImage:[UIImage imageNamed:@"button_line.png"]];
  [pencilButton setImage:[UIImage imageNamed:@"button_cdots.png"]];
}
- (void)setCurrentTool:(id <Tool>)t {
  [currentTool deactivate];
  if (t != currentTool) {
    [currentTool release];
    currentTool = [t retain];
    currentTool.delegate = self;
    [self deselectAllToolButtons];
  }
  [currentTool activate];
  [dudelView setNeedsDisplay];
}
```

www.it-ebooks.info

```
- (IBAction)touchFreehandItem:(id)sender {
}
- (IBAction)touchEllipseItem:(id)sender {
}
- (IBAction)touchRectangleItem:(id)sender {
}
- (IBAction)touchLineItem:(id)sender {
}
- (IBAction)touchPencilItem:(id)sender {
}

- (void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event {
  [currentTool touchesBegan:touches withEvent:event];
  [dudelView setNeedsDisplay];
}
- (void)touchesCancelled:(NSSet *)touches withEvent:(UIEvent *)event {
  [currentTool touchesCancelled:touches withEvent:event];
  [dudelView setNeedsDisplay];
}
- (void)touchesEnded:(NSSet *)touches withEvent:(UIEvent *)event {
  [currentTool touchesEnded:touches withEvent:event];
  [dudelView setNeedsDisplay];
}
- (void)touchesMoved:(NSSet *)touches withEvent:(UIEvent *)event {
  [currentTool touchesMoved:touches withEvent:event];
  [dudelView setNeedsDisplay];
}
- (void)addDrawable:(id <Drawable>)d {
  [dudelView.drawables addObject:d];
  [dudelView setNeedsDisplay];
}
- (UIView *)viewForUseWithTool:(id <Tool>)t {
  return self.view;
}
- (void)drawTemporary {
  [self.currentTool drawTemporary];
}
- (void)viewDidLoad {
  [super viewDidLoad];
  self.fillColor = [UIColor lightGrayColor];
  self.strokeColor = [UIColor blackColor];
  self.strokeWidth = 2.0;
}
// Override to allow orientations other than the default portrait orientation.
- (BOOL)shouldAutorotateToInterfaceOrientation:(UIInterfaceOrientation)orientation {
    return YES;
}
- (void)dealloc {
  self.currentTool = nil;
  self.fillColor = nil;
  self.strokeColor = nil;
  [super dealloc];
}
@end
```

www.it-ebooks.info

That code also refers to a file called *Tool.h*, and the Tool protocol it defines. That's the protocol with which the controller communicates with the selected tool. Create a new protocol header file in your project, name it *Tool.h*, and fill it with this:

```
// Tool.h
#import <UIKit/UIKit.h>

@protocol ToolDelegate;
@protocol Drawable;

@protocol Tool <NSObject>

@property (assign, nonatomic) id <ToolDelegate> delegate;
- (void)activate;
- (void)deactivate;

- (void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event;
- (void)touchesCancelled:(NSSet *)touches withEvent:(UIEvent *)event;
- (void)touchesEnded:(NSSet *)touches withEvent:(UIEvent *)event;
- (void)touchesMoved:(NSSet *)touches withEvent:(UIEvent *)event;

- (void)drawTemporary;
@end

@protocol ToolDelegate

- (void)addDrawable:(id <Drawable>)d;
- (UIView *)viewForUseWithTool:(id <Tool>)t;
- (UIColor *)strokeColor;
- (UIColor *)fillColor;

@end
```

This also defines the ToolDelegate protocol, with which each tool can communicate back to the controller.

At this point, you should be able to build and run your app. You'll still wind up with a blank slate that doesn't do anything, but doing so will at least verify that you're on track.

Next, let's tackle the tools.

The Pencil Tool

The Pencil tool is the simplest one we're going to create. It will place a small dot wherever you tap the screen, or a continuous squiggly line if you continue to drag your finger around.

Make a new NSObject subclass named PencilTool and give it the following content:

```
//  PencilTool.h
#import <Foundation/Foundation.h>
#import "Tool.h"
@interface PencilTool : NSObject <Tool> {
  id <ToolDelegate> delegate;
```

www.it-ebooks.info

```
  NSMutableArray *trackingTouches;
  NSMutableArray *startPoints;
  NSMutableArray *paths;
 }
+ (PencilTool *)sharedPencilTool;
@end

//  PencilTool.m
#import "PencilTool.h"
#import "PathDrawingInfo.h"
#import "SynthesizeSingleton.h"
@implementation PencilTool
@synthesize delegate;
SYNTHESIZE_SINGLETON_FOR_CLASS(PencilTool);
- init {
  if ((self = [super init])) {
    trackingTouches = [[NSMutableArray array] retain];
    startPoints = [[NSMutableArray array] retain];
    paths = [[NSMutableArray array] retain];
  }
  return self;
}
- (void)activate {
}
- (void)deactivate {
  [trackingTouches removeAllObjects];
  [startPoints removeAllObjects];
  [paths removeAllObjects];
}
- (void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event {
  UIView *touchedView = [delegate viewForUseWithTool:self];
  for (UITouch *touch in [event allTouches]) {
    // remember the touch, and its original start point, for future
    [trackingTouches addObject:touch];
    CGPoint location = [touch locationInView:touchedView];
    [startPoints addObject:[NSValue valueWithCGPoint:location]];
    UIBezierPath *path = [UIBezierPath bezierPath];
    path.lineCapStyle = kCGLineCapRound;
    [path moveToPoint:location];
    [path setLineWidth:delegate.strokeWidth];
    [path addLineToPoint:location];
    [paths addObject:path];
  }
}
- (void)touchesCancelled:(NSSet *)touches withEvent:(UIEvent *)event {
  [self deactivate];
}
- (void)touchesEnded:(NSSet *)touches withEvent:(UIEvent *)event {
  for (UITouch *touch in [event allTouches]) {
    // make a line from the start point to the current point
    NSUInteger touchIndex = [trackingTouches indexOfObject:touch];
    // only if we actually remember the start of this touch...
    if (touchIndex != NSNotFound) {
      UIBezierPath *path = [paths objectAtIndex:touchIndex];
      PathDrawingInfo *info = [PathDrawingInfo pathDrawingInfoWithPath:path
        fillColor:[UIColor clearColor] strokeColor:delegate.strokeColor];
      [delegate addDrawable:info];
```

www.it-ebooks.info

```
      [trackingTouches removeObjectAtIndex:touchIndex];
      [startPoints removeObjectAtIndex:touchIndex];
      [paths removeObjectAtIndex:touchIndex];
    }
  }
}
- (void)touchesMoved:(NSSet *)touches withEvent:(UIEvent *)event {
  UIView *touchedView = [delegate viewForUseWithTool:self];
  for (UITouch *touch in [event allTouches]) {
    // make a line from the start point to the current point
    NSUInteger touchIndex = [trackingTouches indexOfObject:touch];
    // only if we actually remember the start of this touch...
    if (touchIndex != NSNotFound) {
      CGPoint location = [touch locationInView:touchedView];
      UIBezierPath *path = [paths objectAtIndex:touchIndex];
      [path addLineToPoint:location];
    }
  }
}
- (void)drawTemporary {
  for (UIBezierPath *path in paths) {
    [delegate.strokeColor setStroke];
    [path stroke];
  }
}
- (void)dealloc {
  [trackingTouches release];
  [startPoints release];
  [paths release];
  self.delegate = nil;
  [super dealloc];
}
@end
```

The interesting parts of this code are all contained in the various “touches” methods, which look at all the current touches (yes, this will work with multitouch just fine). In each case, these methods create or modify a Bézier path, or prepare a completed path, using the fill color from the delegate to create a new instance of a Drawable object called PathDrawingInfo, and pass that to the controller as a complete drawing operation, ready to be added to the DudelView’s stack of Drawable objects.

PathDrawingInfo is a simple class that conforms to the Drawable protocol we defined earlier. It encapsulates a UIBezierPath and two UIColor values for the stroke and fill. Add a new class called PathDrawingInfo to your project, and give it the following content:

```
// PathDrawingInfo.h
#import <Foundation/Foundation.h>
#import "Drawable.h"
@interface PathDrawingInfo : NSObject <Drawable> {
  UIBezierPath *path;
  UIColor *fillColor;
  UIColor *strokeColor;
}
@property (retain, nonatomic) UIBezierPath *path;
@property (retain, nonatomic) UIColor *fillColor;
```

www.it-ebooks.info

```
@property (retain, nonatomic) UIColor *strokeColor;
- (id)initWithPath:(UIBezierPath *)p fillColor:(UIColor *)f strokeColor:(UIColor *)s;
+ (id)pathDrawingInfoWithPath:(UIBezierPath *)p fillColor:(UIColor *)f
strokeColor:(UIColor *)s;
@end

//  PathDrawingInfo.m
#import "PathDrawingInfo.h"
@implementation PathDrawingInfo
@synthesize path, fillColor, strokeColor;
- (id)initWithPath:(UIBezierPath *)p fillColor:(UIColor *)f strokeColor:(UIColor *)s {
  if ((self = [self init])) {
    path = [p retain];
    fillColor = [f retain];
    strokeColor = [s retain];
  }
  return self;
}
+ (id)pathDrawingInfoWithPath:(UIBezierPath *)p fillColor:(UIColor *)f
strokeColor:(UIColor *)s {
  return [[[self alloc] initWithPath:p fillColor:f strokeColor:s] autorelease];
}
- (void)dealloc {
  self.path = nil;
  self.fillColor = nil;
  self.strokeColor = nil;
  [super dealloc];
}
- (void)draw {
  CGContextRef context = UIGraphicsGetCurrentContext();
  CGContextSaveGState(context);
  if (self.fillColor) {
    [self.fillColor setFill];
    [self.path fill];
  }
  if (self.strokeColor) {
    [self.strokeColor setStroke];
    [self.path stroke];
  }
  CGContextRestoreGState(context);
}
@end
```

The `PencilTool` class, and all the rest of the `Tool` classes, also makes use of the `SYNTHESIZE_SINGLETON_FOR_CLASS` macro. This chunk of code, which comes from Matt Gallagher's Cocoa with Love blog, provides a standardized way to make any class into a singleton. This is perfect for Dudel's tools, since we never need more than one of each kind. This macro overrides all of the methods that deal with memory management, making sure that only one instance of this class is ever created. To use it, add a new header file called *SynthesizeSingleton.h* to your project, with the following content:

```
//
//  SynthesizeSingleton.h
//  CocoaWithLove
//
//  Created by Matt Gallagher on 20/10/08.
```

www.it-ebooks.info

```
//  Copyright 2009 Matt Gallagher. All rights reserved.
//
//  Permission is given to use this source code file without charge in any
//  project, commercial or otherwise, entirely at your risk, with the condition
//  that any redistribution (in part or whole) of source code must retain
//  this copyright and permission notice. Attribution in compiled projects is
//  appreciated but not required.
//

#define SYNTHESIZE_SINGLETON_FOR_CLASS(classname) \
 \
static classname *shared##classname = nil; \
 \
+ (classname *)shared##classname \
{ \
        @synchronized(self) \
        { \
                if (shared##classname == nil) \
                { \
                        shared##classname = [[self alloc] init]; \
                } \
        } \
         \
        return shared##classname; \
} \
 \
+ (id)allocWithZone:(NSZone *)zone \
{ \
        @synchronized(self) \
        { \
                if (shared##classname == nil) \
                { \
                        shared##classname = [super allocWithZone:zone]; \
                        return shared##classname; \
                } \
        } \
         \
        return nil; \
} \
 \
- (id)copyWithZone:(NSZone *)zone \
{ \
        return self; \
} \
 \
- (id)retain \
{ \
        return self; \
} \
 \
- (NSUInteger)retainCount \
{ \
        return NSUIntegerMax; \
} \
 \
- (void)release \
{ \
```

www.it-ebooks.info

```
} \
 \
- (id)autorelease \
{ \
        return self; \
}
```

NOTE: This sort of macro definition is inherently sort of tricky to type in on your own, since the backslash character on each line must come right at the end of the line, without any trailing spaces. Your best bet is to copy this file from the book's source code archive.

You're now very close to being able to try things out! At this point, all that's left to enable the Pencil tool is the addition of a few lines in *DudelViewController.m*. Start by importing the header:

```
#import "PencilTool.h"
```

Then populate the touchPencilItem: method as follows:

```
- (IBAction)touchPencilItem:(id)sender {
  self.currentTool = [PencilTool sharedPencilTool];
  [pencilButton setImage:[UIImage imageNamed:@"button_cdots_selected.png"]];
}
```

Finally, to arrange that this tool is selected by default when the app starts, add a line to the viewDidLoad method:

```
- (void)viewDidLoad {
  [super viewDidLoad];
  self.currentTool = [PencilTool sharedPencilTool];
  self.fillColor = [UIColor lightGrayColor];
  self.strokeColor = [UIColor blackColor];
  self.strokeWidth = 2.0;
}
```

You should now be able to build and run your app. Try tapping and dragging all over the screen with the Pencil tool, making dots and squiggles. Figure 4–4 shows an example of a drawing made with our new tool.

www.it-ebooks.info

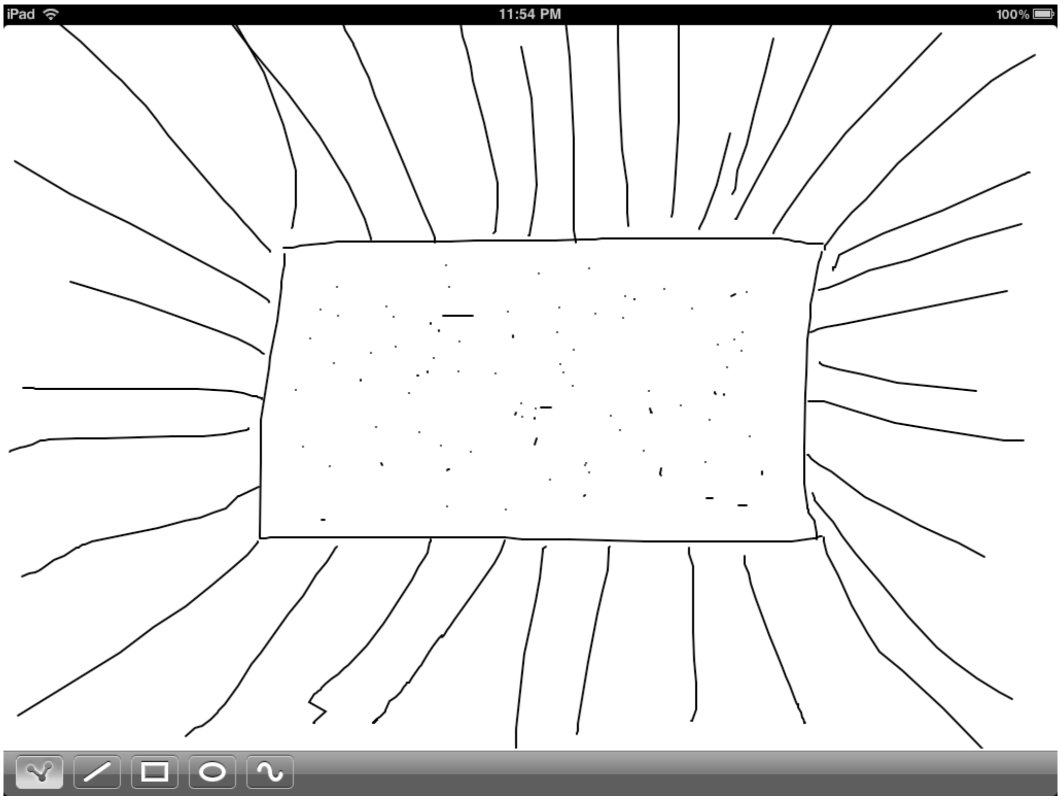


Figure 4–4. *My god, it's full of dots!*

The Line Tool

The Line tool works by letting you touch the screen in one spot and drag to another, creating a line between the two spots when you let go. While you're dragging, a temporary line is drawn between the two points.

Make a new `NSObject` subclass named `LineTool` in your Xcode project. Both the header and implementation files are shown here:

```
//  LineTool.h
#import <Foundation/Foundation.h>
#import "Tool.h"
@interface LineTool : NSObject <Tool> {
  id <ToolDelegate> delegate;
  NSMutableArray *trackingTouches;
  NSMutableArray *startPoints;
}
+ (LineTool *)sharedLineTool;
@end

//  LineTool.m
#import "LineTool.h"
```

www.it-ebooks.info

```
#import "PathDrawingInfo.h"
#import "SynthesizeSingleton.h"

@implementation LineTool
@synthesize delegate;
SYNTHESIZE_SINGLETON_FOR_CLASS(LineTool);
- init {
  if ((self = [super init])) {
    trackingTouches = [[NSMutableArray array] retain];
    startPoints = [[NSMutableArray array] retain];
  }
  return self;
}
- (void)activate {
}
- (void)deactivate {
  [trackingTouches removeAllObjects];
  [startPoints removeAllObjects];
}
- (void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event {
  UIView *touchedView = [delegate viewForUseWithTool:self];
  for (UITouch *touch in [event allTouches]) {
    // remember the touch, and its original start point, for future
    [trackingTouches addObject:touch];
    CGPoint location = [touch locationInView:touchedView];
    [startPoints addObject:[NSValue valueWithCGPoint:location]];
  }
}
- (void)touchesCancelled:(NSSet *)touches withEvent:(UIEvent *)event {}
- (void)touchesEnded:(NSSet *)touches withEvent:(UIEvent *)event {
  UIView *touchedView = [delegate viewForUseWithTool:self];
  for (UITouch *touch in [event allTouches]) {
    // make a line from the start point to the current point
    NSUInteger touchIndex = [trackingTouches indexOfObject:touch];
    // only if we actually remember the start of this touch...
    if (touchIndex != NSNotFound) {
      CGPoint startPoint = [[startPoints objectAtIndex:touchIndex] CGPointValue];
      CGPoint endPoint = [touch locationInView:touchedView];
      UIBezierPath *path = [UIBezierPath bezierPath];
      [path moveToPoint:startPoint];
      [path addLineToPoint:endPoint];
      PathDrawingInfo *info = [PathDrawingInfo pathDrawingInfoWithPath:path
fillColor:delegate.fillColor strokeColor:delegate.strokeColor];
      [delegate addDrawable:info];
      [trackingTouches removeObjectAtIndex:touchIndex];
      [startPoints removeObjectAtIndex:touchIndex];
    }
  }
}
- (void)touchesMoved:(NSSet *)touches withEvent:(UIEvent *)event {}
- (void)drawTemporary {
  UIView *touchedView = [delegate viewForUseWithTool:self];
  for (int i = 0; i<[trackingTouches count]; i++) {
    UITouch *touch = [trackingTouches objectAtIndex:i];
    CGPoint startPoint = [[startPoints objectAtIndex:i] CGPointValue];
    CGPoint endPoint = [touch locationInView:touchedView];
    UIBezierPath *path = [UIBezierPath bezierPath];
```

www.it-ebooks.info

```
    [path moveToPoint:startPoint];
    [path addLineToPoint:endPoint];
    [delegate.strokeColor setStroke];
    [path stroke];
  }
}
- (void)dealloc {
  [trackingTouches release];
  [startPoints release];
  self.delegate = nil;
  [super dealloc];
}
@end
```

This is pretty similar to the Pencil tool. We keep track of all the current touches, as well as the start point for each of them, so that we can make a proper line path for each line segment. We use the `touchesBegan:withEvent:` method to save a reference to each touch we're tracking, as well as each start point. Then in `touchesEnded:withEvent:`, we create a new path object and send it to the delegate.

Now we need to add a bit to `DudelViewController`, so that it knows about this class and can work with it. Start off with another `import` line, somewhere near the top:

```
#import "LineTool.h"
```

Then fill in the `touchLineItem:` action method, like this:

```
- (IBAction)touchLineItem:(id)sender {
  self.currentTool = [LineTool sharedLineTool];
  [lineButton setImage:[UIImage imageNamed:@"button_line_selected.png"]];
}
```

Again, build and run your app, and try out our new tool. One nice feature of this implementation is that since you're tracking multiple touch points, you can touch and drag with several fingers at once, dragging out lines behind each of them. If you're running on the simulator instead of an actual iPad, you can test this a little by holding down the Option key while you're clicking, which simulates an additional click on the other side of the screen. This is mainly in place to help simulate twist and pinch gestures, but we can use it here as well. Figure 4–5 shows some lines.

www.it-ebooks.info

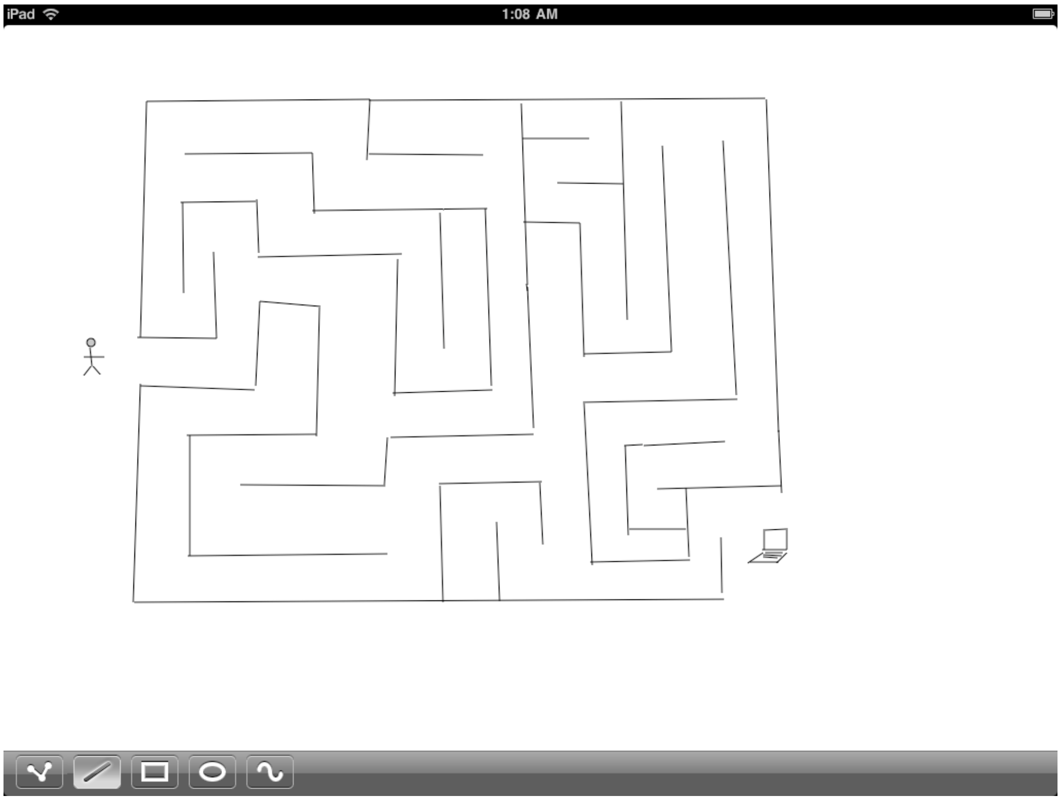


Figure 4–5. *The tyranny of straight lines is keeping this poor fellow away from his beloved MacBook Pro.*

The Ellipse and Rectangle Tools

Next up are the Ellipse and Rectangle tools. They are extremely similar to one another, and also to the Line tool. From a user standpoint, they function similarly: you touch in one corner, drag, and release to define the opposite corner. The Rectangle tool creates a rectangle, and the Ellipse tool creates (you guessed it) an ellipse.

Make a new `RectangleTool` class, and give it this code:

```
//  RectangleTool.h
#import <Foundation/Foundation.h>
#import "Tool.h"
@interface RectangleTool : NSObject <Tool> {
  id <ToolDelegate> delegate;
  NSMutableArray *trackingTouches;
  NSMutableArray *startPoints;
}
+ (RectangleTool *)sharedRectangleTool;
@end

//  RectangleTool.m
#import "RectangleTool.h"
```

www.it-ebooks.info

```
#import "PathDrawingInfo.h"
#import "SynthesizeSingleton.h"
@implementation RectangleTool
@synthesize delegate;
SYNTHESIZE_SINGLETON_FOR_CLASS(RectangleTool);
- init {
  if ((self = [super init])) {
    trackingTouches = [[NSMutableArray array] retain];
    startPoints = [[NSMutableArray array] retain];
  }
  return self;
}
- (void)activate {
}
- (void)deactivate {
  [trackingTouches removeAllObjects];
  [startPoints removeAllObjects];
}
```

As you can see, like the `LineTool` class, this class maintains arrays of `startingPoints` and `trackingTouches`.

The "touches" methods are where the interesting work of this class is done. Like the Line tool, the Rectangle tool is capable of tracking multiple simultaneous touches, ultimately creating a new line for each of them.

```
- (void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event {
  UIView *touchedView = [delegate viewForUseWithTool:self];
  for (UITouch *touch in [event allTouches]) {
    // remember the touch, and its original start point, for future
    [trackingTouches addObject:touch];
    CGPoint location = [touch locationInView:touchedView];
    [startPoints addObject:[NSValue valueWithCGPoint:location]];
  }
}
- (void)touchesCancelled:(NSSet *)touches withEvent:(UIEvent *)event {
}
- (void)touchesEnded:(NSSet *)touches withEvent:(UIEvent *)event {
  UIView *touchedView = [delegate viewForUseWithTool:self];
  for (UITouch *touch in [event allTouches]) {
    // make a rect from the start point to the current point
    NSUInteger touchIndex = [trackingTouches indexOfObject:touch];
    // only if we actually remember the start of this touch...
    if (touchIndex != NSNotFound) {
      CGPoint startPoint = [[startPoints objectAtIndex:touchIndex] CGPointValue];
      CGPoint endPoint = [touch locationInView:touchedView];
      CGRect rect = CGRectMake(startPoint.x, startPoint.y, endPoint.x - startPoint.x,
endPoint.y - startPoint.y);
      UIBezierPath *path = [UIBezierPath bezierPathWithRect:rect];
      PathDrawingInfo *info = [PathDrawingInfo pathDrawingInfoWithPath:path
fillColor:delegate.fillColor strokeColor:delegate.strokeColor];
      [delegate addDrawable:info];
      [trackingTouches removeObjectAtIndex:touchIndex];
      [startPoints removeObjectAtIndex:touchIndex];
    }
  }
}
```

www.it-ebooks.info

```
- (void)touchesMoved:(NSSet *)touches withEvent:(UIEvent *)event {
}
```

The following method draws the current state of the rectangle while you are still dragging it around. Only later does the object being drawn here get added to the view's list of drawable items.

```
- (void)drawTemporary {
  UIView *touchedView = [delegate viewForUseWithTool:self];
  for (int i = 0; i<[trackingTouches count]; i++) {
    UITouch *touch = [trackingTouches objectAtIndex:i];
    CGPoint startPoint = [[startPoints objectAtIndex:i] CGPointValue];
    CGPoint endPoint = [touch locationInView:touchedView];
    CGRect rect = CGRectMake(startPoint.x, startPoint.y, endPoint.x - startPoint.x,
endPoint.y - startPoint.y);
    UIBezierPath *path = [UIBezierPath bezierPathWithRect:rect];
    [delegate.fillColor setFill];
    [path fill];
    [delegate.strokeColor setStroke];
    [path stroke];
  }
}
- (void)dealloc {
  [trackingTouches release];
  [startPoints release];
  self.delegate = nil;
  [super dealloc];
}
@end
```

Now for the Ellipse tool. Its only substantial difference from the Rectangle tool is the creation of `UIBezierPaths` in `touchesEnded:withEvent:` and `drawTemporary`.

```
//  EllipseTool.h
#import <Foundation/Foundation.h>
#import "Tool.h"
@interface EllipseTool : NSObject <Tool> {
  id <ToolDelegate> delegate;
  NSMutableArray *trackingTouches;
  NSMutableArray *startPoints;
}
+ (EllipseTool *)sharedEllipseTool;
@end


//  EllipseTool.m
#import "EllipseTool.h"
#import "PathDrawingInfo.h"
#import "SynthesizeSingleton.h"
@implementation EllipseTool
@synthesize delegate;
SYNTHESIZE_SINGLETON_FOR_CLASS(EllipseTool);
- init {
  if ((self = [super init])) {
    trackingTouches = [[NSMutableArray arrayWithCapacity:100] retain];
    startPoints = [[NSMutableArray arrayWithCapacity:100] retain];
  }
  return self;
```

www.it-ebooks.info

```
}
- (void)activate {
}
- (void)deactivate {
  [trackingTouches removeAllObjects];
  [startPoints removeAllObjects];
}
- (void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event {
  UIView *touchedView = [delegate viewForUseWithTool:self];
  for (UITouch *touch in [event allTouches]) {
    // remember the touch, and its original start point, for future
    [trackingTouches addObject:touch];
    CGPoint location = [touch locationInView:touchedView];
    [startPoints addObject:[NSValue valueWithCGPoint:location]];
  }
}
- (void)touchesCancelled:(NSSet *)touches withEvent:(UIEvent *)event {
}
- (void)touchesEnded:(NSSet *)touches withEvent:(UIEvent *)event {
  UIView *touchedView = [delegate viewForUseWithTool:self];
  for (UITouch *touch in [event allTouches]) {
    // make an ellipse/oval from the start point to the current point
    NSUInteger touchIndex = [trackingTouches indexOfObject:touch];
    // only if we actually remember the start of this touch...
    if (touchIndex != NSNotFound) {
      CGPoint startPoint = [[startPoints objectAtIndex:touchIndex] CGPointValue];
      CGPoint endPoint = [touch locationInView:touchedView];
      CGRect rect = CGRectMake(startPoint.x, startPoint.y, endPoint.x - startPoint.x,
endPoint.y - startPoint.y);
      UIBezierPath *path = [UIBezierPath bezierPathWithOvalInRect:rect];
      PathDrawingInfo *info = [PathDrawingInfo pathDrawingInfoWithPath:path
fillColor:delegate.fillColor strokeColor:delegate.strokeColor];
      [delegate addDrawable:info];
      [trackingTouches removeObjectAtIndex:touchIndex];
      [startPoints removeObjectAtIndex:touchIndex];
    }
  }
}
- (void)touchesMoved:(NSSet *)touches withEvent:(UIEvent *)event {
}
- (void)drawTemporary {
  UIView *touchedView = [delegate viewForUseWithTool:self];
  for (int i = 0; i<[trackingTouches count]; i++) {
    UITouch *touch = [trackingTouches objectAtIndex:i];
    CGPoint startPoint = [[startPoints objectAtIndex:i] CGPointValue];
    CGPoint endPoint = [touch locationInView:touchedView];
    CGRect rect = CGRectMake(startPoint.x, startPoint.y, endPoint.x - startPoint.x,
endPoint.y - startPoint.y);
    UIBezierPath *path = [UIBezierPath bezierPathWithOvalInRect:rect];
    [delegate.fillColor setFill];
    [path fill];
    [delegate.strokeColor setStroke];
    [path stroke];
  }
}
- (void)dealloc {
  [trackingTouches release];
```

www.it-ebooks.info

```
  [startPoints release];
  self.delegate = nil;
  [super dealloc];
}
@end
```

Here are the necessary changes to *DudelViewController.m*:

```
#import "RectangleTool.h"
#import "EllipseTool.h"

- (IBAction)touchEllipseItem:(id)sender {
  self.currentTool = [EllipseTool sharedEllipseTool];
  [ellipseButton setImage:[UIImage imageNamed:@"button_ellipse_selected.png"]];
}
- (IBAction)touchRectangleItem:(id)sender {
  self.currentTool = [RectangleTool sharedRectangleTool];
  [rectangleButton setImage:[UIImage imageNamed:@"button_rectangle_selected.png"]];
}
```

With those in place, the next two buttons at the bottom of the GUI should now be working. Figure 4–6 shows some of the kinds of shapes that can be created with these tools. As with the previous tools, these also work with multitouch, so you should be able to drag multiple fingers at once to create several rectangles or ellipses simultaneously.

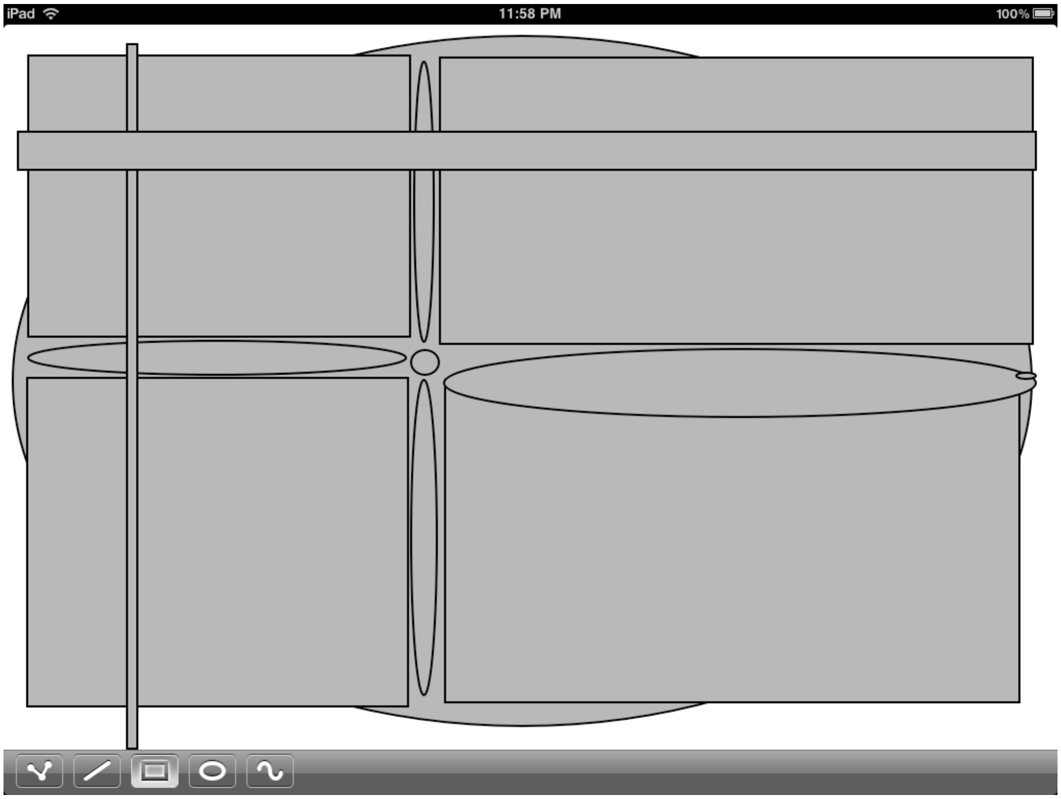


Figure 4–6. *Overlapping blocks and curves*

www.it-ebooks.info

The Freehand Tool

Our final tool, the Freehand tool, lets you create a Bézier path. With this tool, you can draw a big string of curved sections by tapping and dragging. Each touch defines a new point on the curve, and dragging immediately after the touch lets you define control points that determine the curvature around that point. While you're dragging a control point around, it's shown with a dashed red line connecting it to the last point you touched, just to make it stand out a little more. To finish off a path, touch the Freehand button in the toolbar (or any other tool button, for that matter). This Freehand tool corresponds to what most people think of as Bézier curves (if they're thinking of Bézier curves at all).

Due to the additional complexity of the interaction with the Freehand tool, we're not going to consider the use of multitouch here. The Freehand tool relies on making a series of points by touching and releasing multiple times, and if we were to track multiple touches, it would be impossible to guess which subsequent touch belonged with which previous touch. Instead, we have a different set of instance variables that are used to hold the state of the current in-progress curve segment, if there is one. As each curve segment is created, it's added to a `workingPath` object, which is finally sent to the delegate when it's done.

Create a new class called `FreehandTool`, with the following code:

```
//  FreehandTool.h
#import <Foundation/Foundation.h>
#import "Tool.h"
@interface FreehandTool : NSObject <Tool> {
  id <ToolDelegate> delegate;
  UIBezierPath *workingPath;
  CGPoint nextSegmentPoint1;
  CGPoint nextSegmentPoint2;
  CGPoint nextSegmentCp1;
  CGPoint nextSegmentCp2;
  BOOL isDragging;
  BOOL settingFirstPoint;
}
@property (retain, nonatomic) UIBezierPath *workingPath;
+ (FreehandTool *)sharedFreehandTool;
@end

//  FreehandTool.m
#import "FreehandTool.h"
#import "PathDrawingInfo.h"
#import "SynthesizeSingleton.h"
@implementation FreehandTool
@synthesize delegate, workingPath;
SYNTHESIZE_SINGLETON_FOR_CLASS(FreehandTool);
- init {
  if ((self = [super init])) {
  }
  return self;
}
- (void)activate {
  self.workingPath = [UIBezierPath bezierPath];
```

www.it-ebooks.info

```
  settingFirstPoint = YES;
}
- (void)deactivate {
  // this is where we finally tell about our path
  PathDrawingInfo *info = [PathDrawingInfo pathDrawingInfoWithPath:self.workingPath
fillColor:delegate.fillColor strokeColor:delegate.strokeColor];
  [delegate addDrawable:info];
}
- (void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event {
  isDragging = YES;
  UIView *touchedView = [delegate viewForUseWithTool:self];
  UITouch *touch = [[event allTouches] anyObject];
  CGPoint touchPoint = [touch locationInView:touchedView];
  // set nextSegmentPoint2
  nextSegmentPoint2 = touchPoint;
  // establish nextSegmentCp2
  nextSegmentCp2 = touchPoint;
  if (workingPath.empty) {
    // this is the first touch in a path, so set the "1" variables as well
    nextSegmentCp1 = touchPoint;
    nextSegmentPoint1 = touchPoint;
    [workingPath moveToPoint:touchPoint];
  }
}
- (void)touchesCancelled:(NSSet *)touches withEvent:(UIEvent *)event {
  isDragging = NO;
}

- (void)touchesEnded:(NSSet *)touches withEvent:(UIEvent *)event {
  isDragging = NO;
  UIView *touchedView = [delegate viewForUseWithTool:self];
  UITouch *touch = [[event allTouches] anyObject];
  CGPoint touchPoint = [touch locationInView:touchedView];
  nextSegmentCp2 = touchPoint;
  // complete segment and add to list
  if (settingFirstPoint) {
    // the first touch'n'drag doesn't complete a segment, we just
    // note the change of state and move along
    settingFirstPoint = NO;
  } else {
    // nextSegmentCp2, which we've been dragging around, is translated
    // around nextSegmentPoint2 for creation of this segment.
    CGPoint shiftedNextSegmentCp2 = CGPointMake(
      nextSegmentPoint2.x + (nextSegmentPoint2.x - nextSegmentCp2.x),
      nextSegmentPoint2.y + (nextSegmentPoint2.y - nextSegmentCp2.y));
    [workingPath addCurveToPoint:nextSegmentPoint2 controlPoint1:nextSegmentCp1
controlPoint2:shiftedNextSegmentCp2];
    // the "2" values are now copied to the "1" variables
    nextSegmentPoint1 = nextSegmentPoint2;
    nextSegmentCp1 = nextSegmentCp2;
  }
}
- (void)touchesMoved:(NSSet *)touches withEvent:(UIEvent *)event {
  UIView *touchedView = [delegate viewForUseWithTool:self];
  UITouch *touch = [[event allTouches] anyObject];
  CGPoint touchPoint = [touch locationInView:touchedView];
  if (settingFirstPoint) {
```

www.it-ebooks.info

```
      nextSegmentCp1 = touchPoint;
    } else {
      // adjust nextSegmentCp2
      nextSegmentCp2 = touchPoint;
    }
  }
- (void)drawTemporary {
  // draw all the segments we've finished so far
  [workingPath stroke];
  if (isDragging) {
    // draw the current segment that's being created
    if (settingFirstPoint) {
      // just draw a line
      UIBezierPath *currentWorkingSegment = [UIBezierPath bezierPath];
      [currentWorkingSegment moveToPoint:nextSegmentPoint1];
      [currentWorkingSegment addLineToPoint:nextSegmentCp1];
      [[delegate strokeColor] setStroke];
      [currentWorkingSegment stroke];
    } else {
      // nextSegmentCp2, which we've
      // been dragging around, is translated around nextSegmentPoint2
      // for creation of this segment
      CGPoint shiftedNextSegmentCp2 = CGPointMake(
        nextSegmentPoint2.x + (nextSegmentPoint2.x - nextSegmentCp2.x),
        nextSegmentPoint2.y + (nextSegmentPoint2.y - nextSegmentCp2.y));
      UIBezierPath *currentWorkingSegment = [UIBezierPath bezierPath];
      [currentWorkingSegment moveToPoint:nextSegmentPoint1];
      [currentWorkingSegment addCurveToPoint:nextSegmentPoint2
controlPoint1:nextSegmentCp1 controlPoint2:shiftedNextSegmentCp2];
      [[delegate strokeColor] setStroke];
      [currentWorkingSegment stroke];
    }
  }
  if (!CGPointEqualToPoint(nextSegmentCp2, nextSegmentPoint2) && !settingFirstPoint) {
    // draw the guideline to the next segment
    UIBezierPath *currentWorkingSegment = [UIBezierPath bezierPath];
    [currentWorkingSegment moveToPoint:nextSegmentCp2];
    CGPoint shiftedNextSegmentCp2 = CGPointMake(
      nextSegmentPoint2.x + (nextSegmentPoint2.x - nextSegmentCp2.x),
      nextSegmentPoint2.y + (nextSegmentPoint2.y - nextSegmentCp2.y));
    [currentWorkingSegment addLineToPoint:shiftedNextSegmentCp2];
```

To display the temporary curve that the user is dragging around, we will use a dashed line instead of a solid line, to help make it stand out from the background. This dash pattern specifies that the line will be drawn with 10 pixels in the stroke color we set, then skip 7 pixels, repeating forever.

```
    float dashPattern[] = {10.0, 7.0};
    [currentWorkingSegment setLineDash:dashPattern count:2 phase:0.0];
    [[UIColor redColor] setStroke];
    [currentWorkingSegment stroke];
  }
}
- (void)dealloc {
  self.workingPath = nil;
  self.delegate = nil;
  [super dealloc];
```

www.it-ebooks.info

```
}
@end
```

Now add a few lines to *DudelViewController.m* to bring it together:

```
#import "FreehandTool.h"
- (IBAction)touchFreehandItem:(id)sender {
  self.currentTool = [FreehandTool sharedFreehandTool];
  [freehandButton setImage:[UIImage imageNamed:@"button_bezier_selected.png"]];
}
```

That's it! The final drawing tool button in our toolbar is complete. Try it out and see how it works. Figure 4–7 shows a bit of "art" created with our Dudel tools.

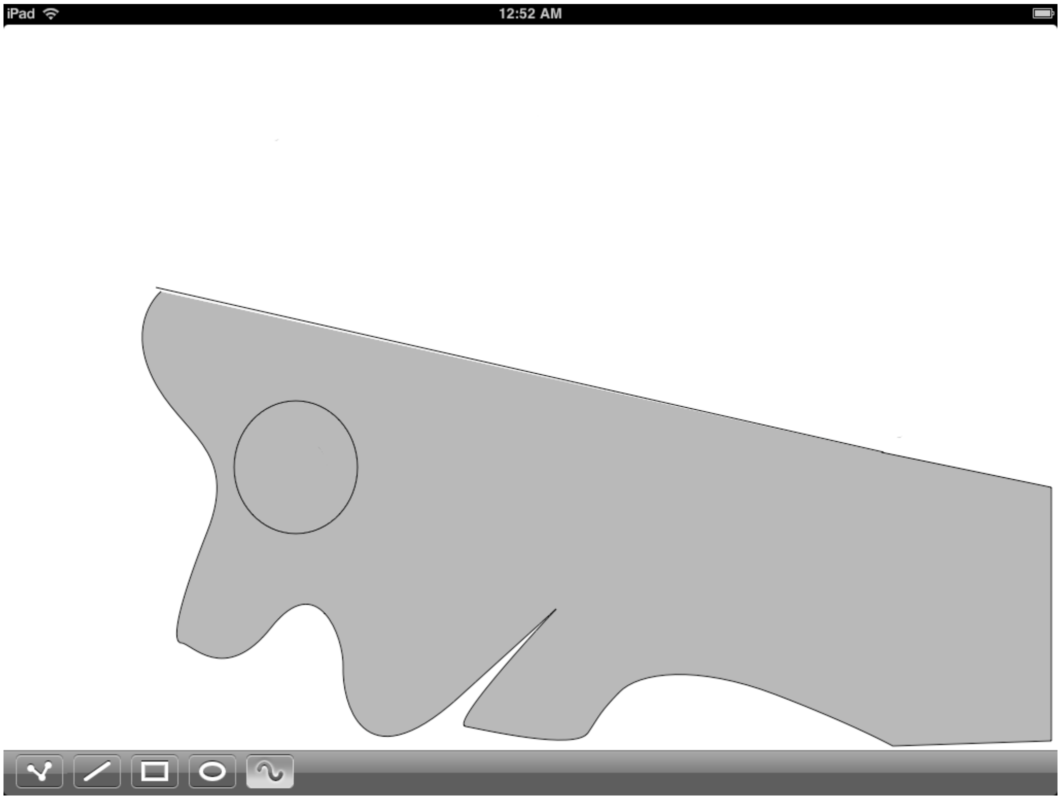


Figure 4–7. *Various paths constructed from Bézier curves, using the Freehand, Ellipse, and Pencil tools*

PDF Generation

The iOS new functionality for rendering to a PDF file means that anything you can draw to the screen using `UIView` can now be drawn straight into a PDF file. Here, you'll learn how to do so by "wrapping" your drawing code in a special PDF-generation context.

www.it-ebooks.info

Generating a PDF is great, but you may ask, “What am I going to do with that PDF file?” In Chapter 10, you’ll learn about some of the new things you can do with files in iOS 3.2 and beyond. For now, we’ll stick to technology that has been a part of iOS ever since *way* back in version 3.0—sending an e-mail message with an attachment using `MFMailComposeViewController`.

We’ll add this new functionality to our Dudel app. First, in the header file for our controller class, we’ll need a new action method, to be triggered from the GUI. We’re also going to declare that our controller conforms to the `MFMailComposeViewControllerDelegate` protocol, so that we’ll be notified when the user has composed and sent the e-mail message or hit Cancel to discard it. We need to include the relevant header for accessing the mail composition GUI. Here’s the updated version of *DudelViewController.h*, containing those changes:

```
//  DudelViewController.h
#import <UIKit/UIKit.h>
#import <MessageUI/MessageUI.h>
#import "Tool.h"
#import "DudelView.h"
@interface DudelViewController : UIViewController <ToolDelegate, DudelViewDelegate,
MFMailComposeViewControllerDelegate> {
  id <Tool> currentTool;
  IBOutlet DudelView *dudelView;
  IBOutlet UIBarButtonItem *freehandButton;
  IBOutlet UIBarButtonItem *ellipseButton;
  IBOutlet UIBarButtonItem *rectangleButton;
  IBOutlet UIBarButtonItem *lineButton;
  IBOutlet UIBarButtonItem *pencilButton;
  UIColor *strokeColor;
  UIColor *fillColor;
}
@property (retain, nonatomic) id <Tool> currentTool;
@property (retain, nonatomic) UIColor *strokeColor;
@property (retain, nonatomic) UIColor *fillColor;
- (IBAction)touchFreehandItem:(id)sender;
- (IBAction)touchEllipseItem:(id)sender;
- (IBAction)touchRectangleItem:(id)sender;
- (IBAction)touchLineItem:(id)sender;
- (IBAction)touchPencilItem:(id)sender;
- (IBAction)touchSendPdfEmailItem:(id)sender;
@end
```

Now go back to project navigation pane in Xcode, and look for the Frameworks section. Right-click it and select **Add ➤ Existing Frameworks…** from the context menu, then double-click `MessageUI.framework` so that Dudel can make use of it.

Next, let’s take care of what we need to hook up in Interface Builder. Open *DudelViewController.xib*, and take a look at the toolbar at the bottom of the window. We want to put a new item in the toolbar that will call our new method, but it should be separated from the drawing tools, so we need some space there as well.

Use the Library to search for “bar button item,” and you’ll see all the sorts of things that you can put into your toolbars. For this example, drag a Flexible Space Bar Button Item out and place it in our toolbar, to the right of everything else. You’ll see it expand to fill

www.it-ebooks.info

all of the available space in the toolbar. Then drag a Bar Button Item from the Library to the far-right end of the toolbar. While it's still selected, use the attribute inspector to make sure its Style is set to Bordered, and set its Title to **Email PDF**. Then control-drag from the new item to File's Owner in the main `.nib` window, and select `touchSendPdfEmailItem:` from the context menu that appears. We're finished with Interface Builder for now, so save your work and go back to Xcode.

It's time to implement the code that will initiate the PDF rendering. Add the following method:

```
- (IBAction)touchSendPdfEmailItem:(id)sender {
  // set up PDF rendering context
  NSMutableData *pdfData = [NSMutableData data];
  UIGraphicsBeginPDFContextToData(pdfData, dudelView.bounds, nil);
  UIGraphicsBeginPDFPage();

  // tell our view to draw
  [dudelView drawRect:dudelView.bounds];

  // remove PDF rendering context
  UIGraphicsEndPDFContext();

  // send PDF data in mail message
  MFMailComposeViewController *mailComposer = [[[MFMailComposeViewController alloc]
init] autorelease];
  mailComposer.mailComposeDelegate = self;
  [mailComposer addAttachmentData:pdfData mimeType:@"application/pdf" fileName:@"Dudel
creation.pdf"];
  [self presentModalViewController:mailComposer animated:YES];
}
```

All we do here is set up a special context using an `NSMutableData` object to contain the PDF content, and call our view object's `drawRect:` method. When we're finished, we pass the PDF data off to the message-composing window.

We also need to implement the following method, so that the e-mail composer can let us know that the user has clicked either Send or Cancel in the window. It's our duty to end the modal session here.

```
- (void)mailComposeController:(MFMailComposeViewController*)controller
didFinishWithResult:(MFMailComposeResult)result error:(NSError*)error {
  [self dismissModalViewControllerAnimated:YES];
}
```

Now you're all set! Build and run, create something with the tools, and then hit the Email PDF button. You'll see that the content of your view is a visible attachment in a new e-mail window, ready for you to send to your admirers!

Things to See and Do

We've covered a lot of ground in this chapter! You've seen several uses of the `NSBezierPath` class, learned how to "print" the content of a `UIView`, and laid the foundation for a fun and functional graphics app. We'll continue building on this foundation throughout the book. Next, in Chapter 5, you'll learn about rendering text on the screen using Core Text.

www.it-ebooks.info

Chapter 5

Using Core Text

This chapter introduces Core Text, a new API available in iOS 3.2. If you're familiar with programming on Mac OS X, you may realize that Core Text isn't really a new API. It has been part of Mac OS X for years, and has now made the transition to iOS.

Here, you'll learn what Core Text is, how it's structured, and how to use it to render text in your applications, by adding a new Text tool to Dudel. After that's in place, you'll see how to use `NSAttributedString` to style your text, letting you change fonts and colors for portions of your string, which Core Text will render with equal ease. In this chapter, we'll keep the text rendering pretty simple, but in Chapter 6, we'll get a little fancier, adding GUI elements for selecting fonts (among other things).

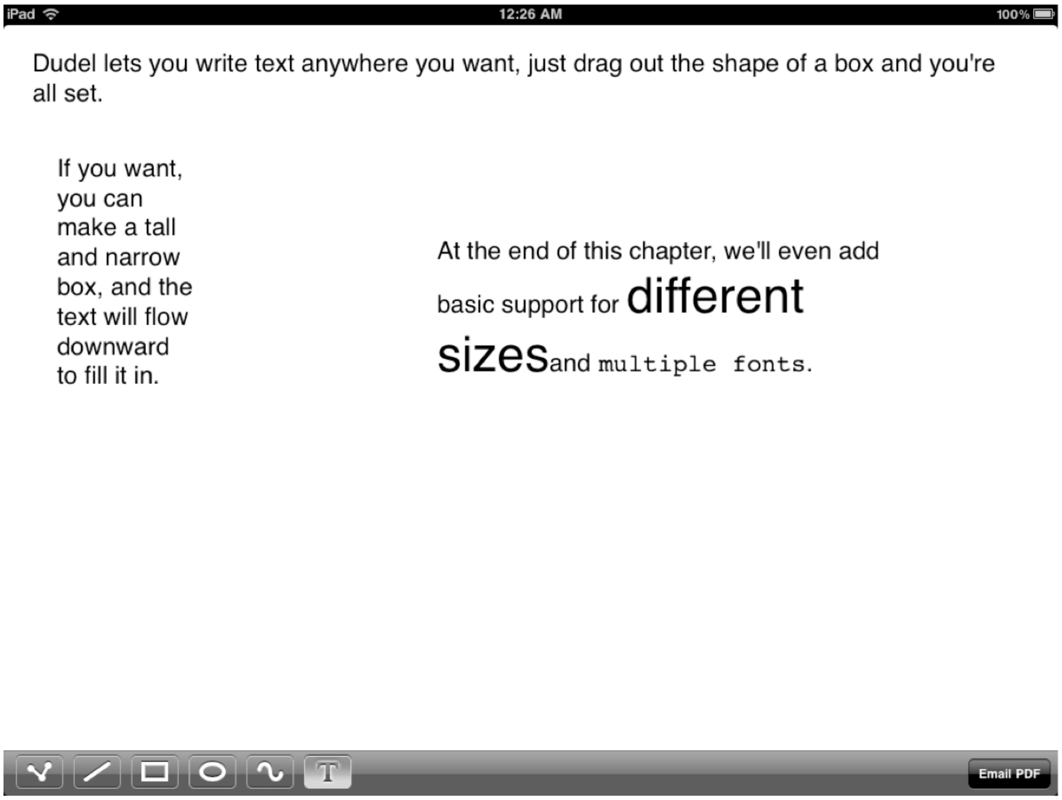
Why Core Text?

Earlier versions of iOS provided a few ways to render text in your own custom views: make a `UILabel` of your view, or tell an `NSString` to draw its contents in a view. But these approaches are somewhat limited. There's no way to specify font, color, or other attributes of a substring to be rendered, for instance. So, if you wanted to render some bold text within a paragraph, you would need to first render everything that came before the bold text, determine the `CGPoint` where the rendering ended, render the bold portion, calculate its end point, start rendering the next piece, and so on. And, of course, at each step, you would need to check to see if you're wrapping to the next line; if so, you would need to start over from there with the rest of the string.

In short, dealing with long strings, or strings with varied styles throughout, has really been kind of a pain in iOS. The level of complexity in laying out rich text is almost on par with laying out a web page, which is the solution that many iPhone applications are built around when displaying styled text, using a `UIWebView` to do the layout. But there are problems with that approach as well. `UIWebView` is a fairly "heavy" class, which can take a while to load and display content—even content that's stored locally on the device.

With Core Text, you now have a chance to skip the web view, and just draw text directly into any graphics context you like. Figure 5–1 shows an example of some text rendered in Dudel using Core Text.

www.it-ebooks.info



Dudel lets you write text anywhere you want, just drag out the shape of a box and you're all set.

If you want, you can make a tall and narrow box, and the text will flow downward to fill it in.

At the end of this chapter, we'll even add basic support for different sizes and multiple fonts.

Figure 5–1. *Dudel now includes basic text rendering.*

Note that Core Text is a fairly low-level way to deal with a piece of text. iOS still doesn't offer anything that is as versatile as the `NSTextView` class in Mac OS X, which will also let you edit rich text, setting fonts and colors as you like. The presence of Core Text is, however, a good step in the right direction. And it's quite possible that Apple or a third party will soon leverage it to provide a general-purpose GUI class for editing rich text.

The Structure of Core Text

Before we start making use of Core Text in our code, an overview of how it works is in order. Unlike most of the new APIs discussed in the book, Core Text is a C-based API, rather than a set of Objective-C classes. For its "home environment" of Mac OS X, it was designed to be a unified API that could be used easily from both Cocoa applications written in Objective-C and Carbon applications written mainly in C and C++. However, like most other modern C-based APIs present in Mac OS X and iOS, Core Text is written in a way that is as close to object-orientation with C as possible, using opaque types for all its structures and accessing those structures only through a comprehensive set of functions. So it's fairly painless.

www.it-ebooks.info

Core Text allows you work with it on a variety of levels. The simplest way lets you take a text string and a rectangle, and with a few lines of code, have the text rendered for you. If you want more fine-grained control, it's possible to reach in and tweak the rendering a bit as well, but most of the time, the high-level functionality is all you'll need. You'll access this through an opaque type called `CTFramesetter` and its associated functions. You create a `CTFramesetter` by passing a special kind of string called an *attributed string* to the `CTFramesetterCreateWithAttributedString()` function, then create another Core Text object called a `CTFrame` by passing the `CTFramesetter` and a `CGPath` (containing just a rectangle) to the `CTFramesetterCreateFrame()` function, and finally render the result with a call to the `CTFrameDraw()` function. Figure 5–2 shows how these different pieces fit together.

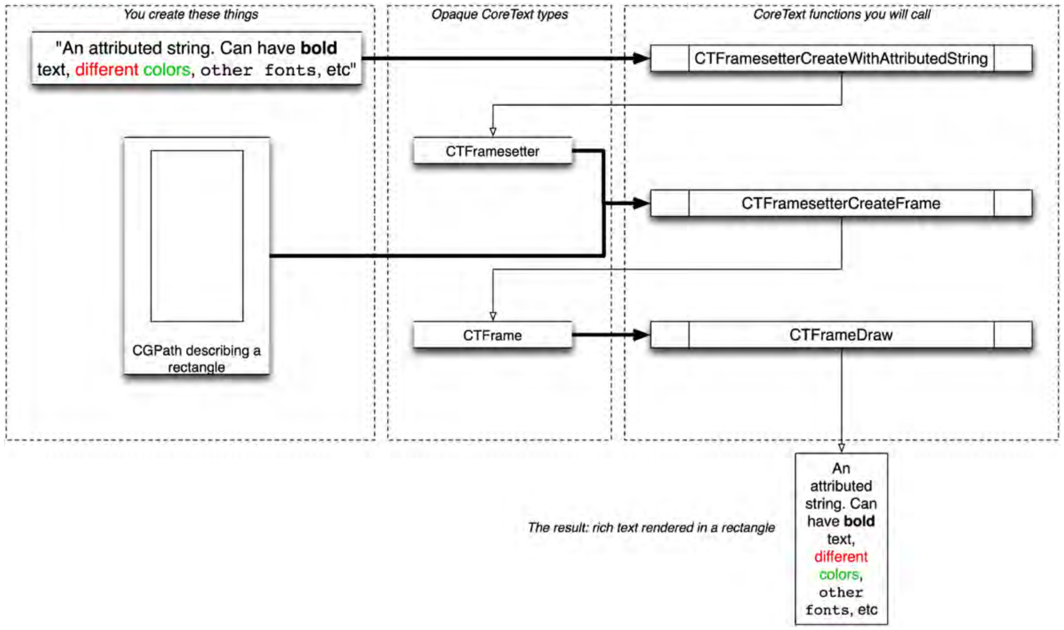

You create these things
Opaque CoreText types
CoreText functions you will call
"An attributed string. Can have bold text, different colors, other fonts, etc"
CTFramesetterCreateWithAttributedString
CTFramesetter
CTFramesetterCreateFrame
CGPath describing a rectangle
CTFrame
CTFrameDraw
The result: rich text rendered in a rectangle
An attributed string. Can have bold text, different colors, other fonts, etc

Figure 5–2. *The basic Core Text workflow*

An attributed string consists of a string and some metadata describing formatting attributes for portions of the string. For instance, you might want to render text where some words are underlined, bold, or in a different font or color. Attributed strings give you a concrete way to represent this sort of thing. In iOS, attributed strings are represented by the Objective-C class `NSAttributedString` and the C type `CFAttributedStringRef`. These are toll-free bridged to one another, so you can use them interchangeably.

www.it-ebooks.info

TOLL-FREE BRIDGING FROM ONE FOUNDATION TO ANOTHER

In both iOS and Mac OS X, there are some kinds of entities that are said to be *toll-free bridged* to one another. This typically refers to an Objective-C class and an opaque C type. In a nutshell, this means that the types are equivalent, and that any function or method that accepts one of them will work just as well with the other (though you may need to do some manual casting to satisfy the compiler). Many of the types that are used in the Core Foundation C library, such as `CFString`, `CFArray`, and so on, are toll-free bridged to a similarly named counterpart in Objective-C's Foundation framework.

The Core Foundation types are all opaque types, which means that each of them is really a pointer to a structure whose contents you, as a user of the API, shouldn't be concerned with. As a way of highlighting the fact that you're dealing with references rather than with the structures themselves, the opaque types defined in Core Foundation have the suffix `Ref` on the end of each type name. The following lines illustrate the concept of passing bridged types around.

```
NSString *objcString;  // assume this exists.
CFStringRef cfString;  // this too.

functionThatWantsCFString(cfString);                // this is fine,
functionThatWantsCFString((CFStringRef)objcString); // and so is this!

[objcString length];           // this returns an integer,
[(NSString *)cfString length]; // and so does this!
```

The opaque types created by Core Foundation abide by the same memory management rules as Objective-C objects do: You're required to release anything that you create or retain. However, whenever you create these objects from C functions instead of Objective-C methods, you should always manage them using functions like `CFRetain()` and `CFRelease()`, instead of the `retain` and `release` methods. This is partly to be stylistically consistent, but also because some of them may not actually be Objective-C objects at all.

As an example, let's look at some code that creates an `NSAttributedString` from an `NSString`, and assigns a bold attribute to a part of the string.

```
NSString *myString = @"A dingo stole my baby!";
NSMutableAttributedString *attrString =
  [[[NSMutableAttributedString alloc] initWithString:myString] autorelease];
[attrString addAttribute:(NSString *)(kCTForegroundColorAttributeName)
                   value:(id)[UIColor redColor].CGColor
                   range:NSMakeRange(0, [myString length])];
```

The specified attribute name, `kCTForegroundColorAttributeName`, defines which attribute of the text we're setting, and then we pass in a value to set for that attribute and the range of the text to cover. In this case, we're just making it all red, but you can do whatever you like. The *Core Text String Attributes Reference*, included with the SDK, contains a list of all the attributes that apply, their meanings, and the type of values expected for each. In addition to color, you can set fonts, paragraph styles, and more.

Once you've created an attributed string, all you really need to do is create a `CTFramesetter`, use that to generate `CTFrame`, and tell it to draw itself.

```
CTFramesetterRef framesetter =
  CTFramesetterCreateWithAttributedString((CFAttributedStringRef)attrString);
```

www.it-ebooks.info

```
CTFrameRef frame = CTFramesetterCreateFrame(framesetter,
                                            CFRangeMake(0, [attrString length]),
                                            myCGPath, // the rect to draw into
                                            NULL);
CTFrameDraw(frame, graphicsContext);  // needs a graphics context to draw into
```

This code is taken out of context, and will need some help before it will really do anything. The next section will provide some context for it, in the form of a new Text tool for Dudel.

Preparing Dudel for a New Tool

We're going to be adding a few files to Dudel, and making changes to a few others, so now might be a good time to make a copy of your Dudel project directory. That way, you can work with the copy, and still have your previous version for reference in case something goes wrong. If you would prefer to start working from a clean slate, you can take a fresh copy of the completed app from Chapter 4, and use it as the basis for what we're doing here.

Preparing the Controller Interface

Let's start off, as before, by dealing with the interface for our controller class. We're going to have a new button for the Text tool, so `DudelViewController` will get a new instance variable to point at that, as well as a new action method for the button to call. Additionally, we'll add a new `font` property, which the Text tool will access to figure out which font to use to draw its text. For now, we're not going to provide any GUI for the user to actually set the font—that will come later, in Chapter 6. The updated *DudelViewController.h* looks like this (the lines shown in bold text are the new parts):

```
//  DudelViewController.h

#import <UIKit/UIKit.h>
#import <MessageUI/MessageUI.h>

#import "Tool.h"
#import "DudelView.h"

@interface DudelViewController : UIViewController <ToolDelegate, DudelViewDelegate,
MFMailComposeViewControllerDelegate> {
  id <Tool> currentTool;
  IBOutlet DudelView *dudelView;
  IBOutlet UIBarButtonItem *textButton;
  IBOutlet UIBarButtonItem *freehandButton;
  IBOutlet UIBarButtonItem *ellipseButton;
  IBOutlet UIBarButtonItem *rectangleButton;
  IBOutlet UIBarButtonItem *lineButton;
  IBOutlet UIBarButtonItem *dotButton;
  UIColor *strokeColor;
  UIColor *fillColor;
  CGFloat strokeWidth;
  UIFont *font;
```

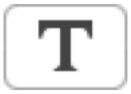
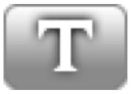
www.it-ebooks.info

```
}

@property (retain, nonatomic) id <Tool> currentTool;
@property (retain, nonatomic) UIColor *strokeColor;
@property (retain, nonatomic) UIColor *fillColor;
@property (assign, nonatomic) CGFloat strokeWidth;
@property (retain, nonatomic) UIFont *font;

- (IBAction)touchTextItem:(id)sender;
- (IBAction)touchFreehandItem:(id)sender;
- (IBAction)touchEllipseItem:(id)sender;
- (IBAction)touchRectangleItem:(id)sender;
- (IBAction)touchLineItem:(id)sender;
- (IBAction)touchDotItem:(id)sender;
- (IBAction)touchSendPdfEmailItem:(id)sender;

@end
```

Setting Up the GUI

We'll also need a pair of images, normal and highlighted, for the Text tool button. Either grab these from the book's code archive or make something on your own, similar to what's shown in Table 5-1.

Table 5-1. *New Buttons for the Text Tool*

Filename	Image
button_text.png	
button_text_selected.png	

Add those to your project alongside the other button images, and then open *DudelViewController.xib* in Interface Builder. Make sure you can see toolbar at the bottom of the Dudel View window. Drag a new `UIBarButtonItem` from the Library to the toolbar, placing it to the right of the other tools, but to the left of the flexible space. Use the attribute inspector to set its Image to *button_text.png* and set its Style to Plain. Figure 5–3 shows the end result.

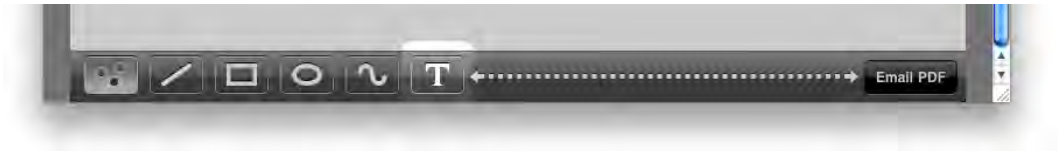


Figure 5–3. *Placement of the new Text tool*

www.it-ebooks.info

Now control-drag from the new button to the File's Owner in the main *.nib* window, and select the `touchTextItem:` action from the menu that appears. Then control-drag from the File's Owner back to the button, and select the `textButton` outlet from the menu. That's all the GUI configuration we need to do, so you can save your work and go back to Xcode.

Implementing Changes to the Controller Class

Let's return to `DudelViewController`, and make the implementation changes to match the new things in the interface. Open *DudelViewController.m*, and start off by adding the following near the top of the file, so that the controller will get access to the new `TextTool` class we'll soon create:

```
#import "TextTool.h"
```

We also have the new `font` property to synthesize. Add it to the existing line:

```
@synthesize currentTool, fillColor, strokeColor, strokeWidth, font;
```

Next, we want to make sure we have a default value for the font, so add a line to `viewDidLoad`, like this:

```
- (void)viewDidLoad {
  [super viewDidLoad];
  self.currentTool = [DotTool sharedDotTool];
  [dotButton setImage:[UIImage imageNamed:@"button_dot_selected.png"]];
  self.fillColor = [UIColor colorWithWhite:0.0 alpha:0.25];
  self.strokeColor = [UIColor blackColor];
  self.font = [UIFont systemFontOfSize:24.0];
}
```

Now let's update the `deselectAllToolButtons` method so that it knows about the new button:

```
- (void)deselectAllToolButtons {
  [textButton setImage:[UIImage imageNamed:@"button_text.png"]];
  [freehandButton setImage:[UIImage imageNamed:@"button_bezier.png"]];
  [ellipseButton setImage:[UIImage imageNamed:@"button_ellipse.png"]];
  [rectangleButton setImage:[UIImage imageNamed:@"button_rectangle.png"]];
  [lineButton setImage:[UIImage imageNamed:@"button_line.png"]];
  [dotButton setImage:[UIImage imageNamed:@"button_dot.png"]];
}
```

Finally, implement the method that the new button will call:

```
- (IBAction)touchTextItem:(id)sender {
  self.currentTool = [TextTool sharedTextTool];
  [self deselectAllToolButtons];
  [textButton setImage:[UIImage imageNamed:@"button_text_selected.png"]];
}
```

Those are all the changes our controller class needs in order to handle the new tool. Next, let's update the `ToolDelegate` protocol to match the new bit of functionality in the controller. Our new Text tool will want to get the currently selected font from the

www.it-ebooks.info

controller, which it knows of only as an object conforming to the ToolDelegate protocol. Edit the *Tool.h file,* and add a line to the ToolDelegate section at the bottom:

```
@protocol ToolDelegate
- (void)addDrawable:(id <Drawable>)d;
- (UIView *)viewForUseWithTool:(id <Tool>)t;
- (UIColor *)strokeColor;
- (UIColor *)fillColor;
- (UIFont *)font;
@end
```

Creating the Text Tool

With that in place, we can now create the TextTool class itself. This class is a lot like the RectangleTool class, but with an extra twist: When the user finishes drawing a rectangle, this tool switches into text-editing mode by placing a UITextView at the location where the rectangle was drawn. It's set up as the first responder so that the keyboard will appear, and users can enter their text. It also shows a gray shade over the rest of the screen, to give the users some focus so they can see where they're typing, as shown in Figure 5–4.

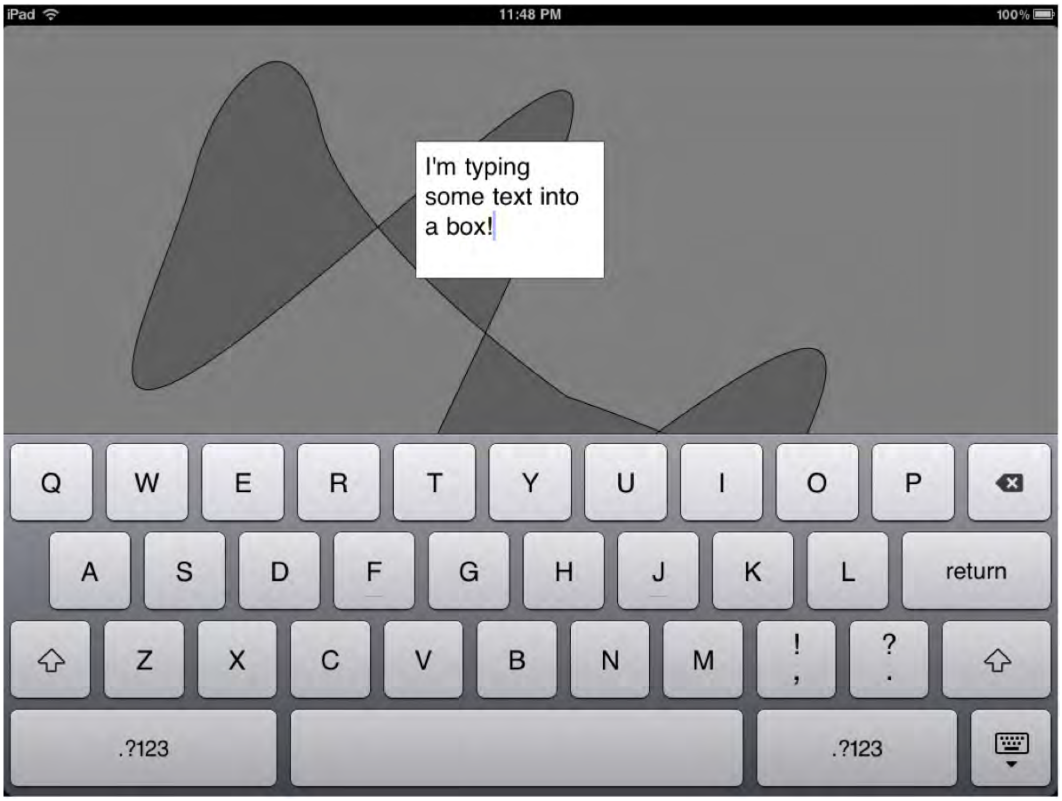


Figure 5–4. *The text field's so bright; you gotta show shade.*

www.it-ebooks.info

When the user presses the bottom-right keyboard button to dismiss the keyboard, or taps anywhere else in the drawing area, the text entry is considered complete. At that point, the tool creates an instance of a new class called `TextDrawingInfo` (which we haven't created yet). That will be another class that implements the `Drawable` protocol (like the `PathDrawingInfo` class from the Chapter 4), and can therefore be added to the list of things that `DudelView` needs to draw.

Declaring the Text Tool Interface

Create a new class called `TextTool`, and start defining it by putting this code into *TextTool.h*:

```
//  TextTool.h
#import <Foundation/Foundation.h>
#import "Tool.h"

@interface TextTool : NSObject <Tool, UITextViewDelegate> {
  id <ToolDelegate> delegate;
  NSMutableArray *trackingTouches;
  NSMutableArray *startPoints;
  UIBezierPath *completedPath;
  CGFloat viewSlideDistance;
}
@property (retain, nonatomic) UIBezierPath *completedPath;
+ (TextTool*)sharedTextTool;
@end
```

Most of what's declared here is pretty similar to what our other tools had. The one new addition is the `viewSlideDistance`, which we'll use to determine how far to shift the view in case it's being covered up by the on-screen keyboard (more on that in a page or two).

Implementing TextTool

The file containing the implementation, *TextTool.m*, is a bit more complicated than the tools we created in Chapter 4, so I'll interject some additional information at the tricky spots as we go through it. Start off with some `#imports`:

```
//  TextTool.m

#import "TextTool.h"
#import "TextDrawingInfo.h"

#import "SynthesizeSingleton.h"
```

After the rectangle is drawn, we'll create a temporary view, which will need to be cleaned up later. Instead of using an instance variable for this, we're going to assign a tag number for later retrieval.

```
#define SHADE_TAG 10000
```

Later on, we'll check the distance between the drawn rectangle's start and end points to see if we think it's big enough to contain any text. This function will help us out.

www.it-ebooks.info

```
static CGFloat distanceBetween(const CGPoint p1, const CGPoint p2) {
  // Pythagoras in the house!
  return sqrt(pow(p1.x-p2.x, 2) + pow(p1.y-p2.y, 2));
}

@implementation TextTool
@synthesize delegate, completedPath;
SYNTHESIZE_SINGLETON_FOR_CLASS(TextTool);
- init {
  if ((self = [super init])) {
    trackingTouches = [[NSMutableArray array] retain];
    startPoints = [[NSMutableArray array] retain];
  }
  return self;
}
- (void)activate {
}
- (void)deactivate {
  [trackingTouches removeAllObjects];
  [startPoints removeAllObjects];
  self.completedPath = nil;
}
- (void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event {
  UIView *touchedView = [delegate viewForUseWithTool:self];
  [touchedView endEditing:YES];
```

Unlike the Rectangle tool introduced in Chapter 4, this tool should allow the user to drag out only one rectangle at a time. If we let it do more, how would we know which one should get the text? So here, instead of dealing with all the touches, we just ask for any one of them.

```
  UITouch *touch = [[event allTouches] anyObject];
```

Remember the touch, and its original start point, for future reference.

```
  [trackingTouches addObject:touch];
  CGPoint location = [touch locationInView:touchedView];
  [startPoints addObject:[NSValue valueWithCGPoint:location]];
}
- (void)touchesCancelled:(NSSet *)touches withEvent:(UIEvent *)event {
}
- (void)touchesEnded:(NSSet *)touches withEvent:(UIEvent *)event {
  UIView *touchedView = [delegate viewForUseWithTool:self];
  for (UITouch *touch in [event allTouches]) {
    NSUInteger touchIndex = [trackingTouches indexOfObject:touch];
```

We continue with the rest only if we actually remember the start of this touch. We might be seeing a simultaneous touch that we ignored earlier.

```
    if (touchIndex != NSNotFound) {
      CGPoint startPoint = [[startPoints objectAtIndex:touchIndex] CGPointValue];
      CGPoint endPoint = [touch locationInView:touchedView];
      [trackingTouches removeObjectAtIndex:touchIndex];
      [startPoints removeObjectAtIndex:touchIndex];
```

Detect short taps that are too small to contain any text. These are probably accidents.

```
      if (distanceBetween(startPoint, endPoint) < 5.0) return;
```

www.it-ebooks.info

Make a rectangle that stretches from the start point to the current point, and wrap that in a path.

```
CGRect rect = CGRectMake(startPoint.x, startPoint.y,
                         endPoint.x - startPoint.x, endPoint.y - startPoint.y);
self.completedPath = [UIBezierPath bezierPathWithRect:rect];
```

Draw a shaded area over the entire view, so that the users can easily see where to focus their attention.

```
UIView *backgroundShade = [[[UIView alloc] initWithFrame:touchedView.bounds]
                           autorelease];
backgroundShade.backgroundColor = [UIColor colorWithWhite:0.0 alpha:0.5];
backgroundShade.tag = SHADE_TAG;
backgroundShade.userInteractionEnabled = NO;
[touchedView addSubview:backgroundShade];
```

Now comes the fun part. We make a temporary `UITextView` for the actual text input, and set ourselves up to receive notifications when that input begins and ends.

```
UITextView *textView = [[[UITextView alloc] initWithFrame:rect] autorelease];
[[NSNotificationCenter defaultCenter] addObserver:self
                                         selector:@selector(keyboardWillShow:)
                                             name:UIKeyboardWillShowNotification
                                           object:nil];
[[NSNotificationCenter defaultCenter] addObserver:self
                                         selector:@selector(keyboardWillHide:)
                                             name:UIKeyboardWillHideNotification
                                           object:nil];
```

Anyone dealing with text input on the iPhone has probably had to tackle the problem of displaying content that may be obscured by the on-screen keyboard. Here in Dudel, we're going to have the same problem, since users can easily drag a text rectangle in the lower half of the screen.

The following code determines how far the main view needs to be shifted to account for the current rectangle, based on the current orientation and the size of the on-screen keyboard. This value is stored in the `viewSlideDistance` variable. It will be used later when the keyboard slides into place, and again when it slides back out. Although a user can still create a text rectangle so tall that it will be partly obscured, by doing the following, we're at least making a solid effort and covering the most common cases.

```
CGFloat keyboardHeight = 0;
UIInterfaceOrientation orientation =
  ((UIViewController*)delegate).interfaceOrientation;
if (UIInterfaceOrientationIsPortrait(orientation)) {
  keyboardHeight = 264;
} else {
  keyboardHeight = 352;
}
CGRect viewBounds = touchedView.bounds;
CGFloat rectMaxY = rect.origin.y + rect.size.height;
CGFloat availableHeight = viewBounds.size.height - keyboardHeight;
if (rectMaxY > availableHeight) {
  // calculate a slide distance so that the dragged box is centered vertically
  viewSlideDistance = rectMaxY - availableHeight;
} else {
```

www.it-ebooks.info

```
        viewSlideDistance = 0;
      }


      textView.delegate = self;
      [touchedView addSubview:textView];
```

This next part is a bit of a trick. Due to a bug in UITextView, just telling it to become the first responder doesn't actually make it happen. The users typically must tap it once on their own to make the keyboard pop up. Toggling the editable flag is a work-around that makes the keyboard actually appear.

```
      textView.editable = NO;
      textView.editable = YES;
      [touchedView becomeFirstResponder];
    }
  }
}
- (void)touchesMoved:(NSSet *)touches withEvent:(UIEvent *)event {
}
- (void)drawTemporary {
  if (self.completedPath) {
    [delegate.strokeColor setStroke];
    [self.completedPath stroke];
  } else {
    UIView *touchedView = [delegate viewForUseWithTool:self];
    for (int i = 0; i<[trackingTouches count]; i++) {
      UITouch *touch = [trackingTouches objectAtIndex:i];
      CGPoint startPoint = [[startPoints objectAtIndex:i] CGPointValue];
      CGPoint endPoint = [touch locationInView:touchedView];
      CGRect rect = CGRectMake(startPoint.x, startPoint.y, endPoint.x - startPoint.x,
                               endPoint.y - startPoint.y);
      UIBezierPath *path = [UIBezierPath bezierPathWithRect:rect];
      [delegate.strokeColor setStroke];
      [path stroke];
    }
  }
}
- (void)dealloc {
  self.completedPath = nil;
  [trackingTouches release];
  [startPoints release];
  self.delegate = nil;
  [super dealloc];
}
```

These are the methods that are triggered by the hiding and showing of the keyboard. When the keyboard slides into place, it covers up the lower portion of the display. Here, we handle this by shifting things a bit if the rectangle we're operating on is covered up.

```
- (void)keyboardWillShow:(NSNotification *)aNotification {
  UIInterfaceOrientation orientation =
    ((UIViewController*)delegate).interfaceOrientation;
  [UIView beginAnimations:@"viewSlideUp" context:NULL];
  UIView *view = [delegate viewForUseWithTool:self];
  CGRect frame = [view frame];
  switch (orientation) {
```

www.it-ebooks.info

```
    case UIInterfaceOrientationLandscapeLeft:
      frame.origin.x -= viewSlideDistance;
      break;
    case UIInterfaceOrientationLandscapeRight:
      frame.origin.x += viewSlideDistance;
      break;
    case UIInterfaceOrientationPortrait:
      frame.origin.y -= viewSlideDistance;
      break;
    case UIInterfaceOrientationPortraitUpsideDown:
      frame.origin.y += viewSlideDistance;
      break;
    default:
      break;
  }
  [view setFrame:frame];
  [UIView commitAnimations];
}
- (void)keyboardWillHide:(NSNotification *)aNotification {
  UIInterfaceOrientation orientation =
    ((UIViewController*)delegate).interfaceOrientation;
  [UIView beginAnimations:@"viewSlideDown" context:NULL];
  UIView *view = [delegate viewForUseWithTool:self];
  CGRect frame = [view frame];
  switch (orientation) {
    case UIInterfaceOrientationLandscapeLeft:
      frame.origin.x += viewSlideDistance;
      break;
    case UIInterfaceOrientationLandscapeRight:
      frame.origin.x -= viewSlideDistance;
      break;
    case UIInterfaceOrientationPortrait:
      frame.origin.y += viewSlideDistance;
      break;
    case UIInterfaceOrientationPortraitUpsideDown:
      frame.origin.y -= viewSlideDistance;
      break;
    default:
      break;
  }
  [view setFrame:frame];
  [UIView commitAnimations];
}
```

This method, declared in the `UITextViewDelegate` protocol, is called when the user taps outside the `textView` or dismisses the keyboard. Here, we create the `TextDrawingInfo` object (which we'll define shortly) that contains the entered text, along with the current font and color choices. Then we get rid of the temporary views we created earlier.

```
- (void)textViewDidEndEditing:(UITextView *)textView {
  NSLog(@"textViewDidEndEditing");
  TextDrawingInfo *info = [TextDrawingInfo textDrawingInfoWithPath:completedPath
                                                              text:textView.text
                                                       strokeColor:delegate.strokeColor
                                                              font:delegate.font];
  [delegate addDrawable:info];
  self.completedPath = nil;
```

www.it-ebooks.info

```
  UIView *superView = [textView superview];
  [[superView viewWithTag:SHADE_TAG] removeFromSuperview];
  [textView resignFirstResponder];
  [textView removeFromSuperview];
  [[NSNotificationCenter defaultCenter] removeObserver:self];
}
@end
```

So that's the Text tool. It's fairly complex, but that's actually a good thing! All the intricacies of entering text are in one place, and the rest of our architecture requires minimal changes in order to deal with it. You've already seen the few changes that `DudelViewController` needed, and the `DudelView` class itself requires no changes at all! All that's left now is to define the `TextDrawingInfo` class.

Creating a New Drawable Class

Once again, create a new class in Xcode, and name it `TextDrawingInfo`. The code for this is pretty similar to the `PathDrawingInfo` class from Chapter 4, but here we're keeping track of a slightly different set of details and providing a different set of methods for creating new instances. *TextDrawingInfo.h* looks like this:

```
//  TextDrawingInfo.h
#import <Foundation/Foundation.h>
#import "Drawable.h"
@interface TextDrawingInfo : NSObject <Drawable> {
  UIBezierPath *path;
  UIColor *strokeColor;
  UIFont *font;
  NSString *text;
}
@property (retain, nonatomic) UIBezierPath *path;
@property (retain, nonatomic) UIColor *strokeColor;
@property (retain, nonatomic) UIFont *font;
@property (copy, nonatomic) NSString *text;
- (id)initWithPath:(UIBezierPath*)p text:(NSString*)t strokeColor:(UIColor*)s
font:(UIFont*)f;
+ (id)textDrawingInfoWithPath:(UIBezierPath *)p text:t strokeColor:(UIColor *)s
font:(UIFont *)f;
@end
```

As for the implementation, *TextDrawingInfo.m* is pretty straightforward. The only interesting method is the `draw` method. Here's the whole thing:

```
//  TextDrawingInfo.m
#import "TextDrawingInfo.h"
#import <CoreText/CoreText.h>

@implementation TextDrawingInfo
@synthesize path, strokeColor, font, text;
- initWithPath:(UIBezierPath*)p text:(NSString*)t strokeColor:(UIColor*)s
font:(UIFont*)f {
  if ((self = [self init])) {
    path = [p retain];
    strokeColor = [s retain];
    font = [f retain];
```

www.it-ebooks.info

```
    text = [t copy];
  }
  return self;
}
+ (id)textDrawingInfoWithPath:(UIBezierPath *)p text:t strokeColor:(UIColor *)s
font:(UIFont *)f {
  return [[[self alloc] initWithPath:p text:t strokeColor:s font:f] autorelease];
}
- (void)dealloc {
  self.path = nil;
  self.strokeColor = nil;
  self.font = nil;
  self.text = nil;
  [super dealloc];
}
- (void)draw {
  CGContextRef context = UIGraphicsGetCurrentContext();

  NSMutableAttributedString *attrString =
    [[[NSMutableAttributedString alloc] initWithString:self.text] autorelease];
  [attrString addAttribute:(NSString *)(kCTForegroundColorAttributeName)
                     value:(id)self.strokeColor.CGColor
                     range:NSMakeRange(0, [self.text length])];
  CTFramesetterRef framesetter =
    CTFramesetterCreateWithAttributedString((CFAttributedStringRef)attrString);

  CTFrameRef frame = CTFramesetterCreateFrame(framesetter,
                                              CFRangeMake(0, [attrString length]),
                                              self.path.CGPath, NULL);
  CFRelease(framesetter);
  if (frame) {
    CGContextSaveGState(context);

    // Core Text wants to draw our text upside down! This flips it the
    // right way.
    CGContextTranslateCTM(context, 0, path.bounds.origin.y);
    CGContextScaleCTM(context, 1, -1);
    CGContextTranslateCTM(context, 0, -(path.bounds.origin.y +
path.bounds.size.height));

    CTFrameDraw(frame, context);
    CGContextRestoreGState(context);
    CFRelease(frame);
  }
}
@end
```

You may recognize some pieces of the `draw` method. They're very similar to the example shown earlier in this chapter. We create an attributed string and set its color, use it to create a `CTFramesetterRef`, and then use that to create a `CTFrameRef`. Here, we've added a check to make sure the `CTFrameRef` is created and also a few `CGContext` function calls. These are here for a specific reason: On the Mac OS X platform, the y axis is flipped (relative to the way it's done on iOS) for normal drawing. Core Text, which comes from Mac OS X, expects that flipped axis, which means that when it draws in an iOS context, the results are upside down! The translate/scale/translate triad in the preceding code makes sure that the text appears right side up.

www.it-ebooks.info

Before this will compile, we need to add the Core Text framework to the Xcode project. Right-click the Frameworks folder, select **Add ➤ Existing Frameworks…** from the context menu, select *CoreText.framework* from the list that appears, and then click the Add button.

Now that everything is in place, you should be able to build and run Dudel in the simulator, and use the new Text tool to add text to your drawings. The masterpiece on display in Figure 5–5 just scratches the surface of what can be done here.

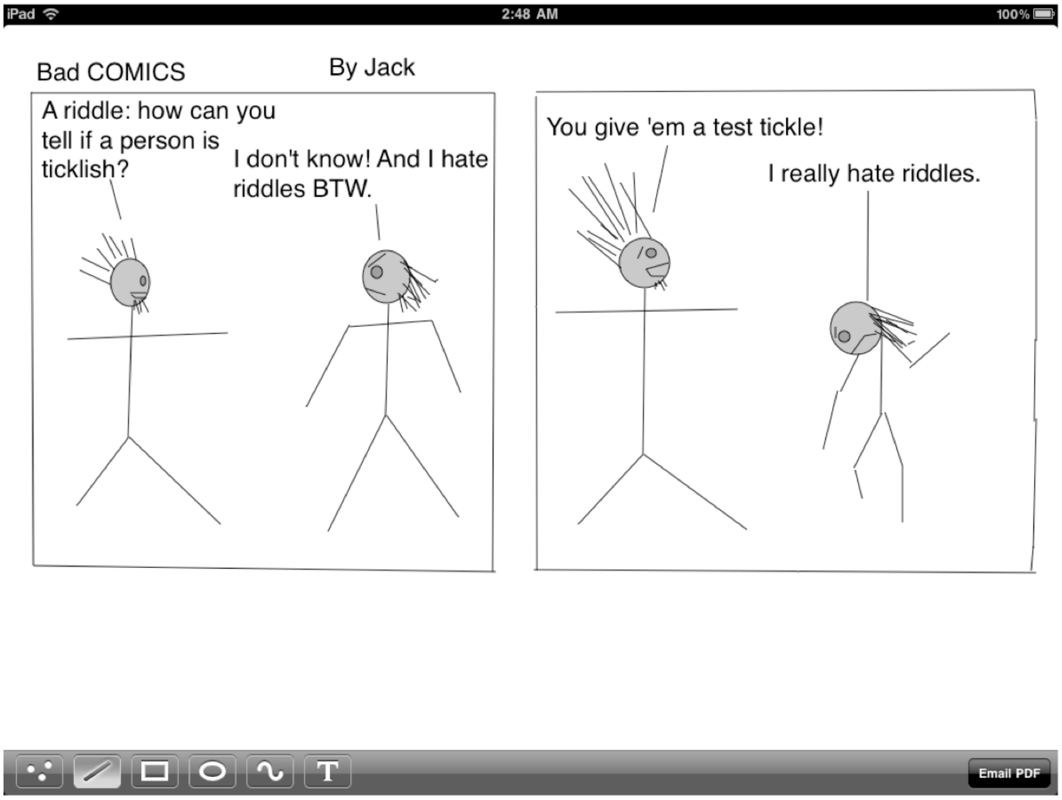

iPad
2:48 AM
100%
Bad COMICS
By Jack
A riddle: how can you tell if a person is ticklish?
I don't know! And I hate riddles BTW.
You give 'em a test tickle!
I really hate riddles.
Email PDF

Figure 5–5. *This is the only joke I can consistently remember, people—seriously.*

Rendering Multiple Styles

So, we now have a tool for drawing text, but we still haven't reached the core of what's interesting about Core Text: rendering multiple styles. At this point, lacking a standard GUI widget that gives us anything like a WYSIWYG display while editing the text, there's no really nice way to enter rich text as you can in a word processor, with buttons to change fonts or set colors.

Fortunately for me, I know that you're a computer programmer, and chances are you're already familiar with a way of marking text attributes that isn't as nice, but *is* applicable to a wide range of problems: HTML! Let's extend our text-rendering algorithm to include

www.it-ebooks.info

a very basic parsing of the text that the user enters, looking for embedded tags that we can use to assign attributes to the text.

I'm going to show you a very simple approach that uses an NSScanner object to scan through the entire text string, searching for just a single kind of tag: <font> (and its matching end tag). It will use the specified values to add attributes to the text. What we're doing here is just barely what I would call "parsing," and will probably make you cringe if your computer science education is less rusty than mine. I'm also well aware that the font tag has been deprecated for years, but it's sure an easy way to do quick-'n-dirty markup compared to using CSS! And it works well for our purposes here.

Edit the beginning of the draw method of TextDrawingInfo as shown here, removing the crossed-out lines and replacing them with the bold line:

```
- (void)draw {
  CGContextRef context = UIGraphicsGetCurrentContext();

~~  //NSMutableAttributedString *attrString = [[[NSMutableAttributedString alloc]~~
~~initWithString:self.text] autorelease];~~
~~  //[attrString addAttribute:(NSString *)(kCTForegroundColorAttributeName)~~
~~value:(id)self.strokeColor.CGColor range:NSMakeRange(0, [self.text length])];~~
  **NSAttributedString *attrString = [self attributedStringFromMarkup:self.text];**
  CTFramesetterRef framesetter =
    CTFramesetterCreateWithAttributedString((CFAttributedStringRef)attrString);
```

Now we need to define the attributedStringFromMarkup: method in the same class (anywhere above the draw method should be fine). This uses NSScanner to look for the tags it knows about, and makes one big NSMutableAttributedString out of a number of smaller NSAttributedStrings generated between tags. Here, you also see a little usage of CTFontRef, which is Core Text's own way of referring to fonts.

```
- (NSAttributedString *)attributedStringFromMarkup:(NSString *)markup {
  NSMutableAttributedString *attrString =
    [[[NSMutableAttributedString alloc] initWithString:@""] autorelease];
  NSString *nextTextChunk = nil;
  NSScanner *markupScanner = [NSScanner scannerWithString:markup];
  CGFloat fontSize = 0.0;
  NSString *fontFace = nil;
  UIColor *fontColor = nil;
  while (![markupScanner isAtEnd]) {
    [markupScanner scanUpToString:@"<" intoString:&nextTextChunk];
    [markupScanner scanString:@"<" intoString:NULL];
    if ([nextTextChunk length] > 0) {
      CTFontRef currentFont =
        CTFontCreateWithName((CFStringRef)(fontFace ? fontFace : self.font.fontName),
                             (fontSize != 0.0 ? fontSize : self.font.pointSize),
                             NULL);
      UIColor *color = fontColor ? fontColor : self.strokeColor;
      NSDictionary *attrs = [NSDictionary dictionaryWithObjectsAndKeys:
                             (id)color.CGColor, kCTForegroundColorAttributeName,
                             (id)currentFont, kCTFontAttributeName,
                             nil];
      NSAttributedString *newPiece = [[[NSAttributedString alloc]
        initWithString:nextTextChunk attributes:attrs] autorelease];
      [attrString appendAttributedString:newPiece];
```

www.it-ebooks.info

```
      CFRelease(currentFont);
    }
    NSString *elementData = nil;
    [markupScanner scanUpToString:@">" intoString:&elementData];
    [markupScanner scanString:@">" intoString:NULL];
    if (elementData) {
      if ([elementData length] > 3 &&
        [[elementData substringToIndex:4] isEqual:@"font"]) {
        fontFace = fontFaceNameFromString(elementData);
        fontSize = fontSizeFromString(elementData);
        fontColor = fontColorFromString(elementData);
      } else if ([elementData length] > 4 &&
        [[elementData substringToIndex:5] isEqual:@"/font"]) {
        // reset all values
        fontSize = 0.0;
        fontFace = nil;
        fontColor = nil;
      }
    }
  }
  return attrString;
}
```

This method, in turn, offloads the parsing of the `font` element attributes to the following three functions. Put these directly above the `attributedStringFromMarkup:` method. (Although it may seem wrong, putting them inside the `@implementation` block is totally fine.)

```
static NSString *fontFaceNameFromString(NSString *attrData) {
  NSScanner *attributeDataScanner = [NSScanner scannerWithString:attrData];
  NSString *faceName = nil;
  if ([attributeDataScanner scanUpToString:@"face=\"" intoString:NULL]) {
    [attributeDataScanner scanString:@"face=\"" intoString:NULL];
    if ([attributeDataScanner scanUpToString:@"\"" intoString:&faceName]) {
      return faceName;
    }
  }
  return nil;
}
static CGFloat fontSizeFromString(NSString *attrData) {
  NSScanner *attributeDataScanner = [NSScanner scannerWithString:attrData];
  NSString *sizeString = nil;
  if ([attributeDataScanner scanUpToString:@"size=\"" intoString:NULL]) {
    [attributeDataScanner scanString:@"size=\"" intoString:NULL];
    if ([attributeDataScanner scanUpToString:@"\"" intoString:&sizeString]) {
      return [sizeString floatValue];
    }
  }
  return 0.0;
}
static UIColor *fontColorFromString(NSString *attrData) {
  return nil;
}
```

You'll notice that the third method, `fontColorFromString()`, isn't shown in a completed form here. In the interests of time and space, and not wandering too far afield from our main topic, let's leave that as an exercise for the reader, shall we?

www.it-ebooks.info

With this in place, we now have a way to define some characteristics of the text we enter! Build and run Dudel, and create some new objects using the Text tool to try it out. Here are some suggestions for putting it through its paces:

- Create a paragraph with some `<font size=”64”>really big text</font>` and then more normal-sized text.
- Try sticking some `<font face=”Courier”>Courier into the mix</font>` to see how multiple fonts are rendered

Mix and match these however you like. Our parser is far from perfect, and throwing something like nested `font` tags at it will probably confuse it, but at least it’s something!

The Lessons of Core Text

In this chapter, you learned how to use Core Text to render text, including some styled text. You also saw how this works in the context of a real application, and how you can leverage different components to achieve a decent user experience, by using a tried-and-tested input control (`UITextView`) to let the user enter text that’s displayed in a different way by your own component. Understanding when to use the components included in Cocoa Touch, and how to make them work together with your own components, is really important for building larger, more complex applications.

In Chapter 6, you’ll see how to give the user a whole lot more functionality than simple buttons in a toolbar will allow. This is possible through the use of the new `UIPopoverController` class.

www.it-ebooks.info

www.it-ebooks.info

Chapter 6

Popovers

Up until recently, the iOS user interface paradigm supported showing only a limited amount of material on the screen at any point in time. In a Cocoa Touch application, there's typically one view controller in focus at a time, and that view controller is in charge of the whole screen (or most of it). The notable exceptions are classes like `UINavigationController` and `UITabBarController`, which don't display any interesting content on their own, but instead help developers organize other view controllers.

On the small screen of the iPhone and the iPod touch, this makes a lot of sense. Instead of a profusion of tiny widgets fighting for space on the screen, iOS users have gotten used to being able to focus on one thing at a time, with new views sliding into place when on-screen objects or controls are used. This paradigm is so widely used that even controls that would take up just a small space on a desktop computer, such as a popup list, fill the iPhone's screen when you activate them. On the iPad, however, this behavior isn't always suitable. Sometimes, you need to display a little GUI in order to choose an option, such as from a popup list. Filling the larger iPad screen with a simple list of items would feel both unnatural and wasteful of that nice screen real estate!

The new `UIPopoverController` class in iOS 3.2 lets you display an auxiliary view that floats in front of the other on-screen content, without filling the entire screen. Like the `UINavigationController` and `UITabBarController`, `UIPopoverController` doesn't display any interesting content on its own. Instead, it serves an organizational role and acts as a container for your own view controllers.

In this chapter, you'll learn how to use `UIPopoverController` in a variety of ways. We'll add popover views to Dudel for setting fonts, stroke width, and colors.

Popover Preparations

So far, Dudel serves as a nice demo of a few features, but it's extremely limited in comparison to the vector-drawing applications that have been around for decades. One of the main features it lacks is the ability to change the properties of what you're drawing. Right now, you're stuck with the line width, stroke and fill colors, and font that the app gives you from the outset. It's time to change all that!

www.it-ebooks.info

In this chapter, we're going to create GUIs that let users change all those attributes, giving users much more control over their creations. Each of these attributes requires a little different approach to setting them, and therefore a different sort of GUI:

- Selecting a font will occur through a simple list that displays the name of each font, rendered in that font itself.
- The font's size will be set using a slider in a popup, with a preview showing a piece of text rendered at the chosen size.
- A popup with a slider will let you set the line width, again with a built-in preview.
- Another popup will let you choose the fill and stroke colors from a predefined grid of colors.

The idea is for you to learn several ways that popovers can be used in a real application, starting with the simplest type and working up to more complicated examples.

Before we proceed, let's clarify a point about the concept of a selected object, and the context to which the attributes you set will be applied. Most vector-drawing applications include some sort of selector tool that lets you click an object you've drawn, which then becomes highlighted and editable in some way. Any changes you make to color settings, line width, and so on are typically applied immediately to the selected object. In Dudel, however, we have none of that. There's never a selected object, and therefore never any visible item to which your attribute settings are applied. Instead, the settings are remembered in a central spot (the `DudelViewController` class), where they will be used for the *next* thing you draw.

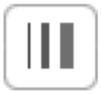
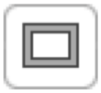
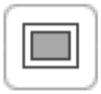
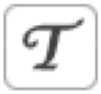
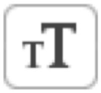
The Basic GUI

Before you begin making any changes, make a copy of your entire project directory. Or, if you haven't been following along in earlier chapters, grab a fresh copy of the completed Chapter 5 project from the book's source code archive and work from there.

Let's start off by making some modifications to the main GUI in `DudelViewController`. We're going add a set of new `UIBarButtonItems` at the bottom of the screen for the popovers, each with a new icon. Unlike the icons for the tools we created earlier, these don't need to have any sort or highlighting state, so just a single icon for each is fine. Table 6-1 shows the icons you'll need for this chapter. You'll find these in the Chapter 6 project from the book' source code archive, or use your own creations if you prefer. Add these images, using the filenames listed in Table 6–1, to your project.

www.it-ebooks.info

Table 6–1. *New Buttons for the Popovers*

Filename	Image
button_strokewidth.png	
button_strokecolor.png	
button_fillcolor.png	
button_fontname.png	
button_fontsize.png	

Now let's add action methods to our controller class's interface for connecting these buttons. Open *DudelViewController.h*, and somewhere near the end of the file, but before the `@end` line, add the following lines:

```
- (IBAction)popoverFontName:(id)sender;
- (IBAction)popoverFontSize:(id)sender;
- (IBAction)popoverStrokeWidth:(id)sender;
- (IBAction)popoverStrokeColor:(id)sender;
- (IBAction)popoverFillColor:(id)sender;
```

Then, just to keep our code in a compilable state, switch over to *DudelViewController.m* and insert some empty implementations for those methods inside the `@implementation DudelViewController` section:

```
- (IBAction)popoverFontName:(id)sender {
}
- (IBAction)popoverFontSize:(id)sender {
}
- (IBAction)popoverStrokeWidth:(id)sender {
}
- (IBAction)popoverStrokeColor:(id)sender {
}
- (IBAction)popoverFillColor:(id)sender {
}
```

We'll go back and fill in the implementations of those methods a little later, but first we want to hook up the GUI. Save your work, and then open *DudelViewController.xib* in Interface Builder. We'll add the new buttons as a group, between the group of tools on the left and the e-mail action on the right, as shown in Figure 6–1.

www.it-ebooks.info

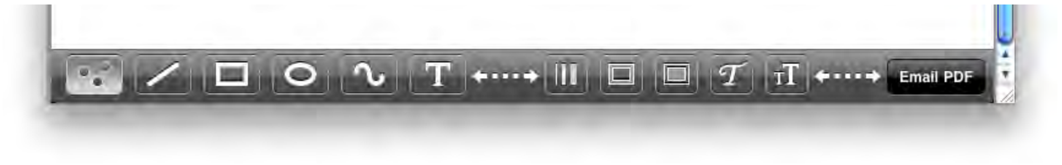


Figure 6–1. *Positioning the settings buttons*

First, duplicate the flexible space object in place, by selecting it and pressing ⌘D. That will give you a location to put more buttons. Then use the Library to find a `UIBarButtonItem` and drag it out between the two flexible spaces. Next, open the attribute inspector. Set the new item's Style to Plain, and set its image to *button_strokewidth.png*. This gives us the basic template for how all five buttons will appear. While the new item is still selected, press ⌘D four times to make a row of five identical items.

Now we need to add the actions and images to the buttons. Select the leftmost item, control-drag to the File's Owner icon in the main *.nib* window, and click `popoverStrokeWidth:` in the list of actions that appears. Then go along the rest of the row, configuring each item's action and connecting it to the appropriate image. The second item should get the *popover_strokecolor.png* image and be connected to `popoverStrokeColor:`. The third should use *popover_fillcolor.png* and `popoverFillColor:`. The last two items are for choosing a font name and font size, and I'll bet that by now, you can figure out which images and actions to use for them.

Popover Considerations

One of the main uses for popovers is to present a list of selectable items, not unlike the menus available in Mac OS X and other desktop operating systems. When using menus in a Mac OS X application, the system takes care of things such as making sure that only one menu is shown at a time and making the menu disappear when an item is selected. But the popover in iOS is a different beast.

A popover won't automatically disappear when the user selects something inside it, and opening one popover doesn't remove any previously opened popover from the screen. This means that you could easily wind up with multiple popovers on the screen at once, overlapping each other.

The only time the system automatically closes a popover is when you touch some part of the screen outside the popover (except, notably, touching an item in a `UIToolbar`, which leaves the popover just as it is). The rest of the time, you'll need to dismiss the popover yourself any time a user action warrants it.

However, this apparent lack of automation actually gives you some amount of flexibility compared to what's typically possible with a menu. A popover can, for instance, contain interactive controls, such as sliders or check boxes, to let the user quickly try out different possibilities and see the results instantly. That wouldn't be possible if the popover went away as soon as someone clicked it. Similarly, allowing multiple popovers

www.it-ebooks.info

to be displayed simultaneously may be useful in situations where you want to let the user quickly change multiple settings or attributes. For example, in a word processing app, you might want to let the user open two popovers: one for selecting from a list of fonts, and one for toggling attributes (bold, italics, underline, and so on).

NOTE: Apple recommends against displaying multiple popovers at once, in order to avoid "confusing" your users, so think twice before going that route.

In Dudel, we're going to allow for only one popover at a time by keeping an instance variable in DudelViewController that points at the current popover, and taking steps to make sure that it's properly managed. Start off by editing *DudelViewController.h*, adding the following code shown in bold. In addition to adding the instance variable (and its matching property declaration), here we're also adding UIPopoverControllerDelegate to the list of protocols this class implements.

```
@interface DudelViewController : UIViewController <ToolDelegate, DudelViewDelegate,
MFMailComposeViewControllerDelegate, UIPopoverControllerDelegate> {
  id <Tool> currentTool;
  IBOutlet DudelView *dudelView;
  IBOutlet UIBarButtonItem *textButton;
  IBOutlet UIBarButtonItem *freehandButton;
  IBOutlet UIBarButtonItem *ellipseButton;
  IBOutlet UIBarButtonItem *rectangleButton;
  IBOutlet UIBarButtonItem *lineButton;
  IBOutlet UIBarButtonItem *dotButton;
  UIColor *strokeColor;
  UIColor *fillColor;
  UIFont *font;
  CGFloat strokeWidth;
  UIPopoverController *currentPopover;
}

@property (retain, nonatomic) id <Tool> currentTool;
@property (retain, nonatomic) UIColor *strokeColor;
@property (retain, nonatomic) UIColor *fillColor;
@property (retain, nonatomic) UIFont *font;
@property (assign, nonatomic) CGFloat strokeWidth;
@property (retain, nonatomic) UIPopoverController *currentPopover;
```

Follow up by switching over to *DudelViewController.m* to synthesize the currentPopover property, and clean it up in the dealloc method.

@synthesize currentTool, fillColor, strokeColor, font, strokeWidth, **currentPopover**;

```
- (void)dealloc {
  self.currentTool = nil;
  self.fillColor = nil;
  self.strokeColor = nil;
  self.currentPopover = nil;
  [super dealloc];
}
```

www.it-ebooks.info

One tricky aspect of dealing with popovers has to do with cleanup after a popover has been dismissed. If the user clicked outside the popover, causing it to be automatically dismissed, then a method will be called in the `UIPopoverController`'s delegate. But if you dismiss the popup from within code, that method *isn't* called. We'll handle this discrepancy by just making the delegate method call our own cleanup method, `handleDismissedPopoverController:`, which we'll be careful to call every time we dismiss a popup manually.

```
- (void)handleDismissedPopoverController:(UIPopoverController*)popoverController {
  self.currentPopover = nil;
}
- (void)popoverControllerDidDismissPopover:(UIPopoverController *)popoverController {
  [self handleDismissedPopoverController:popoverController];
}
```

As you can see, our current cleanup method doesn't do much cleanup yet, but that will change!

The main thing we're going to need to do in our cleanup method, besides clearing our `currentPopover` instance variable, is to get whatever values we need from the popover's displayed controller. In Dudel, we'll implement this by checking for the specific classes we're using for the view controllers. So the `handleDismissedPopoverController:` method will end up containing a series of `if/else` blocks, like this:

```
// just for explanatory purposes, not for copy-and-paste!
if ([popoverController.contentViewController isMemberOfClass:[SomeController class]]) {
  // now we know which view controller we're dealing with
  SomeController *sc = (SomeController *)popoverController.contentViewController;
  // retrieve some values from the controller, to see what the user selected/adjusted
  self.something = sc.something;
  ...
} else if (...)
```

Yes, I agree that this sort of `if/else` pileup is distasteful. But it's the simplest solution in this case, and our project is small enough that it's not introducing too much painful ugliness.

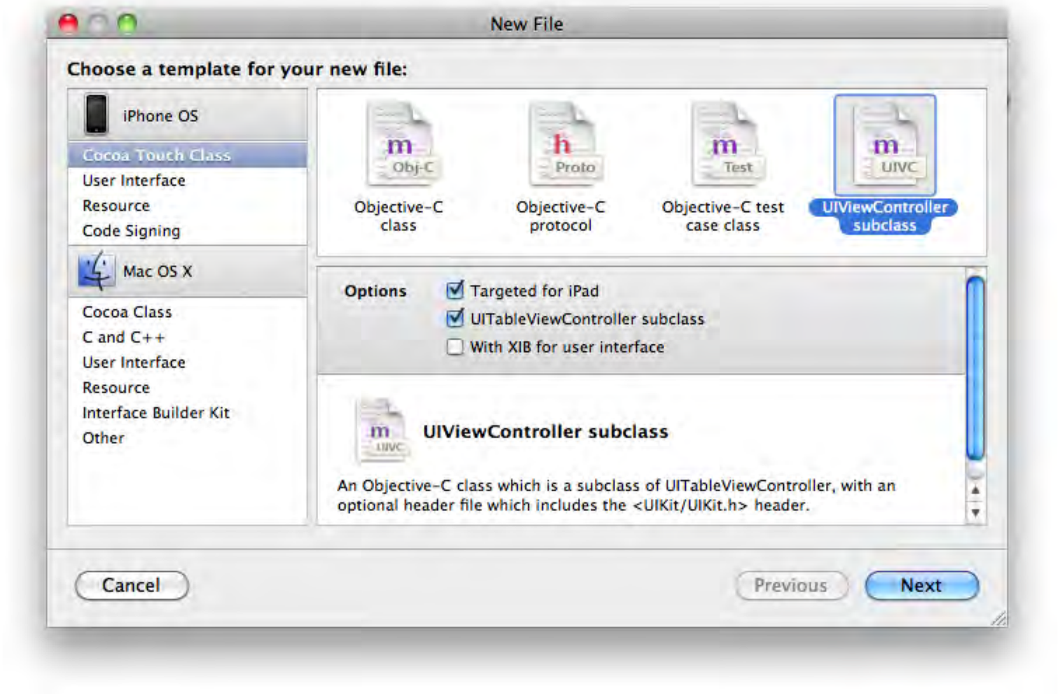
The Font Name Popover

The first popover we're going to create is for choosing the font used by the Text tool. We'll simply display a list of all available font names on the device; specifying the size will be the job of our next popover. What we need here is a view controller that will let us display a list of selectable items, and this list may be larger than the screen itself, so it should be scrollable—sounds like a job for a `UITableView`!

Add a new class to your project, and use the assistant that comes up to specify that you want a Cocoa Touch class, specifically a `UIViewController` subclass. If you've ever created a `UIViewController` subclass for iPhone in the past—and I suspect you have—this should look pretty familiar. The main difference is the inclusion of a new Targeted for iPad check box. If that's checked, Xcode will use a slightly different template for creating your class. Go ahead and make sure that's turned on, along with the

www.it-ebooks.info

UITableViewController subclass check box, but not the XIB check box, as shown in Figure 6–2. Click Next, and then enter FontListController as the name of your new class.



Options
- [✓] Targeted for iPad
- [✓] UITableViewController subclass
- [ ] With XIB for user interface

UIViewController subclass

An Objective-C class which is a subclass of UITableViewController, with an optional header file which includes the <UIKit/UIKit.h> header.

Figure 6–2. *Creating a new controller class*

The Simplest Popover You'll Ever Create

Thanks to the power and flexibility of UITableView, creating this class is going to be a breeze. All we need to do is add a few instance variables for hanging onto a list of fonts as well as the current selection, and fill in a few short methods in the controller class.

Start with *FontListController.h*, adding the instance variables and matching properties as shown in the following code. We're also defining a string constant, which will be used to let the main view controller know that the user has selected something.

```
#import <UIKit/UIKit.h>
// we'll use a notification with this name, to let the main
// view controller know that something was selected here.
#define FontListControllerDidSelect @"FontListControllerDidSelect"
@interface FontListController : UITableViewController {
  NSArray *fonts;
  NSString *selectedFontName;
  UIPopoverController *container;
}
@property (retain, nonatomic) NSArray *fonts;
@property (copy, nonatomic) NSString *selectedFontName;
```

www.it-ebooks.info

```
@property (assign, nonatomic) UIPopoverController *container;
@end
```

As you can see, we also created an instance variable for pointing to the `UIPopoverController` that acts as the container for an instance of this class. `FontListController` doesn't have any use for this itself, but it will be used later when `DudelViewController` needs to close the containing `UIPopoverController`.

Now switch over to *FontListController.m*, where we have a series of small changes to make to the default template. Apart from the changes shown here, you can leave the rest of the autogenerated class as is. First, synthesize all the declared properties by adding this line inside the `@implementation FontListController` section:

```
@synthesize fonts, selectedFontName, container;
```

Then uncomment the `viewDidLoad` method, remove most of the code in there (except for the call to `[super viewDidLoad]`), and add the bold lines shown here to its body:

```
- (void)viewDidLoad {
  [super viewDidLoad];
  NSArray *familyNames = [UIFont familyNames];
  NSMutableArray *fontNames = [NSMutableArray array];
  for (NSString *family in familyNames) {
    [fontNames addObjectsFromArray:[UIFont fontNamesForFamilyName:family]];
  }
  self.fonts = [fontNames sortedArrayUsingSelector:@selector(compare:)];
}
```

In a nutshell, this goes through an array of strings containing font family names, gets all the fonts that belong to each family, and adds them to an array. Finally, it puts them in alphabetical order and saves the sorted array in the `fonts` instance variable.

Next, uncomment the `viewWillAppear:` method, and add the bold lines shown here:

```
- (void)viewWillAppear:(BOOL)animated {
  [super viewWillAppear:animated];
  NSInteger fontIndex = [self.fonts indexOfObject:self.selectedFontName];
  if (fontIndex != NSNotFound) {
    NSIndexPath *indexPath = [NSIndexPath indexPathForRow:fontIndex inSection:0];
    [self.tableView scrollToRowAtIndexPath:indexPath
        atScrollPosition:UITableViewScrollPositionMiddle animated:NO];
  }
}
```

This code tries to find the location of the selected font in the array of fonts, and scrolls the table view to make it visible. This works under the assumption that the code that initializes this class (which we'll add to `DudelViewController` soon) also sets the `selectedFontName` property. The check against `NSNotFound` makes sure that we don't crash in case that value hasn't been set or is set to something invalid (a font name that isn't in our list).

Next, we fill in the blanks for the basic `UITableViewDatasource` methods that every `UITableViewController` subclass must implement:

```
- (NSInteger)numberOfSectionsInTableView:(UITableView *)tableView {
```

www.it-ebooks.info

```
    // Return the number of sections.
    return 1;
}

- (NSInteger)tableView:(UITableView *)tableView numberOfRowsInSection:(NSInteger)section
{
    // Return the number of rows in the section.
    return [fonts count];
}

// Customize the appearance of table view cells.
- (UITableViewCell *)tableView:(UITableView *)tableView
cellForRowAtIndexPath:(NSIndexPath *)indexPath {

  static NSString *CellIdentifier = @"Cell";

  UITableViewCell *cell = [tableView dequeueReusableCellWithIdentifier:CellIdentifier];
  if (cell == nil) {
    cell = [[[UITableViewCell alloc] initWithStyle:UITableViewCellStyleDefault
reuseIdentifier:CellIdentifier] autorelease];
  }

  // Configure the cell...
  NSString *fontName = [fonts objectAtIndex:indexPath.row];
  cell.textLabel.text = fontName;
  cell.textLabel.font = [UIFont fontWithName:fontName size:17.0];
  if ([self.selectedFontName isEqual:fontName]) {
    cell.accessoryType = UITableViewCellAccessoryCheckmark;
  } else {
    cell.accessoryType = UITableViewCellAccessoryNone;
  }
  return cell;
}
```

The first two methods are self-explanatory, and the last one isn't much more complicated. It just sees which font name is at the specified index, and uses that name both as the display value and to look up a font. That way, each font name is displayed in its own font! It also sets a check box on the cell if (and only if) the current font name matches the selected font, so the user can see the current selection while scrolling through the list.

Next, we're going to implement the method that's called when the user selects a row. The idea is to make a note of which font the user selected, update the display of the affected rows (so the check box appears in the correct cell) to give the user some immediate feedback, and then post a notification so that whoever is listening, such as `DudelViewController`, will get a chance to do something. This method already exists in the template code, but contains some commented-out example code that isn't relevant here. Delete that, and add the code shown in bold:

```
- (void)tableView:(UITableView *)tableView didSelectRowAtIndexPath:(NSIndexPath
*)indexPath {
  // determine two affected table columns: the one that was selected before,
  // and the one that's selected now.
  NSInteger previousFontIndex = [self.fonts indexOfObject:self.selectedFontName];
```

www.it-ebooks.info

```
  // don't do any updating etc. if the user touched the already-selected row
  if (previousFontIndex != indexPath.row) {
    NSArray *indexPaths = nil;
    if (previousFontIndex!= NSNotFound) {
      NSIndexPath *previousHighlightedIndexPath = [NSIndexPath
        indexPathForRow:previousFontIndex inSection:0];
      indexPaths = [NSArray arrayWithObjects:indexPath, previousHighlightedIndexPath,
nil];
    } else {
      indexPaths = [NSArray arrayWithObjects:indexPath, nil];
    }

    // notice the new selection
    self.selectedFontName = [self.fonts objectAtIndex:indexPath.row];

    // then reload
    [self.tableView reloadRowsAtIndexPaths:indexPaths
      withRowAnimation:UITableViewRowAnimationFade];
    [[NSNotificationCenter defaultCenter]
postNotificationName:FontListControllerDidSelect
      object:self];
  }
}
```

Finally, we need to add a bit of cleanup, so that the list of font names doesn't hang around forever:

```
- (void)viewDidUnload {
  // relinquish ownership of anything that can be re-created in viewDidLoad or on
demand.
  // For example: self.myOutlet = nil;
  self.fonts = nil;
}

- (void)dealloc {
  self.fonts = nil;
  self.selectedFontName = nil;
  [super dealloc];
}
```

That should be all we need for the `FontListController` class itself. At this point, you should try to build your app, just to make sure no syntax errors have snuck in, but you won't see any difference when you run the app just yet. Our next step here will be enabling `DudelViewController` to use our new class.

The Back End

Now it's time to implement the portions of `DudelViewController` that will fire up the `FontListController`, dismiss its popover when the user makes a selection, and grab the selected value. Start with an import:

```
#import "FontListController.h"
```

www.it-ebooks.info

Then fill in this previously empty method:

```
- (IBAction)popoverFontName:(id)sender {
  FontListController *flc = [[[FontListController alloc]
    initWithStyle:UITableViewStylePlain] autorelease];
  flc.selectedFontName = self.font.fontName;
  [self setupNewPopoverControllerForViewController:flc];
  flc.container = self.currentPopover;
  [[NSNotificationCenter defaultCenter] addObserver:self
    selector:@selector(fontListControllerDidSelect:)
    name:FontListControllerDidSelect
    object:flc];
  [self.currentPopover presentPopoverFromBarButtonItem:sender
    permittedArrowDirections:UIPopoverArrowDirectionAny
    animated:YES];
}
```

This method creates and configures a `FontListController` instance. Part of the configuration involves calling a method named `setupNewPopoverControllerForViewController:` (which we're going to create in just a minute). It also sets us up as an observer for a notification, which will tell us that the user selected something, and then displays a popover.

What's not really clear here is the final line, which contains `self.currentPopover`. We haven't set that, have we? Well, the following auxiliary method does! Insert this method somewhere above all the popover action methods:

```
- (void)setupNewPopoverControllerForViewController:(UIViewController *)vc {
  if (self.currentPopover) {
    [self.currentPopover dismissPopoverAnimated:YES];
    [self handleDismissedPopoverController:self.currentPopover];
  }
  self.currentPopover = [[[UIPopoverController alloc] initWithContentViewController:vc]
    autorelease];
  self.currentPopover.delegate = self;
}
```

We'll use this method every time we're going to present a popover. By doing so, we save a few lines in each popover action method, and also ensure that we're doing things the same way each time. Now, I'm not going to pretend that this separate method sprang from my forehead in one piece. The fact is that I had this code, or something very much like it, inside each action method as I was working on this chapter. Eventually, I realized that there was a sizable chunk that was identical in each method, and refactored it into a method on its own.

TIP: Any time you find yourself doing cut-and-paste coding, consider chunking things off into separate methods, because someone is probably going to revisit your code someday. The mind you save may be your own.

www.it-ebooks.info

So now we have code in place to fire up the popover, but we still need to handle the user selecting something. Begin by creating a method to be called when the notification we're observing is triggered:

```
- (void)fontListControllerDidSelect:(NSNotification *)notification {
  FontListController *flc = [notification object];
  UIPopoverController *popoverController = flc.container;
  [popoverController dismissPopoverAnimated:YES];
  [self handleDismissedPopoverController:popoverController];
  self.currentPopover = nil;
}
```

Here, you see the reason for putting the `container` property in `FontListController`. After the user makes a selection, `FontListController` shoots off a notification. `DudelViewController` picks it up, and uses the container property to dismiss the popover. This method also calls the main handler for all our popovers, which you should now revise to this:

```
- (void)handleDismissedPopoverController:(UIPopoverController*)popoverController {
  if ([popoverController.contentViewController isMemberOfClass:
       [FontListController class]]) {
    // this is the font list, grab the new selection
    FontListController *flc = (FontListController *)
      popoverController.contentViewController;
    self.font = [UIFont fontWithName:flc.selectedFontName size:self.font.pointSize];
  }
  self.currentPopover = nil;
}
```

This is where the selection in the popover finally ends up having an effect. The font is now set according to what the user picked.

You should now be able to build and run the app, and then touch the font list button in the toolbar to see something like the popup shown in Figure 6–3.

Try selecting a different font, and then using the Text tool to create some text. Neat! You now have the full complement of fonts included with the iPad at your disposal. You’re still stuck with just one size, so let’s tackle that next.

www.it-ebooks.info

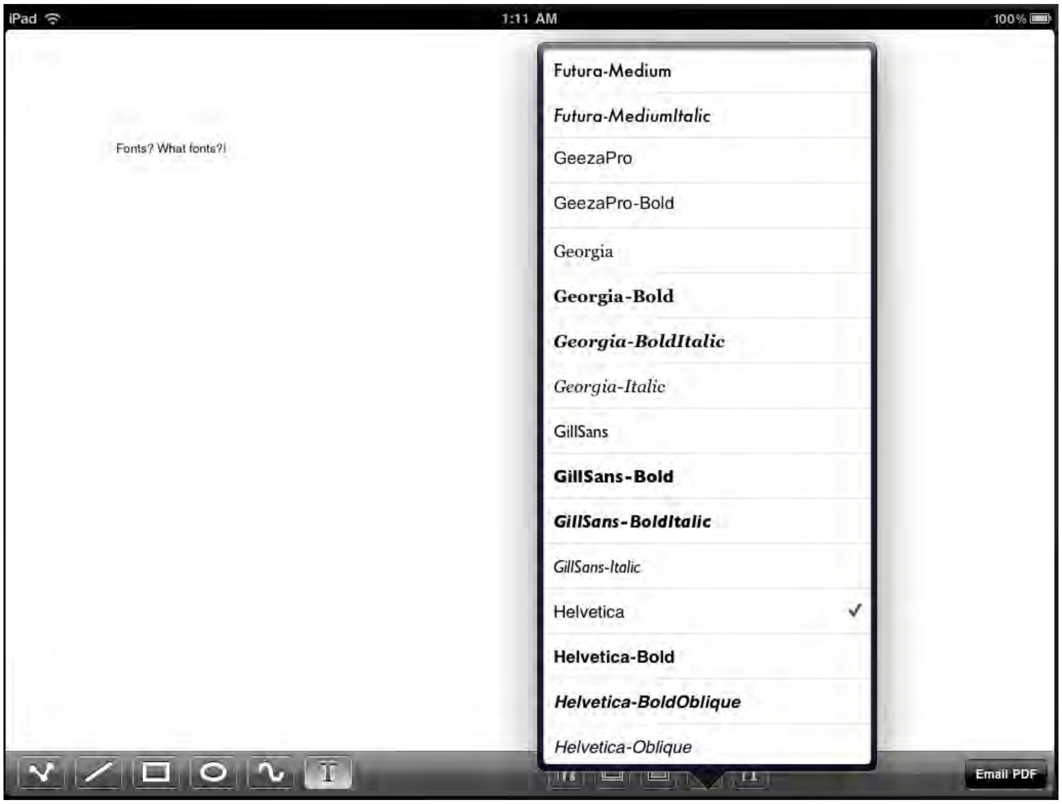


Figure 6–3. *Selecting a font*

The Font Size Popover

To keep the font selector simple, we'll have the size selection as a separate operation, in its own popover. The GUI for the font size selector will consist of a slider, a label showing the slider's value, and a text view showing a preview of the chosen font at the chosen size. As the user operates the slider, the preview immediately reflects the slider's value. Figure 6–4 shows the GUI in action.

www.it-ebooks.info

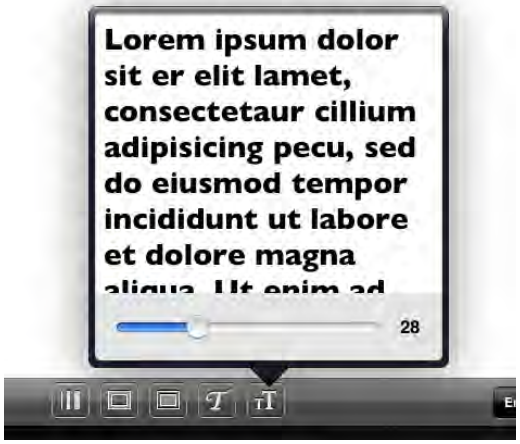

Lorem ipsum dolor sit er elit lamet, consectetaur cillium adipisicing pecu, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad

Figure 6–4. *Setting the font size, with live preview*

Unlike the font list, the font size popover shouldn't go away as soon as the user touches it—that would be a pretty surprising response from touching a slider. That means that the interaction between `DudelViewController` and this new popover will be a little simpler than it was for the font list, since the font size popover will never need to be explicitly dismissed in our code. We'll rely on the system to dismiss it when the user clicks outside it.

Creating the GUI

In Xcode, use the New File Assistant to make a new Cocoa Touch class. Select `UIViewController` as its superclass, and configure the check boxes to target iPad and create an *.xib* file, but to *not* make it a subclass of `UITableViewController`. Name the new class `FontSizeController`.

Start by editing *FontSizeController.h*, which will contain outlets for each GUI object we need to interact with, an instance variable containing the current chosen font, and an action method for the slider to call. Here's the entire content of the file:

```
// FontSizeController.h
#import <UIKit/UIKit.h>
@interface FontSizeController : UIViewController {
  IBOutlet UITextView *textView;
  IBOutlet UISlider *slider;
  IBOutlet UILabel *label;
  UIFont *font;
}
@property (retain, nonatomic) UIFont *font;
- (void)takeIntValueFrom:(id)sender;
@end
```

www.it-ebooks.info

Now open *FontSizeController.xib* in Interface Builder. The first thing you'll notice is that the default view contained within is the size of the entire iPad screen. That's way too big for our purposes! It's also showing a black status bar at the top, as if it were a full-screen view.

Select the view, and use the attribute inspector to set the Status Bar value to Unspecified. Then use the size inspector to set its size to 320 by 320. You actually need to do these two steps in that order, since if the Status Bar value is set to Black, Interface Builder wants to treat that view as a full-screen view, and won't let you change its size.

NOTE: Apple's documentation says that the minimum width for a popover is 320 pixels.

Now use the Library to grab a `UITextView`, a `UISlider`, and a `UILabel`, and put them each into the view, laid out something like what you saw in Figure 6–4. In my version, I've broken up the monotony of the white background by setting the main view's background color to light gray. You can skip this step if you like, or use a different background color.

Next, connect each outlet from File's Owner to one of the GUI objects by control-clicking that icon, dragging to a GUI object, and selecting the appropriate outlet from the list that appears. Then connect the slider's target and action by control-clicking the slider, dragging to File's Owner, and selecting the `takeIntValueFrom:` method.

Select the `UITextView` you created earlier. This contains some "Lorem ipsum" text by default, which is fine for our purposes, so leave that bit alone. However, we don't want users interacting with this text view, so use the attribute inspector to turn off the Editable check box, as well as all of the check box options that affect scrolling behavior.

Finally select the slider, and set its minimum and maximum values to 1 and 96, respectively. The GUI is now complete! Save your work, and go back to Xcode.

Making It Work

Now it's time for the `FontListController` implementation, which is shown here in an abbreviated form after deleting extra comments and unneeded overrides from the template:

```
//  FontSizeController.m
#import "FontSizeController.h"
@implementation FontSizeController
@synthesize font;
// Implement viewDidLoad to do additional setup after loading the view, typically from a
nib.
- (void)viewDidLoad {
  [super viewDidLoad];
  textView.font = self.font;
  NSInteger i = self.font.pointSize;
  label.text = [NSString stringWithFormat:@"%d", i];
  slider.value = i;
}
```

www.it-ebooks.info

```
- (BOOL)shouldAutorotateToInterfaceOrientation:
    (UIInterfaceOrientation)interfaceOrientation {
    // Overridden to allow any orientation.
    return YES;
}
- (void)dealloc {
  self.font = nil;
  [super dealloc];
}
- (void)takeIntValueFrom:(id)sender {
  NSInteger size = ((UISlider *)sender).value;
  self.font = [self.font fontWithSize:size];
  textView.font = self.font;
  label.text = [NSString stringWithFormat:@"%d", size];
}
@end
```

As you can see, this class is quite simple. It basically just responds to the user dragging the slider by modifying the `font` property and updating the display.

Now all that's left to do is integrate this new class with the view controller. Switch back to *DudelViewController.m*, where we have a few changes to make. Add this near the top:

```
#import "FontSizeController.h"
```

Then update the `handleDismissedPopoverController:` method like this:

```
- (void)handleDismissedPopoverCo2ntroller:(UIPopoverController*)popoverController {
  if ([popoverController.contentViewController isMemberOfClass:
    [FontListController class]]) {
    // this is the font list, grab the new selection
    FontListController *flc = (FontListController *)
      popoverController.contentViewController;
    self.font = [UIFont fontWithName:flc.selectedFontName size:self.font.pointSize];
  } else if ([popoverController.contentViewController isMemberOfClass:
    [FontSizeController class]]) {
    FontSizeController *fsc = (FontSizeController *)
      popoverController.contentViewController;
    self.font = fsc.font;
  }
  self.currentPopover = nil;
}
```

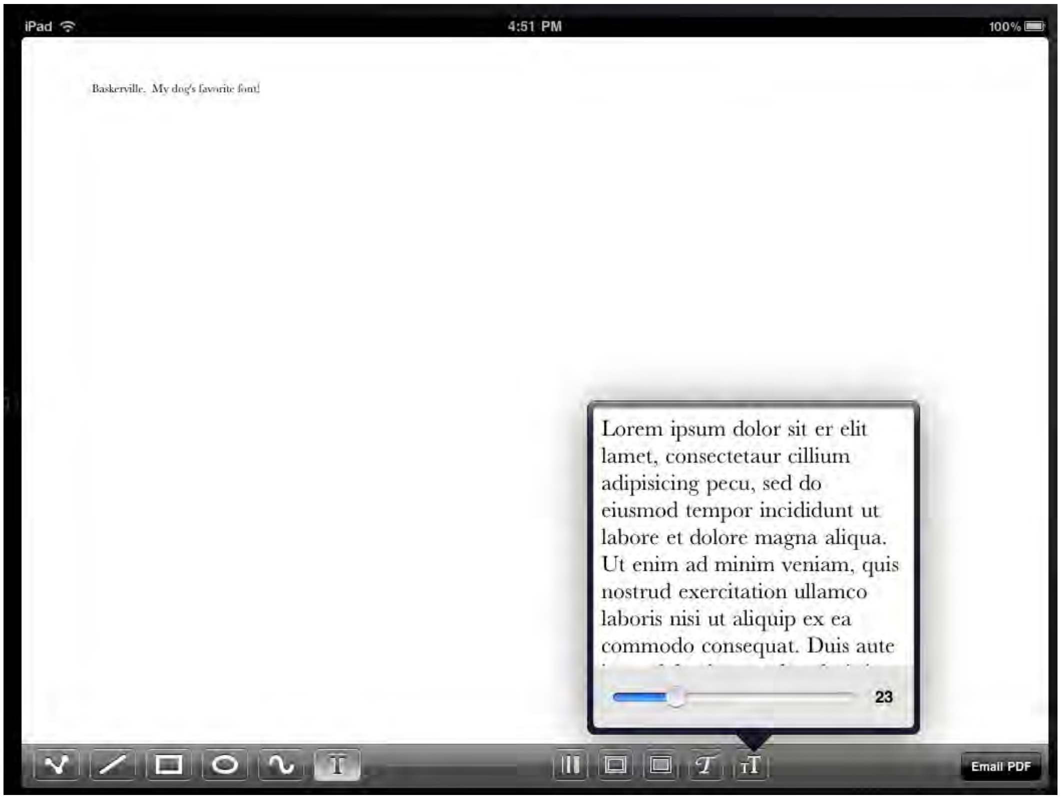
After that, we just need to fill in the `popoverFontSize:` method, like this:

```
- (IBAction)popoverFontSize:(id)sender {
  FontSizeController *fsc = [[[FontSizeController alloc] initWithNibName:nil bundle:nil]
    autorelease];
  fsc.font = self.font;
  [self setupNewPopoverControllerForViewController:fsc];
  self.currentPopover.popoverContentSize = fsc.view.frame.size;
  [self.currentPopover presentPopoverFromBarButtonItem:sender
    permittedArrowDirections:UIPopoverArrowDirectionAny animated:YES];
}
```

www.it-ebooks.info

The one new thing we're doing in this method is setting the `popoverContentSize` property on the `currentPopover`. If we didn't do this, the popup would automatically fill the maximum height of the screen.

With that in place, we're finished with the font size selection popup. Build and run the app, and you should see something like Figure 6–5. Notice that since this popover knows about the complete `UIFont` object that is currently set, not just the size, we can display the font preview with the correct font and size.



Baskerville. My dog's favorite font!

Lorem ipsum dolor sit er elit lamet, consectetaur cillium adipisicing pecu, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute

Figure 6–5. *Setting a font size*

Now we have pretty good control over the fonts we're using for the Text tool. This is still far from a word processor or page layout app, but it's a pretty decent start, especially considering how little code we've written!

The Stroke Width Popover

Next up is the popover for setting the stroke width. This one is pretty similar to the one for font size. We'll give the user a slider to drag back and forth for setting the width, as well as a preview. This time, the preview will draw a few lines and curves in a

www.it-ebooks.info

UIBezierPath just like the ones the user can make, clearly showing the result of the user's selection.

Paving the Way

Start by creating yet another UIViewController subclass, using the New File Assistant. Just like FontSizeController, this one should *not* be a subclass of UITableViewController, but it *should* have an *.xib* file and be targeted for iPad. Name it StrokeWidthController. After Xcode creates it, open *StrokeWidthController.h* and give it the following content:

```
//  StrokeWidthController.h
#import <UIKit/UIKit.h>
@class StrokeDemoView;
@interface StrokeWidthController : UIViewController {
  IBOutlet UISlider *slider;
  IBOutlet UILabel *label;
  IBOutlet StrokeDemoView *strokeDemoView;
  CGFloat strokeWidth;
}
@property (assign, nonatomic) CGFloat strokeWidth;
- (void)takeIntValueFrom:(id)sender;
@end
```

This class is pretty similar to FontSizeController. The main differences are that here, we're keeping track of a simple floating-point value for the width, and we're also referencing a new class we're about to create, called StrokeDemoView, which we'll use to display the preview of the selected stroke width.

Before we create the GUI, we also need to create the StrokeDemoView class. Using the New File Assistant once again, make a new UIView subclass and name it StrokeDemoView. Just creating the class in our project is all we need to do in order to make Interface Builder know about the class and let us use it in the GUI. We'll go back and fill in the actual content later.

Creating the GUI

To begin, open *StrokeWidthController.xib* in Interface Builder. Once again, you'll see that the UIView it contains is meant to take up an entire screen, which isn't what we want here either. Use the attribute inspector to set the view's Status Bar to Unspecified, and then use the size inspector to make its size 320 by 320.

Now use the Library to find the three classes that are needed for our GUI: UISlider, UILabel, and StrokeDemoView. Drag each of them to the view, laying them out as shown in Figure 6–6.

www.it-ebooks.info

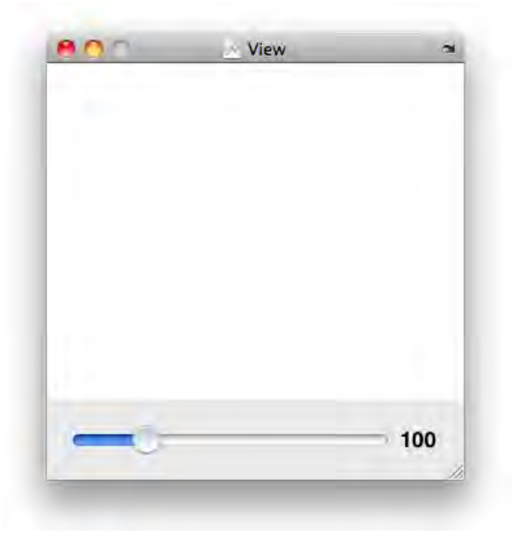


Figure 6–6. *Creating the GUI for StrokeWidthController*

You can't really tell from the figure, but the large, white rectangle filling most of the view is an instance of `StrokeDemoView`. For best results with our preview-drawing code, make this view 320 by 257, since the `StrokeDemoView` is going to have hard-coded locations for the lines and curves it draws. Here, I've once again given the entire view a light-gray background, to make the control area stand out from the preview a bit. Use the attribute inspector to give the slider a reasonable range by setting its minimum value to 1 and its maximum value to 20.

Make all the connections described in the header file, by control-dragging from File's Owner to each of the GUI components and making the connection, then control-dragging from the slider back to File's Owner and selecting the `takeIntValueFrom:` action. Now the basic GUI configuration is complete, so let's return to Xcode and make it work!

Previewing the Stroke Width with a Custom View

Now we're going to define the `StrokeDemoView` class. This class will be pretty simple. It defines a property called `strokeWidth`, which determines how it draws its path. Our controller will set this each time the user moves the slider. *StrokeDemoView.h* looks like this:

```
//  StrokeDemoView.h
#import <UIKit/UIKit.h>
@interface StrokeDemoView : UIView {
  CGFloat strokeWidth;
  UIBezierPath *drawPath;
}
@property (assign, nonatomic) CGFloat strokeWidth;
@end
```

The implementation is also pretty simple. It defines the path to draw when it's initialized, and implements the `setStrokeWidth:` method in order to mark itself as "dirty" by calling `[self setNeedsDisplay]`, so the view is scheduled for redrawing. The `drawRect:` method simply draws the path. Here's the whole thing:

```
//  StrokeDemoView.m
#import "StrokeDemoView.h"
@implementation StrokeDemoView
@synthesize strokeWidth;
- (void)setStrokeWidth:(CGFloat)f {
  strokeWidth = f;
  drawPath.lineWidth = f;
  [self setNeedsDisplay];
}
- (id)initWithCoder:(NSCoder *)aDecoder {
  if ((self = [super initWithCoder:aDecoder])) {
    drawPath = [[UIBezierPath bezierPathWithRect:CGRectMake(10, 10, 145, 100)] retain];
    [drawPath appendPath:
      [UIBezierPath bezierPathWithOvalInRect:CGRectMake(165, 10, 145, 100)]];

    [drawPath moveToPoint:CGPointMake(10, 120)];
    [drawPath addLineToPoint:CGPointMake(310, 120)];

    [drawPath moveToPoint:CGPointMake(110, 140)];
    [drawPath addLineToPoint:CGPointMake(310, 200)];

    [drawPath moveToPoint:CGPointMake(100, 180)];
    [drawPath addLineToPoint:CGPointMake(310, 140)];

    [drawPath moveToPoint:CGPointMake(90, 200)];
    [drawPath addCurveToPoint:CGPointMake(300, 230)
                controlPoint1:CGPointMake(0, 0)
                controlPoint2:CGPointMake(-100, 300)];
  }
  return self;
}
- (void)dealloc {
  [drawPath dealloc];
  [super dealloc];
}
- (void)drawRect:(CGRect)rect {
  [[UIColor blackColor] setStroke];
  [drawPath stroke];
}
@end
```

Note that since this class is instantiated only from within an *.xib* file, and never directly in code, we implement `initWithCoder:` and not `initWithFrame:`. If we wanted to also be able to instantiate this class in code, we would need to implement the latter as well in order to create the path.

www.it-ebooks.info

Implementing the Controller

Now that we have a working StrokeDemoView, our next move is to go back and implement StrokeWidthController. This class is quite simple, and quite similar to the FontSizeController we built earlier. Here's the entire content of *StrokeWidthController.m*:

```
//  StrokeWidthController.m
#import "StrokeWidthController.h"
#import "StrokeDemoView.h"
@implementation StrokeWidthController
@synthesize strokeWidth;
- (void)viewDidLoad {
  [super viewDidLoad];
  NSInteger i = self.strokeWidth;
  strokeDemoView.strokeWidth = i;
  label.text = [NSString stringWithFormat:@"%d", i];
  slider.value = i;
}
-
(BOOL)shouldAutorotateToInterfaceOrientation:(UIInterfaceOrientation)interfaceOrientatio
n {
    // Overriden to allow any orientation.
    return YES;
}
- (void)takeIntValueFrom:(id)sender {
  NSInteger i = ((UISlider *)sender).value;
  self.strokeWidth = i;
  strokeDemoView.strokeWidth = self.strokeWidth;
  label.text = [NSString stringWithFormat:@"%d", i];
  slider.value = self.strokeWidth;
}
@end
```

Making it Work

Now all we need to do is update our main controller to make it aware of the new popover. Open *DudelViewController.m*, and start with an import:

```
#import "StrokeWidthController.h"
```

Once again, we update the handleDismissedPopoverController: method, this time grabbing the new stroke width after completion:

```
- (void)handleDismissedPopoverController:(UIPopoverController*)popoverController {
  if ([popoverController.contentViewController isMemberOfClass:
    [FontListController class]]) {
    // this is the font list, grab the new selection
    FontListController *flc = (FontListController *)
      popoverController.contentViewController;
    self.font = [UIFont fontWithName:flc.selectedFontName size:self.font.pointSize];
  } else if ([popoverController.contentViewController isMemberOfClass:
    [FontSizeController class]]) {
    FontSizeController *fsc = (FontSizeController *)
      popoverController.contentViewController;
```

www.it-ebooks.info

```
      self.font = fsc.font;
    } else if ([popoverController.contentViewController isMemberOfClass:
      [StrokeWidthController class]]) {
      StrokeWidthController *swc = (StrokeWidthController *)
        popoverController.contentViewController;
      self.strokeWidth = swc.strokeWidth;
    }
    self.currentPopover = nil;
}
```

And finally, we implement the action that sets it in motion:

```
- (IBAction)popoverStrokeWidth:(id)sender {
  StrokeWidthController *swc = [[[StrokeWidthController alloc] initWithNibName:nil
    bundle:nil] autorelease];
  swc.strokeWidth = self.strokeWidth;
  [self setupNewPopoverControllerForViewController:swc];
  self.currentPopover.popoverContentSize = swc.view.frame.size;
  [self.currentPopover presentPopoverFromBarButtonItem:sender
    permittedArrowDirections:UIPopoverArrowDirectionAny animated:YES];
}
```

Build and run the app, and try out the new popover. Now you can finally see what that `UIBezierPath` we defined in the `StrokeDemoView` class looks like! Figure 6–7 shows the stroke width popover in action.

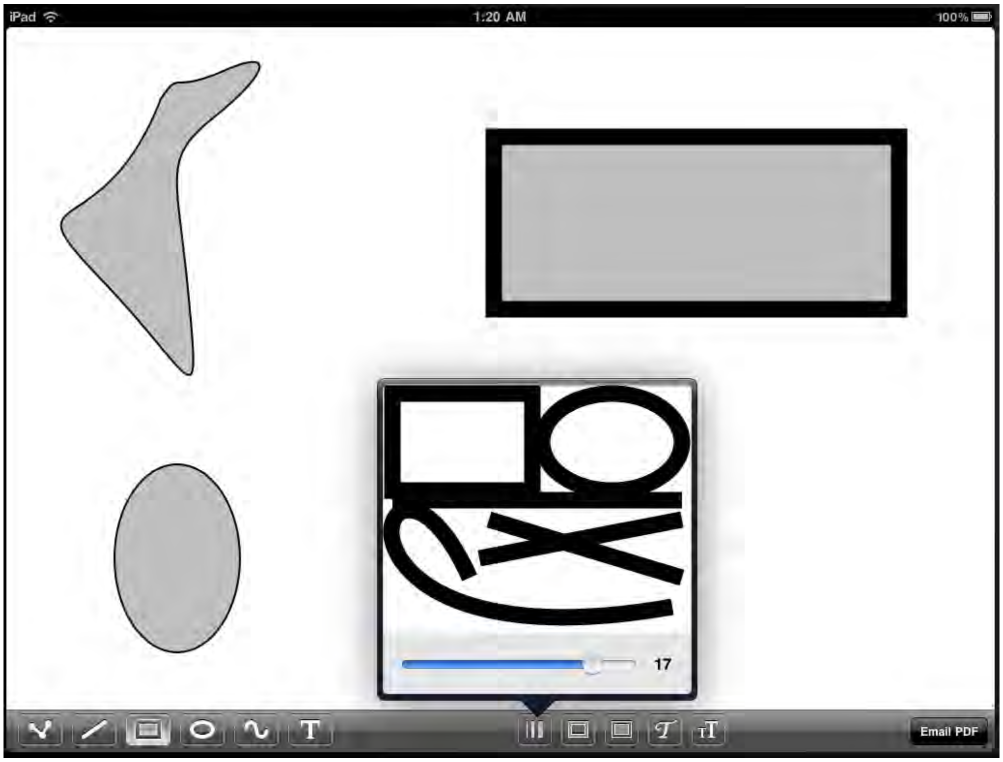


Figure 6–7. *Setting a stroke width*

www.it-ebooks.info

Drag the slider back and forth, and the stroke width changes. As we discussed, there's no concept of an active selection in Dudel, so changing this affects only the stroke width of the next graphic you draw, leaving the existing graphics unchanged.

The Dual-Action Color Popover

We're down to just two popovers left to implement, and they're both actually the same. What we need is a simple color picker that lets the user set colors for either stroke or fill, depending on which button is clicked.

NOTE: We wouldn't need to implement a color selector popover if iOS included some sort of color picker (along the lines of Mac OS X's `NSColorPanel`, for instance), but it currently does not.

Recall that our implementation of `DudelViewController` works by checking each dismissed popover by class to see which one it was. So, we'll implement the color selector GUI in one class, but use two subclasses to create the popovers, so that we can tell which is which when it's dismissed.

To keep things simple, we're just going to let the user pick from a simple grid that shows 12 colors, as shown in Figure 6–12. An additional view at the top of the GUI will show the currently selected color.

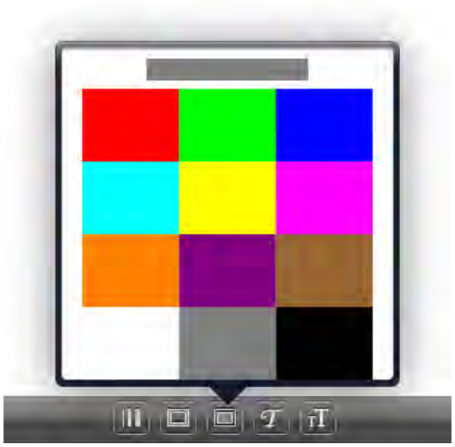


Figure 6–8. *Our simple color picker (I know you're probably seeing this in black and white, so please take my word for it when I tell you that those are colors.)*

As for the user interaction, it seems natural that this popover should have a "touch-and-dismiss" policy, unlike the stroke width and font size popovers, which hang around to let users move the slider multiple times until they got it just right. For the color selector, we'll let the users drag their finger around the grid, always displaying the latest color in the view at the top, and dismiss the popover as soon as they release their finger.

www.it-ebooks.info

Creating a Simple Color Grid

Let's start by making a view class that just knows how to display a grid of colors, and respond to touch events by sending notifications containing the touched color. Later, our view controller class will register as an observer for those notifications. Create a new `UIView` subclass called `ColorGrid`, and put the following code in *ColorGrid.h*:

```
//  ColorGrid.h
#import <UIKit/UIKit.h>
// notification names
#define ColorGridTouchedOrDragged @"ColorGridTouchedOrDragged"
#define ColorGridTouchEnded @"ColorGridTouchEnded"
// key into the notification's userInfo dictionary
#define ColorGridLatestTouchedColor @"ColorGridLatestTouchedColor"
@interface ColorGrid : UIView {
  NSArray *colors;
  NSUInteger columnCount;
  NSUInteger rowCount;
}
@property (retain, nonatomic) NSArray *colors;
@property (nonatomic) NSUInteger columnCount;
@property (nonatomic) NSUInteger rowCount;
@end
```

This interface shows all the elements we'll need in order to use this class: a set of properties for specifying the colors, as well as the number of columns and rows to display, all of which need to be set in order for the view to draw properly. Here, we also define a pair of `NSString` constants that interested parties (such as our controller class) will use to register themselves as `NSNotification` observers, and another string that's used as a key into the `userInfo` dictionary passed along with the notification for retrieving the chosen color. It's a good idea to define strings that will be used in multiple spots this way, instead of putting the literal strings, quotes and all, in your code. With the defined version, Xcode will help autocomplete as you type, and the compiler will complain if you misspell it.

Now for the implementation. Switch to *ColorGrid.m*, and start things off with the basics:

```
//  ColorGrid.m
#import "ColorGrid.h"
@implementation ColorGrid
@synthesize colors, columnCount, rowCount;
- (void)dealloc {
  self.colors = nil;
  [super dealloc];
}
```

Next up is the `drawRect:` method. This method relies on `columnCount` and `rowCount` being set to a nonzero value before being drawn. Those values determine the layout of the grid as a whole. The `UIColor` objects stored in the `colors` array will be used to fill rectangles in the grid, row by row. If there aren't enough colors in the array to fill the grid, the rest of the "cells" will be filled with white.

```
- (void)drawRect:(CGRect)rect {
  CGRect b = self.bounds;
```

www.it-ebooks.info

```
  CGContextRef myContext = UIGraphicsGetCurrentContext();
  CGFloat columnWidth = b.size.width / columnCount;
  CGFloat rowHeight = b.size.height / rowCount;
  for (NSUInteger rowIndex = 0; rowIndex < rowCount; rowIndex++) {
    for (NSUInteger columnIndex = 0; columnIndex < columnCount; columnIndex++) {
      NSUInteger colorIndex = rowIndex * columnCount + columnIndex;
      UIColor *color = [self.colors count] > colorIndex ?
                         [self.colors objectAtIndex:colorIndex] :
                         [UIColor whiteColor];
      CGRect r = CGRectMake(b.origin.x + columnIndex * columnWidth,
                            b.origin.y + rowIndex * rowHeight,
                            columnWidth, rowHeight);
      CGContextSetFillColorWithColor(myContext, color.CGColor);
      CGContextFillRect(myContext, r);
    }
  }
}
```

We also need to be able to determine the color shown at any given point, for the touch methods to be able to report with a notification. Rather than putting that directly into the touch methods, we split it off into a separate `colorAtPoint:` method that each of them can use. This is basically the inverse of what's going on in the innermost loop of the `drawRect:` method.

```
- (UIColor *)colorAtPoint:(CGPoint)point {
  if (!CGRectContainsPoint(self.bounds, point)) return nil;

  CGRect b = self.bounds;
  CGFloat columnWidth = b.size.width / columnCount;
  CGFloat rowHeight = b.size.height / rowCount;
  NSUInteger rowIndex = point.y / rowHeight;
  NSUInteger columnIndex = point.x / columnWidth;
  NSUInteger colorIndex = rowIndex * columnCount + columnIndex;
  return [self.colors count] > colorIndex ?
           [self.colors objectAtIndex:colorIndex] :
           nil;
}
```

Finally, we get to the touch methods themselves. This class responds to both initial touches and drags in the same way, so `touchesMoved:` just calls `touchesBegan:`. However, `touchesEnded:` uses a different notification name, so we'll let it have its own code.

```
- (void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event {
  CGPoint location = [[touches anyObject] locationInView:self];
  UIColor *color = [self colorAtPoint:location];
  if (color) {
    NSDictionary *userDict = [NSDictionary dictionaryWithObject:color
                                     forKey:ColorGridLatestTouchedColor];
    [[NSNotificationCenter defaultCenter] postNotificationName:ColorGridTouchedOrDragged
                                    object:self userInfo:userDict];
  }
}
- (void)touchesMoved:(NSSet *)touches withEvent:(UIEvent *)event {
  [self touchesBegan:touches withEvent:event];
}
- (void)touchesEnded:(NSSet *)touches withEvent:(UIEvent *)event {
```

www.it-ebooks.info

```
  CGPoint location = [[touches anyObject] locationInView:self];
  UIColor *color = [self colorAtPoint:location];
  if (color) {
    NSDictionary *userDict = [NSDictionary dictionaryWithObject:color
                                           forKey:ColorGridLatestTouchedColor];
    [[NSNotificationCenter defaultCenter] postNotificationName:ColorGridTouchEnded
                                          object:self userInfo:userDict];
  }
}
@end
```

Hooking Up the Grid

Now that we have a view class ready to go, we can create a view controller that makes use of it. Use the New File Assistant to create a new `UIViewController` subclass, making it include the creation of an *.xib* file, but *not* making it a `UITableViewController` subclass. Name it `SelectColorController`, and give it the following interface in *SelectColorController.h*:

```
//  StrokeColorController.h
#import <UIKit/UIKit.h>
@class ColorGrid;
// a notification name
#define ColorSelectionDone @"ColorSelectionDone"
@interface SelectColorController : UIViewController {
  IBOutlet ColorGrid *colorGrid;
  IBOutlet UIView *selectedColorSwatch;
  UIColor *selectedColor;
  UIPopoverController *container;
}
@property (retain, nonatomic) ColorGrid *colorGrid;
@property (retain, nonatomic) UIColor *selectedColor;
@property (assign, nonatomic) UIPopoverController *container;
@end
```

The GUI for this class will contain a `ColorGrid` instance, as well as a simple `UIView` for displaying the selected color. We make the `colorGrid` an accessible property so that our main controller, `DudelViewController`, can set its properties (`rowCount`, `columnCount`, and `colors`) when setting things up. This class also has properties for the currently selected color and the `UIPopoverController` that displays it. And, like the `ColorGrid` class, it will communicate “upstream” to the `DudelViewController` indirectly, through the use of a notification, whose name is defined here.

Now open *SelectColorController.xib* in Interface Builder. Once again, the default view is meant to be full-screen, so use the attribute inspector to turn off the Status Bar by setting it to Unspecified, and then the size inspector to make it 320 by 320. Use the Library to get instances of `UIView` and `ColorGrid`, and lay them out as shown in Figure 6–9.

www.it-ebooks.info

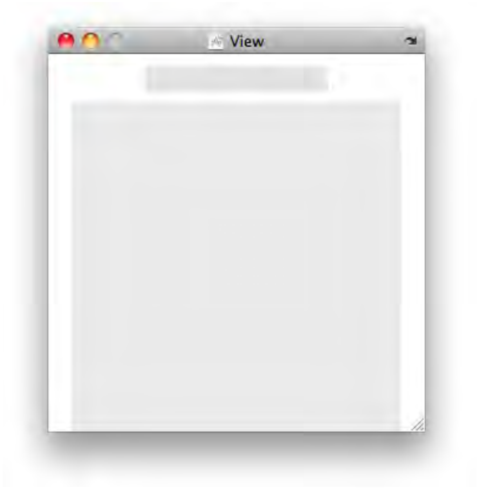


Figure 6–9. *The basic layout of our color picker in Interface Builder*

The upper view there is the `UIView`, and the lower one is the `ColorGrid`. You don't need to get too picky about the sizes of these views, since `ColorGrid` will adjust to whatever you throw at it, but it's good to have the `ColorGrid` reasonably large and both views centered in the parent view. Connect the outlets from File's Owner to the `colorGrid` and `selectedColorSwatch` views. The GUI is complete! Now we just need to make it work.

Here's the code for *SelectColorController.m*:

```
//
//  StrokeColorController.m
#import "SelectColorController.h"
#import "ColorGrid.h"
@implementation SelectColorController
@synthesize colorGrid, selectedColor, container;
- (void)viewDidLoad {
  [super viewDidLoad];
  [[NSNotificationCenter defaultCenter] addObserver:self
    selector:@selector(colorGridTouchedOrDragged:)
    name:ColorGridTouchedOrDragged object:colorGrid];
  [[NSNotificationCenter defaultCenter] addObserver:self
    selector:@selector(colorGridTouchEnded:)
    name:ColorGridTouchEnded object:colorGrid];
  selectedColorSwatch.backgroundColor = self.selectedColor;
}
- (BOOL)shouldAutorotateToInterfaceOrientation:(UIInterfaceOrientation)orientation {
    // Overriden to allow any orientation.
    return YES;
}
- (void)viewDidUnload {
  [super viewDidUnload];
  [[NSNotificationCenter defaultCenter] removeObserver:self];
}
- (void)dealloc {
  [[NSNotificationCenter defaultCenter] removeObserver:self];
  self.colorGrid = nil;
```

www.it-ebooks.info

```
  [super dealloc];
}
- (void)colorGridTouchedOrDragged:(NSNotification *)notification {
  NSDictionary *userDict = [notification userInfo];
  self.selectedColor = [userDict objectForKey:ColorGridLatestTouchedColor];
  selectedColorSwatch.backgroundColor = self.selectedColor;
}
- (void)colorGridTouchEnded:(NSNotification *)notification {
  NSDictionary *userDict = [notification userInfo];
  self.selectedColor = [userDict objectForKey:ColorGridLatestTouchedColor];
  selectedColorSwatch.backgroundColor = self.selectedColor;
  [[NSNotificationCenter defaultCenter] postNotificationName:ColorSelectionDone
    object:self];
}
@end
```

This should be pretty straightforward. We register methods to listen for activity from the `ColorGrid`, each of which grabs the latest touched color from the notification object. If the touch ended, we send out yet another notification, so that our main controller gets the message that a color has been set.

Serving Two Masters

Before we use this new controller from `DudelViewController`, we need to make two subclasses of the `SelectColorController` class, since `DudelViewController` uses the class of the currently active popover controller to determine exactly which controller it's dealing with.

NOTE: Purists may object to the creation of subclasses that don't have any behavior or data of their own, but then again, purists object to a lot of things. Being reality-based, we'll do it this way, both because it's simple and because it lets `DudelViewController` deal with all the popovers as consistently as possible.

Use the New File Assistant to create a new class. The assistant doesn't know about the `SelectColorController` class, so we need to use `NSObject` as the superclass and change it later. Name the class `StrokeColorController`, and give its *.h* and *.m* files the following contents:

```
//  StrokeColorController.h
#import <Foundation/Foundation.h>
#import "SelectColorController.h"
@interface StrokeColorController : SelectColorController {}
@end

//  StrokeColorController.m
#import "StrokeColorController.h"
@implementation StrokeColorController
@end
```

Create another new class named `FillColorController`, and define it like this:

www.it-ebooks.info

```
//  FillColorController.h
#import "SelectColorController.h"
@interface FillColorController : SelectColorController {}
@end

//  FillColorController.m
#import "FillColorController.h"
@implementation FillColorController
@end
```

Now let's add support for both of these to `DudelViewController` in one fell swoop. Open *DudelViewController.m*, and start off adding these includes:

```
#import "StrokeColorController.h"
#import "FillColorController.h"
#import "ColorGrid.h"
```

Next, let's look at the code that will launch each of the font selectors. The two action methods are very similar, so instead of repeating a lot of code, we put most of it into a separate method, shown here:

```
// both of the color popover action methods call this method.
- (void)doPopoverSelectColorController:(SelectColorController*)scc sender:(id)sender {
  [self setupNewPopoverControllerForViewController:scc];
  scc.container = self.currentPopover;
  self.currentPopover.popoverContentSize = scc.view.frame.size;

  // these have to be set after the view is already loaded (which happened
  // a couple of lines ago, thanks to scc.view...)
  scc.colorGrid.columnCount = 3;
  scc.colorGrid.rowCount = 4;
  scc.colorGrid.colors = [NSArray arrayWithObjects:
                          [UIColor redColor],
                          [UIColor greenColor],
                          [UIColor blueColor],
                          [UIColor cyanColor],
                          [UIColor yellowColor],
                          [UIColor magentaColor],
                          [UIColor orangeColor],
                          [UIColor purpleColor],
                          [UIColor brownColor],
                          [UIColor whiteColor],
                          [UIColor lightGrayColor],
                          [UIColor blackColor],
                          nil];
  [[NSNotificationCenter defaultCenter] addObserver:self
    selector:@selector(colorSelectionDone:) name:ColorSelectionDone object:scc];

  [self.currentPopover presentPopoverFromBarButtonItem:sender
    permittedArrowDirections:UIPopoverArrowDirectionAny animated:YES];
}
- (IBAction)popoverStrokeColor:(id)sender {
  StrokeColorController *scc = [[[StrokeColorController alloc]
    initWithNibName:@"SelectColorController" bundle:nil] autorelease];
  scc.selectedColor = self.strokeColor;
```

www.it-ebooks.info

```
  [self doPopoverSelectColorController:scc sender:sender];
}
- (IBAction)popoverFillColor:(id)sender {
  FillColorController *fcc = [[[FillColorController alloc]
    initWithNibName:@"SelectColorController" bundle:nil] autorelease];
  fcc.selectedColor = self.fillColor;
  [self doPopoverSelectColorController:fcc sender:sender];
}
```

In each of those cases, our main controller is set up to listen for notifications from the color selector. Here's the method that will handle the notifications:

```
- (void)colorSelectionDone:(NSNotification *)notification {
  SelectColorController *object = [notification object];
  UIPopoverController *popoverController = object.container;
  [popoverController dismissPopoverAnimated:YES];
  [self handleDismissedPopoverController:popoverController];
}
```

Finally, we take care of the main popover dismissal handler. Add the bold lines to the following method, which will make us notice new values in the color selectors:

```
- (void)handleDismissedPopoverController:(UIPopoverController*)popoverController {
  if ([popoverController.contentViewController isMemberOfClass:
    [FontListController class]]) {
    // this is the font list, grab the new selection
    FontListController *flc = (FontListController *)
      popoverController.contentViewController;
    self.font = [UIFont fontWithName:flc.selectedFontName size:self.font.pointSize];
  } else if ([popoverController.contentViewController isMemberOfClass:
    [FontSizeController class]]) {
    FontSizeController *fsc = (FontSizeController *)
      popoverController.contentViewController;
    self.font = fsc.font;
  } else if ([popoverController.contentViewController isMemberOfClass:
    [StrokeWidthController class]]) {
    StrokeWidthController *swc = (StrokeWidthController *)
      popoverController.contentViewController;
    self.strokeWidth = swc.strokeWidth;
  } else if ([popoverController.contentViewController isMemberOfClass:
    [StrokeColorController class]]) {
    StrokeColorController *scc = (StrokeColorController *)
      popoverController.contentViewController;
    self.strokeColor = scc.selectedColor;
  } else if ([popoverController.contentViewController isMemberOfClass:
    [FillColorController class]]) {
    FillColorController *fcc = (FillColorController *)
      popoverController.contentViewController;
    self.fillColor = fcc.selectedColor;
  }
  self.currentPopover = nil;
}
```

You should now be able to build and run your app, and use the new color popovers to define stroke and fill colors for all of the tools.

y

www.it-ebooks.info

With this functionality in place, we have all we need for a bare-bones vector-drawing app. No one's going to mistake this for Adobe Illustrator, but it's easy and functional enough for people to use for simple creations. Figure 6–10 shows an example of what you can create with Dudel.

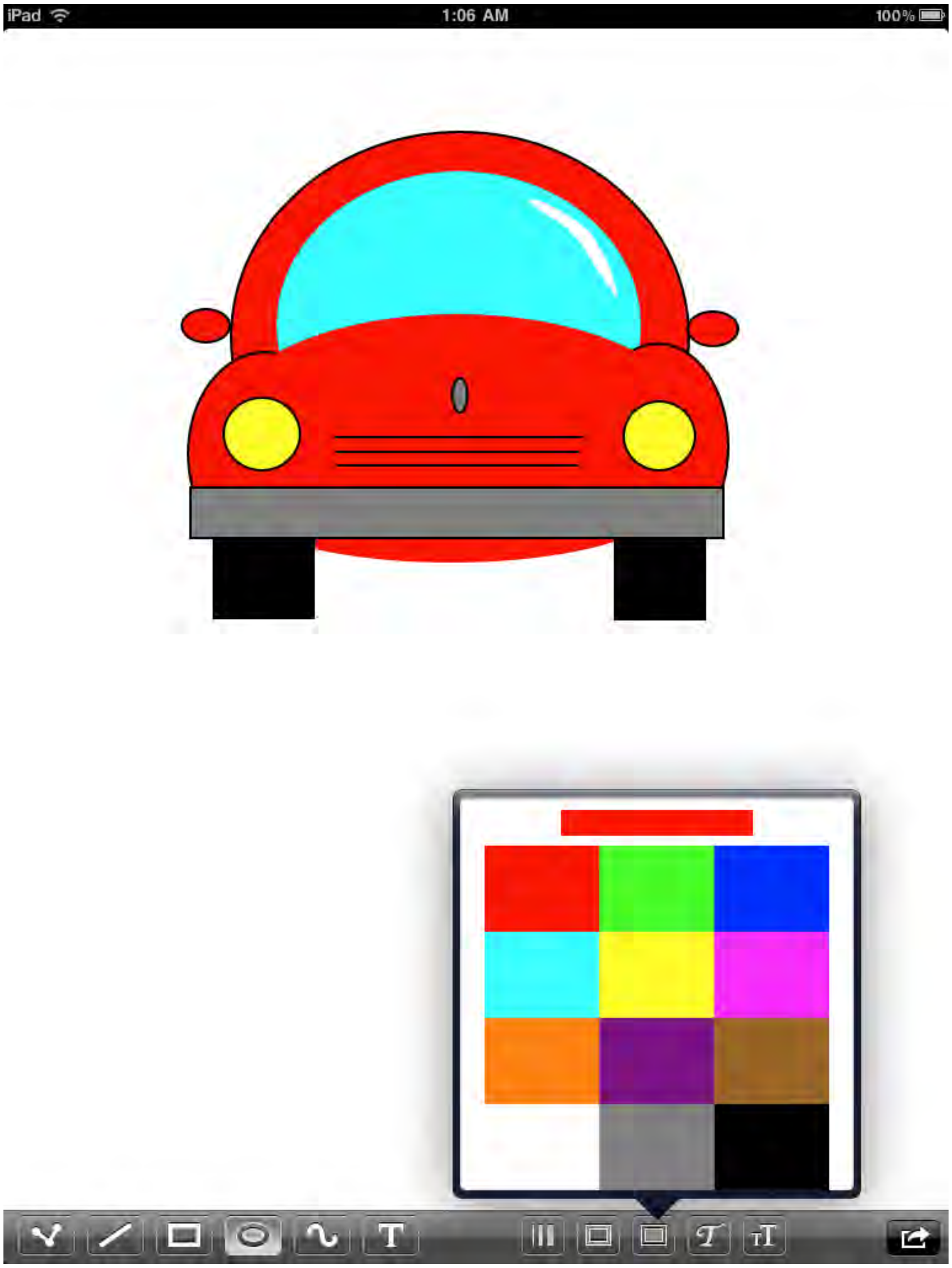


Figure 6–10. *Drawing with Dudel (Dave Wooldridge's creation)*

www.it-ebooks.info

Your Popover-Fu Is Strong

You've now seen a wide range of views presented as popovers. The examples in this chapter demonstrated some of the various ways you can deal with the popover interface, such as choosing whether to let the popover stick around while the user works with controls, and how to pass changes upstream using notifications. These techniques are already used by a wide variety of iPad apps. Adding them to your own apps will let your users access application features in ways that are similar to the menus, palettes, and inspectors of desktop applications, while still keeping your interface free from clutter.

With that, we wrap up the basic features of Dudel. We'll continue adding more to Dudel throughout the book, but now it's time to take a side trip and dig into the new possibilities for displaying video and using external screens. Chapter 7 covers video and display options for iPad apps.

www.it-ebooks.info

Chapter 7

Video and Display Output

In older versions of iOS, displaying video was an all-or-nothing proposition. The `MPMoviePlayerController` included as part of the Media Player framework allowed you to display video that took over the whole screen, and that was it! There was no system-supported way to display video in any other way—for example, as a small video displayed within a web page.

Starting with iOS 3.2 for the iPad, the `MPMoviePlayerController` has changed a bit. Now, instead of taking over the screen, the default behavior is to display its video directly on the screen. In this chapter, you'll see how this is accomplished by creating an app that displays multiple videos on the screen simultaneously (with some limitations).

Also new in iOS 3.2 for the iPad is the ability for third-party developers to display content on an external screen connected through an adapter to the iPad's dock connector. While this sort of thing was possible in previous versions of iPhone OS, it required the use of private APIs, which meant that you couldn't actually ship software that made use of an external screen through the App Store. That privilege was exclusively Apple's! This has changed with iOS 3.2. In this chapter, you'll learn how to handle an external screen with ease.

Displaying Multiple Videos

Let's start by looking at how to display multiple videos on the screen using `MPMoviePlayerController`. If you haven't used this class in previous iPhone projects, you may not aware that it's not what you think of as a typical view controller in Cocoa Touch; in fact, it doesn't inherit from `UIViewController`. In MVC terms, it's a controller in the sense that it serves as an intermediary between a video file or stream (the model) and the on-screen view, but it doesn't fit into the `UIKit` scheme of shuffling view controllers around as users navigate the app. However, a new class included in iOS 3.2 fits that purpose perfectly—the similarly named `MPMoviePlayerViewController`.

The `MPMoviePlayerViewController` works just like any other `UIViewController` subclass. You create one (specifying its contents using the `initWithContentURL:` method), and push it onto a view controller navigation stack, just like any other view controller. This

www.it-ebooks.info

usage is so simple that we're not going to give it any more ink here. Instead, we'll focus on MPMoviePlayerController, and demonstrate how to display content from several of these controllers at once. We're going to create an app that shows a table view, with each row displaying a video.

Creating the Video App Project

In Xcode, create a new view-based application, targeted for iPad only, and name it VideoToy. This will create a few items, including the VideoToyAppDelegate and VideoAppViewController classes.

Before we start working on the code, take a few minutes to find some video clips that you can use in this project. Any sort of iPhone-friendly video will do nicely. If you don't already have some *.mp4* or *.m4u* files on your computer, you can find some on the Internet. An easy way is to browse the iTunes U section of iTunes and download videos from there. Then drag a few into your project.

Specifying Your Video Files

Now let's get started on the code. We're going to make a single change to *VideoAppDelegate.m* to specify the names of the video files we're using. We're passing this list along to the VideoAppViewController instance.

```
- (BOOL)application:(UIApplication *)application
  didFinishLaunchingWithOptions:(NSDictionary *)launchOptions {
  [window addSubview:viewController.view];
  [window makeKeyAndVisible];

  viewController.urlPaths = [NSMutableArray arrayWithObjects:
      [[NSBundle mainBundle] pathForResource:@"looking_for_my_leopard"
        ofType:@"mp4"],
      [[NSBundle mainBundle] pathForResource:@"knight_rider_season2intro"
        ofType:@"mp4"],
      [[NSBundle mainBundle] pathForResource:@"muppets" ofType:@"mp4"],
      [[NSBundle mainBundle] pathForResource:@"opengl" ofType:@"mp4"],
      nil];
  [viewController.tableView reloadData];
  return YES;
}
```

This code references a few properties that don't yet exist in VideoToyViewController, but don't worry. We're going to turn VideoToyViewController into a UITableViewController subclass that holds onto a list of video files to display.

The GUI will be created entirely from the table view delegate and dataSource methods, so we have no need for the *VideoToyViewController.xib* that was created along with the project. Delete *VideoToyViewController.xib* from the project. Then open *MainWindow.xib* so we can remove the reference it contains to that *.xib* file. Select the VideoToyViewController object, open the attribute inspector, and clear out the Nib Name field.

www.it-ebooks.info

Next, make the following changes to *VideoToyViewController.h*:

```
//  VideoToyViewController.h
#import <UIKit/UIKit.h>
#import <MediaPlayer/MediaPlayer.h>
@class VideoCell;
@interface VideoToyViewController : UITableViewController {
  NSMutableArray *urlPaths;
  IBOutlet VideoCell *videoCell;
}
@property (retain, nonatomic) NSMutableArray *urlPaths;
@end
```

Here, we change the superclass, declare the `urlPaths` property that we referenced earlier in the app delegate, and also lay the foundation for a more detailed part of the GUI by creating the `videoCell` instance variable. If you're wondering why we declared `videoCell` as an `IBOutlet`, when this class isn't loading its GUI from a *.xib* file, then good for you—you're really paying attention here! That will be explained in just a minute, so hang in there.

Switch over to *VideoToyViewController.m*, and add an `import` near the top:

```
//  VideoToyViewController.m
#import "VideoCell.h"
```

Now add these methods to define the basic properties of the table view that will be displayed:

```
- (CGFloat)tableView:(UITableView *)tableView
  heightForRowAtIndexPath:(NSIndexPath *)indexPath {
  return [VideoCell rowHeight];
}
- (NSInteger)numberOfSectionsInTableView:(UITableView *)tableView {
  return 1;
}
- (NSInteger)tableView:(UITableView *)tableView
  numberOfRowsInSection:(NSInteger)section {
  return [urlPaths count];
}
```

Using the videoCell Outlet to Load the GUI

Now for the slightly trickier spot and the explanation for the existence of the `videoCell` outlet. The following shows a good way to load a table view's cell content from a *.nib* file, rather than defining your layout entirely in code.

```
- (UITableViewCell *)tableView:(UITableView *)tableView
  cellForRowAtIndexPath:(NSIndexPath *)indexPath {
  VideoCell *cell = (VideoCell *)[tableView
    dequeueReusableCellWithIdentifier:[VideoCell reuseIdentifier]];
  if (!cell) {
    [[NSBundle mainBundle] loadNibNamed:@"VideoCell" owner:self
      options:nil];
    cell = videoCell;
    cell.selectionStyle = UITableViewCellSelectionStyleNone;
```

www.it-ebooks.info

```
    [videoCell autorelease];
    videoCell = nil;
  }
  cell.urlPath = [urlPaths objectAtIndex:indexPath.row];
  return cell;
}
```

The first line tries to find a cell to reuse, as usual. If it doesn't find one, then instead of programmatically creating a cell, we load one from a *.nib* file. The trick is that we load this file with `self` as the File's Owner, which will have the side effect of setting the `videoCell` outlet to point to whatever the File's Owner proxy in the *.xib* file has its `videoCell` outlet pointing to.

Remember to add a little cleanup for the new `urlPaths` property:

```
- (void)dealloc {
  self.urlPaths = nil;
  [super dealloc];
}
```

Now use the New File Assistant to make a new `UITableViewCell` subclass called `VideoCell`. Open its *.h* file, and add the bold lines shown here:

```
//  VideoCell.h
#import <UIKit/UIKit.h>
#import <MediaPlayer/MediaPlayer.h>
@interface VideoCell : UITableViewCell {
  IBOutlet UIView *movieViewContainer;
  IBOutlet UILabel *urlLabel;
  NSString *urlPath;
  MPMoviePlayerController *mpc;
}
@property (retain, nonatomic) NSString *urlPath;
@property (retain, nonatomic) MPMoviePlayerController *mpc;
+ (NSString *)reuseIdentifier;
+ (CGFloat)rowHeight;
@end
```

Switch to *VideoCell.m*, and remove the `initWithStyle:reuseIdentifier:` method, since we won't be needing it. Then make the rest of the file look like this:

```
//  VideoCell.m
#import "VideoCell.h"
@implementation VideoCell
@synthesize urlPath, mpc;
+ (NSString *)reuseIdentifier {
  return @"VideoCell";
}
+ (CGFloat)rowHeight {
  return 200;
}
- (void)setupMpc {
  if (mpc) {
    // we've already got one of these, time to get rid of it
    [mpc.view removeFromSuperview];
    self.mpc = nil;
```

www.it-ebooks.info

```
  }
  if (urlPath) {
    NSURL *url = [NSURL fileURLWithPath:self.urlPath];
    self.mpc = [[[MPMoviePlayerController alloc] initWithContentURL:url]
      autorelease];
    mpc.shouldAutoplay = NO;
    mpc.view.frame = movieViewContainer.bounds;
    [movieViewContainer addSubview:mpc.view];
  }
}
- (void)setUrlPath:(NSString *)p {
  if (![p isEqual:urlPath]) {
    [urlPath autorelease];
    urlPath = [p retain];
    if (urlPath && !mpc) {
      [self setupMpc];
    }
    urlLabel.text = urlPath;
  }
}
- (void)awakeFromNib {
  if (urlPath && !mpc) {
    [self setupMpc];
  }
  urlLabel.text = urlPath;
}
- (void)dealloc {
  self.urlPath = nil;
  self.mpc = nil;
  [super dealloc];
}
- (void)setSelected:(BOOL)selected animated:(BOOL)animated {
  [super setSelected:selected animated:animated];
  // Configure the view for the selected state
  [mpc play];
}
@end
```

In this code, we're referring to a `UILabel` and an `MPMoviePlayerController` within our GUI, as well as a plain-old `UIView`. The `MPMoviePlayerController` that we create exposes a property called `view`, which lets us access the view object it uses to render the video. We don't know the class of this view—it's essentially a private class that the Media Player framework doesn't expose to us. When the time comes, after the *.nib* file has loaded and the `urlPath` has been set, we create an `MPMoviePlayerController`, grab its view, and put it into our view hierarchy by making it a child of the empty `UIView`.

Creating the VideoCell User Interface

Now use the New File Assistant to create a new empty GUI file called *VideoCell.xib*. Open the new file in Interface Builder, add a `VideoCell` instance from the Library, and resize it to 768 by 200. Use the attribute inspector to set the identifier to `VideoCell`, just as we did in code. Add a label on the left to display a URL, and use the attribute

www.it-ebooks.info

inspector to configure it for 0 lines and character wrap. Also add a plain-old `UIView` on the right. Figure 7–1 shows the basic layout.

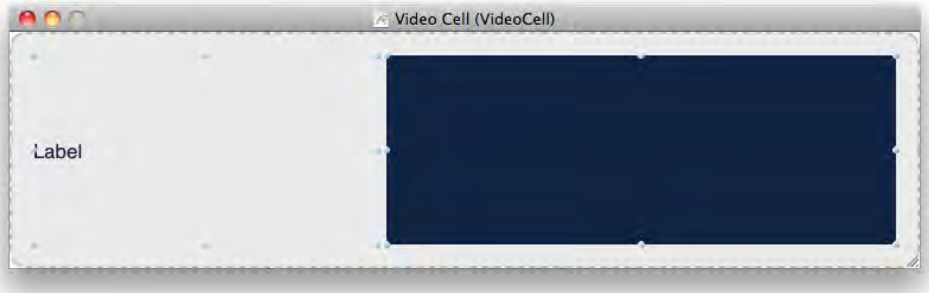


Figure 7–1. *Not exactly a polished user interface, but it's a start.*

Connect the outlets from `VideoCell` to the appropriate objects. Use the size inspector to configure the autosizing attribute of each of them so that they'll expand horizontally, while still remaining tied to their respective window edges when the cell is resized (such as when the iPad rotates). Finally, use the identity inspector to make File's Owner a `VideoToyViewController`, and connect its `videoCell` outlet to the one you just created.

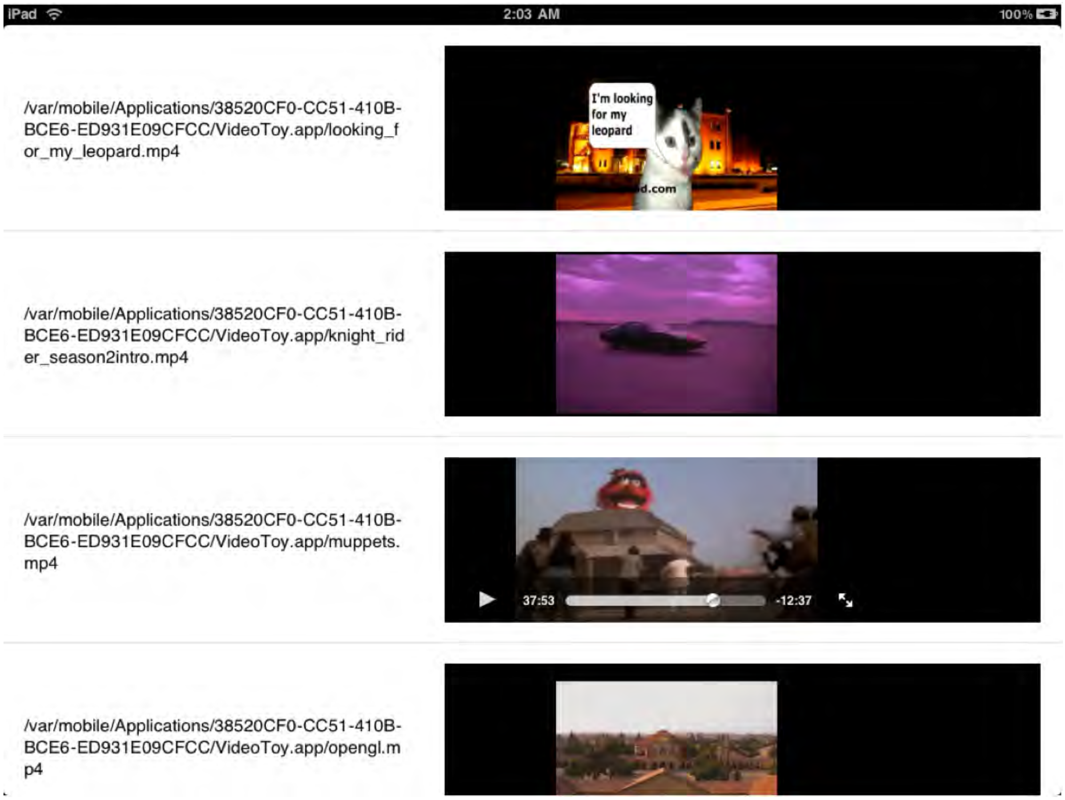
Running the Video App

Now you should be able to build and run the app, and see the vertical list of video views. They will all be empty rectangles, except for the last one displayed, which will probably start playing on its own. Touching any other cell in the table view will activate the movie view in that row and start playing it. As you activate each video by tapping its row, you get a full set of video controls, including a button that switches to full-screen display.

You will also notice one of the main limitations of using multiple video views: only one of them can play at a time! Starting playback on an `MPMoviePlayerController` will simultaneously pause playback on any others that are running. Figure 7–2 shows the display after loading four videos and playing each of them a bit.

You now have the beginnings of a sort of video browser. This could be useful in a number of applications, letting you preview several videos in a list. That's fine, but what if you also wanted to display the video on an external screen? You'll be able to do that with a few adjustments, as described in the next section.

www.it-ebooks.info



/var/mobile/Applications/38520CF0-CC51-410B-BCE6-ED931E09CFCC/VideoToy.app/looking_for_my_leopard.mp4

/var/mobile/Applications/38520CF0-CC51-410B-BCE6-ED931E09CFCC/VideoToy.app/knight_rider_season2intro.mp4

/var/mobile/Applications/38520CF0-CC51-410B-BCE6-ED931E09CFCC/VideoToy.app/muppets.mp4

/var/mobile/Applications/38520CF0-CC51-410B-BCE6-ED931E09CFCC/VideoToy.app/opengl.mp4

Figure 7–2. *Displaying a few of the videos floating around my hard drive.*

Outputting to an External Screen

The `UIScreen` class has been a part of iOS since the beginning. Its `mainScreen` class method gives you the screen of the device you're running on, which you can query for geometry information such as its bounds or frame.

In iOS 3.2, `UIScreen` gets a new class method, `screens`, which returns an array of all currently connected screens, including the iPad's own screen. This is the key to accessing an attached external screen. If it contains more than one item, all but the first are external screens. Also new in iOS 3.2 is the ability to ask any screen which resolutions it supports via the `availableModes` method, along with the `currentMode` method for determining and setting which resolution is in use.

You can move any of your content to an external screen simply by creating a new `UIWindow` object, adding your views to it, and setting its `screen` property to point to the external screen.

www.it-ebooks.info

Extending the Video App to Handle an External Screen

When using an external screen, you need to consider how to properly handle when the user plugs in or unplugs a screen. To help with this, `UIScreen` defines a few notifications that let you know when a screen is connected or disconnected, so you can act accordingly.

In this section, we'll extend our `VideoToy` project so that if a screen is connected, the video you choose will play on it; if you disconnect the screen, the video will continue playing on the device. This will require a bit of extra bookkeeping on our part—we'll need to keep track of the currently selected video and its corresponding views, so that we can switch things around as the external screen comes and goes.

Start off by editing *VideoCell.h*, adding a few lines to define a delegate, a protocol the delegate should implement, and a property declaration for `movieViewContainer` so that we can reach it from other classes.

```
//  VideoCell.h
#import <UIKit/UIKit.h>
#import <MediaPlayer/MediaPlayer.h>
@interface VideoCell : UITableViewCell {
  IBOutlet UIView *movieViewContainer;
  IBOutlet UILabel *urlLabel;
  NSString *urlPath;
  MPMoviePlayerController *mpc;
  id delegate;
}
@property (retain, nonatomic) UIView *movieViewContainer;
@property (retain, nonatomic) NSString *urlPath;
@property (retain, nonatomic) MPMoviePlayerController *mpc;
@property (assign, nonatomic) id delegate;
+ (NSString *)reuseIdentifier;
+ (CGFloat)rowHeight;
@end
@protocol VideoCellDelegate
- (void)videoCellStartedPlaying:(VideoCell *)cell;
@end
```

Now switch to *VideoCell.m*, and add synthesized accessors for `movieViewContainer` and `delegate`.

```
@synthesize urlPath, mpc, movieViewContainer, delegate;
```

Then free up one additional resource in `dealloc`:

```
- (void)dealloc {
  self.urlPath = nil;
  self.mpc = nil;
  self.movieViewContainer = nil;
  [super dealloc];
}
```

Next, implement the following change, to let the delegate know when the video has been selected. This way, the view can be shifted to the external screen (if it's connected).

www.it-ebooks.info

```
- (void)setSelected:(BOOL)selected animated:(BOOL)animated {
  [super setSelected:selected animated:animated];
  if ([delegate respondsToSelector:@selector(videoCellStartedPlaying:)]) {
    [delegate videoCellStartedPlaying:self];
  }
  // Configure the view for the selected state
  [mpc play];
}
```

We really didn't do too much here. The most interesting part—handling a user selection that should put the video on the `externalScreen`—has been foisted off on a vaguely defined `delegate` object. Let's make that a bit more concrete, by having `VideoToyViewController` act as the delegate for `VideoCell`. This controller will now keep track of the selected `VideoCell` instance, as well as a `UIWindow` assigned to an external screen, if there is one.

Open *VideoToyViewController.h*, and make the changes shown here:

```
//  VideoToyViewController.h
#import <UIKit/UIKit.h>
#import <MediaPlayer/MediaPlayer.h>
@class VideoCell;
@interface VideoToyViewController : UITableViewController {
  NSMutableArray *urlPaths;
  IBOutlet VideoCell *videoCell;
  UIWindow *externalWindow;
  VideoCell *selectedCell;
}
@property (retain, nonatomic) NSMutableArray *urlPaths;
@property (retain, nonatomic) UIWindow *externalWindow;
@property (retain, nonatomic) VideoCell *selectedCell;
@end
```

Now it's time for *VideoToyViewController.m*, which is where the real work of managing the external screen happens. Synthesize the new properties, like this:

```
@synthesize urlPaths, externalWindow, selectedCell;
```

And make sure that resources are properly freed:

```
- (void)dealloc {
  self.urlPaths = nil;
  self.externalWindow = nil;
  self.selectedCell = nil;
  [super dealloc];
}
```

Next, move on to the `viewDidLoad` method. Here, we're going to first call the `updateExternalWindow` method (which we'll define in just a moment). We’ll also set up notifications whenever an external screen is connected or disconnected. Both of these events will also call the `updateExternalWindow` method.

```
- (void)viewDidLoad {
  [super viewDidLoad];
  [self updateExternalWindow];
  [[NSNotificationCenter defaultCenter] addObserver:self
    selector:@selector(updateExternalWindow)
```

www.it-ebooks.info

```
    name:UIScreenDidConnectNotification
    object:nil];
  [[NSNotificationCenter defaultCenter] addObserver:self
    selector:@selector(updateExternalWindow)
    name:UIScreenDidDisconnectNotification
    object:nil];
}
```

And now for the `updateExternalWindow` method itself. As you just saw, this method is called when our view is loaded, as well as every time the external screen is connected or disconnected. It's fairly complicated, since it is designed to handle the variety of situations it may encounter and do the right thing. The comments in the code provide more details.

```
- (void)updateExternalWindow {
  if ([[UIScreen screens] count] > 1) {
    //
    // An external screen is connected. Find the screen, put a
    // UIWindow on it.
    //
    UIScreen *externalScreen = [[UIScreen screens] lastObject];
    // Screen modes are sorted in order of increasing resolution.
    // Let's take the highest.
    UIScreenMode *highestScreenMode = [[externalScreen availableModes]
      lastObject];
    CGRect externalWindowFrame = CGRectMake(0, 0,
      [highestScreenMode size].width, [highestScreenMode size].height);
    self.externalWindow = [[[UIWindow alloc] initWithFrame:
      externalWindowFrame] autorelease];
    externalWindow.screen = externalScreen;
    [externalWindow.screen setCurrentMode:highestScreenMode];
    [externalWindow makeKeyAndVisible];
    if (selectedCell) {
      // A cell is selected. Move its view to the external window.
      [externalWindow addSubview:selectedCell.mpc.view];
      selectedCell.mpc.view.frame = externalWindow.bounds;
    }
  } else if ([[UIScreen screens] count] == 1) {
    //
    // No external screen is connected. Let's make sure we have no
    // dangling references
    // to anything off the main screen.
    //
    if ([[externalWindow subviews] count] > 0) {
      // externalWindow used to be attached to a screen which is no
      // longer there! Move its view back to where it came from.
      UIView *v = [[externalWindow subviews] lastObject];
      v.frame = selectedCell.movieViewContainer.bounds;
      [selectedCell.movieViewContainer addSubview:v];
    }
    self.externalWindow.screen = nil;
    self.externalWindow = nil;
  }
}
```

www.it-ebooks.info

NOTE: The updateExternalWindow method could have been split into three methods: one for each of the notifications, and one for after the nib file loaded. In fact, the first version I wrote did just that. But I noticed there was some functional overlap, so I refactored a bit. To me, it seems that compressing it into one method brings it together a bit better.

The next thing to tackle is the creation of the VideoCell instances. Each needs to be told who its delegate is. Find the relevant section in tableView:cellForRowAtIndexPath:, and add the bold line shown here:

```
[[NSBundle mainBundle] loadNibNamed:@"VideoCell" owner:self
  options:nil];
cell = videoCell;
cell.selectionStyle = UITableViewCellSelectionStyleNone;
cell.delegate = self;
```

Implementing the VideoCell Delegate Method

Finally, let's implement the delegate method itself. As you may recall, this method is called whenever the user selects a VideoCell in the GUI. Here, we need to check whether an external screen is connected and whether another video is currently running. Yes, this method is even more complicated than the updateExternalWindow method, but it has a lot to do. Again, the code comments provide more explanation.

```
- (void)videoCellStartedPlaying:(VideoCell *)cell {
  if (selectedCell != cell) { // Skip everything if it's the same cell.
    if ([[UIScreen screens] count] > 1) {
      // Switching external from one video (or blank) to another
      UIScreen *externalScreen = [[UIScreen screens] lastObject];
      UIScreenMode *highestScreenMode = [[externalScreen availableModes]
        lastObject];
      CGRect externalWindowFrame = CGRectMake(0, 0,
        [highestScreenMode size].width, [highestScreenMode size].height);
      if ([[externalWindow subviews] count] > 0) {
        // There's already a movie there. Put its view back in the cell
        // it came from.
        UIView *v = [[externalWindow subviews] lastObject];
        v.frame = selectedCell.movieViewContainer.bounds;
        [selectedCell.movieViewContainer addSubview:v];
      }
      // We're done with the old movie and cell.
      self.selectedCell = cell;
      // Get rid of the old window and screens; create new ones.
      self.externalWindow = [[[UIWindow alloc] initWithFrame:
        externalWindowFrame] autorelease];
      externalWindow.screen = externalScreen;
      [externalWindow.screen setCurrentMode:highestScreenMode];
      [externalWindow makeKeyAndVisible];
      if (selectedCell) {
        // Move the selected cell's movie view to the external screen.
```

www.it-ebooks.info

```
            [externalWindow addSubview:selectedCell.mpc.view];
            selectedCell.mpc.view.frame = externalWindow.bounds;
          }
        } else if ([[UIScreen screens] count] == 1) {
          // No external screen is connected.
          if ([[externalWindow subviews] count] > 0) {
            // We seem to have an old external window hanging around. Move
            // its view back to the cell it came from.
            UIView *v = [[externalWindow subviews] lastObject];
            v.frame = selectedCell.movieViewContainer.bounds;
            [selectedCell.movieViewContainer addSubview:v];
          }
          self.externalWindow.screen = nil;
          self.externalWindow = nil;
          // Keep track of the selected cell.
          self.selectedCell = cell;
        }
      }
    }
```

Testing the External Screen Functionality

Now build and run the app, and then unplug your iPad from your computer. This will probably crash the app, but that's OK.

Start up the app again directly on the iPad. Now whip out your trusty iPad-VGA adapter, find a convenient monitor with a VGA input, and plug it in. If you do that while a video is playing, you'll see that the video jumps to the external monitor. Otherwise, start playing a video, and you'll see it appear there.

The code we wrote for this functionality is quite robust. You should be able to unplug and reattach the screen during playback, before starting playback, while switching songs, before launching the app, and so on. You'll find that "it just works."

Display Solutions

The app we've built in this chapter is not meant as any sort of commercial product. We've kept it bare-bones, just to focus on the new ways of dealing with video playback and with the new facility for putting views onto an external display. Both of these areas are pretty easy to implement. The most complicated thing we needed to do was to make sure our app keeps informed about external screen connections and disconnections, and behaves appropriately.

Now it's up to you to determine if any of your iPad apps could make use of video playback and/or an external screen, and apply what you've learned here to your own projects.

In the meantime, continue on to Chapter 8 to learn how to use the UIKit's new split view to let your iPad apps display both content and navigation at the same time.

www.it-ebooks.info

Chapter 8

Split Views and Modal Modes

With the iPhone’s tiny screen, it’s natural to build interfaces that focus on one small portion of your app at a time. Cocoa Touch includes specialized `UIViewController` subclasses to facilitate this, letting you organize different views into tabs or navigable trees. On the iPad, however, we have a whole lot more space, so it makes sense to make better use of it!

In this chapter, you’ll learn about the `UISplitViewController`, which lets you move some of your application’s navigation structure into a view that appears to the left of your main content or in a floating popover accessible via a button in a toolbar.

We’ll also take a look at the new types of modal displays that can be used on the iPad, which give you added control over the way modal interactions are displayed and handled. To demonstrate these techniques, we’ll continue to improve our Dudel app.

The Split View Concept

Up to this point, Dudel has been a fun toy, but it has at least one quite severe limitation: Apart from sending your drawing as e-mail, you have no way of saving what you’ve drawn. As soon as you quit the app, your work is gone! That’s clearly not the way any iPhone or iPad app should work, so we’re going to remedy that, and give the user a way to save any number of Dudel documents. We’ll use a `UISplitViewController` to help us out here, so we can display an additional view controller that shows a list of all relevant files, letting the user switch between them easily.

The `UISplitViewController`, like the `UINavigationController` and `UITabBarController`, serves an organizational function. Rather than displaying any content on its own, it shows the view for an additional controller next to the main controller view in landscape mode, as shown in Figure 8–1.

www.it-ebooks.info

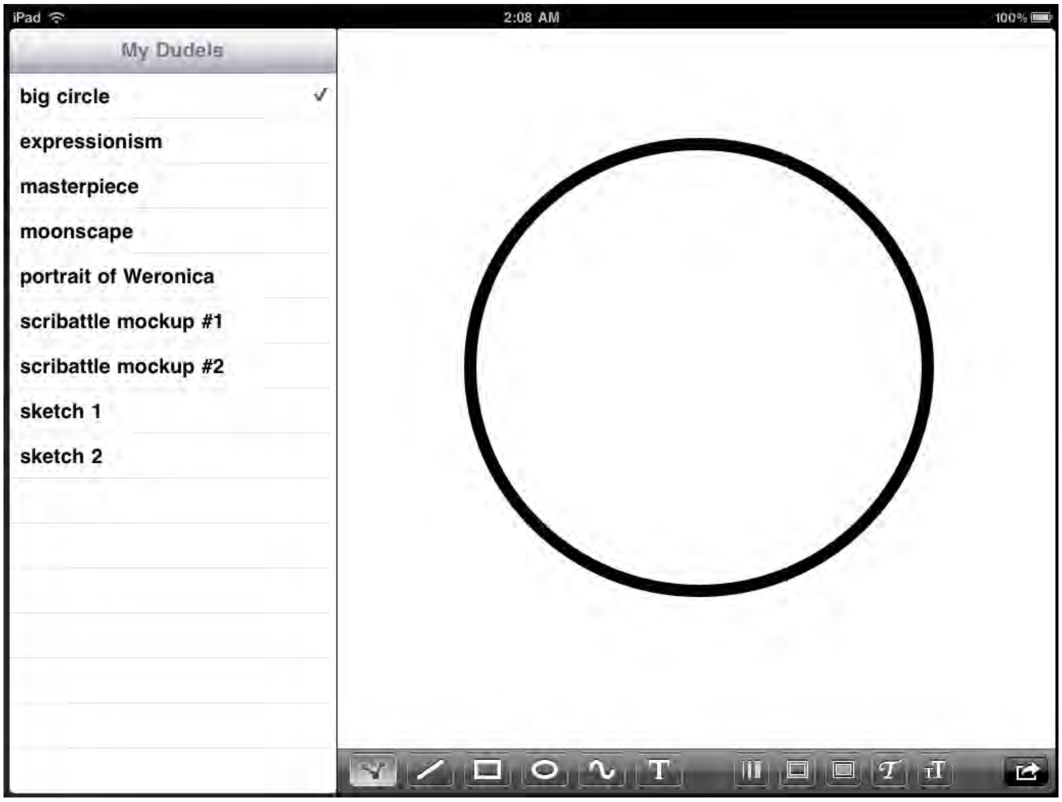


Figure 8–1. *Some inspiring Dudel art. Notice the list of available Dudel files on the left.*

NOTE: The use of the `UISplitViewController`, in combination with a toolbar at the bottom of the main view, is somewhat unorthodox. The `UISplitViewController` always creates the left-side view with a title row at the top ("My Dudels" here), and Apple's recommendation when using `UISplitViewController` is to put the main view's toolbar (if any) at the top as well. I didn't do this for Dudel, and it does give the screen a slightly lopsided appearance. But I think this adds character! Of course, you're free to move the main view's toolbar to the top if you wish, to bring it more in line with what Apple recommends. And when you use a `UISplitViewController` in your own apps, you should probably put your main view's toolbar at the top, unless you have a good reason not to (a better reason than my claims of adding character!).If you rotate the device to portrait mode, something interesting happens. The `UISplitViewController` switches gears, and no longer shows a list of files on the left. Instead, it gives a `UIBarButtonItem` to its delegate (a view controller of our own), which can then add it to a toolbar. That button item, when touched, brings up the same view that was shown on the left side in landscape mode, this time displayed using a `UIPopoverController`, as shown in Figure 8–2.

www.it-ebooks.info

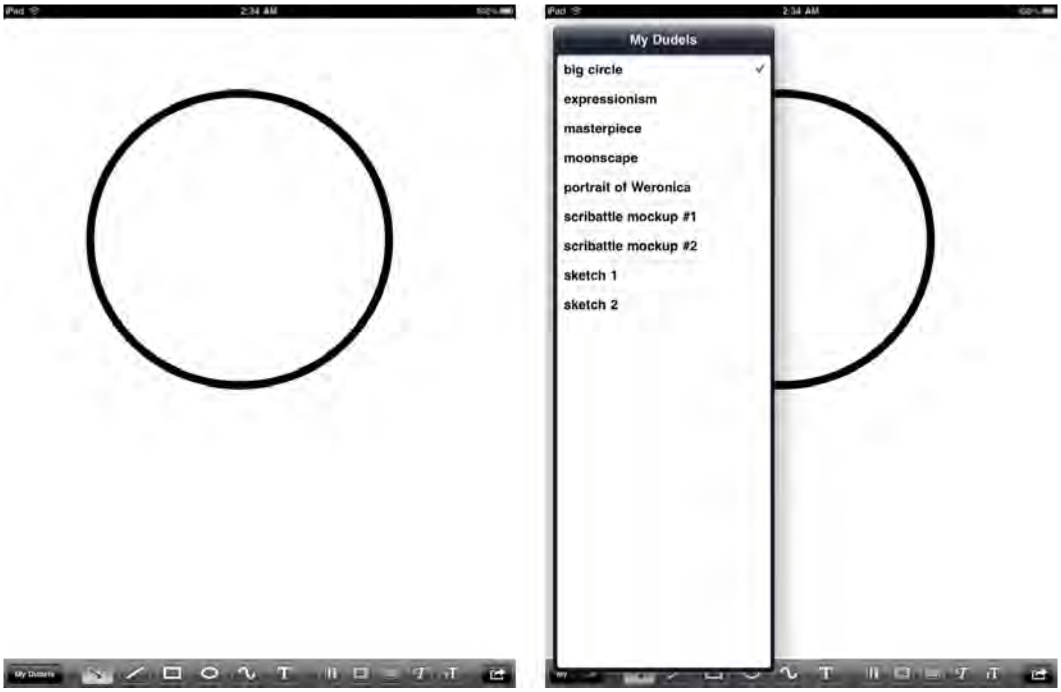


Figure 8–2. *In portrait mode, the file list is shown only when you click the button that brings up the popover.*

Unlike the popovers we set up in Chapter 6, this one will require no configuration on our part, since the `UISplitViewController` sets it up for us. However, we will still be required to dismiss the popover after the user makes a selection.

You might also notice in Figures 8-1 and 8-2 that the Email PDF button at the lower right has been swapped out for a generic action icon button. This will bring up a menu containing a handful of operations such as creating, renaming, and deleting files, which set up in this chapter.

The Basics of Saving and Loading

Before we can start thinking about showing a list of files, we need to add support for reading and writing files in the first place! Fortunately, the `NSCoding` protocol included in Cocoa Touch provides a solution that is easy to implement and perfectly adequate for our purposes.

The idea behind `NSCoding` is to add a couple of methods to each class that represents an object that needs to be saved: one method to save each of an object's instance variables to an archive, and another to populate an object's instance variables using values retrieved from an archive. When it's time to save, you just tell the root or top-level object to archive itself, and then when you want to load, you do the inverse. In our case, the top-level object is the `NSArray` containing the list of `Drawable` items. `NSArray` already

www.it-ebooks.info

implements the NSCoding protocol, but we need to do the same for our Drawable classes.

Confused? Let's start looking at some code that should clear this up.

Make a fresh copy of your Dudel project directory for this chapter's work, and open the Xcode project inside the new directory. Next, add the following lines to *PathDrawingInfo.m*:

```
- (void)encodeWithCoder:(NSCoder *)encoder {
  [encoder encodeObject:self.path forKey:@"path"];
  [encoder encodeObject:self.fillColor forKey:@"fillColor"];
  [encoder encodeObject:self.strokeColor forKey:@"strokeColor"];
}

- (id)initWithCoder:(NSCoder *)decoder {
  if ((self = [self init])) {
    self.path = [decoder decodeObjectForKey:@"path"];
    self.fillColor = [decoder decodeObjectForKey:@"fillColor"];
    self.strokeColor = [decoder decodeObjectForKey:@"strokeColor"];
  }
  return self;
}
```

Then add the following to *TextDrawingInfo.m*:

```
- (void)encodeWithCoder:(NSCoder *)encoder {
  [encoder encodeObject:self.path forKey:@"path"];
  [encoder encodeObject:self.strokeColor forKey:@"strokeColor"];
  [encoder encodeObject:self.font forKey:@"font"];
  [encoder encodeObject:self.text forKey:@"text"];
}

- (id)initWithCoder:(NSCoder *)decoder {
  if ((self = [self init])) {
    self.path = [decoder decodeObjectForKey:@"path"];
    self.strokeColor = [decoder decodeObjectForKey:@"strokeColor"];
    self.font = [decoder decodeObjectForKey:@"font"];
    self.text = [decoder decodeObjectForKey:@"text"];
  }
  return self;
}
```

Each of those methods receives an NSCoder object as an argument, which is primed to either receive values from our object or provide values for creating a new object, depending on which method we're talking about. All you do is set or retrieve values for all your instance variables, in a style similar to using an NSDictionary. That's all we need to do in the model classes, so move on to *DudelViewController.m*, where we'll add the machinery that starts these operations.

We'll start off by doing something quite simple. We'll add code that will make Dudel save the current state of the document it's working on when the user quits the app, and then reload that same document state when the app launches. For now, we'll just use a single file named *Untitled.dudeldoc* to save the user's work. Later in this chapter, we'll extend this to let the users name their own files.

www.it-ebooks.info

Begin by adding the following utility method to the DudelViewController class. This method will save the current document (which is, in our running app, simply the contents of the DudelView's drawables array) to the specified filename, and return a Boolean value indicating whether or not the save was successful. Add a method declaration to the *.h* file, and the implementation itself to the *.m* file.

```
// DudelViewController.h
- (BOOL)saveCurrentToFile:(NSString *)filename;

// DudelViewController.m
- (BOOL)saveCurrentToFile:(NSString *)filename {
  return [NSKeyedArchiver archiveRootObject:dudelView.drawables toFile:filename];
}
```

Next, add the following method, which will do the inverse, attempting to read some object data from the specified file:

```
// DudelViewController.h
- (BOOL)loadFromFile:(NSString *)filename;

// DudelViewController.m
- (BOOL)loadFromFile:(NSString *)filename {
  id root = [NSKeyedUnarchiver unarchiveObjectWithFile:filename];
  if (root) {
    dudelView.drawables = root;
  }
  [dudelView setNeedsDisplay];
  return (root != nil);
}
```

The NSKeyedArchiver and NSKeyedUnarchiver classes used in these methods are both subclasses of NSCoder, the class that we treated somewhat like a dictionary in those earlier methods. These classes know how to open a file, and either write or read its contents (depending on which class and which method you're using). They save an existing object graph by traversing all its relationships to other objects, or create a new object graph from the data in the file.

Next, let's implement the code that will actually call these utility methods, inside the existing viewDidLoad method. The first new section comes up with a fully qualified file path, including a path to our app's documents directory, where our *Untitled.dudeldoc* file will reside, and calls the loadFromFile: method. The final section sets up this class as an observer of UIApplicationWillTerminateNotification, so that we can intervene when the app is about to exit.

```
// Implement viewDidLoad to do additional setup after loading the view, typically from a
nib.
- (void)viewDidLoad {
  [super viewDidLoad];
  self.currentTool = [PencilTool sharedPencilTool];
        [dotButton setImage:[UIImage imageNamed:@"button_cdots_selected.png"]];
  self.fillColor = [UIColor colorWithWhite:0.0 alpha:0.25];
  self.strokeColor = [UIColor blackColor];
  self.font = [UIFont systemFontOfSize:12.0];
  self.strokeWidth = 2.0;
```

www.it-ebooks.info

```
  // reload default document
  NSArray *dirs = NSSearchPathForDirectoriesInDomains(NSDocumentDirectory,
    NSUserDomainMask, YES);
  NSString *filename = [[dirs objectAtIndex:0]
    stringByAppendingPathComponent:@"Untitled.dudeldoc"];
  [self loadFromFile:filename];

  [[NSNotificationCenter defaultCenter] addObserver:self
    selector:@selector(applicationWillTerminate:)
    name:UIApplicationWillTerminateNotification
    object:[UIApplication sharedApplication]];
}
```

Now for the method that's actually triggered when the app is exiting. This method does the same work to determine the filename, and calls `saveCurrentToFile:` to ensure that the user's work isn't lost.

```
- (void)applicationWillTerminate:(NSNotification *)n {
  NSArray *dirs = NSSearchPathForDirectoriesInDomains(NSDocumentDirectory,
    NSUserDomainMask, YES);
  NSString *filename = [[dirs objectAtIndex:0]
    stringByAppendingPathComponent:@"Untitled.dudeldoc"];
  [self saveCurrentToFile:filename];
}
```

Note that the repetitive file-path creation is only temporary. A little later on, when we're ready to start thinking about handling multiple document files, we'll create a new class that will take care of the paths for us.

Now compile and run your app. The first time you run it, you won't notice anything different. You'll get a blank screen, where you can go ahead and doodle something. The fun part comes when you quit the app and then restart it. You'll find that your drawing is intact! This small change leads to a huge difference in how users perceive your app, since they're able to leave their work and come back to it at any time, in a seamless manner.

Document Management

The changes we made in the previous section gave us the basics for saving and loading a single file, but we'll need more than that! The iOS doesn't have anything like the Finder, so the only access our users will have to the drawings they make is what we provide for them in our app. That means we should offer at least the following capabilities:

- See a list of all Dudel documents
- Create a new document
- Rename a document
- Delete a document
- Keep track of which document was last used

www.it-ebooks.info

Rather than putting the code for all that into one of our existing controller classes, we'll make a new class to manage the Dudel documents for us. This class will provide a single shared instance for all our other classes to use, and anyone who needs to access documents in any way will go through that shared instance.

Listing Files

Make a new class (a direct subclass of NSObject) and name it FileList. Here's the interface for *FileList.h*, which includes a notification name used to tell interested parties that the list of files has changed, properties for reading the list of all available documents and accessing the current document, and methods to do the other document-management operations:

```
//  FileList.h

#import <Foundation/Foundation.h>

// notification name
#define FileListChanged @"FileListChanged"

@interface FileList : NSObject {
  NSMutableArray *allFiles;
  NSString *currentFile;
}

@property (nonatomic, readonly) NSArray *allFiles;
@property (nonatomic, copy) NSString *currentFile;

+ (FileList *)sharedFileList;

- (void)deleteCurrentFile;
- (void)renameFile:(NSString *)oldFilename to:(NSString *)newFilename;
- (void)renameCurrentFile:(NSString *)newFilename;
- (NSString *)createAndSelectNewUntitled;

@end
```

As for the implementation, FileList uses NSFileManager to do file operations, and NSUserDefaults to keep track of the current file. This is all fairly standard Objective-C activity.

```
//  FileList.m
#import "FileList.h"
#import "SynthesizeSingleton.h"

// key for storing current filename in user defaults
#define DEFAULT_FILENAME_KEY @"defaultFilenameKey"

@implementation FileList
@synthesize allFiles;
@synthesize currentFile;

SYNTHESIZE_SINGLETON_FOR_CLASS(FileList)
```

www.it-ebooks.info

```
- init {
  if (self = [super init]) {
    NSArray *dirs = NSSearchPathForDirectoriesInDomains(NSDocumentDirectory,
      NSUserDomainMask, YES);
    NSString *dirPath = [dirs objectAtIndex:0];
    NSArray *files = [[NSFileManager defaultManager] contentsOfDirectoryAtPath:dirPath
      error:NULL];
    NSArray *sortedFiles = [[files pathsMatchingExtensions:[NSArray
      arrayWithObject:@"dudeldoc"]] sortedArrayUsingSelector:@selector(compare:)];
    allFiles = [[NSMutableArray array] retain];
    // the filenames returned by pathsMatchingExtensions: don't include the whole
    // file path, so we add it to each file here.
    for (NSString *file in sortedFiles) {
      [allFiles addObject:[dirPath stringByAppendingPathComponent:file]];
    }
    currentFile = [[[NSUserDefaults standardUserDefaults]
      stringForKey:DEFAULT_FILENAME_KEY] retain];
    if ([allFiles count]==0) {
      // there are no documents, make one!
      [self createAndSelectNewUntitled];
    } else if (![allFiles containsObject:currentFile]) {
      // user defaults are suggesting a file that doesn't exist in our documents
      // directory, so just use the first file in the list.
      self.currentFile = [allFiles objectAtIndex:0];
    }
  }
  return self;
}
- (void)setCurrentFile:(NSString *)filename {
  if (![currentFile isEqual:filename]) {
    [currentFile release];
    currentFile = [filename copy];
    [[NSUserDefaults standardUserDefaults] setObject:currentFile
      forKey:DEFAULT_FILENAME_KEY];
    [[NSNotificationCenter defaultCenter] postNotificationName:FileListChanged
      object:self];
  }
}
- (void)deleteCurrentFile {
  if (self.currentFile) {
    NSUInteger filenameIndex = [self.allFiles indexOfObject:self.currentFile];
    NSError *error = nil;
    BOOL result = [[NSFileManager defaultManager] removeItemAtPath:self.currentFile
      error:&error];

    if (filenameIndex != NSNotFound) {
      [allFiles removeObjectAtIndex:filenameIndex];
      // now figure out which file to make current
      if ([self.allFiles count]==0) {
        [self createAndSelectNewUntitled];
      } else {
        if ([self.allFiles count]==filenameIndex) {
          filenameIndex--;
        }
        self.currentFile = [self.allFiles objectAtIndex:filenameIndex];
      }
```

www.it-ebooks.info

```
    }
    [[NSNotificationCenter defaultCenter] postNotificationName:FileListChanged
      object:self];
  }
}
- (void)renameFile:(NSString *)oldFilename to:(NSString *)newFilename {
  [[NSFileManager defaultManager] moveItemAtPath:oldFilename toPath:newFilename
    error:NULL];
  if ([self.currentFile isEqual:oldFilename]) {
    self.currentFile = newFilename;
  }
  int nameIndex = [self.allFiles indexOfObject:oldFilename];
  if (nameIndex != NSNotFound) {
    [allFiles replaceObjectAtIndex:nameIndex withObject:newFilename];
  }
  [[NSNotificationCenter defaultCenter] postNotificationName:FileListChanged
    object:self];
}
- (void)renameCurrentFile:(NSString *)newFilename {
  [self renameFile:self.currentFile to:newFilename];
}
- (NSString *)createAndSelectNewUntitled {
  NSString *defaultFilename = [NSString stringWithFormat:@"Dudel %@.dudeldoc",
    [NSDate date]];
  NSArray *dirs = NSSearchPathForDirectoriesInDomains(NSDocumentDirectory,
    NSUserDomainMask, YES);
  NSString *filename = [[dirs objectAtIndex:0] stringByAppendingPathComponent:
    defaultFilename];
  [[NSFileManager defaultManager] createFileAtPath:filename contents:nil
    attributes:nil];
  [allFiles addObject:filename];
    [allFiles sortUsingSelector:@selector(compare:)];
  self.currentFile = filename;
  [[NSNotificationCenter defaultCenter] postNotificationName:FileListChanged
    object:self];
  return self.currentFile;
}
@end
```

Adding a File List Controller

The FileList class will be used by both DudelViewController and our application delegate, as well as one other class we haven't yet created: FileListViewController, which will display a list of all the files and let the user select a file to change the current selection. Create a new UIViewController subclass, and click the check boxes to make it iPad-ready and a subclass of UITableViewController (but no *.xib* file). Then name it FileListViewController. This is going to be a quite standard controller for a table view, using FileList to see what it should be displaying. Here's the code for both the *.h* and *.m* files (by now, the structure of a table view controller should look familiar to you):

```
//  FileListController.h
#import <UIKit/UIKit.h>

// notification name
#define FileListControllerSelectedFile @"FileListControllerSelectedFile"
```

www.it-ebooks.info

```
#define FileListControllerFilename @"FileListControllerFilename"

@interface FileListViewController : UITableViewController {
  NSString *currentDocumentFilename;
  NSArray *documents;
}
@property (nonatomic, copy) NSString *currentDocumentFilename;
@property (nonatomic, retain) NSArray *documents;
@end

//  FileListController.m
#import "FileListViewController.h"
#import "FileList.h"

@implementation FileListViewController
@synthesize currentDocumentFilename, documents;
- (void)reloadData {
  self.currentDocumentFilename = [FileList sharedFileList].currentFile;
  self.documents = [FileList sharedFileList].allFiles;
  [self.tableView reloadData];
}
- (void)fileListChanged:(NSNotification *)n {
  [self reloadData];
}
- (void)viewDidLoad {
  [super viewDidLoad];
  [[NSNotificationCenter defaultCenter] addObserver:self
    selector:@selector(fileListChanged:) name:FileListChanged
    object:[FileList sharedFileList]];
}
- (void)viewWillAppear:(BOOL)animated {
  [super viewWillAppear:animated];
  [self reloadData];
}
- (BOOL)shouldAutorotateToInterfaceOrientation:(UIInterfaceOrientation)orientation {
  return YES;
}
- (NSInteger)numberOfSectionsInTableView:(UITableView *)tableView {
  return 1;
}
- (NSInteger)tableView:(UITableView *)tableView numberOfRowsInSection:(NSInteger)s {
  return [self.documents count];
}
- (UITableViewCell *)tableView:(UITableView *)tableView
  cellForRowAtIndexPath:(NSIndexPath *)indexPath {
  static NSString *CellIdentifier = @"Cell";
  UITableViewCell *cell = [tableView dequeueReusableCellWithIdentifier:
    CellIdentifier];
  if (cell == nil) {
    cell = [[[UITableViewCell alloc] initWithStyle:UITableViewCellStyleDefault
      reuseIdentifier:CellIdentifier] autorelease];
  }
  NSString *file = [self.documents objectAtIndex:indexPath.row];
  cell.textLabel.text = [[file lastPathComponent] stringByDeletingPathExtension];
  if ([file isEqual:self.currentDocumentFilename]) {
    cell.accessoryType = UITableViewCellAccessoryCheckmark;
  } else {
```

www.it-ebooks.info

```
    cell.accessoryType = UITableViewCellAccessoryNone;
  }
  return cell;
}
- (void)tableView:(UITableView *)tv didSelectRowAtIndexPath:(NSIndexPath *)indexPath {
  NSDictionary *userInfo = [NSDictionary dictionaryWithObject:[documents
    objectAtIndex:indexPath.row] forKey:FileListControllerFilename];
  [[NSNotificationCenter defaultCenter]
    postNotificationName:FileListControllerSelectedFile object:self userInfo:userInfo];
  [self reloadData];
}
- (void)dealloc {
  [[NSNotificationCenter defaultCenter] removeObserver:self];
  self.currentDocumentFilename = nil;
  self.documents = nil;
  [super dealloc];
}
@end
```

The only interesting thing this class does is register for FileList's notification about changes, so that the view can be updated automatically. Also, whenever the user selects a row here, FileListViewController posts a notification to that effect. We'll use that later in this chapter, to be able to update our main DudelViewController and DudelView whenever that happens.

Changing the App Delegate

For the first time since we started on Dudel, it's time to make some changes to the app delegate. Changes are required here because we're going to rearrange the top-level view arrangement of our application. Inside the *.xib* file, we'll be making a UISplitViewController the root view controller, with instances of FileListViewController and DudelViewController as its "children" (up until now, DudelViewController was the root view controller).

Before we edit the *.xib* file, let's make the necessary preparations to *DudelAppController.h* (the app delegate), basically just adding a couple of outlets:

```
//  DudelAppDelegate.h
#import <UIKit/UIKit.h>
@class DudelViewController;
@class FileListViewController;
@interface DudelAppDelegate : NSObject <UIApplicationDelegate> {
  UIWindow *window;
  DudelViewController *viewController;
  FileListViewController *fileListController;
  UISplitViewController *splitViewController;
}
@property (nonatomic, retain) IBOutlet UIWindow *window;
@property (nonatomic, retain) IBOutlet DudelViewController *viewController;
@property (nonatomic, retain) IBOutlet FileListViewController *fileListController;
@property (nonatomic, retain) IBOutlet UISplitViewController *splitViewController;
@end
```

www.it-ebooks.info

Now open *MainWindow.xib* in Interface Builder. By default, the main *.xib* window shows you only the top-level structure of the items contained in the *.xib*, but we're going to need to fix the plumbing here a bit. First, switch to the column view by clicking the appropriate button, as shown in Figure 8–3.

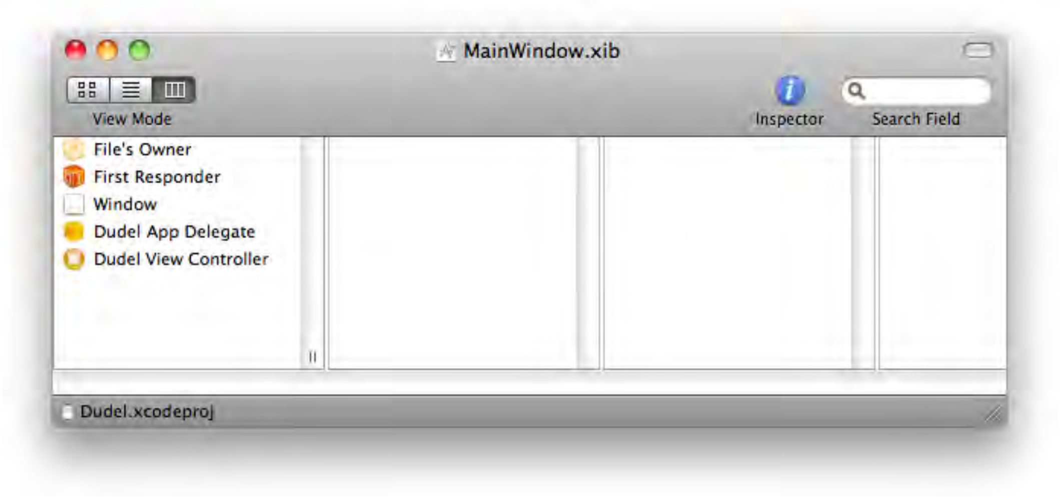


Figure 8–3. *The default contents of the MainWindow.xib file created with your project*

We're going to add a `UISplitViewController`, move the `DudelViewController` into it, and then add a `FileListController` to the mix. Start by finding a `UISplitViewController` in the Library and dragging it to the first column shown in the main window. Then click the new split view controller to see what's inside it, and click the navigation controller in there to see what it contains. You should see something like Figure 8–4.

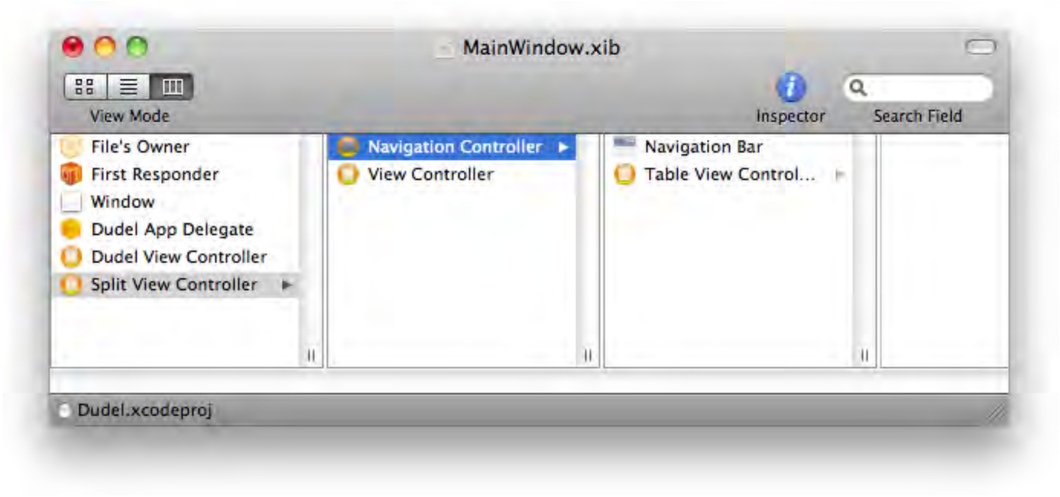


Figure 8–4. *We've put a split view in place. Now we just need to give it the correct contents.*

www.it-ebooks.info

In the second column, the item labeled View Controller is where we want to have our `DudelViewController` now. And we want our `FileListViewController` to be in the item labeled Table View Controller in the third column.

This window does resemble a Finder window, so you might think you could just drag the Dudel view controller already at the top level into the split view controller, but that won't work. Instead, delete the top-level Dudel View Controller item, then click the Split View Controller item and select the view controller it contains. Open the identity inspector, and change its class to `DudelViewController`. Then click the Navigation Controller item, select the Table View Controller item it contains, and change its class to `FileListViewController`. You should now see something like Figure 8–5.

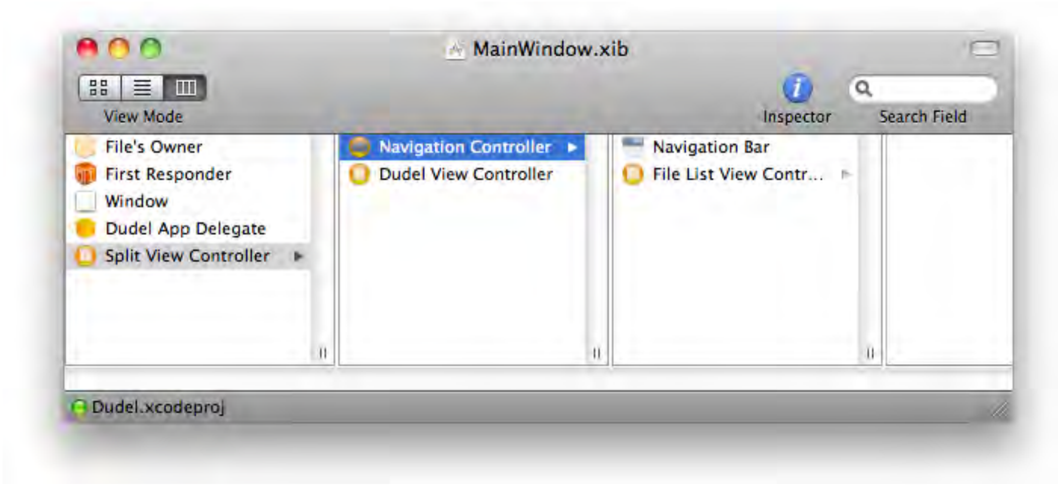


Figure 8–5. *A look at the completed reorganization*

Now all that's left here is to make some connections between objects in the *.xib* file. Connect each of `DudelAppDelegate`'s outlets to the appropriate view controllers in the nib file: `splitViewController`, `fileListController`, and `viewController`. Previously, the `viewController` outlet was connected to the old `DudelViewController`, but since we deleted that and are using a new one instead, you'll need to reconnect that outlet to the new `DudelViewController` inside the split view. Also, connect the `UISplitViewController`'s `delegate` outlet to the `DudelViewController`. This seems tricky, since the latter is contained inside the former, but as long as you're in column view, you shouldn't have a problem.

Save your work, and go back to Xcode, where it's time to finish the changes required for *DudelAppDelegate.m*. We're doing two main things here: switching out references to the top-level view controller and observing a notification from the `FileListViewController` class, so that whenever the user selects a file, we can set up the `DudelViewController` with the contents of the newly selected file.

```
//  DudelAppDelegate.m
#import "DudelAppDelegate.h"
#import "DudelViewController.h"
```

www.it-ebooks.info

```
#import "FileListViewController.h"
#import "FileList.h"
@implementation DudelAppDelegate
@synthesize window;
@synthesize viewController;
@synthesize fileListController;
@synthesize splitViewController;
- (BOOL)application:(UIApplication *)application
didFinishLaunchingWithOptions:(NSDictionary *)launchOptions {
  // Override point for customization after app launch
~~  [window addSubview:viewController.view];~~
  [window addSubview:splitViewController.view];
  [window makeKeyAndVisible];
  [[NSNotificationCenter defaultCenter] addObserver:self
    selector:@selector(fileListControllerSelectedFile:)
    name:FileListControllerSelectedFile object:fileListController];
  return YES;
}
- (void)fileListControllerSelectedFile:(NSNotification *)n {
  NSString *oldFilename = [FileList sharedFileList].currentFile;
  [viewController saveCurrentToFile:oldFilename];
  NSString *filename = [[n userInfo] objectForKey:FileListControllerFilename];
  [FileList sharedFileList].currentFile = filename;
  [viewController loadFromFile:filename];
}
- (void)dealloc {
  [[NSNotificationCenter defaultCenter] removeObserver:self];
~~  [viewController release];~~
  [splitViewController release];
  [window release];
  [super dealloc];
}
@end
```

With that in place, we're getting very close to having a working split view up and running. Hang tight! The next step is to modify *DudelViewController.m*, removing the temporary "hack" we put in place for loading and saving a file to a single, hard-coded location. Start by importing the header for `FileList` somewhere at the top of the file:

```
#import "FileList.h"
```

Then, in both `viewDidLoad` and `applicationWillTerminate:`, make the following change. This eliminates the lengthy path construction, and instead just asks `FileList` for the current file.

```
~~  NSArray *dirs = NSSearchPathForDirectoriesInDomains(NSDocumentDirectory,~~
~~    NSUserDomainMask, YES);~~
~~  NSString *filename = [[dirs objectAtIndex:0]~~
~~    stringByAppendingPathComponent:@"Untitled.dudeldoc"];~~
  NSString *filename = [FileList sharedFileList].currentFile;
```

We also need to add an outlet from `DudelViewController` to the toolbar in its display, so that we can add and remove toolbar items as the iPad rotates. Open *DudelViewController.h* and add the following line:

```
  IBOutlet UIToolbar *toolbar;
```

www.it-ebooks.info

To connect it, open *DudelViewController.xib* in Interface Builder, double-click to open the DudelView if it's not already open, then control-drag from the File's Owner icon to the toolbar at the bottom of the DudelView, and select toolbar from the pop-up menu.

One final step is necessary for the split view controller. I mentioned earlier that the actual split view is shown only in landscape mode, and that when switching to portrait mode, we instead get a UIBarButtonItem (which is set up to open a popover) passed to the UISplitViewController's delegate (our DudelViewController instance). Likewise, another delegate method is called when switching to landscape mode, telling us that the UIBarButtonItem we were passed earlier is no longer valid. We'll implement these two methods, so that in the first method, we add the item to our toolbar, along with a flexible spacer so that it stands slightly removed from the tools. In the second method, we remove the two items.

```
- (void)splitViewController:(UISplitViewController*)svc
  willHideViewController:(UIViewController *)aViewController
  withBarButtonItem:(UIBarButtonItem*)barButtonItem
  forPopoverController:(UIPopoverController*)pc {
  // insert the new item and a spacer into the 33
  NSMutableArray *newItems = [[toolbar.items mutableCopy] autorelease];
  [newItems insertObject:barButtonItem atIndex:0];
  UIBarButtonItem *spacer = [[[UIBarButtonItem alloc]
    initWithBarButtonSystemItem:UIBarButtonSystemItemFlexibleSpace target:nil
    action:nil] autorelease];
  [newItems insertObject:spacer atIndex:1];
  [toolbar setItems:newItems animated:YES];
  // configure display of the button
  barButtonItem.title = @"My Dudels";
}
- (void)splitViewController:(UISplitViewController*)svc
  willShowViewController:(UIViewController *)aViewController
  invalidatingBarButtonItem:(UIBarButtonItem *)button {
  // remove the button, and the spacer that is beside it
  NSMutableArray *newItems = [[toolbar.items mutableCopy] autorelease];
  if ([newItems containsObject:button]) {
    [newItems removeObject:button];
    [newItems removeObjectAtIndex:0];
    [toolbar setItems:newItems animated:YES];
  }
}
```

We also need to implement a third delegate method, which is called when the popover created by the UISplitViewController is about to be displayed:

```
- (void)splitViewController:(UISplitViewController*)svc
  popoverController:(UIPopoverController*)pc
  willPresentViewController:(UIViewController *)aViewController {
  // we don't create this popover on our own, but we want to notice it so that
  // we can dismiss any other popovers, and also remove it later.
  if (self.currentPopover) {
    [self.currentPopover dismissPopoverAnimated:YES];
    [self handleDismissedPopoverController:self.currentPopover];
  }
  self.currentPopover = pc;
}
```

www.it-ebooks.info

The point of this is mainly just to make sure that we're not showing multiple popovers, as we've done with all the other popovers.

At this point, you should now be able to build and run your app, and—finally!—see the split view in action. Rotate to landscape mode, and the file list appears on the left. Rotate to portrait mode, and the file list disappears, but in its place, there's a button at the left edge of the toolbar that brings up the file list in a popover.

That's great, but there's still a bit of a problem. The file list has only one item, which you can't rename or delete, and there's no way to make a new item! To remedy this, we need to add a few more things.

Creating and Deleting Files

All this time, we've had a button in the lower-right corner of our toolbar just for the purpose of sending our drawing as a PDF in an e-mail message. That's still a nice piece of functionality, but we can do more with that space—namely, replace it with a button that launches a small menu in another popover.

Let's start by creating yet another `UIViewController` subclass, once again a `UITableViewController` subclass with no *.xib* file, named `ActionsMenuController`. Like some of the other view controllers we've made, this one defines a notification name that's used when the user selects an item in the list it's going to display. It also defines an enumerated type that will show which of the menu items was selected. Here's the entire content of both the *.h* and *.m* files:

```
//  ActionsMenuController.h
#import <UIKit/UIKit.h>
#define ActionsMenuControllerDidSelect @"ActionsMenuControllerDidSelect"
typedef enum SelectedActionType {
  NoAction = -1,
  NewDocument,
  RenameDocument,
  DeleteDocument,
  EmailPdf,
  ShowAppInfo
} SelectedActionType;
@interface ActionsMenuController : UITableViewController {
  SelectedActionType selection;
  UIPopoverController *container;
}
@property (readonly) SelectedActionType selection;
@property (assign, nonatomic) UIPopoverController *container;
@end

//  ActionsMenuController.m
#import "ActionsMenuController.h"
@implementation ActionsMenuController
@synthesize selection, container;
- (void)viewWillAppear:(BOOL)animated {
  [super viewWillAppear:animated];
  selection = NoAction;
}
```

www.it-ebooks.info

```
- (BOOL)shouldAutorotateToInterfaceOrientation:(UIInterfaceOrientation)orientation {
  return YES;
}
- (NSInteger)numberOfSectionsInTableView:(UITableView *)tableView {
return 1;
}
- (NSInteger)tableView:(UITableView *)tableView numberOfRowsInSection:(NSInteger)s {
return 5;
}
- (UITableViewCell *)tableView:(UITableView *)tableView
  cellForRowAtIndexPath:(NSIndexPath *)indexPath {
  static NSString *CellIdentifier = @"Cell";
  UITableViewCell *cell = [tableView dequeueReusableCellWithIdentifier:CellIdentifier];
  if (cell == nil) {
    cell = [[[UITableViewCell alloc] initWithStyle:UITableViewCellStyleDefault
      reuseIdentifier:CellIdentifier] autorelease];
  }
  switch (indexPath.row) {
    case NewDocument:
      cell.textLabel.text = @"New Dudel";
      break;
    case RenameDocument:
      cell.textLabel.text = @"Rename this Dudel";
      break;
    case DeleteDocument:
      cell.textLabel.text = @"Delete this Dudel";
      break;
    case ShowAppInfo:
      cell.textLabel.text = @"Dudel App Info";
      break;
    case EmailPdf:
      cell.textLabel.text = @"Send PDF via email";
      break;
    default:
      break;
  }
  return cell;
}
- (void)tableView:(UITableView *)tableView didSelectRowAtIndexPath:(NSIndexPath
*)indexPath {
  selection = indexPath.row;
  [[NSNotificationCenter defaultCenter]
    postNotificationName:ActionsMenuControllerDidSelect object:self];
}
@end
```

Switch over to `DudelViewController`, where we're going to make some slight changes:

Add this to the instance variables:

```
IBOutlet UIToolbar *toolbar;
```

Remove this action method:

~~- (IBAction)touchSendPdfEmailItem:(id)sender;~~

And add this action method:

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

www.it-ebooks.info

Open *DudelViewController.xib* in Interface Builder, and select the button farthest to the right in the toolbar. Open the Inspector panel, clear out the Title field, and set the Identifier to Action. Then control-drag from the button to `DudelViewController`, and select the `popoverActionsMenu:` action. Now save your work, and go back to Xcode.

We'll need to make some changes to *DudelViewController.m* to match the header file's changes, and to handle this new menu controller. For starters, add this near the top:

```
#import "ActionsMenuController.h"
```

Then create the following methods:

```
- (IBAction)popoverActionsMenu:(id)sender {
  ActionsMenuController *amc = [[[ActionsMenuController alloc] initWithNibName:nil
    bundle:nil] autorelease];
  [self setupNewPopoverControllerForViewController:amc];
  amc.container = self.currentPopover;
  self.currentPopover.popoverContentSize = CGSizeMake(320, 44*5);
  [[NSNotificationCenter defaultCenter] addObserver:self
    selector:@selector(actionsMenuControllerDidSelect:)
    name:ActionsMenuControllerDidSelect object:amc];
  [self.currentPopover presentPopoverFromBarButtonItem:sender
    permittedArrowDirections:UIPopoverArrowDirectionAny animated:YES];
}
- (void)actionsMenuControllerDidSelect:(NSNotification *)notification {
 ActionsMenuController *amc = [notification object];
 UIPopoverController *popoverController = amc.container;
 [popoverController dismissPopoverAnimated:YES];
 [self handleDismissedPopoverController:popoverController];
 self.currentPopover = nil;
}
- (void)createDocument {
  [self saveCurrentToFile:[FileList sharedFileList].currentFile];
  [[FileList sharedFileList] createAndSelectNewUntitled];
  dudelView.drawables = [NSMutableArray array];
  [dudelView setNeedsDisplay];
}
- (void)deleteCurrentDocumentWithConfirmation {
  [[[[UIAlertView alloc] initWithTitle:@"Delete current Dudel" message:
  @"This will remove your current drawing completely. Are you sure you want to do that?"
  delegate:self cancelButtonTitle:@"Cancel" otherButtonTitles:@"Delete it!", nil]
  autorelease] show];
}
- (void)renameCurrentDocument {
  // hold on, we're not quite ready for this yet
}
- (void)showAppInfo {
  // not ready for this one, either!
}
// UIAlertView delegate method, called by the delete confirmation alert.
// we're only using one UIAlertView right now, so no need to check which
// one this is, just which button was pressed.
- (void)alertView:(UIAlertView *)alertView clickedButtonAtIndex:(NSInteger)buttonIndex {
  if (buttonIndex == 1) {
    [[FileList sharedFileList] deleteCurrentFile];
    [self loadFromFile:[FileList sharedFileList].currentFile];
  }
```

www.it-ebooks.info

```
}
```

Then replace this:

~~- (IBAction)touchSendPdfEmailItem:(id)sender {~~

with this:

```
- (void)sendPdfEmail {
```

We need to add an additional check, which contains another bit of checking all its own, to our `handleDismissedPopoverController:` method, down near the end of the method (just before the return):

```
} else if ([popoverController.contentViewController
isMemberOfClass:[ActionsMenuController class]]) {
    ActionsMenuController *amc = (ActionsMenuController
*)popoverController.contentViewController;
    switch (amc.selection) {
      case NewDocument:
        [self createDocument];
        break;
      case RenameDocument:
        [self renameCurrentDocument];
        break;
      case DeleteDocument:
        [self deleteCurrentDocumentWithConfirmation];
        break;
      case EmailPdf:
        [self sendPdfEmail];
        break;
      case ShowAppInfo:
        [self showAppInfo];
        break;
      default:
        break;
    }
}
```

Now build and run your app, and try that on for size! You can now both create and delete Dudel documents, and switch between them. Each new document you create will be given a default name containing a timestamp, to ensure its uniqueness. All that's left to do now is to add the ability to rename the files you create here, and to display a brief info panel with another menu item. Both of these will be accomplished using modal displays.

Renaming Files

First, let's get the file renaming working. The idea here is that creating a new document should be instantaneous, with the default filename working as a placeholder until the time when the user decides to give it a name. At that point, the user can invoke this functionality through the menu.

Start by making a new `UIViewController` subclass. This one will *not* be a `UITableViewController` subclass, and it needs to have a matching *.xib* file, where we'll

www.it-ebooks.info

define a simple GUI for renaming a file. Name this controller class `FileRenameViewController`. Here's the content of the *FileRenameViewController.h* file:

```
//  FileRenameViewController.h
#import <UIKit/UIKit.h>
@protocol FileRenameViewControllerDelegate;
@interface FileRenameViewController : UIViewController {
  id <FileRenameViewControllerDelegate> delegate;
  NSString *originalFilename;
  NSString *changedFilename;
  IBOutlet UILabel *textLabel;
  IBOutlet UITextField *textField;
}
@property (nonatomic, retain) id <FileRenameViewControllerDelegate> delegate;
@property (nonatomic, copy) NSString *originalFilename;
@property (nonatomic, copy) NSString *changedFilename;
@end
@protocol FileRenameViewControllerDelegate
- (void)fileRenameViewController:(FileRenameViewController *)c
  didRename:(NSString *)oldFilename to:(NSString *)newFilename;
@end
```

One of the new features in iOS 3.2 is the ability to use presentation styles when displaying a modal view. In older versions of iOS, modal views always filled the screen, but here we're going to use a presentation style called `UIModalPresentationFormSheet`, which presents a 540-by-620 view, centered in the screen, with the rest of screen grayed out. This view slides in from the bottom of the screen, and since it doesn't cover the entire screen, it's slightly less jarring. Let's set it up now.

Open *FileRenameViewController.xib* in Interface Builder. Select the view, and use the attribute inspector to disable its status bar (as we've done for several other views in Dudel), which will let us resize the view. Use the size inspector to set its size to 540 by 620. Next, use the Library to find a `UILabel` and a `UITextfield`, dragging each of them into the view. Switch back to the attribute inspector, and set the font for each of those components to 24-point Helvetica. For the `UILabel`, also set its # Lines to 0, which will let it display text on multiple lines if necessary. Then make them each nearly fill the width of the view, and position them well above center (in order to leave space for the keyboard), something like what you see in Figure 8–6. For bonus points, use the attributes inspector to set the text field's placeholder text to **Entire Filename**.

www.it-ebooks.info

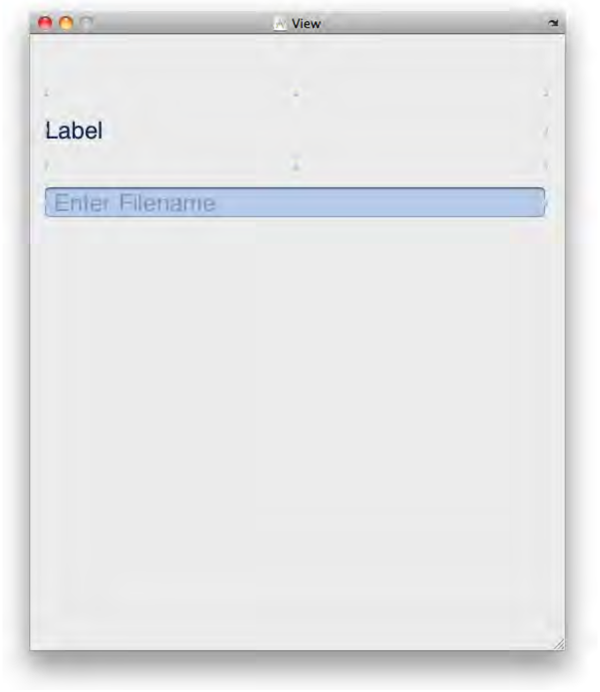


Figure 8–6. *Laying out GUI components for the FileRenameViewController. Notice the blue resize handles that extend nearly to the sides of the view.*

Now all we need to do is connect the `textField` and `textLabel` buttons from the File's Owner icon to the appropriate GUI components, and connect the text field's `delegate` outlet back to File's Owner. Save your *.xib*. This GUI is done!

Switch back to Xcode, and enter this code for *FileRenameViewController.m*:

```
//  FileRenameViewController.m
#import "FileRenameViewController.h"
#import "FileList.h"
@implementation FileRenameViewController
@synthesize delegate;
@synthesize originalFilename;
@synthesize changedFilename;
- (void)viewWillAppear:(BOOL)animated {
  [super viewWillAppear:animated];
  textField.text = [[originalFilename lastPathComponent] stringByDeletingPathExtension];
  textLabel.text = @"Please enter a new file name for the current Dudel.";
}
- (void)viewDidAppear:(BOOL)animated {
  [super viewDidAppear:animated];
  [textField becomeFirstResponder];
}
- (BOOL)shouldAutorotateToInterfaceOrientation:(UIInterfaceOrientation)orientation {
  return YES;
}
- (void)dealloc {
```

www.it-ebooks.info

```
  self.delegate = nil;
  self.originalFilename = nil;
  self.changedFilename = nil;
  [super dealloc];
}
- (void)textFieldDidEndEditing:(UITextField *)tf {
  NSString *dirPath = [originalFilename stringByDeletingLastPathComponent];
  self.changedFilename = [[dirPath stringByAppendingPathComponent:tf.text]
    stringByAppendingPathExtension:@"dudeldoc"];
  if ([[FileList sharedFileList].allFiles containsObject:self.changedFilename]) {
    textLabel.text =
      @"A file with that name already exists! Please enter a different file name.";
  } else {
    [[FileList sharedFileList] renameFile:self.originalFilename
      to:self.changedFilename];
    [delegate fileRenameViewController:self didRename:originalFilename
      to:changedFilename];
  }
}
- (BOOL)textFieldShouldReturn:(UITextField *)tf {
  [tf endEditing:YES];
  return YES;
}
@end
```

Now let's set up DudelViewController to use the new renaming mechanism. Starting in the header, add this:

```
#import "FileRenameViewController.h"
```

Then add a protocol to the list of protocols our controller implements:

```
@interface DudelViewController : UIViewController <ToolDelegate, DudelViewDelegate,
  MFMailComposeViewControllerDelegate, UIPopoverControllerDelegate,
  FileRenameViewControllerDelegate> {
```

Switch to the *.m* file, and fill in this method's body:

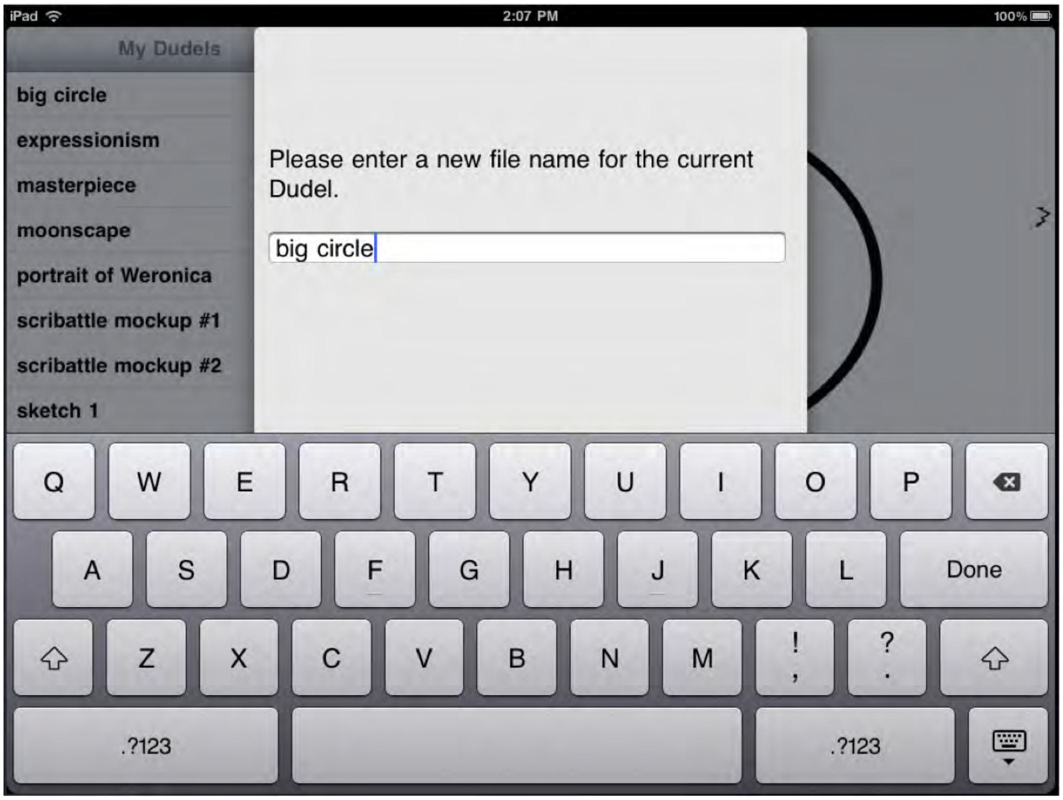
```
- (void)renameCurrentDocument {
  FileRenameViewController *controller = [[[FileRenameViewController alloc]
    initWithNibName:@"FileRenameViewController" bundle:nil] autorelease];
  controller.delegate = self;
  controller.modalPresentationStyle = UIModalPresentationFormSheet;
  controller.originalFilename = [FileList sharedFileList].currentFile;
  [self presentModalViewController:controller animated:YES];
}
```

Finally, implement the delegate method that gets called when the view's modal session is done:

```
- (void)fileRenameViewController:(FileRenameViewController *)c
  didRename:(NSString *)oldFilename to:(NSString *)newFilename {
  [self dismissModalViewControllerAnimated:YES];
}
```

Now build and run your app. You should be able to select a file in the list, and rename it using the **Rename this Dudel** menu item, as shown in Figure 8-7.

www.it-ebooks.info



Please enter a new file name for the current Dudel.

big circle

Figure 8–7. *The FileRenameViewController in action*

Implementing an About Panel in a Modal Way

The final feature we want in this chapter is to implement a sort of About panel, similar to what you can typically find in Mac OS X applications. iPhone applications don't often have these panels, but on the iPad, we have a little more space. In Dudel, we already have an action menu that's a good place to access such a feature, so off we go.

We'll present a `UIWebView` with information about Dudel and links at the bottom to get additional information. This web view will be presented modally, just like the file-renaming view.

As you may know, the Dudel application created during the writing of this book is actually available on the App Store as a free download. This is partly because while creating it, we found it was fun to use and worthy of making available to others. But we also figured that it could be a good way to promote the book itself!

The shipping version of Dudel includes an info screen that describes this book, tells the users that they can buy this book if they want to see how the app was made, and provides links to the Apress and Amazon web sites. In order to keep our promise to include full details on how to make this app, we're going to make the same info screen

www.it-ebooks.info

that could lead you to buying this book—if you didn't already have a copy, which you do. How self-referential is this?

Creating the Modal Web View Controller

Start by creating a new `UIViewController` class called `ModalWebViewController`, again with an *.xib* file and without being a subclass of `UITableViewController`. Give it the following interface declaration in *ModalWebViewController.h*:

```
//  ModalWebViewController.h
#import <UIKit/UIKit.h>
@protocol ModalWebViewControllerDelegate;
@interface ModalWebViewController : UIViewController {
  id <ModalWebViewControllerDelegate> delegate;
  UIWebView *webView;
}
@property (nonatomic, assign) id <ModalWebViewControllerDelegate> delegate;
@property (nonatomic, retain) IBOutlet UIWebView *webView;
- (IBAction)done;
- (IBAction)apressSite;
- (IBAction)amazonSite;

@end
@protocol ModalWebViewControllerDelegate
- (void)modalWebViewControllerDidFinish:(ModalWebViewController *)controller;
@end
```

Now open *ModalWebViewController.xib* in Interface Builder, and once again use the attribute inspector to disable the view's status bar. Then use the size inspector to set the view's size to 540 by 620. Use the Library to find a `UIToolbar` and put it at the bottom of the view, and then add a `UIWebView` to fill up the rest of the view. Now put five `UIBarButtonItems` into the toolbar. Use the attribute inspector to set the fourth button's identifier to **Flexible Space**, and the fifth item's identifier to **Done**. Then set the titles on the remaining three to **More Info**, **Buy the eBook**, and **Buy the Print Book**, respectively, as shown in Figure 8–8.

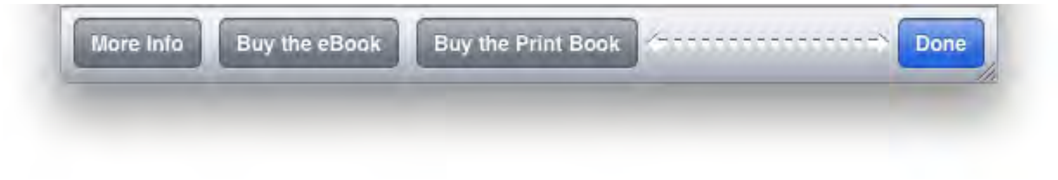


Figure 8–8. *The toolbar for our ModalWebViewController*

Control-drag to connect each of the four clickable buttons to the appropriate action methods in File's Owner: `apressSite`, `apressSite`, `amazonSite`, and `done`. (Yes, we're reusing the same method for both the More Info and Buy the eBook actions.)

Now control-drag from the File's Owner icon to the `UIWebView`, and connect the `webView` outlet. The GUI is complete. Save your work and switch back to Xcode to finish up this class. Here's the code for *ModalWebViewController.m*:

www.it-ebooks.info

```
//  ModalWebViewController.m
#import "ModalWebViewController.h"
@implementation ModalWebViewController
@synthesize delegate;
@synthesize webView;
- (void)viewDidLoad {
  // Load the bookInfo.html file into the UIWebView.
  NSString *path = [[NSBundle mainBundle] pathForResource:@"bookInfo" ofType:@"html"];
  NSURL *url = [NSURL fileURLWithPath:path];
  NSURLRequest *request = [NSURLRequest requestWithURL:url];
  [self.webView loadRequest:request];
  [super viewDidLoad];
}
- (IBAction)done {
  // The Done button was tapped, so close Modal Web View.
  [self.delegate modalWebViewControllerDidFinish:self];
}
- (IBAction)apressSite {
  // Go to the Apress.com book web page in Mobile Safari.
  NSURL *url = [NSURL URLWithString:@"http://www.apress.com/book/view/9781430230212"];
  [[UIApplication sharedApplication] openURL:url];
}
- (IBAction)amazonSite {
  // Go to the Amazon.com book web page in Mobile Safari.
  NSURL *url = [NSURL URLWithString:@"http://www.amazon.com/dp/1430230215/"];
  [[UIApplication sharedApplication] openURL:url];
}
- (BOOL)shouldAutorotateToInterfaceOrientation:(UIInterfaceOrientation)orientation {
  // Overridden to allow any orientation.
  return YES;
}
- (void)dealloc {
  [webView release];
  [super dealloc];
}
@end
```

Displaying a Web Page

Our modal web view controller code references an HTML page whose content should be displayed in the info panel. This is a standard HTML/CSS document, which displays some text and an image. That file is included in the source code archive for this book (along with the image file it references, *booktitle.png*). You can copy both files from the source code archive into your project, or just make a new empty *bookInfo.html* file in your Xcode project using the New File Assistant (pick Empty File from the Other section) and giving it something like the following content:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta name="viewport" content="width=540" />
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
<title>Beginning iPad Development for iPhone Developers: Mastering the iPad SDK</title>
<style type="text/css">
```

www.it-ebooks.info

```
<!--
body {
  background: #000 url(booktitle.png) fixed bottom no-repeat;
  margin: 0px;
  font-family: Arial, Helvetica, sans-serif;
  font-size: 20px;
  color: #FC3;
}
a:link { color: #FFF; }
a:visited { color: #FFF; }
a:hover { color: #FFF; }
a:active { color: #FFF; }
.intro { padding: 30px; -align: center; }
-->
</style></head>
<body><div class="intro">Leverage your iPhone development skills to build apps for the
iPad. Learn how to utilize all of the new iPad SDK features from <b>Dudel</b> in your
own apps, plus so much more! This book includes the full source code for <b>Dudel</b>.
</p>
</body>
</html>
```

Integrating with the Dudel View Controller

Now let's deal with DudelViewController, where we'll tie up the loose ends needed to display this web view. Start with the *DudelViewController.h* file, adding this line:

```
#import "ModalWebViewController.h"
```

Also add one more protocol to the list for the DudelViewController class:

```
@interface DudelViewController : UIViewController <ToolDelegate, DudelViewDelegate,
  MFMailComposeViewControllerDelegate, UIPopoverControllerDelegate,
  FileRenameViewControllerDelegate, ModalWebViewControllerDelegate > {
```

Then switch to *DudelViewController.m* and fill out the following method, which was previously empty:

```
- (void)showAppInfo {
  // The About the Book button was tapped, so display the Modal Web View.
  ModalWebViewController *controller = [[[ModalWebViewController alloc]
    initWithNibName:@"ModalWebViewController" bundle:nil] autorelease];
  controller.delegate = self;
  // UIModalPresentationFormSheet has a fixed 540 pixel width and 620 pixel height.
  controller.modalPresentationStyle = UIModalPresentationFormSheet;
  [self presentModalViewController:controller animated:YES];
}
```

Finally, handle the removal of the modal web view by implementing the following delegate method to dismiss it:

```
- (void)modalWebViewControllerDidFinish:(ModalWebViewController *)controller {
  [self dismissModalViewControllerAnimated:YES];
}
```

www.it-ebooks.info

Now you should be able to build and run the app, bring up the info panel through the action menu, and see something very much like the handsome page shown in Figure 8–9.

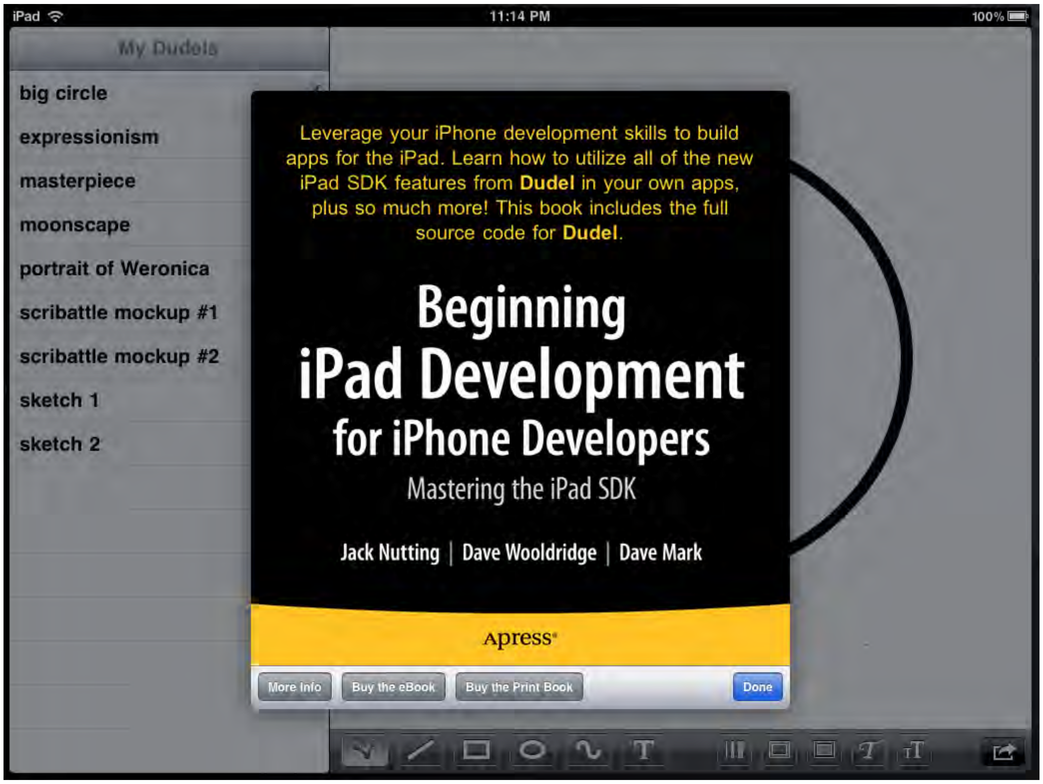

iPad
11:14 PM
100%
My Dudels
big circle
expressionism
masterpiece
moonscape
portrait of Weronica
scribattle mockup #1
scribattle mockup #2
sketch 1
sketch 2
Leverage your iPhone development skills to build apps for the iPad. Learn how to utilize all of the new iPad SDK features from Dudel in your own apps, plus so much more! This book includes the full source code for Dudel.
Beginning iPad Development for iPhone Developers
Mastering the iPad SDK
Jack Nutting | Dave Wooldridge | Dave Mark
Apress
More Info
Buy the eBook
Buy the Print Book
Done

Figure 8–9. *The black ink used for this image alone is enough to print an entire book in a third-world country.*

Obviously, you'll be missing most of that if you didn't copy the image from the book's source code archive, but you get the idea.

Let's Split

We've covered a lot of ground in this chapter. You've seen how the `UISplitViewController` works and how to display a modal view without taking up the entire screen. You've also learned the basics of how to save and load documents in a way that can be somewhat simpler to use than Core Data (though for document types that are more complex than just a single array of items, you should really invest the time and effort to use Core Data instead). Along the way, you saw one possible approach to the issue of dealing with data files stored by your application, which is worth some consideration, since iOS doesn't provide any way for users to deal with them otherwise.

In the next chapter, we'll move on to another important aspect of apps. There, you'll learn some great new tricks that iOS 3.2 introduces for getting input from users.

www.it-ebooks.info

www.it-ebooks.info

Chapter 9

New Input Methods

You've already seen how iOS 3.2 provides several new techniques for displaying and arranging content, giving you more flexibility in presenting data to your users. But using iOS isn't a one-way street. Interactivity is crucial to the iPhone/iPad user experience.

Each major new release of iOS has provided new ways for users to interact with applications. iOS 3.2 is no exception, adding customizable menus for text editing, built-in gesture recognition for interpreting and handling sequences of touches, and new functionality for extending or even replacing the standard on-screen keyboard. This chapter explores each of these areas, demonstrating how to put them to use in a variety of applications. First, we'll look at how to add items to the little text-editing menu that pops up in response to a user pressing and hold a finger over a text-input object. Next, we'll deal with gesture recognition. Finally, you'll see how to customize the on-screen keyboard.

Menu Additions

Anyone who has edited text on an iOS device is probably familiar with the text-editing menu that appears at various times, hovering over the text area in response to user actions, as shown in Figure 9–1. This is a context-sensitive menu that displays only items relevant to the current selection (unlike menus in Mac OS X, where unavailable items are grayed out).

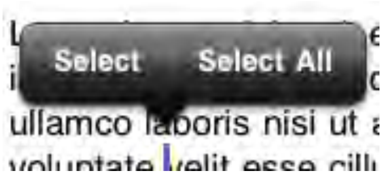


Figure 9–1. *The basic menu that's shown when no text is selected*

Starting with i OS 3.2, it's now possible for developers to tap into this functionality. You can set up a list of your own menu items that will be added to the menu, and also implement functionality to enable and disable your menu items on the fly, depending on

www.it-ebooks.info

the current selection or any other factors you want to take into account. (You can't do anything about the menu items that the system provides.)

The gateway for accessing all this functionality is the `UIMenuController` class. `UIMenuController` is a singleton class, whose single instance is used across all text views in your app. The idea is that you create one or more `UIMenuItem` instances (each of which specifies an action and a title), put them in an array, and pass them off to the `UIMenuController` instance. When it's time to display the menu, `UIMenuController` will use the responder chain to locate an object that implements the method for each menu item, and determine whether or not that item should be displayed.

To demonstrate this in action, we'll create a quick little test-bed app. This app won't really do anything apart from letting us edit a piece of text, and attach a menu item of our own design to the menu controller. The menu item will let users select a URL in their text and open that URL in Safari.

Start by creating a new view-based project in Xcode, targeted at iPad. I named my app `TextMangler`. The project that Xcode creates will have a `TextManglerViewController` class. Add the following instance variable to *TextManglerViewController.h*, between the two curly braces in the class declaration:

```
IBOutlet UITextView *textView;
```

Now open *TextManglerViewController.xib* in Interface Builder, and drag a `UITextView` object into the view, filling it completely. Then control-drag from the File's Owner icon to the text view and select `textView` from the small context menu that appears. This GUI is done, so save your work and go back to Xcode.

TextManglerViewController.m has a number of predefined methods that were put there when the project was created. You can leave all of those in place, and just add definitions for the few methods described here. Start by defining the `viewDidLoad` method, where we do our initialization. This code creates a menu item, and then passes it along to `UIMenuController` so that it can appear alongside the other menu items.

```
- (void)viewDidLoad {
  [super viewDidLoad];
  UIMenuItem *menuItem = [[[UIMenuItem alloc] init] autorelease];
  menuItem.title = @"Open URL in Safari";
  menuItem.action = @selector(openUrlInSafari:);
  [UIMenuController sharedMenuController].menuItems = [NSArray
    arrayWithObject:menuItem];
}
```

Note that in defining a `UIMenuItem`, we specify an action, but not a target. The target is determined dynamically by traversing the responder chain, sending each object in the chain the `canPerformAction:` method until one of them returns `YES` (or until there's nothing left in the responder chain to ask). Since this controller class will be in the responder chain, we'll implement the method here.

The `canPerformAction:` method first checks to make sure that the relevant action is being asked about, and then checks the text view's selected text to see if it's an URL

www.it-ebooks.info

that can be opened. If so, it returns `YES`. If it doesn't know what else to do with this query, it passes the call along by calling the superclass's implementation.

```
- (BOOL)canPerformAction:(SEL)action withSender:(id)sender {
  if (action == @selector(openUrlInSafari:)) {
    NSString *selectedText = [textView.text substringWithRange:textView.selectedRange];
    NSURL *url = [NSURL URLWithString:selectedText];
    return [[UIApplication sharedApplication] canOpenURL:url];
  }
  return [super canPerformAction:action withSender:sender];
}
```

NOTE: You might think that you should explicitly return `YES` or `NO` here (depending on how you think about it) if you don't know what to do with the action in question, but you would be wrong. The `canPerformAction:` method is called for each and every menu item, so you would potentially be enabling (or disabling) all of them, not just yours!

Finally, here's the action method itself, which is called when the user selects the menu item we created.

```
- (void)openUrlInSafari:(id)sender {
  NSString *selectedText = [textView.text substringWithRange:textView.selectedRange];
  NSURL *url = [NSURL URLWithString:selectedText];
  [[UIApplication sharedApplication] openURL:url];
}
```

Build and run your app, and a big text view will fill the screen. Go ahead and play with it, pressing and holding somewhere, selecting some text, and so on. You'll see the same menu items as usual. Then type in a URL, such as `http://apress.com`, and select the text. Now you get the extra menu item, as shown in Figure 9–2. Touching it should launch Safari and bring up the page.

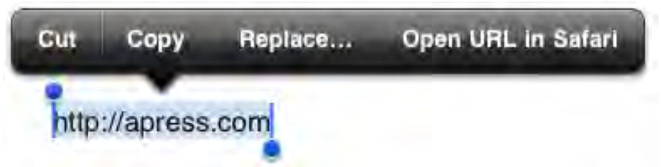


Figure 9–2. *Our new menu item has been added to the mix.*

Since `UIMenuItemController` is shared throughout your application, the menu items you add to it will be available in every `UITextField` and `UITextView` that the user sees. Since the menu items are enabled and disabled using the responder chain, you can decide the level of granularity you want. Use `canPerformAction:` to enable items in your app delegate if you want them to always be enabled, or in individual view controllers if you want more fine-grained control.

www.it-ebooks.info

Gesture Recognition

Starting with iOS 3.2, `UIViews` can handle not only individual touch events, but they can also look for particular kinds of touch actions and let your code know when they occur. Some of this isn't entirely new. `UIScrollView`, for instance, has always known how to watch for pinch and drag gestures, which it uses for controlling zoom levels and panning the view. What's new is that you can now tell any `UIView` to watch for specific gestures and let you know when they occur.

To make this work, you first create an instance of the new `UIGestureRecognizer` class, or rather, an instance of one of its many subclasses:

- `UILongPressGestureRecognizer`
- `UIPanGestureRecognizer`
- `UIPinchGestureRecognizer`
- `UIRotationGestureRecognizer`
- `UISwipeGestureRecognizer`
- `UITapGestureRecognizer`

Each of those is fine-tuned to detect a particular user gesture, clearly indicated in the class name. Most of them have at least one property that allows you to set some configuration options or read a value back.

After creating a gesture recognizer, you just pass it to a view using the `addGestureRecognizer:` method. Then the method you specified when creating the gesture recognizer will be called whenever the user performs that gesture. Let's put this into action using Dudel.

Adding Undo to Dudel

One key feature that Dudel is missing is any sort of undo action. Each stroke you make in a drawing is a part of your drawing forever. We're going to remedy that by assigning a gesture to open a small popover containing a single item that lets us remove the most recently created `Drawable` object in `dudelView`'s array. It's going to end up looking like Figure 9–3.

www.it-ebooks.info

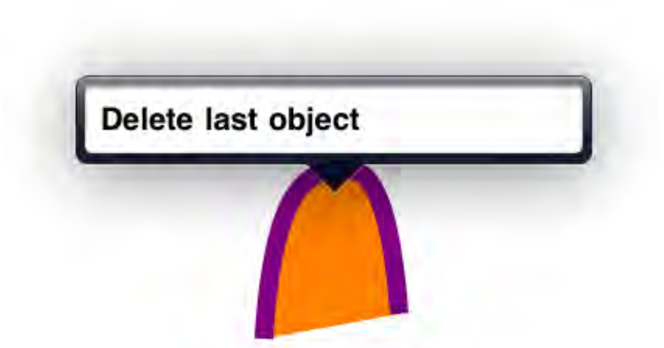


Figure 9–3. *A small menu that appears after a long touch. The arrow at the bottom edge of the popover points at the location of the touch, which in this case just happens to be the location of the last object.*

As you've done before, make a fresh copy of the Dudel project from the previous chapter, to contain your changes for this chapter. Then open the Xcode project in the new directory.

Let's start by making the view controller that will display a small pop-up menu in Dudel. Create a new `UIViewController` subclass, this time as a subclass of `UITableViewController`, without a matching *.xib* file, and name it `DudelEditController`. Here's the entire content of both the *.h* and *.m* files:

```
//  DudelEditController.h
#import <UIKit/UIKit.h>
#define DudelEditControllerDelete @"DudelEditControllerDelete"
@interface DudelEditController : UITableViewController {
  UIPopoverController *container;
}
@property (assign, nonatomic) UIPopoverController *container;
@end

//  DudelEditController.m
#import "DudelEditController.h"
@implementation DudelEditController
@synthesize container;
- (BOOL)shouldAutorotateToInterfaceOrientation:(UIInterfaceOrientation)orientation {
    // Override to allow orientations other than the default portrait orientation.
    return YES;
}
- (NSInteger)numberOfSectionsInTableView:(UITableView *)tableView {
    // Return the number of sections.
    return 1;
}
- (NSInteger)tableView:(UITableView *)tableView numberOfRowsInSection:(NSInteger)s {
    // Return the number of rows in the section.
    return 1;
}
- (UITableViewCell *)tableView:(UITableView *)tableView
    cellForRowAtIndexPath:(NSIndexPath *)indexPath {
  static NSString *CellIdentifier = @"Cell";
  UITableViewCell *cell = [tableView dequeueReusableCellWithIdentifier:CellIdentifier];
  if (cell == nil) {
```

www.it-ebooks.info

```
    cell = [[[UITableViewCell alloc] initWithStyle:UITableViewCellStyleDefault
           reuseIdentifier:CellIdentifier] autorelease];
  }
  cell.textLabel.text = @"Delete last object";
  return cell;
}

- (void)tableView:(UITableView *)tableView didSelectRowAtIndexPath:(NSIndexPath *)ip {
  [[NSNotificationCenter defaultCenter] postNotificationName:DudelEditControllerDelete
    object:self];
}
@end
```

Now open *DudelViewController.m* and add this line near the top of the file:

```
#import "DudelEditController.h"
```

Then add these lines to the end of `viewDidLoad`:

```
  UILongPressGestureRecognizer *longPress =
    [[[UILongPressGestureRecognizer alloc] initWithTarget:self
    action:@selector(handleLongPress:)] autorelease];
  [dudelView addGestureRecognizer:longPress];
```

Next, implement the `handleLongPress:` method referenced earlier.

```
- (void)handleLongPress:(UIGestureRecognizer *)gr {
  if (gr.state == UIGestureRecognizerStateBegan) {
    DudelEditController *c = [[[DudelEditController alloc]
      initWithStyle:UITableViewStylePlain] autorelease];
    [self setupNewPopoverControllerForViewController:c];
    self.currentPopover.popoverContentSize = CGSizeMake(320, 44*1);
    c.container = self.currentPopover;
    [[NSNotificationCenter defaultCenter] addObserver:self
      selector:@selector(dudelEditControllerSelectedDelete:)
      name:DudelEditControllerDelete object:c];
    CGRect popoverRect = CGRectZero;
    popoverRect.origin = [gr locationInView:dudelView];
    [self.currentPopover presentPopoverFromRect:popoverRect inView:dudelView
      permittedArrowDirections:UIPopoverArrowDirectionAny animated:YES];
  }
}
```

Now it's time to implement the method that is called when the menu item is actually selected.

```
- (void)dudelEditControllerSelectedDelete:(NSNotification *)n {
  DudelEditController *c = [n object];
  UIPopoverController *popoverController = c.container;
  [popoverController dismissPopoverAnimated:YES];
  [self handleDismissedPopoverController:popoverController];
  self.currentPopover = nil;
  if ([dudelView.drawables count] > 0) {
    [dudelView.drawables removeLastObject];
    [dudelView setNeedsDisplay];
  }
}
```

www.it-ebooks.info

Build and run your app, do some doodling, and then press and hold anywhere on the screen until the popover appears. Select its one item, and watch as the last shape or stroke you made suddenly disappears! This is great, but now try to draw a couple more shapes using whatever tool you already had selected. You'll see that things get a little screwy. The screen doesn't seem to update properly while you drag, and the first shape you draw will disappear when you start drawing the next one. This is all due to the gesture activity leaving the chosen tool in an inconsistent state, which is easily remedied.

Resetting the Selected Tool's State

We need to make sure that when active touches are canceled (which happens when the gesture recognizer decides that a gesture is happening), the selected tool's state is reset so that it doesn't think it's still in the middle of tracking a drag.

Open the *.m* file for each of the tool classes (except `FreehandTool`), and add the following line to the `touchesCancelled:withEvent:` method:

```
- (void)touchesCancelled:(NSSet *)touches withEvent:(UIEvent *)event {
  // If you already had any other code here, leave it alone and add this:
  [self deactivate];
}
```

The `FreehandTool` class is the exception. It already implements this method. Here, just add a single line:

```
- (void)touchesCancelled:(NSSet *)touches withEvent:(UIEvent *)event {
  [self activate];
  isDragging = NO;
}
```

This one is a little different from the others. It calls the `activate` method instead of `deactivate`. That's due to a peculiarity in the creation of this class, where `deactivate` finishes the current drawing action—just what we want to avoid!

Now you should be able to build and run the app, and see everything behaving as it should.

We've used only `UILongPressGestureRecognizer` for this example, but the other gesture recognizers work similarly.

Keyboard Extensions and Replacements

Did you ever notice that when you're filling in a web-based form in Mobile Safari, a small row of additional buttons appears above the keyboard, containing buttons labeled Previous, Next, and so on? iOS 3.2 gives us a way to do that sort of thing, too, using the new `inputAccessoryView` property of `UITextView` and `UITextField`. You just put anything you like into a `UIView` (buttons, labels, sliders … you name it), and pass that view off to the text-input object in question.

www.it-ebooks.info

As if that weren't enough, starting with OS 3.2 we can now replace the entire keyboard! The idea is similar to that for the accessory view. `UIView` has a new `inputView` property that you can set, using any `UIView` you like.

Adding a Keyboard Button in Dudel

To demonstrate how to extend the keyboard, we're going to add something that has been missing from the `FileRenameViewController` in Dudel: the ability to cancel the renaming operation. We'll do this by adding a Cancel button to the keyboard.

Open *FileRenameViewController.m*, and add the following method to set up the input accessory view:

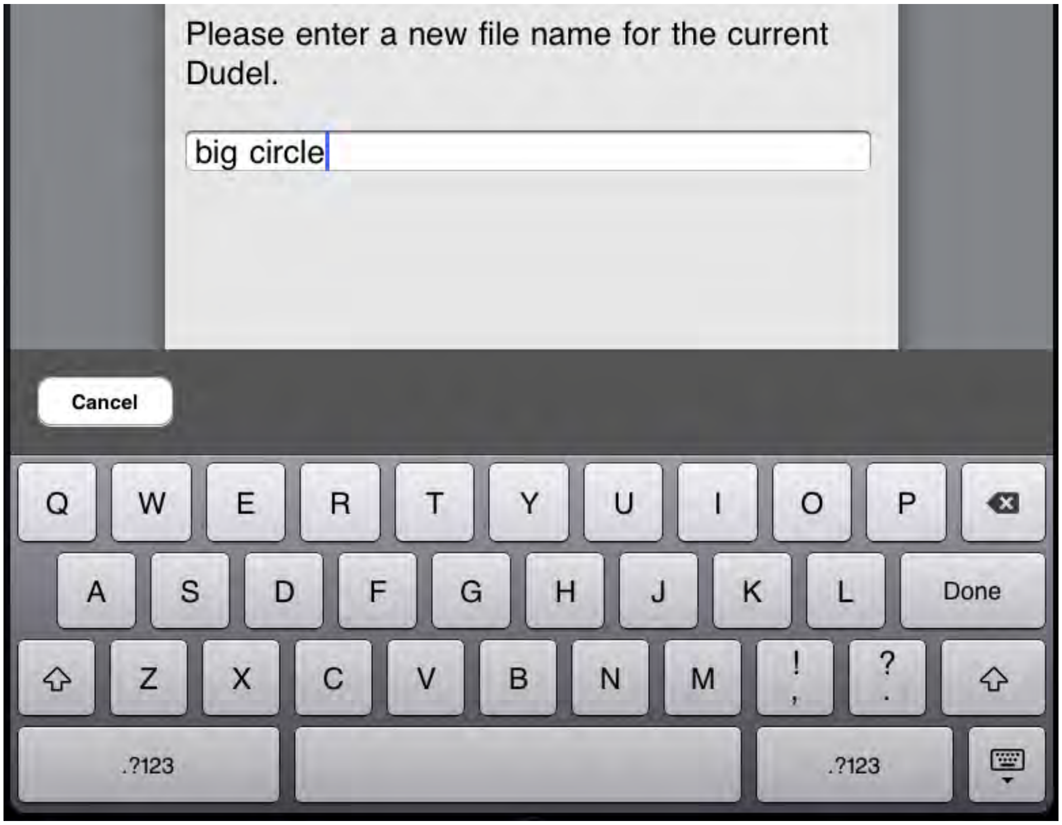
```
- (void)viewDidLoad {
  [super viewDidLoad];
  UIView *inputAccessoryView = [[UIView alloc] initWithFrame:
    CGRectMake(0.0, 0.0, 768.0, 77.0)];
  inputAccessoryView.backgroundColor = [UIColor darkGrayColor];
  UIButton *cancelButton = [UIButton buttonWithType:
    UIButtonTypeRoundedRect];
  cancelButton.frame = CGRectMake(20.0, 20.0, 100.0, 37.0);
  [cancelButton setTitle: @"Cancel" forState:UIControlStateNormal];
  [cancelButton setTitleColor:[UIColor blackColor] forState:
    UIControlStateNormal];
  [cancelButton addTarget:self action:@selector(cancel:)
    forControlEvents:UIControlEventTouchUpInside];
  [inputAccessoryView addSubview:cancelButton];
  textField.inputAccessoryView = inputAccessoryView;
}
```

Also add the method that actually does the canceling:

```
- (void)cancel:(id)sender {
  [delegate fileRenameViewController:self didRename:originalFilename
    to:originalFilename];
}
```

Now build and run your app, bring up the file renaming view, and you'll see something like Figure 9-4.

www.it-ebooks.info


Please enter a new file name for the current Dudel.

big circle

Figure 9–4. *Here's the Cancel button we've just added, making this keyboard view more useful.*

Replacing the Keyboard

In this section, we're going to implement a simple calculator that uses a normal `UITextView` object as an input field. Instead of letting the users enter any sort of text they want, we'll present an input view that contains buttons for numbers, as well as buttons for the calculator functions.

Our calculator will use Reverse Polish Notation (RPN). With RPN, the mathematical operators are shown after the numbers on which they should operate. For instance, 1 + 3 would be written as 1 3 + in RPN. In a longer sequence, the result of an operation can be used as input to the next operator. For example, 10 / 2 + 3 would be written as 10 2 / 3 + in RPN.

One consequence of this notation is that it eliminates the need for parentheses in expressions. RPN expressions are always evaluated left to right. To change the order of operations, you just need to shift the operators around. For example, (3 * 4) + 10 becomes 3 4 * 10 + in RPM, and 3 * (4 + 10) becomes 3 4 10 + *. When entering expressions like this in an RPN calculator, normally you press some sort of Enter key between entering numbers, as in 3 [Enter] 4 [Enter] 10 [+] [*].

www.it-ebooks.info

Another consequence of using RPN is that creating a calculator app becomes really simple! At the core of the implementation lies a stack (in our case, an `NSMutableArray` will do nicely) onto which each number is pushed. Each mathematical operator uses the top item of the stack, along with the number currently in the text view, to perform an operation and leave the result in the text view. We don't need to worry about parsing parentheses or keeping track of pending operations that are waiting for a higher-precedence operation to take place first.

As a bonus, our app will make use of the iPad's screen real estate to show more than just the single number being entered. We'll show the entire stack of all numbers that have been entered and are waiting to be acted upon, as shown in Figure 9–5.

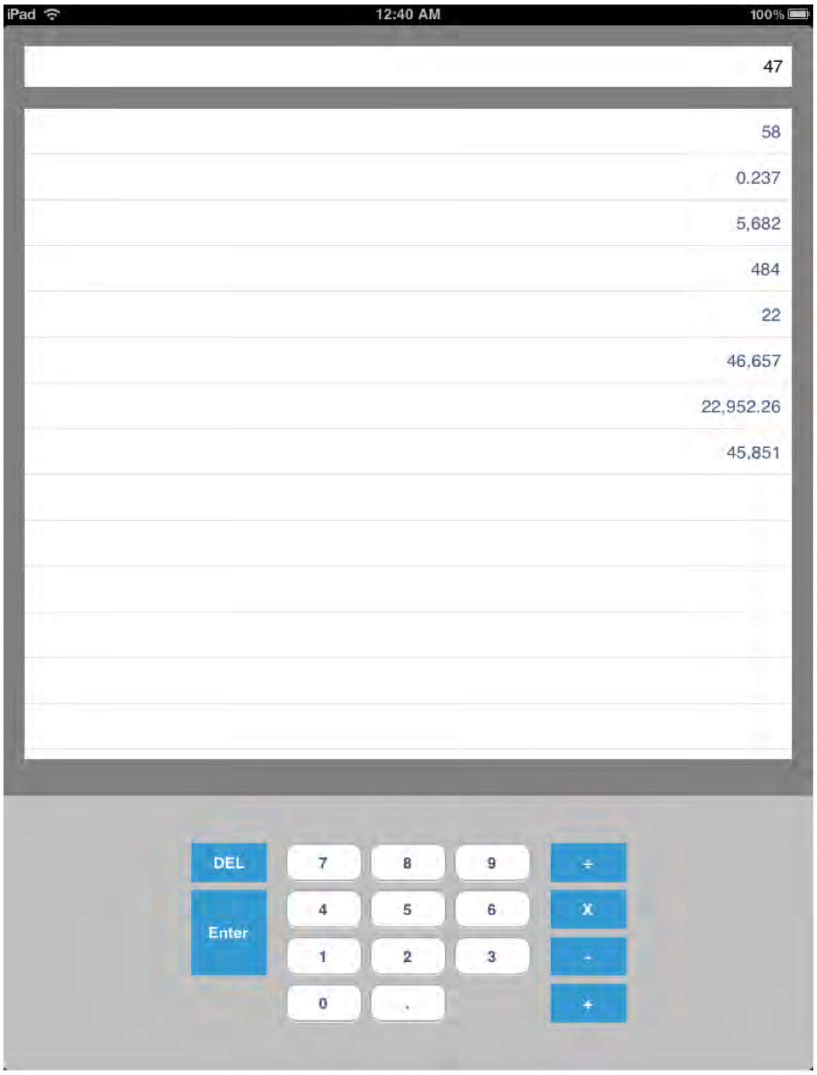


Figure 9–5. *An RPN calculator worthy of a strange name. The text area at the top is the editing area. Below that is the stack of entered numbers.*

www.it-ebooks.info

In honor of this calculator's reverse Polish heritage, we're going to name it ClacHsilop. (Read it backward. If I have to explain it, that means it's not funny!)

Open Xcode, make a new view-based iPad project, and name it `ClacHsilop`. This class will have a single view controller, which will manage the text view and the table view in the display. For the text view, rather than just setting a property to specify the `inputView`, we're going to subclass `UITextView` and override the `inputView` method, returning a pointer to a view of our own.

Defining the InputView Class

Let's start by creating the new view. We'll subclass `UITextView`, and use Interface Builder to define the content for our `inputView`, laying out buttons the way we want, and connecting them to action methods in our `UITextView` subclass. Our text view class will also define a delegate protocol for passing along calculator command actions (+, –, and so on) to its delegate.

Use the New File Assistant to create a new Objective-C class, a subview of `UIView` (since `UITextView` isn't one of the choices), and name it `InputView`. The `InputView` class will a have a method that allows buttons in the `inputView` GUI to enter text directly (in our case, strings containing numeric digits), as well as a method that will let a button trigger a calculator action based on the sender's tag. Here's the complete source of the `InputView` class:

```
//  InputView.h
#import <UIKit/UIKit.h>
typedef enum ActionTag {
  ActionEnter = 0,
  ActionDivide,
  ActionMultiply,
  ActionSubtract,
  ActionAdd
} ActionTag;
@protocol InputViewDelegate;
@interface InputView : UITextView {
  UIView *inputView;
  id <InputViewDelegate> ivDelegate;
}
- (IBAction)takeInputFromTitle:(id)sender;
- (IBAction)doDelete:(id)sender;
- (IBAction)doTaggedAction:(id)sender;
@end
@protocol InputViewDelegate
- (void)doTaggedAction:(ActionTag)tag forInputView:(InputView *)iv;
@end

//  InputView.m
#import "InputView.h"
@implementation InputView
- (void)dealloc {
  [inputView release];
  [super dealloc];
}
```

www.it-ebooks.info

```
- (UIView *)inputView {
  if (!inputView) {
    NSArray *objects = [[NSBundle mainBundle] loadNibNamed:@"RpnKeyboard" owner:self
      options:nil];
    inputView = [[objects objectAtIndex:0] retain];
  }
  return inputView;
}
- (IBAction)takeInputFromTitle:(id)sender {
  // remove the initial zero;
  if ([self.text isEqual:@"0"]) {
    self.text = @"";
  }
  self.text = [self.text stringByReplacingCharactersInRange:self.selectedRange
    withString:((UIButton *)sender).currentTitle];
}
- (IBAction)doDelete:(id)sender {
  NSRange r = self.selectedRange;
  if (r.length > 0) {
    // the user has highlighted some text, fall through to delete it
  } else {
    // there's just an insertion point
    if (r.location == 0) {
      // cursor is at the beginning, forget about it.
      return;
    } else {
      r.location -= 1;
      r.length = 1;
    }
  }
  self.text = [self.text stringByReplacingCharactersInRange:r withString:@""];
  r.length = 0;
  self.selectedRange = r;
}
- (IBAction)doTaggedAction:(id)sender {
  ActionTag tag = [sender tag];
  [ivDelegate doTaggedAction:tag forInputView:self];
}
@end
```

There are just a couple tricky parts here. The first is in the `takeInputFromTitle:` action, which is the one that all our numeric digit buttons will call. Like most handheld calculators, ours will display a 0 (zero) instead of an empty display when its value is zero. The small check for a 0 in that method makes that 0 go away when the user starts typing.

The other fussy bit is the `doDelete:` action, which will be called by the delete/backspace key on the keyboard. Since the user can always highlight a section of the number by touching it, as well as put the insertion cursor at the beginning of the number, we need to consider a few things there before deleting any text.

www.it-ebooks.info

Creating the Keyboard Input View

Now let's create the GUI. Use the New File Assistant once again to make a new view *.xib* resource, naming it `RpnKeyboard`. Open the *RpnKeyboard.xib* file in Interface Builder, select the File's Owner proxy icon, and use the identity inspector to set its class to `InputView`.

The *.xib* should already contain a `UIView`, which you should now open. If you created the *.xib* file as an iPad resource (instead of an iPhone resource), it may be preconfigured for full-screen usage. In that case, use the attribute inspector to disable its status bar.

Now resize the view so that it can accommodate the buttons we need. The view will be resized to the correct iPad keyboard space before being displayed, so the exact size isn't too important—anything around 500 by 250 pixels should be just fine.

Use the attribute inspector to set the view's background color to light gray by clicking the color well for the background color, choosing the grayscale slider in the color picker that appears, and selecting 75%. This will help give your keyboard an appearance that's similar to the normal keyboard.

Now drag in a basic Round Rect Button from the Library, and then control-drag from it to the File's Owner icon, connecting it to the `takeInputFromTitle:` action. Then duplicate the button ten times with ⌘D, and arrange the buttons as shown in Figure 9–6, which also shows the titles you should set on each button. Let Interface Builder help you define spacing between these buttons. There should be a natural spot at which the buttons snap into place, 8 pixels apart from each other.

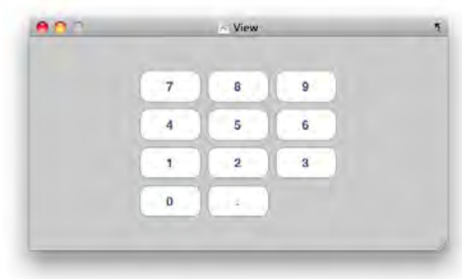


Figure 9–6. *The basic calculator-style numeric keypad*

Next, drag in another Round Rect Button from the Library, placing it to the right of the 9 button, and connect this one to the `doTaggedAction:` method in File's Owner. To make the function buttons stand out a bit from the numeric input buttons, use the attribute inspector to change this button's Type from Rounded Rect to Custom, and set its background color to something you like (I chose a slightly greenish blue). Duplicate this button three times, and put the buttons in a column to the right of the others, as shown in Figure 9–7. This figure also shows which mathematical symbol to use as the title for each button.

www.it-ebooks.info

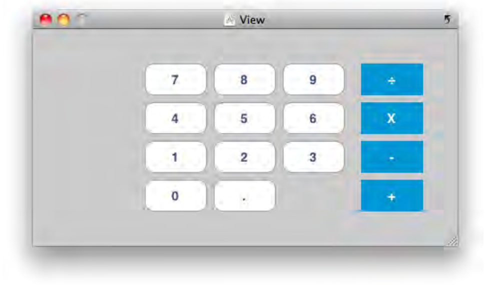


Figure 9–7. *Mathematical symbols*

Remember that the action method these buttons trigger looks at the sender's tag to see what it's supposed to do, so we're going to give each button a tag matching its function. Unfortunately, we can't use the tag names we defined as an enumerated type in `InputView's` header file (wouldn't that be sweet?). Instead, we need to use the corresponding integers: 1 for division, 2 for multiplication, 3 for subtraction, and 4 for addition. Open the attribute inspector, and then select one button at a time, setting each tag value in turn.

The final set of buttons will be the ones marked DEL and Enter on the left side. Select one of the buttons you just made on the right, duplicate it twice, and drag them both over to the left. Lay them out as shown in Figure 9–8, making the Enter button as tall as two normal buttons.

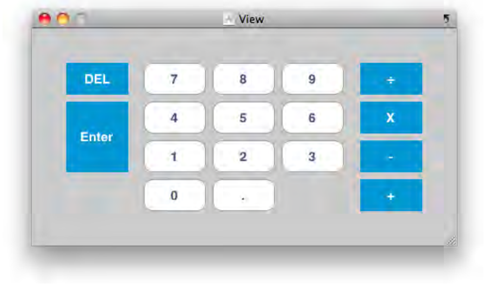


Figure 9–8. *The DEL and Enter buttons*

Change the Enter button's tag to `0` so that it activates the correct bit of functionality. Next, retarget the DEL button by control-dragging from the button to the File's Owner icon and selecting the `doDelete:` action. At this point, the DEL button will actually be misconfigured. A button click can trigger multiple action method calls, and setting a new connection doesn't delete any of the old ones. To fix this, open the connections inspector, where you can see the multiple actions that are configured for the DEL button. Delete the connection to `doTaggedAction:` so we don't have a DEL button doing crazy things.

I mentioned earlier that the view we create here will be resized automatically to the correct iPad keyboard size. Let's take control of how the resizing affects our buttons,

www.it-ebooks.info

making sure that the entire button group will remain a constant size and centered in the overall view.

Use the mouse to drag a rectangle across all 17 buttons, so that they're all highlighted, and then select **Layout ➤ Embed Objects In ➤ View** from the menu. With the new view selected, open the size inspector. To make sure this enclosing view remains centered, click to turn off all the red arrows and bars in the Autosizing section.

The special input view is now complete! Save your work, and then switch back to Xcode.

Creating the Calculator

We're now going to define our app's view controller, which does the actual work of being a calculator.

Open the *ClacHsilopViewController.h* file, and add the instance variables shown in the following listing. We also declare it to be a delegate of the InputView class.

```
//  ClacHsilopViewController.h
#import <UIKit/UIKit.h>
#import "InputView.h"
@interface ClacHsilopViewController : UIViewController <InputViewDelegate> {
  IBOutlet InputView *inputView;
  IBOutlet UITableView *stackTableView;
  NSNumberFormatter *decimalFormatter;
  NSMutableArray *stack;
}
@end
```

Before implementing that class, let's set up the GUI. Open *ClacHsilopViewController.xib* in Interface Builder. Select the main view and set its background color to a darker gray, just to make our other components stand out. Then use the Library to find an InputView, and drag it to the top of the view. Our table will eventually display the stack in a right-justified column, so right-justify the InputView as well, using the attribute inspector.

Next, drag out a UITableView, making it fill most of the view, as shown in Figure 9–9. The table doesn't need to extend to the bottom of the view, since the customized inputView will be appearing on top of it.

Connect the table view's dataSource and delegate outlets to the File's Owner proxy icon, as well as the InputView's ivDelegate outlet. Then connect the inputView and stackTableView outlets to the appropriate objects. Save your work now, and switch back to Xcode.

www.it-ebooks.info

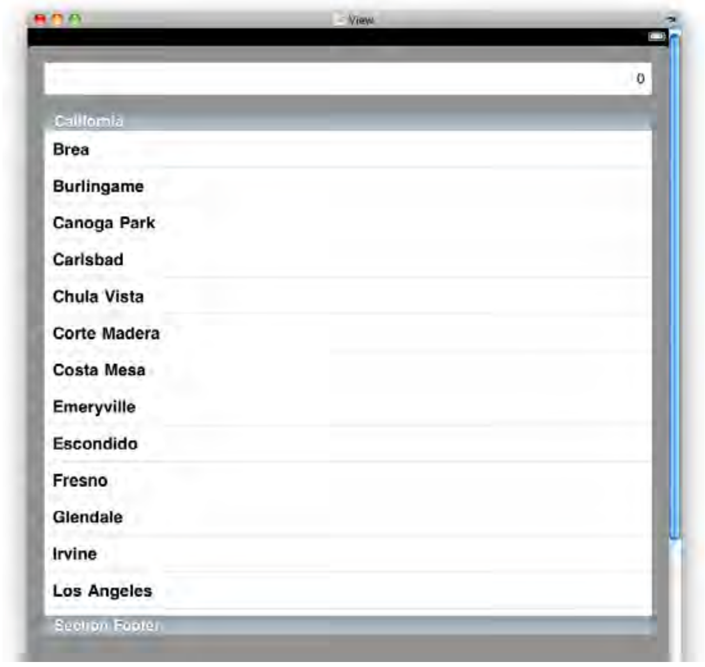


Figure 9–9. *Text and a table*

Open *ClacHsilopViewController.m*. This class will have two primary functions: presenting the contents of the stack in a table view and handling the actual calculator functionality in response to the user working the controls.

```
//  ClacHsilopViewController.m
#import "ClacHsilopViewController.h"
@implementation ClacHsilopViewController
- (void)viewDidLoad {
  [super viewDidLoad];
  stack = [[NSMutableArray alloc] init];
  decimalFormatter = [[NSNumberFormatter alloc] init];
  decimalFormatter.numberStyle = NSNumberFormatterDecimalStyle;
  [stackTableView reloadData];
  [inputView becomeFirstResponder];
}
- (BOOL)shouldAutorotateToInterfaceOrientation:(UIInterfaceOrientation)o {
  return YES;
}
- (void)dealloc {
  [stack release];
  [decimalFormatter release];
  [super dealloc];
}
- (NSInteger)numberOfSectionsInTableView:(UITableView *)tableView {
  return 1;
}
- (NSInteger)tableView:(UITableView *)tableView
  numberOfRowsInSection:(NSInteger)s {
```

www.it-ebooks.info

```
  return [stack count];
}
- (UITableViewCell *)tableView:(UITableView *)tableView
    cellForRowAtIndexPath:(NSIndexPath *)indexPath {
  static NSString *CellIdentifier = @"Cell";
  UITableViewCell *cell = [tableView dequeueReusableCellWithIdentifier:
    CellIdentifier];
  if (cell == nil) {
    cell = [[[UITableViewCell alloc] initWithStyle:
      UITableViewCellStyleValue1
    reuseIdentifier:CellIdentifier] autorelease];
  }
  cell.detailTextLabel.text = [decimalFormatter stringFromNumber:[stack
    objectAtIndex:indexPath.row]];

  return cell;
}
- (void)handleError {
  // in case of an error, push the current number
  // onto the stack instead of just tossing it
  NSDecimalNumber *inputNumber = [NSDecimalNumber
    decimalNumberWithString:inputView.text];
  [stack insertObject:inputNumber atIndex:0];
  inputView.text = @"Error";
}
- (void)doEnter {
  NSDecimalNumber *inputNumber = [NSDecimalNumber
    decimalNumberWithString:inputView.text];
  [stack insertObject:inputNumber atIndex:0];
  [stackTableView reloadData];
  inputView.text = @"0";
}
- (void)doDecimalArithmetic:(SEL)method {
  if ([stack count] > 0) {
    NSDecimalNumber *inputNumber = [NSDecimalNumber
      decimalNumberWithString:inputView.text];
    NSDecimalNumber *stackNumber = [stack objectAtIndex:0];
    NSDecimalNumber *result = [stackNumber performSelector:method
      withObject:inputNumber];
    inputView.text = [decimalFormatter stringFromNumber:result];
    [stack removeObjectAtIndex:0];
  } else {
    [self handleError];
  }
  [stackTableView reloadData];
}
- (void)doTaggedAction:(ActionTag)tag forInputView:(InputView *)iv {
  switch (tag) {
    case ActionEnter:
      [self doEnter];
      break;
    case ActionDivide:
      [self doDecimalArithmetic:@selector(decimalNumberByDividingBy:)];
      break;
    case ActionMultiply:
      [self doDecimalArithmetic:
        @selector(decimalNumberByMultiplyingBy:)];
```

www.it-ebooks.info

```
      break;
    case ActionSubtract:
      [self doDecimalArithmetic:@selector(decimalNumberBySubtracting:)];
      break;
    case ActionAdd:
      [self doDecimalArithmetic:@selector(decimalNumberByAdding:)];
      break;
    default:
      break;
  }
}
@end
```

That's it! With this code in place, you should now be able to build and run the app, see the GUI appear, and immediately have the RPN input keypad at your disposal.

That's All the Input You Need

In this chapter, you've learned about the great new input features included in iOS 3.2. While things like gesture recognition have been theoretically possible all along by tracking events, including an API for recognizing them will help developers add interactivity that they might have otherwise skipped. Adding items to the text-editing menu feels like a surprise, considering that the menu itself has been around for only about a year, but this could be a useful addition for some kinds of applications. Enhancing the keyboard has been a sore point for many iPhone developers for years, so the ability to extend or even completely redefine the keyboard as we wish is a welcome change indeed!

This concludes our coverage of the new GUI features in iOS 3.2. In Chapter 10, you'll learn about techniques for letting your app "play nice" with other apps by passing files and other kinds of data back and forth between them.

www.it-ebooks.info

Chapter 10

Working with Documents

For anyone with a background in programming desktop apps, the iPhone presents some unique challenges. Lacking anything like the Mac OS Finder or the Windows Explorer, the system does not provide a general-purpose technique for displaying a file or a collection of files. There has been no way to deal with files as discrete chunks of interchangeable data, and no way to determine if the device has any other installed apps that could have a use for your app's data. Furthermore, moving documents back and forth between the iPhone and the desktop has not been easy. Many developers have included a built-in web server in their apps for the sole purpose of letting a web browser on the desktop connect to an iPhone on the same Wi-Fi network and exchange files with it.

Starting with iOS 3.2 for the iPad, the support for working with files has improved greatly. Although there's still nothing like the Finder, apps can now declare their ability to open particular types of files. Additionally, each app that deals with a file of any kind can ask the system to display a list of other apps that can open that file, and pass the file directly to the app that the user chooses. Apple has also added basic document synchronization support, letting apps declare that particular documents should be shared with the desktop computer, where they'll show up in iTunes.

In this chapter, you'll learn how to use these new features so that your apps can play well with others. Once again, we'll be working with Dudel. You'll learn how to pass files to other apps, how to register a file type that an app can open, and how to deal with files that are passed in from other apps. You'll also learn how to use the new synchronization capabilities of iOS 3.2 (combined with iTunes) to copy files from your iPad to your computer, as well as the other way around.

Passing Files to Another Application

The new document-interaction features found in iOS 3.2 let apps work together in a whole new way, somewhat compensating for the operating system's lack of multitasking, or its inability to let you drag files or other objects from one app to another. With the new document-interaction facility, you can take the output of one app and pass it to another app, where the user might do some additional work on it, and then pass it along to yet another app.

www.it-ebooks.info

Each iPhone app can register itself, via its *Info.plist* file, as being able to open particular types of files. The operating system itself keeps track of which apps are registered for which file types. All you need to do in order to pass a file to another app is call a single method that figures out the type of the file, determines which apps can open it, presents the user with a list of valid apps in a popover, and lets the user choose one. If the user chooses an app from the list, your app will exit, and the operating system will start the other app, passing along the file.

We'll implement this in Dudel by creating a PDF file and letting the user pass it off to another app. In order for this work, you'll first need to install an app from the App Store that will accept PDF files. I'm using GoodReader, which is an inexpensive, full-featured app for dealing with PDF and other image formats. But feel free to use any other PDF viewer you prefer. As long as it can open PDF files, it should work for our purposes.

NOTE: All the examples used in this chapter involve communicating with other apps. Since the iPad Simulator lets you use only a small subset of the iPad's included apps (plus any apps you build and install yourself), you'll need to use an actual iPad connected to your computer to test the code in this chapter.

Adding a PDF-Sending Menu Action

Let's start by adding an item to the ActionsMenuController so that we have a way to activate the PDF-sending method. Add a line to the enumerated types declared in *ActionsMenuController.h*:

```
typedef enum SelectedActionType {
  NoAction = -1,
  NewDocument,
  RenameDocument,
  DeleteDocument,
  EmailPdf,
  OpenPdfElsewhere,
  ShowAppInfo
} SelectedActionType;
```

Then extend the implementation in *ActionsMenuController.m* to include one more row:

```
- (NSInteger)tableView:(UITableView *)tableView numberOfRowsInSection:
  (NSInteger)section {
    // Return the number of rows in the section.
    ~~return 5;~~
    return 6;
}
```

In the same file, make sure the tableView:cellForRowAtIndexPath: method provides a value for the new row as well. Add the following inside its large switch construct:

```
    case OpenPdfElsewhere:
      cell.textLabel.text = @"Open PDF in another app";
      break;
```

www.it-ebooks.info

Now it's time to switch our attention to the DudelViewController, starting with a few changes to the interface in *DudelViewController.h*. First, add yet another protocol to the growing list of protocols that this class implements:

```
@interface DudelViewController : UIViewController <ToolDelegate, DudelViewDelegate,
MFMailComposeViewControllerDelegate, UIPopoverControllerDelegate,
ModalWebViewControllerDelegate, FileRenameViewControllerDelegate,
UIDocumentInteractionControllerDelegate> {
```

We're also going to need to have access to the menu bar item that brings up the action menu, so add the following instance variable:

```
  IBOutlet UIBarButtonItem *actionsMenuButton;
```

Save your changes, and open *DudelViewController.xib* in Interface Builder. Open the view it contains, scroll to the bottom (if it's not already visible), and then control-drag from File's Owner to the relevant menu bar item and connect the actionsMenuButton outlet. Then save your changes.

Preparing a File for Sending

Switch back to Xcode and open *DudelViewController.m*, where we'll implement the rest of this functionality. Start off in handleDismissedPopoverController, adding this chunk to its switch construct:

```
      case OpenPdfElsewhere:
        [self openPdfElsewhere];
        break;
```

Before we implement the openPdfElsewhere method, we need to refactor some work we did earlier. Back in Chapter 4, we added the sendPdfEmail method, which creates a PDF drawing context that writes into an NSMutableData object, does the drawing, and then uses the resulting data object as an attachment in an outbound e-mail message. In order to send a PDF representation of our document to another app, we need to do some of the same things. Rather than duplicating that functionality, let's break up the old method into two: one that generates the PDF data and another that sends the e-mail message. That way, our new method will be able to reuse the data-creation method, but then do its own thing with the data that's produced.

```
// Remove sendPdfEmail, and replace it with these two methods:
- (NSData *)pdfDataForCurrentDocument {
  // set up PDF rendering context
  NSMutableData *pdfData = [NSMutableData data];
  UIGraphicsBeginPDFContextToData(pdfData, dudelView.bounds, nil);
  UIGraphicsBeginPDFPage();

  // tell our view to draw
  [dudelView drawRect:dudelView.bounds];

  // remove PDF rendering context
  UIGraphicsEndPDFContext();
```

www.it-ebooks.info

```
  return pdfData;
}
- (void)sendPdfEmail {
  NSData *pdfData = [self pdfDataForCurrentDocument];
  // send PDF data in mail message
  MFMailComposeViewController *mailComposer = [[[MFMailComposeViewController alloc]
init] autorelease];
  mailComposer.mailComposeDelegate = self;
  [mailComposer addAttachmentData:pdfData mimeType:@"application/pdf" fileName:@"Dudel
creation.pdf"];
  [self presentModalViewController:mailComposer animated:YES];
}
```

Invoking the Document Interaction Controller

Now we get to the interesting part. The `openPdfElsewhere` method takes the generated PDF data, saves it to a temporary file, and passes it off to a newly created `UIDocumentInteractionController`. This is a special-purpose controller that knows how to see which installed apps can open a given file, based on their declared file-opening abilities.

```
- (void)openPdfElsewhere {
  NSData *pdfData = [self pdfDataForCurrentDocument];
  NSString *filePath = [NSTemporaryDirectory() stringByAppendingPathComponent:@"Dudel
creation.pdf"];
  NSURL *fileURL = [NSURL fileURLWithPath:filePath];
  NSError *writeError = nil;
  [pdfData writeToURL:fileURL options:0 error:&writeError];
  if (writeError) {
    NSLog(@"Error writing file '%@' :\n%@", filePath, writeError);
    return;
  }
  UIDocumentInteractionController *docController =
  [UIDocumentInteractionController interactionControllerWithURL:fileURL];
  docController.delegate = self;
  BOOL result = [docController presentOpenInMenuFromBarButtonItem:actionsMenuButton
animated:YES];
}
```

Here, we call the `presentOpenInMenuFromBarButtonItem` method, which will bring up a popover containing the names of all installed apps that can handle that file, as shown in Figure 10–1. If no apps can handle the given file, you won't see this popover. But if one or more apps will accept that file, the popover appears and waits for you to select an app. In either case, this popover is taken care of by the `UIDocumentInteractionController` itself, so we will never need to dismiss it or otherwise deal with it.

www.it-ebooks.info

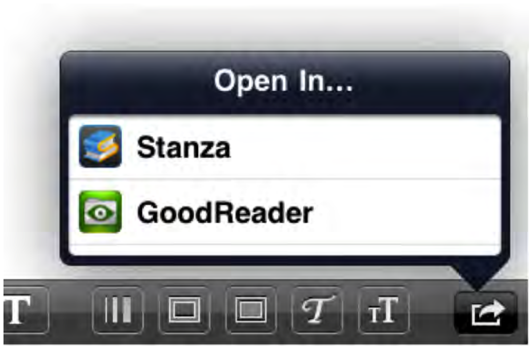


Figure 10–1. *If any of the installed apps can open the file, a popover appears with a list.*

So, what happens when you select an app from the list? Basically, the current app will exit, and the chosen app will start up with some parameters telling it which file to open. Before the current app quits, the `UIDocumentInteractionController` will call some methods in its delegate (if the methods are implemented) that give you a chance to do some things. But at that stage, your app should really be prepared to just let go, so the chosen app can open as quickly and seamlessly as possible.

So that's all you need to do in order to send a file to another app. Most of the code we've added so far in this chapter has just been to facilitate a new item in our action menu. Passing a file to another app is basically just a matter of creating an instance of `UIDocumentInteractionController` and giving it the URL to your file. The rest is handled for you. So how do you go about receiving a file?

Receiving Files

As it turns out, receiving a file can be a bit trickier than sending one. The first step is to register your application as a suitable viewer/editor for the file type you want to have passed your way. This registration shouldn't be confused with anything like stuffing values into the Windows registry. In fact, this type of registration is completely passive. All you need to do is specify in your app's *Info.plist* (which is called *Dudel-Info.plist* in our case) which types of files it should open. The operating system itself will examine your declared type compatibilities and use that information to figure out where other apps may be able to send their data. The second step will be to implement a `UIApplication` delegate method that will let your app notice the URL of a file being sent its way at launch time and do something with it.

Registering As a Recipient

iOS, like its predecessor, Mac OS X, has a rather complex system of determining the type of data that's represented by a file or a data stream. Depending on whether the data is in a file, coming from a web server or a mail message, or being accessed in some other way, the operating system might use a filename extension, MIME type

www.it-ebooks.info

declaration, or a UTI to determine just what this hunk of data is and figure out which application should deal with it.

Everyone knows about filename extensions, and you probably have seen MIME type declarations in one place or another. But you may not be familiar with the UTI concept. The idea of UTI, introduced by Apple a few years ago, is to use a reverse domain name scheme (like the system used for naming Java packages or identifiers for iPhone apps) to identify data types.

Apple defines UITs for a large number of common data types, such as text and images, which all begin with the prefix `public` (such as `public.text`, `public.image`, and `public.png`). You should use `public` UTIs for describing your data whenever possible.

When you have your own data type, such as the format in which Dudel saves its document files, you should make up your own UTI. You can use some combination of the name of your company, the name of the primary application that uses the type, and the type name itself. In the case of our Dudel document files, we'll create a `com.rebisoft.dudeldoc` UTI. (Rebisoft is the name of the company through which Dudel is published on the App Store.)

Declaring a Data Type's Existence with UTI

To begin the registration, open *Dudel-Info.plist* in your Xcode project. By default, you'll be editing this file using a graphical editor that gives you an outline view of the property list document. For the changes we need to make, you're better off editing the XML directly. Right-click anywhere in the document view and select **Open As ➤ Plain Text File** from the context menu, forcing the editor to redisplay the property list's context as plain XML.

You'll see that the XML contains a `<plist>` tag, followed by a `<dict>` tag. To declare the existence of our new document type, insert the following chunk of XML immediately after the `<dict>` line, so that it ends up inside that element.

```
<key>UTExportedTypeDeclarations</key>
<array>
        <dict>
                <key>UTTypeConformsTo</key>
                <array>
                        <string>public.data</string>
                </array>
                <key>UTTypeDescription</key>
                <string>Dudel Document File</string>
                <key>UTTypeIdentifier</key>
                <string>com.rebisoft.dudeldoc</string>
                <key>UTTypeTagSpecification</key>
                <dict>
                        <key>public.filename-extension</key>
                        <string>dudeldoc</string>
                </dict>
        </dict>
</array>
```

www.it-ebooks.info

Here, we first create a one-element array of dictionaries associated with the `UTExportedTypeDeclarations` key, which is used by the operating system to see which special data types our app knows about. We can declare one or more other UTIs to which our new type conforms.

The idea is that these types, like Objective-C classes, exist in a hierarchy. In this case, it's a multiple-inheritance hierarchy, since a type can have multiple parents. These parent types can be used by the operating system to determine which context a resource can be used in. In our case, we just declare `public.data` as a parent for our new type. This is about the most generic thing you can do. It basically just tells the operating system that our UTI exists and represents a chunk of data.

Note that the preceding XML doesn't say anything about our app. In fact, it's only there to declare the existence of our new data type, and establish a potential mapping between the filename extension we specified (*dudeldoc*) and the UTI (`com.rebisoft.dudeldoc`). With this bit of metadata at hand, the operating system can look at a *.dudeldoc* file and at least determine a UTI for it.

Declaring Data Type Ownership Using UTI

But what will the operating system do with that UTI? The idea is that it will find an app that declares, “Hey, I know how to open that file!” That's where another piece of XML comes in. Add the following to the *Dudel-Info.plist* file, directly below the previous XML you added.

```
<key>CFBundleDocumentTypes</key>
<array>
        <dict>
                <key>CFBundleTypeIconFiles</key>
                <array>
                        <string>Dudel_AppIcon_320x320.png</string>
                        <string>Dudel_AppIcon_64x64.png</string>
                </array>
                <key>CFBundleTypeName</key>
                <string>Dudel Document File</string>
                <key>CFBundleTypeRole</key>
                <string>Editor</string>
                <key>LSHandlerRank</key>
                <string>Owner</string>
                <key>LSItemContentTypes</key>
                <array>
                        <string>com.rebisoft.dudeldoc</string>
                </array>
        </dict>
</array>
```

Here, we have another single-entity array, this time declaring that our app knows how to open and edit documents of the `com.rebisoft.dudeldoc` type. It also specifies a couple of icons that the operating system can use to display a graphical representation of this type. The source archive for this book includes these two icons, which you should add to your Xcode project. This XML also defines a human-readable name for this document type.

www.it-ebooks.info

Next are a couple of key/value pairs that give the operating system more information. The first says that Dudel is an `Editor` for this type, capable of making changes to it (as opposed to a `Viewer`, which can only display the file). The next definition says that our app should be considered the `Owner` for this data type. This means that if the system must choose between one of several apps capable of opening a *.dudeldoc* file, it will tend to pick Dudel first. These definitions are direct carryovers from Mac OS X, where they are commonly used by the Finder to figure out how to deal with files you double-click, for example. In iOS, there's still not a lot of concrete use for these settings. However, if your app is dealing with files, it's best to set these up, in case future versions of iOS make better use of this metadata.

Testing the File-Receiving Feature

Now that we've declared our UTIs, Dudel is nearly ready to have files sent to it. In fact, if you build and run Dudel on your iPad, as far as iPhone OS can tell, Dudel really seems ready. For example, if Mail encounters a *.dudeldoc* file in a mail message, it will try to open it in Dudel.

You can test this now by using your computer to e-mail yourself an empty document named *something.dudeldoc* and then viewing that e-mail on your iPad. You should see something like Figure 10–2, with Mail displaying an icon for the empty document. If you tap and hold on the document icon, you'll see a popover that gives you an option to open the file in Dudel. But don't try that just yet. We haven't implemented the method in our app delegate to handle the file. We'll do that next.

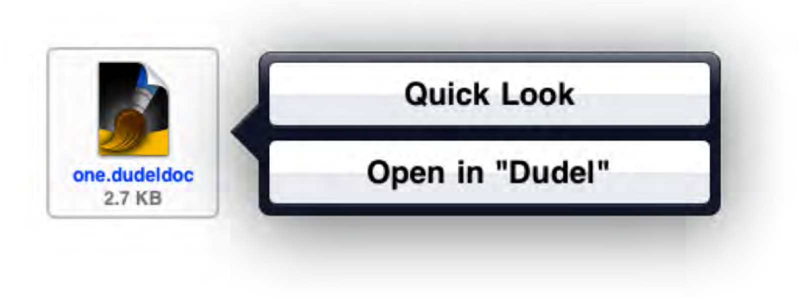


Figure 10–2. *About to open a Dudel document in Mail*

NOTE: As of this writing, using iOS 3.2, Mail's handling of custom UTIs seems a bit incomplete. A UTI declared with only `public.data` as its parent will sometimes show up in Mail, but other times it will not. If you're suffering from this problem, try making it declare `public.text` as the parent type. That will add another somewhat messy view to Mail (since it will want to try to preview the “text,” which isn't text at all), but at least it will let you try out the workflow of sending and receiving documents.

www.it-ebooks.info

Retrieving File Information from Launch Options

Now we need to write some code that will be run when our app is launched, making it check to see if it's being asked to open a file. We'll add code to the `application:didFinishLaunchingWithOptions:` method in `DudelAppDelegate`. If the app is being asked to open a file, it happens here, via one of the values in the `launchOptions` dictionary.

```
- (BOOL)application:(UIApplication *)application
didFinishLaunchingWithOptions:(NSDictionary *)launchOptions {
  // Override point for customization after app launch
  [window addSubview:splitViewController.view];
  [window makeKeyAndVisible];
  [[NSNotificationCenter defaultCenter] addObserver:self
    selector:@selector(fileListControllerSelectedFile:)
    name:FileListControllerSelectedFile object:fileListController];

  NSURL *openedUrl = [launchOptions
    objectForKey:UIApplicationLaunchOptionsURLKey];
  if (openedUrl) {
    if ([openedUrl isFileURL]) {
      // Handle the file that's passed in
      [[FileList sharedFileList] importAndSelectFromURL:openedUrl];
    }
  }
  return YES;
}
```

The URL that's passed in here points to a temporary location where the system is holding a copy of the file we sent. If we want to keep it, we need to make our own copy in the app's normal Documents directory. Instead of doing it right there in the app delegate, we pass that responsibility along to a class that has taken care of many other file-management issues for us: `FileList`. Add the following lines to *FileList.h* and *FileList.m* to make the magic happen.

```
// FileList.h
- (void)importAndSelectFromURL:(NSURL *)url;
// FileList.m
- (void)importAndSelectFromURL:(NSURL *)url {
  NSString *importFilePath = [url path];
  NSString *importFilename = [importFilePath lastPathComponent];
  NSArray *dirs =
    NSSearchPathForDirectoriesInDomains(NSDocumentDirectory,
    NSUserDomainMask, YES);
  NSString *dir = [dirs objectAtIndex:0];
  NSString *filename = importFilename;
  NSFileManager *fm = [NSFileManager defaultManager];
  if ([fm fileExistsAtPath:[dir
    stringByAppendingPathComponent:filename]]) {
    NSString *filenameWithoutExtension = [filename
      stringByDeletingPathExtension];
    NSString *extension = [filename pathExtension];
    BOOL filenameAlreadyInUse = YES;
    for (NSUInteger counter = 1; filenameAlreadyInUse; counter++) {
      filename = [NSString stringWithFormat:@"%@-%d.%@",
```

www.it-ebooks.info

```
                filenameWithoutExtension,
                counter,
                extension];
        filenameAlreadyInUse = [fm fileExistsAtPath:[dir
          stringByAppendingPathComponent:filename]];
      }
    }
    NSError *error = nil;
    [fm copyItemAtPath:importFilePath toPath:[dir
      stringByAppendingPathComponent:filename] error:&error];
    [allFiles addObject:filename];
    [allFiles sortUsingSelector:@selector(compare:)];
    self.currentFile = filename;
    [[NSNotificationCenter defaultCenter]
      postNotificationName:FileListChanged object:self];
}
```

This method basically just copies the file, but not before checking to make sure the filename isn't already taken. If the name exists, it will come up with a new filename by tacking on a number.

Sending a Dudeldoc File

Now we are ready to accept files sent to Dudel. But so far, there aren't any *.dudeldoc* files anywhere outside Dudel's own documents directory, so no one else can send us anything! Let's solve that by adding yet another item to our action menu to let us send a *.dudeldoc* file as an e-mail attachment. This should be very familiar to you by now. Make the additions and changes to the various files as follows:

```
// ActionsMenuController.h
typedef enum SelectedActionType {
  NoAction = -1,
  NewDocument,
  RenameDocument,
  DeleteDocument,
  EmailDudelDoc,
  EmailPdf,
  OpenPdfElsewhere,
  ShowAppInfo
} SelectedActionType;

// ActionsMenuController.m
- (NSInteger)tableView:(UITableView *)tableView numberOfRowsInSection:
  (NSInteger)section {
    // Return the number of rows in the section.
    ~~return 6;~~
    return 7;
}
// Add this near the end of tableView:cellForRowAtIndexPath:
    case EmailDudelDoc:
      cell.textLabel.text = @"Send DudelDoc via email";
      break;

// DudelViewController.m
- (void)sendDudelDocEmail {
```

www.it-ebooks.info

```
  NSString *filepath = [FileList sharedFileList].currentFile;

  [self saveCurrentToFile:filepath];
  NSData *fileData = [NSData dataWithContentsOfFile:filepath];
  MFMailComposeViewController *mailComposer =
    [[[MFMailComposeViewController alloc] init] autorelease];
  mailComposer.mailComposeDelegate = self;
  [mailComposer addAttachmentData:fileData
    mimeType:@"application/octet-stream"
    fileName:[filepath lastPathComponent]];
  [self presentModalViewController:mailComposer animated:YES];
}
// Add this near the end of handleDismissedPopoverController:
      case EmailDudelDoc:
        [self sendDudelDocEmail];
        break;
```

Now you should be able to test the entire workflow for handling Dudel documents. You can create a document, e-mail it to yourself, switch to Mail and see the attachment, and press and hold to see the popover that lets you send the file to Dudel. After you send the file, you can see it in your list with a new filename (in order to avoid overwriting the original file).

Desktop Synchronization

Another new piece of document-related functionality on the iPad is the ability to synchronize your app's files with the desktop. Starting with iOS 3.2, applications can register for desktop synchronization by adding the following key/value pair to their *Info.plist* file:

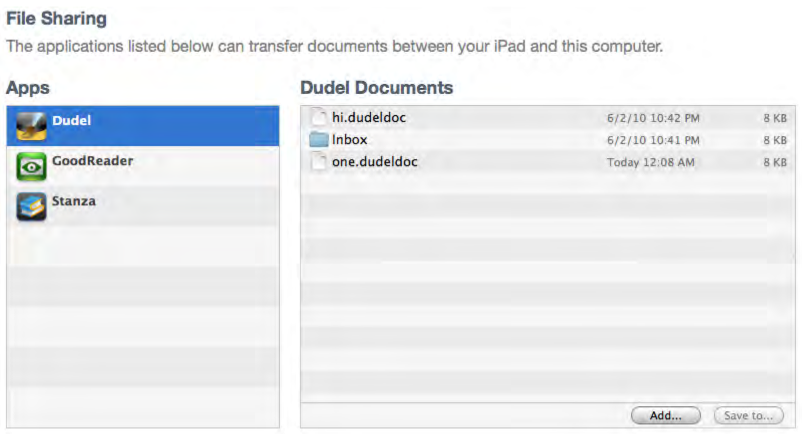
```
<key>UIFileSharingEnabled</key>
<true/>
```

That's all! Add this to *Dudel-Info.plist*, and build and run that on your iPad.

Now, while your iPad is still connected to your computer, go into iTunes on your computer, select your iPad in the navigation area on the left, and then click the Apps tab to bring it forward. You'll see all the installed apps as usual, but with one new twist: a new File Sharing section at the bottom of the window's content area. This section shows all the apps that have sharable document content, and Dudel is there! Select Dudel, and you can see everything in our Documents directory, as shown in Figure 10–3.

You can drag files out to the Finder to save them on your computer, and you can drag other files from your computer back in. You'll see whatever documents you've created in Dudel, and perhaps also a subdirectory called Inbox. (The Inbox directory is the place where the system puts files temporarily when your app is asked to open them. You should probably just leave it alone.)

www.it-ebooks.info


File Sharing

The applications listed below can transfer documents between your iPad and this computer.

Apps	Dudel Documents		
Dudel	hi.dudeldoc	6/2/10 10:42 PM	8 KB
GoodReader	Inbox	6/2/10 10:41 PM	8 KB
Stanza	one.dudeldoc	Today 12:08 AM	8 KB

Figure 10–3. *The iTunes view of your Dudel document directory*

NOTE: If the interface for accessing documents from your iPad seems uninspired to you, you're not alone. When Apple announced this technology initially, many of us were hoping for something better, like having the Documents directories from the iPad show up directly in the Finder, or even allowing you to automatically synchronize files between an iPad and a computer (instead of manually dragging them around as the current setup requires). However, the iPad is still a new product, and surely the software (both in the device and on the computer it links with) will evolve over time. Let's keep our fingers crossed for some improvements from Apple in this area.

Share and Share Alike

Now you've gotten a taste of the various document-management features that the iPad offers. What you've seen here only scratches the surface.

For instance, we've assumed that when importing a document whose filename already exists, we should just generate a new filename. However, you might want to make a suite of applications that are basically playing "hot potato" with a document, quickly passing it around to have different things done with it in each place. So, you might want to instead let a duplicate-named imported document just replace the old one. Or you may want to ask the user what to do each time it happens.

This way of dealing with documents opens a lot of possibilities for making apps that interoperate smoothly with one another. It will be interesting to see how this evolves.

Speaking of evolving, it's now time for you to head on to Chapter 11. There, you'll learn how to take an existing iPhone app and prepare it for the iPad, so that it can make the most of the new screen size and new GUI paradigms.

www.it-ebooks.info

Chapter 11

From iPhone to iPad

Over the course of this book, you've gone through the creation of several apps made just for iPad. But if you're coming into this as an iPhone developer, chances are you already have one or more apps that you would like to bring over to the iPad. Yes, you could just run them on the iPad, but that's far from ideal. Both of the iPad's methods for displaying an iPhone app—either showing it at actual size in the middle of the screen or stretching the display to make it fill the entire iPad screen—are pretty disappointing for most applications.

In this chapter, you'll learn how to take an existing iPhone application and turn it into a first-class citizen on the iPad, making full use of the extra screen real estate, as well as the new user interface functionality available in the iPhone SDK. We'll start by creating a new iPhone application that includes drill-down navigation and a detail view, and then consider how to adapt it to the iPad. We'll walk through the steps to take in Xcode and Interface Builder, and reorganize the view controllers to make the application work more nicely on the big screen.

Introducing NavApp for iPhone

To demonstrate the process of preparing an iPhone app for the new world of the iPad, we're going to create a simple application called NavApp, which follows a typical iPhone app pattern. You pick an item from a table view, which leads you to another table view full of items. Pick one of those, and you see some sort of detail about the item you chose. This basic arrangement can be seen in standard iPhone and iPad apps such as Mail and Settings.

Figure 11–1 shows the basic flow of NavApp, starting with a top-level view and drilling down to the details. The functionality here is really bare-bones, in order to keep the project small. That way, we can focus on the techniques needed specifically for transforming an iPhone app into an iPad app. These techniques are applicable to any sort of app that should behave differently on the iPad than on the iPhone, particularly applications that make use of `UINavigationController` and should be updated to display the navigation views in a more iPad-friendly way (which is probably most of them).

www.it-ebooks.info

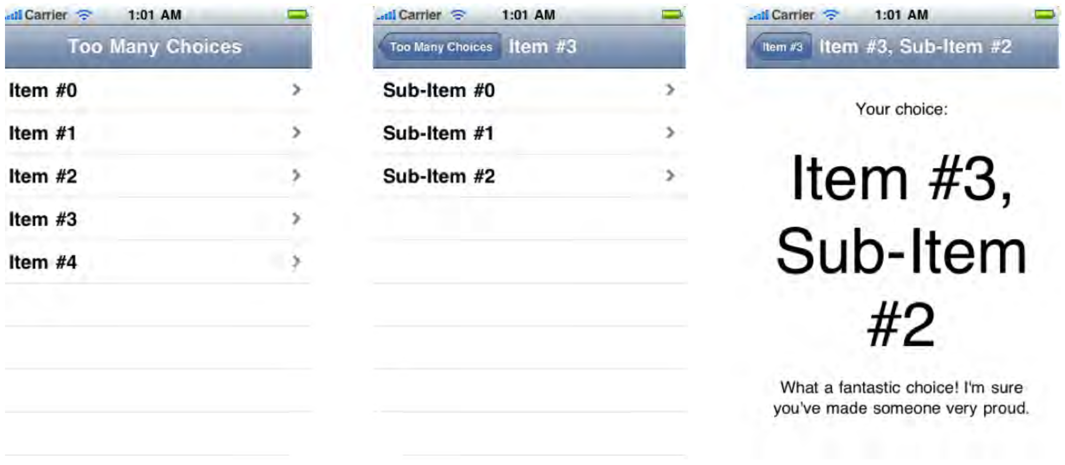


Figure 11–1. *The views of NavApp, shown in sequence*

Creating the NavApp Project

Start by creating a new project in Xcode, and choose the Navigation-based Application template, which automatically forces you to target iPhone instead of iPad. Name your project `NavApp` and save it somewhere appropriate.

Xcode creates classes called `RootViewController` and `NavAppAppDelegate` for you, along with GUI layouts in *RootViewController.xib* and *MainWindow.xib*. If you've done much iPhone development in the past, you're familiar with this arrangement. The *MainWindow.xib* file, which is loaded when the app launches, contains a `UINavigationViewController`, which itself contains a `RootViewController`. The `RootViewController` will be presented as the initial view of the `UINavigationViewController`, and when the user selects an item, it will push the next view controller that should be displayed onto the `UINavigationViewController`.

We'll add a class that represents the next level of items that the user can drill down into, and then, a little later, another class that displays details about the user's selection. First, let's modify the classes that Xcode created for us, to make them do our bidding. As it turns out, the `NavAppAppDelegate` class created by Xcode is just fine, as is the *RootViewController.h* file, but we'll need to make some changes to *RootViewController.m*.

Enhancing the Root View Controller

First, import the header for the second-level view controller. We haven't created that class yet, but we'll get to it soon enough, so we may as well add it now:

```
#import "SecondLevelViewController.h"
```

Next, find the `viewWillAppear:` method. By default, it's commented out and doesn't really do anything. Remove the surrounding `/* */` and give it the following content:

www.it-ebooks.info

```
- (void)viewWillAppear:(BOOL)animated {
  [super viewWillAppear:animated];
  self.navigationItem.title = @"Too Many Choices";
}
```

Now make a small change so that our table view has some rows:

```
- (NSInteger)tableView:(UITableView *)tableView numberOfRowsInSection:(NSInteger)section
{
~~  return 0;~~
  return 5;
}
```

Next, add a few lines near the end of `tableView:cellForRowAtIndexPath:`, to give those rows some content:

```
  // Configure the cell.
  cell.textLabel.text = [NSString stringWithFormat:@"Item #%d",
    indexPath.row];
  cell.accessoryType = UITableViewCellAccessoryDisclosureIndicator;

  return cell;
```

Finally, remove the commented sample code from `tableView:didSelectRowAtIndexPath:`, and give it the following content instead:

```
- (void)tableView:(UITableView *)tableView
  didSelectRowAtIndexPath:(NSIndexPath *)indexPath {
  SecondLevelViewController *detailViewController =
    [[SecondLevelViewController alloc] initWithStyle:
      UITableViewStylePlain];
  detailViewController.choice = [NSString stringWithFormat:@"Item #%d",
    indexPath.row];
  // Pass the selected object to the new view controller.
  [self.navigationController pushViewController:detailViewController
    animated:YES];
  [detailViewController release];
}
```

That's all we need to do for our `RootViewController` class.

Defining the Second Level View Controller

Now use the New File Assistant to create a new controller class called `SecondLevelViewController`, making it a subclass of `UITableViewController` with no *.xib* file. Like the `RootViewController` class, `SecondLevelViewController` requires just a few changes to the Xcode-generated default to make it work the way we want.

Begin in the *.h* file, adding the `choice` instance variable and property to keep track of which item the user chose at the root level:

```
@interface SecondLevelViewController : UITableViewController {
  NSString *choice;
}
@property (copy, nonatomic) NSString *choice;
@end
```

www.it-ebooks.info

Now switch over to *SecondLevelViewController.m*, where we'll make a few changes, similar to those we made for RootViewController. Import the header for the detail class that we'll use to present the final choice:

```
#import "ChoiceViewController.h"
```

Then take care of the choice property by adding this line right after the @implementation line:

```
@synthesize choice;
```

Next, uncomment the viewWillAppear: method, and use it to set the title for the navigation bar.

```
- (void)viewWillAppear:(BOOL)animated {
  [super viewWillAppear:animated];
  self.navigationItem.title = self.choice;
}
```

Add the following details to give our table view a section and a few rows:

```
- (NSInteger)numberOfSectionsInTableView:(UITableView *)tableView {
  // Return the number of sections.
  return 1;
}
- (NSInteger)tableView:(UITableView *)tableView
  numberOfRowsInSection:(NSInteger)section {
  // Return the number of rows in the section.
  return 3;
}
```

Then fill in the end of the tableView:cellForRowAtIndexPath: method:

```
  // Configure the cell...
  cell.textLabel.text = [NSString stringWithFormat:@"Sub-Item #%d",
    indexPath.row];
  cell.accessoryType = UITableViewCellAccessoryDisclosureIndicator;
  return cell;
```

Now implement the tableView:didSelectRowAtIndexPath: method, to pass along the final selection to the detail view:

```
- (void)tableView:(UITableView *)tableView didSelectRowAtIndexPath:(NSIndexPath
*)indexPath {
  ChoiceViewController *detailViewController = [[ChoiceViewController
    alloc] initWithNibName:@"ChoiceViewController" bundle:nil];
  detailViewController.choice = [NSString stringWithFormat:
    @"%@, Sub-Item #%d", self.choice, indexPath.row];
  // Pass the selected object to the new view controller.
  [self.navigationController pushViewController:detailViewController
    animated:YES];
  [detailViewController release];
}
```

Finally, complete the handling of the choices property by clearing it in the dealloc method:

```
- (void)dealloc {
```

www.it-ebooks.info

```
  self.choice = nil;
  [super dealloc];
}
```

That takes care of the SecondLevelViewController class.

Defining the Choice View Controller

The final class we need for this project is a UIViewController subclass, containing a *.xib* file, named ChoiceViewController. This class is very simple. We just need to add a couple of instance variables, implement a single method, and define the GUI in Interface Builder. Start by adding the bold lines here to the header file:

```
@interface ChoiceViewController : UIViewController {
  NSString *choice;
  IBOutlet UILabel *choiceLabel;
}
@property (copy, nonatomic) NSString *choice;
@end
```

Then switch to *ChoiceViewController.m*, and add the following directly after the @implementation line:

```
@synthesize choice;
```

Unlike the template-generated table view controller classes, ChoiceViewController doesn't have a commented-out viewWillAppear: method just waiting for us to fill in, so we'll need to give it one, as follows:

```
- (void)viewWillAppear:(BOOL)animated {
  [super viewWillAppear:animated];
  self.navigationItem.title = self.choice;
  choiceLabel.text = choice;
}
```

Finally, clear out the choices property in the dealloc method:

```
- (void)dealloc {
  self.choice = nil;
  [super dealloc];
}
```

The last step to complete this project is to set up the *ChoiceViewController.xib* file. Open it in Interface Builder, and use three UILabel objects from the Library window to create a GUI like the one shown in Figure 11–2.

www.it-ebooks.info

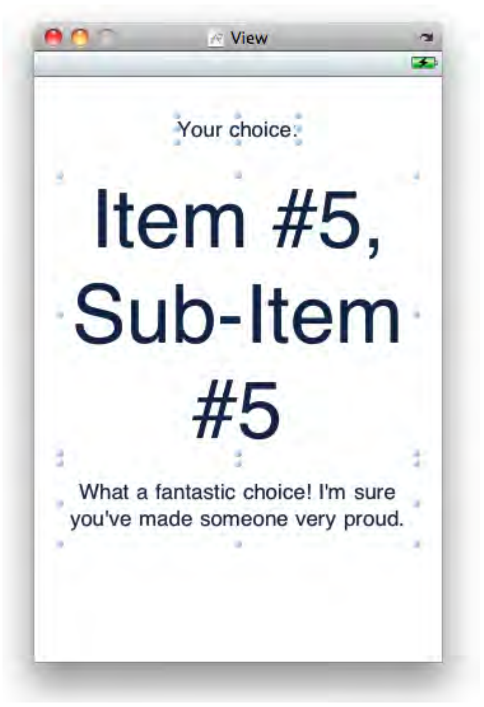

View
Your choice:
Item #5, Sub-Item #5
What a fantastic choice! I'm sure you've made someone very proud.

Figure 11–2. *The ChoiceViewController GUI layout. The little resizing controls show where the labels are placed.*

With the GUI in place, control-drag from the File's Owner icon to the central `UILabel` and connect the `choiceLabel` outlet to it. After doing that, save your work and go back to Xcode.

Now you're almost ready to build and run NavApp. Before that, though, you need to know about a nuance in Xcode that you may not have thought much about: the Overview popup in the Xcode toolbar, particularly the Active SDK section, where a number of SDK choices are made available.

Choosing the Active SDK

The Active SDK choice is actually dual-purpose. It controls both which SDK your app is built with (affecting which APIs are available to your app) and which SDK is used for launching the iPhone Simulator when you try things out on your machine. At the time of this writing, the latest public iPhone OS release for iPhone devices is 3.1.3; for iPad, it's 3.2.

To see what these settings do, first set the Active SDK to iPhone Simulator 3.1.3, and then hit the Build and Run button in the toolbar. Xcode will start the Simulator in iPhone mode and launch your application. You should now be able to navigate through the app as you would expect, making selections and seeing results in the final screen. Now quit your app.Back in Xcode, switch the Active SDK to iPhone Simulator 3.2. Then start your app, without first building, by selecting **Run ➤ Run** from the menu. Xcode will relaunch the

www.it-ebooks.info

simulator in iPad mode, and launch your app in the Simulator. But it won't work! You'll find that your app crashes immediately. This seems to be a problem inherent to the interaction between Xcode and the Simulator. Trying to run an app compiled for OS 3.1.3 in the Simulator running OS 3.2 just doesn't work.

However, the reverse is OK. You can build your app for OS 3.2 and still run it on OS 3.1.3, as long as you do some sort of runtime checks to make sure you're not using any OS 3.2 features. That's the suggested approach for creating universal apps that will run on both iPhone and iPad, and it's the direction we'll be taking with NavApp.

With iPhone Simulator 3.2 still selected, choose **Build ➤ Build and Run** from the menu. Your app will start up with the Simulator in iPad mode, and this time it will work, at least in a rudimentary way.

You'll have the sort of iPad experience that anyone with an iPad has probably had when running iPhone software that hasn't yet been updated for iPad. With a Cocoa Touch app such as this, your options are fairly uninteresting:

- You can display the app at actual size in the middle of the screen, ignoring most of the space available on the iPad.
- You can expand the app to fill the entire screen by doubling the number of pixels used. This misuses the nice big screen by just stretching all the GUI elements to grotesque proportions.

Fortunately, we can do better. Xcode gives us a way to upgrade an existing iPhone project to support iPad as well, automatically paving the way for our app to run on both iPhone and iPad by adding a new main *.xib* file, which is laid out to fit the iPad display, and configuring it to be loaded when the app is launched on an iPad.

Adding iPad to the Mix

Now we’re ready to upgrade NavApp for the iPad. Before kicking off this process, make a backup copy of your project directory. This can be useful in case you want to compare your original app with the iPad-ready version that Xcode sets up.

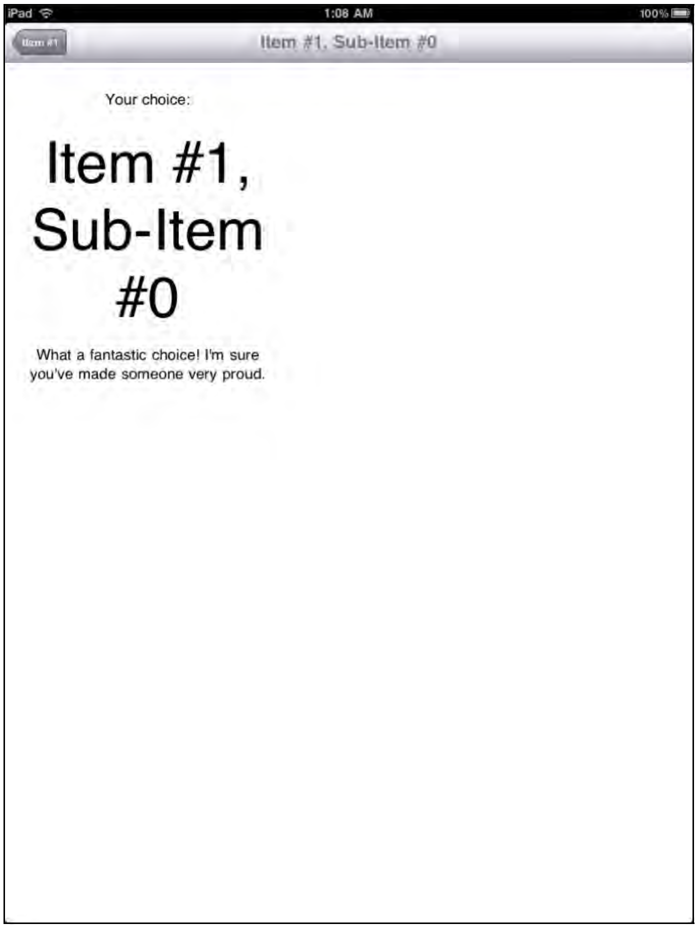
Next, in the Groups & Files section of your Xcode project window, open the Targets section and select NavApp. Then select **Project ➤ Upgrade Current Target for iPad** from the menu. Xcode will present you with a modal sheet that asks whether you want to create a single universal application that will run on both iPhone and iPad, or create a second target for an iPad application in your project, leaving the original target intact. Select the One Universal application option, and then click OK.

Xcode now does a few simple things. It copies the *MainWindow.xib* file to *MainWindow-iPad.xib*, making a few changes to the file's contents, such as specifying the iPad's screen size. The new *.xib* file is added to the project, and a line is added to the *NavAppCompare-Info.plist* file, specifying that this new *.xib* file should be used when launching on the iPad. It also makes a few changes to your project, such as setting the base SDK to 3.2.

www.it-ebooks.info

Taking the Upgraded NavApp for a Spin

Make sure that Simulator 3.2 is chosen as the Active SDK in Xcode, and then build and run the upgraded app. You should see NavApp spring to life, full size and at full resolution, in the Simulator. It will work just as it did in iPhone form, letting you drill down through the structure we laid out previously and displaying a similar result, as shown in Figure 11–3.



Your choice:

Item #1, Sub-Item #0

What a fantastic choice! I'm sure you've made someone very proud.

Figure 11–3. *The ChoiceViewController GUI, running on the iPad*

Although this type of upgrade works, it isn't what you really want for an iPad app. The popular iPhone pattern of drilling down into structures, with the entire content of the screen sliding out the side, isn't prevalent on the iPad. In fact, Apple actually recommends against that usage, for a couple of reasons:

www.it-ebooks.info

- The full-screen wipe in response to simple tap on the screen, which works well enough on a small display such as the iPhone's, begins to feel a little off on a larger screen. The full-screen swish is best reserved for situations where the user actually made a swiping gesture.
- Since the iPad has so much more screen real estate, you can easily show a drill-down navigation view alongside of, or hovering in front of, the main content, by using a split view or a popover view (as you've seen in earlier chapters).

So, let's rethink the NavApp GUI for the iPad version.

Reconsidering iPhone Design ChoicesWhile the navigation views are the first things you see in NavApp, they're actually just stepping-stones leading to the app's main content view, which is handled by the `ChoiceViewController` class.

For the iPad version of this app, let's rework the design so that the final view is now front and center. We'll go about this by reconfiguring some *.xib* files, conditionally changing the behavior of the navigation views in response to user actions, and extending the `ChoiceViewController` class so that it can display something reasonable, even when the user hasn't selected anything yet. The navigation views will end up being displayed in the left-hand side of a split view, or in a floating popover view, depending on whether the iPad is in landscape or portrait mode. This is similar to what we did in the Dudel application earlier in this book, and even goes a step closer to the way that the iPad's built-in Mail application handles drilling down through accounts and folders to reach your messages.

The first step toward making this work will be to redefine what the `NavAppAppDelegate` class does, both in code and in its related *.xib* files. This class was created automatically when we created the Xcode project, and in its original form, it sets up the navigation interface (since that's the kind of project we created). We're going to add a bit of code that checks at runtime to see if we're running on an iPad, and if so, instead set up a split view interface. The other half of this redesign will be configuring the *MainWindow-iPad.xib* file so that it actually wraps things up in a split view.

Conditional Behavior: Know Your Idioms

Open *NavAppAppDelegate.h*, and add an outlet for a future `UISplitViewController` as shown here:

```
@interface NavAppAppDelegate : NSObject <UIApplicationDelegate> {
  UIWindow *window;
  UINavigationController *navigationController;
  UISplitViewController *splitViewController;
}
@property (nonatomic, retain) IBOutlet UIWindow *window;
@property (nonatomic, retain) IBOutlet UINavigationController *navigationController;
@property (nonatomic, retain) IBOutlet UISplitViewController *splitViewController;
@end
```

www.it-ebooks.info

NOTE: Throughout this book, we've been creating outlets by putting `IBOutlet` in front of the instance variable declaration. So why are we suddenly putting it in the property declaration here? The two are equivalent, and the choice of where to put `IBOutlet` is really a matter of style. In this case, we're working with a class template that was generated by Xcode and contains its `IBOutlet` markers in the property declarations. Rather than modifying the generated source code to make a change that has no quantifiable effect, we're just following the example of the surrounding code here. When in Rome …

Now switch over to *NavAppAppDelegate.m*, and configure the basics for the new outlet we created by adding this line near the top of the `@implementation` section:

```
@synthesize splitViewController;
```

Don't forget to free up that new resource as well:

```
- (void)dealloc {
  [splitViewController release];
  [navigationController release];
  [window release];
  [super dealloc];
}
```

Next is the interesting part of this class:

```
- (BOOL)application:(UIApplication *)application
    didFinishLaunchingWithOptions:(NSDictionary *)launchOptions {
  if (UI_USER_INTERFACE_IDIOM() == UIUserInterfaceIdiomPad) {
    [window addSubview:[splitViewController view]];
  } else {
    [window addSubview:[navigationController view]];
  }
  [window makeKeyAndVisible];
  return YES;
}
```

This method uses the `UI_USER_INTERFACE_IDIOM` function to determine whether the app is running on an iPad. If it is, we'll present a different view than what we show for the iPhone.

This is a pretty subtle shift. Keep in mind that the app delegate class is loaded from the app's main *.xib* file, and once it's loaded and the application has finished launching, this is the method that actually gives the application a view to display. With this small change, we radically alter the entire appearance and flow of the app! Of course, to make that really happen, we'll need to make sure that our new `splitViewController` outlet is actually pointing at something.

www.it-ebooks.info

Configuring the Main iPad GUI

It's time to reorganize the main GUI, putting the navigation view inside a split view. Open *MainWindow-iPad.xib* in Interface Builder, and switch the main window to column view, revealing something like Figure 11–4.

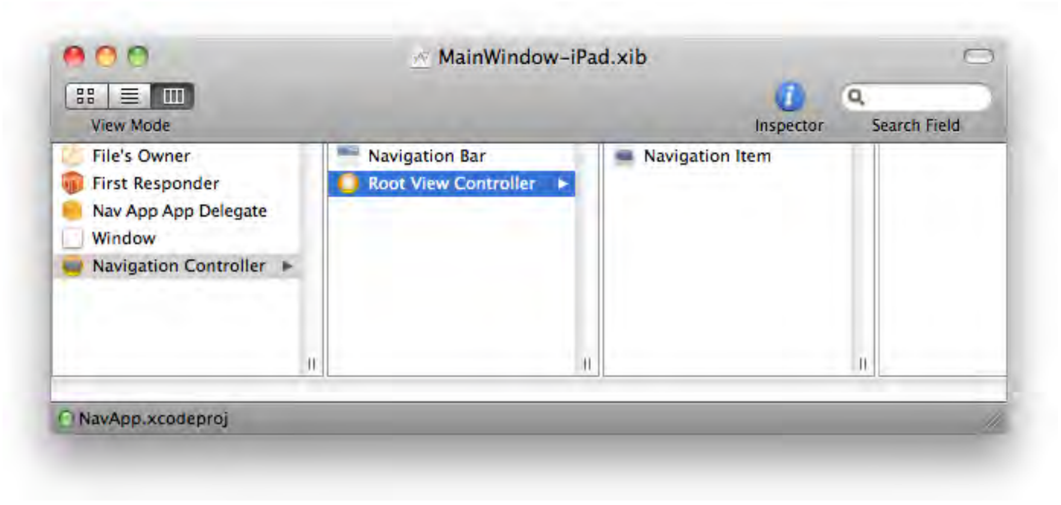


Figure 11–4. *The primary .xib file for the iPad version of our app, before our enhancements*

Here, we've drilled down into the Navigation Controller and Root View Controller objects, revealing the complete structure of the objects in this *.xib* file. If you select the Nav App App Delegate object in this window, and then open the connections inspector, you'll see that it has outlets connected to the window and to the navigation controller—the two objects that are tied together in code when the app launches—by virtue of adding the view controller's view to the window. We're going to add a `UISplitViewController` to this *.xib* file, and configure it so that the preceding conditional code adds the split view to the window.

Start by finding a `UISplitViewController` in the Library, and dragging it to the leftmost column of the *.xib* window. Then control-drag from the app delegate to the new split view controller, and hook up the `splitViewController` outlet.

The split view is meant to display views for two view controllers at once, and by default, the one we created will have a navigation controller and a generic view controller. The navigation controller contains a generic table view controller. We need to modify these objects, making them instances of the real classes we're using in our app.

Drill down into the Navigation Controller object inside the Split View Controller object, and select the Table View Controller object it contains. Open the identity inspector, and change that controller's class to `RootViewController`. Then switch to the attribute inspector to configure the controller a little more. Set the Title to **Changes**, and the NIB Name to *RootViewController*.

www.it-ebooks.info

Now backtrack a bit, and select the generic View Controller object inside the Split View Controller object. Once again, bring up the identity inspector, and set this controller's class to `ChoiceViewController`

. Then switch back to the attribute inspector, and set the NIB Name to *ChoiceViewController-iPad*. That's the name of an *.xib* file that doesn't exist yet, but soon will, since we'll create it in the next section. At this point, your *.xib* window should look something like Figure 11–5.

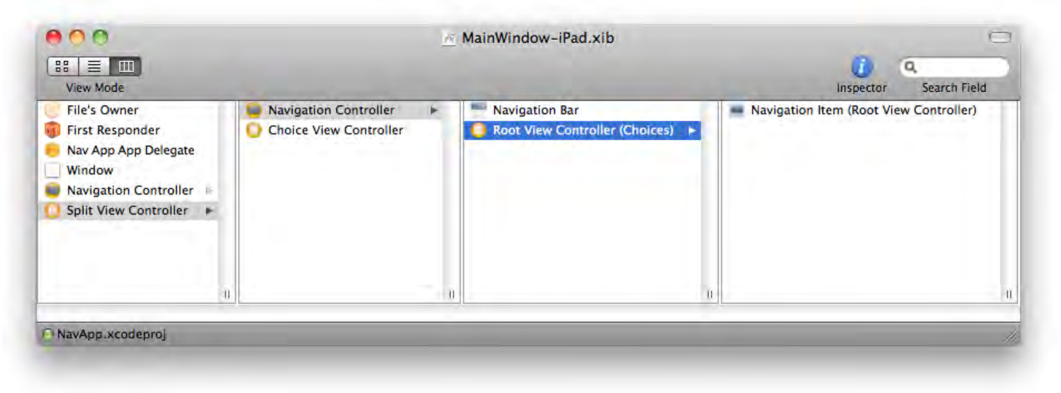


Figure 11–5. *The iPad version of the main .xib file after reconfiguring for iPad navigation*

As you learned in Chapter 8, proper use of a split view requires you to provide the split view controller with a delegate, which plays a role in juggling between a split view and a popover when the iPad is rotated. In this case, we'll connect the split view controller's delegate outlet to the choice view controller, and later we'll implement the delegate code there. Select the Split View Controller item so that you can see its children, control-drag from the Split View Controller object to the Choice View Controller object, and then connect the `delegate` outlet.

The final change for the *MainWindow-iPad.xib* file is to delete the navigation controller that's at the top level of the *.xib* file. The app delegate still has an outlet to it, but that doesn't really matter. When our app runs on an iPad, the iPad-specific *.xib* file is loaded, and the unconnected outlet is ignored.

Creating the Choice View Controller GUI for iPad

Earlier, we configured the `ChoiceViewController` instance in our main iPad GUI to use a special iPad-friendly *.xib* file. Let's create that now. Switch back to Xcode, and use the New File Assistant to create a new View XIB file, located in the iPhone OS / User Interface section. Make sure the product menu is displaying iPad, and click Next. Then name it *ChoiceViewController-iPad.xib* and click Next. You'll see the new file added to your project.

Before editing the new GUI, open *ChoiceViewController.h* and add the following instance variable:

www.it-ebooks.info

```
  IBOutlet UIToolbar *toolbar;
```

This new `toolbar` outlet will point at a toolbar in the iPad version of the GUI, which we'll create soon.

Open both *ChoiceViewController.xib* and *ChoiceViewController-iPad.xib* in Interface Builder. Unlike the original *.xib* file, which was created along with the class, the new one is kind of a blank slate. Select the File's Owner icon, and use the identity inspector to set its class to `ChoiceViewController`. Now switch to *ChoiceViewController.xib*, open its view, select all the GUI objects in there, and press ⌘C to copy them. Then switch back to the new *ChoiceViewController-iPad.xib* file, open its view (which you'll see is iPad-size), and paste in the GUI objects. You'll want to center them in the display, and should probably resize them to fill the width of the display as well—no sense letting all that screen real estate go to waste!

Now use the Library to find a `UIToolbar`, and drag it to the new iPad-ready `ChoiceViewController` view, dropping it at the top of the view so that the toolbar appears up there. The toolbar contains a single default item, which you should go ahead and delete. Finally, connect the outlets from the File's Owner icon to the GUI objects in the *.xib* file: `choiceLabel` to the big label in the middle, `toolbar` to the toolbar you just created, and `view` (which we didn't define in out class, but inherited from `UIViewController`) to the entire containing view.

Implementing the Split View Delegate Methods

Go back to Xcode, and open the *ChoiceViewController.m* file. Add the two required methods for the `UISplitViewController`:

```
- (void)splitViewController:(UISplitViewController*)svc
  willHideViewController:(UIViewController *)aViewController
  withBarButtonItem:(UIBarButtonItem*)barButtonItem
  forPopoverController:(UIPopoverController*)pc {
  // add the new button item to our toolbar
  NSArray *newItems = [toolbar.items arrayByAddingObject:barButtonItem];
  [toolbar setItems:newItems animated:YES];

  // configure the button
  barButtonItem.title = @"Choices";
}
- (void)splitViewController:(UISplitViewController*)svc
  willShowViewController:(UIViewController *)aViewController
  invalidatingBarButtonItem:(UIBarButtonItem *)button {
  // remove the button
  NSMutableArray *newItems = [[toolbar.items mutableCopy] autorelease];
  if ([newItems containsObject:button]) {
    [newItems removeObject:button];
    [toolbar setItems:newItems animated:YES];
  }
}
```

www.it-ebooks.info

That's all we need to do in order to handle switching between portrait and landscape orientation. The split view controller will call the first method when switching to portrait mode, and the second method when switching to landscape mode.

At this point, you should be able to run the app. You'll see that it works … to some extent. The split view kicks in, displaying itself on the left side in landscape mode, and shrinking down to a button in the toolbar in portrait mode. Rotating from one to another isn't working yet, but we'll get to that a little later.

The problem is in the interaction between the view controllers themselves. All the action—not only the navigation, but also the display of the final selection—is constrained to the navigation view, whether it's appearing in the split view or in a popover view. The big view for displaying the choice just displays the default “dummy” text all the time! Clearly, we need to update our table view controllers so that they do different things in response to the user selecting a row, depending on whether the app is running on an iPhone or iPad.

Tweaking the Navigation Logic

First, we need to make a pair of identical changes for both *RootViewController.m* and *ChoiceViewController.m*, to ensure that the views can rotate properly. In each of those files, uncomment the `shouldAutorotateToInterfaceOrientation:` method, and make it always return `YES`:

```
- (BOOL)shouldAutorotateToInterfaceOrientation:(UIInterfaceOrientation)o{
  // Return YES for supported orientations.
  return YES;
}
```

Now switch over to *SecondLevelViewController.m*, where we'll make some rather more critical changes. Start by adding this somewhere near the top of the file:

#import "NavAppAppDelegate.h"

Next, uncomment the `shouldAutorotateToInterfaceOrientation:` method, and make it always return `YES`, as we just did for the `RootViewController` and `ChoiceViewController` classes.

Then, in the `tableView:cellForRowAtIndexPath:` method, add a bit of code so that we don't show the final disclosure indicator (the little right-pointing arrow/chevron that lets the users know that they can keep on digging):

```
  cell.textLabel.text = [NSString stringWithFormat:@"Sub-Item #%d", indexPath.row];
  if (UI_USER_INTERFACE_IDIOM() != UIUserInterfaceIdiomPad) {
    cell.accessoryType = UITableViewCellAccessoryDisclosureIndicator;
  }
```

Then change the behavior of the final selection here, so that instead of creating and pushing another view controller onto the navigation stack, we grab the “global” `ChoiceViewController` and just tell it what the selection is:

```
- (void)tableView:(UITableView *)tableView didSelectRowAtIndexPath:(NSIndexPath
*)indexPath {
```

www.it-ebooks.info

```
  if (UI_USER_INTERFACE_IDIOM() == UIUserInterfaceIdiomPad) {
    NavAppAppDelegate *appDelegate =
     [[UIApplication sharedApplication] delegate];
    UISplitViewController *splitViewController =
      appDelegate.splitViewController;
    ChoiceViewController *detailViewController =
      [splitViewController.viewControllers objectAtIndex:1];
    detailViewController.choice = [NSString stringWithFormat:
    @"%@, Sub-Item #%d", self.choice, indexPath.row];
  } else {
    ChoiceViewController *detailViewController = [[ChoiceViewController
      alloc] initWithNibName:@"ChoiceViewController" bundle:nil];
    detailViewController.choice = [NSString stringWithFormat:
      @"%@, Sub-Item #%d", self.choice, indexPath.row];

    // Pass the selected object to the new view controller.
    [self.navigationController pushViewController:detailViewController
      animated:YES];
    [detailViewController release];
  }
}
```

Now you should be able to build and run the app, and see something closer to what we're shooting for. You can pick an item and a subitem, and your choice is displayed in the main view (and not inside the navigation view). However, this is still a bit off. That main view is just showing default dummy values (whatever you entered in Interface Builder) until the user selects something, and that's not what we want.

Let's enhance `ChoiceViewController` a bit, so that we can display something special for the no-selection state, before the user has navigated anywhere.

Enhancing the Main View with a No-Selection State

Basically, the new no-selection state will consist of hiding the labels at the top and bottom, and putting a special text in the large center label.

Start by adding two new outlets to the class definition *ChoiceViewController.h* so that we can access the top and bottom labels:

```
  IBOutlet UILabel *topLabel;
  IBOutlet UILabel *bottomLabel;
```

Now open *ChoiceViewController-iPad.xib*, and connect each of the new outlets by control-dragging from File's Owner to each of the labels and selecting the proper outlet. Save your changes, and go back to Xcode.

NOTE: If you're worried about the fact that the new outlets won't be used in the non-iPad version of the GUI, don't be! When this code runs on an iPhone and the iPhone version of the GUI is loaded, those unconnected outlets will simply be left as pointers to `nil`—no harm done.

www.it-ebooks.info

Open *ChoiceViewController.m* to make a few quick changes. The first changes will be for the `viewWillAppear:` method, to make it display the appropriate content depending on whether or not the `choice` property is populated:

```
- (void)viewWillAppear:(BOOL)animated {
  [super viewWillAppear:animated];
  if (self.choice) {
    self.navigationItem.title = self.choice;
    choiceLabel.text = choice;
    topLabel.hidden = NO;
    bottomLabel.hidden = NO;
  } else {
    choiceLabel.text = @"Make your choice!";
    topLabel.hidden = YES;
    bottomLabel.hidden = YES;
  }
}
```

Next, we're going to implement the `setChoice:` method. So far, we've relied on the `@synthesized` version of this, but now that we need to update the display once the value is set, we should actually do something here.

```
- (void)setChoice:(NSString *)c {
  if (![c isEqual:choice]) {
    [choice release];
    choice = [c copy];
    self.navigationItem.title = self.choice;
    choiceLabel.text = choice;
    topLabel.hidden = NO;
    bottomLabel.hidden = NO;
  }
}
```

Note that we don't need to do anything here to handle the case where the new value for `choice` is `nil` (which would theoretically require us to once again put “Make your choice!” in the main label and hide the other labels), since in practice, this will never occur. The only time that `choice` is set is when the user has just selected something, and in this app, that “something” is never `nil`.

At this point, you should be able to run the app and see it working the way that we intended and that makes the most sense, without any surprises for the users. When you first launch the app, nothing is selected in the navigation view, and the main display reflects this. Once you select something, your selection sticks around in the main view until you navigate to something else. This is pretty much identical to the behavior of other iPad apps such as Mail, so users should feel right at home with another app that works this way.

Thanks to the way we've written the app, it should also continue to work on iPhone just as it used to. To launch your app on the Simulator in iPhone mode, the key is to build the app using the 3.2 target, then switch to the 3.1.3 (or other iPhone OS) target, and select Run ➤ Run from the menu.

www.it-ebooks.info

Running on Multiple Targets

Earlier in this chapter, when you were first upgrading the NavApp iPhone project to include iPad support, you were given the choice to create a single, universal application or create separate targets for iPad and iPhone. We told you to go the universal route for NavApp, but what about the other option?

There are a number of reasons you may want to have separate iPhone and iPad apps. Maybe your application contains a lot of graphics at different sizes for the iPhone and iPad, and you want to keep the total file size down by eliminating iPhone-specific resources from the iPad version and vice versa. Or maybe you want to have separate products, with the iPad version including additional features and commanding a higher price, while still keeping everything in the same Xcode project.

Fortunately, those needs are easy to accommodate.

If you choose the multiple-target option when upgrading your project in Xcode, it really does most of the heavy lifting for you, and all the code changes you need to make can be done in the same way as you've seen here. In fact, if you were to start over with the original NavApp project, and upgrade it using the multiple-target option, you should be able make the exact same code and GUI changes described in this chapter and achieve the desired result: the ability to build different versions of the app for the iPhone and iPad. The only differences in the process are administrative in nature.

Using the multiple-target option, when creating a new resource such as an *.xib* file, you're prompted to pick which target or targets to include the resource in, so you need to make the appropriate choice there. And instead of just selecting a different Active SDK in the multipurpose pop-up control in Xcode to switch between launching the Simulator in iPad or iPhone mode, you need to select the target (iPhone or iPad) you want to execute. Otherwise, the steps are pretty similar to those described in this chapter. You can do the same sort of conditional coding, using the `UI_USER_INTERFACE_IDIOM` function to determine whether the app is running on an iPhone or iPad and adjusting accordingly. Sticking with this approach, rather than branching your code into separate projects, also makes things easier if you change your mind later. You can create a universal app without needing to also merge code bases that have diverged.

Juggling iDevices

In this chapter, you created a simple iPhone app from scratch, and then upgraded it to also work well on the iPad by creating some iPad-specific GUIs and using conditional coding to decide at runtime how the app should behave, depending on which platform it's running on. The code shown for tackling this situation is, of course, fitted to match the situation at hand. However, the strategies embodied by the code are general strategies for dealing with combined iPhone/iPad applications. No matter what sort of iPhone app you're dealing with, the lessons you've learned here should help you bring it to the iPad.

www.it-ebooks.info

www.it-ebooks.info

Chapter 12

Additional Resources for iPad Development

You've made your way through this book, mastering all of the new iPad features in the iPhone SDK. Are you ready for more? We've got you covered. This chapter points you to resources for additional iPad development assistance, handy programming tips, design aids, and other information.

Here, you'll find our own curated lists of recommended web sites, blogs, community forums, and books. Some of the links lead to commercial products, but most of the listed resources are free. The lists are by no means comprehensive, but they should serve as a helpful starting point for continuing your iPad development journey.

Logging in to the Mother Ship

As you might expect, the center of your iPhone and iPad universe is the Apple Developer Center. Your first stop in acquiring more iPad knowledge should be the vast online archive of developer documentation, sample code, and resources that Apple provides. Much of the primary documentation is available both online and as downloadable PDFs.

If you don't have time to comb through Apple's entire treasure trove of documentation, the essential must-read items for all iPad developers are the *iPad Programming Guide* and the *iPad Human Interface Guidelines*. These outline important tips and rules for creating high-quality iPad applications for the iTunes App Store.

NOTE: Some links may require logging into the iPhone Developer Program for access.

www.it-ebooks.info

iPad Development

The Apple Developer Center offers the following iPad development documentation:

- *The iPhone OS Reference Library and Sample Code*: Available online at `http://developer.apple.com/iphone/library/navigation/`
- *iPad Programming Guide*: Available online at `http://developer.apple.com/iphone/library/documentation/General/Conceptual/iPadProgrammingGuide/` and downloadable from `http://developer.apple.com/iphone/library/documentation/General/Conceptual/iPadProgrammingGuide/iPadProgrammingGuide.pdf`
- *iPad Human Interface Guidelines*: Available online at `http://developer.apple.com/iphone/library/documentation/General/Conceptual/iPadHIG/` and downloadable from `http://developer.apple.com/iphone/library/documentation/General/Conceptual/iPadHIG/iPadHIG.pdf`
- *Introduction to Creating Universal Applications:* `http://devimages.apple.com/iphone/resources/introductiontouniversalapps.pdf`

Objective-C and Cocoa Touch

The Apple Developer Center offers the following Objective-C and Cocoa Touch documentation:

- *Objective-C Programming Language Reference*: Available online at `http://developer.apple.com/iphone/library/documentation/Cocoa/Conceptual/ObjectiveC/` and downloadable from `http://developer.apple.com/iphone/library/documentation/Cocoa/Conceptual/ObjectiveC/ObjC.pdf`
- *Cocoa Fundamentals Guide*: Available online at `http://developer.apple.com/iphone/library/documentation/Cocoa/Conceptual/CocoaFundamentals/` and downloadable from `http://developer.apple.com/iphone/library/documentation/Cocoa/Conceptual/CocoaFundamentals/CocoaFundamentals.pdf`
- *UIKit Framework Reference:* Available online at `http://developer.apple.com/iphone/library/documentation/UIKit/Reference/UIKit_Framework/` and downloadable from `http://developer.apple.com/iphone/library/documentation/UIKit/Reference/UIKit_Framework/UIKit_Framework.pdf`

www.it-ebooks.info

iPad App Deployment

The following are Apple resources for iPad App deployment:

- iPhone Provisioning Portal User Guide for App Testing and Ad-Hoc Distribution: `http://developer.apple.com/iphone/manage/overview/index.action`
- iTunes Connect Developer Guide: `http://itunesconnect.apple.com/docs/iTunesConnect_DeveloperGuide.pdf`
- App Store Resource Center: `https://developer.apple.com/iphone/appstore/`
- News and Announcements for iPhone App Developers: `https://developer.apple.com/iphone/news/`

Learning from the Experts

Wading through Apple's dense sea of online documentation can sometimes feel like searching for a needle in a haystack. Books provide a more focused, structured approach to learning specific topics. You can also get valuable information from the blogs and web sites of leading app developers.

> **Note:** All of the code examples listed in this book, along with the full source code of the iPad drawing app, Dudel, can be downloaded from `http://www.apress.com/book/view/9781430230212`.

Books

Apress offers many comprehensive books on Objective-C, Cocoa Touch, and iPhone and iPad development, including the following:

- *Beginning iPhone and iPad Development with SDK 4: Exploring the iPhone SDK* by Jack Nutting, Dave Mark, and Jeff LaMarche (`http://www.apress.com/book/view/9781430230243`)
- *More iPhone and iPad Development: Further Explorations of the iPhone SDK* by Jack Nutting, Dave Mark, and Jeff LaMarche (`http://www.apress.com/book/view/9781430232520`)
- *The Business of iPhone App Development: Making and Marketing Apps that Succeed* by Dave Wooldridge with Michael Schneider (`http://www.apress.com/book/view/9781430227335`)
- *Building iPhone OS Accessories: Use the iPhone Accessories API to Control and Monitor Devices* by Ken Maskrey (`http://www.apress.com/book/view/9781430229315`)

www.it-ebooks.info

Tutorials and Code Examples

The following are some of our favorite blogs and web sites, which offer helpful tutorials, example projects, and code snippets for iPad apps:

- iCodeBlog app programming tutorials (`http://icodeblog.com/`)
- Dr. Touch's development blog (`http://www.drobnik.com/touch/`)
- Games from Within, indie iPhone/iPad game development (`http://gamesfromwithin.com/`)
- iPhoneDev Central (`http://www.iphonedevcentral.com/`)
- iPhoneFlow Development Community Links (`http://www.iphoneflow.com/`)
- iPhone Development Bits (`http://iphonedevelopmentbits.com/`)
- iPhone Developer Tips (`http://iphonedevelopertips.com/`)
- iPhone Dev FAQ (`http://www.iphonedevfaq.com/`)
- iPhone Development Blog (`http://iphoneincubator.com/blog/`)
- Jeff LaMarche's iPhone development blog (`http://iphonedevelopment.blogspot.com/`)
- Majic Jungle's development blog (`http://majicjungle.com/blog/`)
- ManicDev's iPhone and iPad SDK development tutorials and tips (`http://maniacdev.com/`)
- Mark Johnson's developer blog (`http://www.markj.net/`)
- Matt Legend Gemmell's blog (`http://mattgemmell.com/`)
- Ray Wenderlich's developer blog (`http://www.raywenderlich.com/`)

Designing User Interfaces for iPad Apps

It should go without saying that an attractive, intuitive, easy-to-use interface is a major key to the success of your iPad app. And beyond the importance of usability, the iPad's large screen demands a beautiful, visual experience. To help you in this quest, we've listed links to several time-saving templates, graphics collections, and design tools.

www.it-ebooks.info

NOTE: If you haven't read Apple's *iPad Human Interface Guidelines* yet, do yourself a favor and check it out. Not only does it offer essential design tips and recommendations for building effective user interfaces, but following Apple's guidelines can also help prevent UI-related rejections when submitting your iPad application to the iTunes App Store.

Paper Prototyping

The following are some paper prototyping products for iPad app design:

- Kapsoft's iPad Stencil (`http://www.mobilesketchbook.com/`)
- *iPad Application Sketch Book* by Dean Kaplan (`http://www.apress.com/book/view/9781430232049`)
- *The Developer Sketchbook for iPad Apps* by Dave Wooldridge (`http://developersketchbook.com/`)
- UI Stencils' iPad Stencil Kit (`http://www.uistencils.com/products/ipad-stencil-kit`)

Digital Mockups

The following digital templates are available for iPad app design:

- Endloop's iMockups (`http://www.imockups.com/`)
- Balsamiq Mockups (`http://www.balsamiq.com/products/mockups/`)
- Briefs, a Cocoa Touch framework for live wireframes (`http://giveabrief.com/`)
- Teehan+Lax's iPad GUI PSD, for Photoshop (`http://www.teehanlax.com/blog/2010/02/01/ipad-gui-psd/`)
- Kevin Andersson's iPad editable PSD, for Photoshop (`http://blog.kevinandersson.dk/2010/01/29/apple-ipad-fully-editable-psd/`)
- RawApps' iPad GUI Kit PSD, for Photoshop (`http://www.rawapps.com/849/ipad-gui-kit-in-psd-format-is-here/`)
- Iconshock's iPad vector GUI elements, for Illustrator (`http://iconlibrary.iconshock.com/icons/ipad-vector-gui-elements-tabs-buttons-menus-icons/`)
- Dave Morford's iPhone/iPad stencil, for OmniGraffle (`http://www.morford.org/iphoneosdesignstencil/`)
- iA's iPad Stencil, for OmniGraffle (`http://informationarchitects.jp/ipad-stencil-for-omnigraffle/`)

d

www.it-ebooks.info

User Interface Icons

Here are some places where you can find icons for iPad apps:

- app-bits iPhone Toolbar Icon Set (`http://www.app-bits.com/downloads/iphone-toolbar-icon-set.html`)
- Cocoia blog's iPhone/iPad icon PSD, for Photoshop (`http://blog.cocoia.com/2010/iphone-ipad-icon-psd-template/`)
- Dezinerfolio vector icons (`http://www.dezinerfolio.com/freebie/30-free-vector-icons`)
- eddit iPhone UI Icon Set (`http://eddit.com/shop/iphone_ui_icon_set/`)
- Glyphish icons for iPhone and iPad apps (`http://glyphish.com/`)
- iconSweets Photoshop icons (`http://www.iconsweets.com/`)
- PixelPressIcon Whitespace Icon Collection (`http://www.pixelpressicons.com/?page_id=118`)
- RawApps iPad icon set (`http://www.rawapps.com/4905/rawapps-com-launches-ipad-icon-set-ver-1-download-it-today/`)
- The Working Group's iPhone toolbar icons (`http://blog.twg.ca/2009/09/free-iphone-toolbar-icons/`)

Design Considerations and Inspirations

Learn more about iPad interface design from these resources:

- Matt Legend Gemmell's observations on iPad application design (`http://mattgemmell.com/2010/03/05/ipad-application-design`)
- *Smashing Magazine*'s "Useful Design Tips for Your iPad App" (`http://www.smashingmagazine.com/2010/04/16/design-tips-for-your-ipad-app/`)
- iA's "Designing for iPad: Reality Check" "(`http://informationarchitects.jp/designing-for-ipad-reality-check/`)
- Landing Pad—A Showcase of Beautiful iPad App Design (`http://landingpad.org/`)
- iPad Apps That Don't Suck (`http://ipadappsthatdontsuck.com/`)
- *Touch Gesture Reference Guide* (`http://www.lukew.com/touch/`)

www.it-ebooks.info

Finding Answers in Online Forums

What if you need a little one-on-one help? Thankfully, the iPhone and iPad developer community members are very generous and willing to share their wealth of knowledge online for the greater good of the group.

NOTE: What goes around comes around. If people take the time to provide you with assistance, be sure to pay it forward by replying to posted questions if you know the answers.

Here are a few popular web forums worth visiting:

- The Official Book Forum for Beginning iPad Development, and Beginning iPhone Development and More iPhone Development (`http://iphonedevbook.com/forum/`)
- Apple Developer Forums (`https://devforums.apple.com/community/iphone`) iPhone Developer Program login required
- iPhone Dev SDK Forum (`http://www.iphonedevsdk.com/`)
- iDevApps iPhone and iPad Programming Forum (`http://www.idevapps.com/forum/`)
- iPhone Dev Forums (`http://www.iphonedevforums.com/`)
- iDevGames iPhone and iPad Game Developers Forum (`http://www.idevgames.com/forum/`)
- Mac Rumors iPhone/iPad Programming Forum (`http://forums.macrumors.com/forumdisplay.php?f=135`)
- Stack Overflow, collaborative questions and answers for programmers (`http://stackoverflow.com/`)

Holding the Future

Congratulations on working your way through each and every chapter as we tackled all of the new iPad features in the iPhone SDK. We certainly covered a lot of ground, so you should now feel confident in creating your own iPad apps. And this is just the beginning. We can’t wait to see what cool new features Apple has in store for future versions of its wildly popular iPad tablet. Until then, thanks for reading!

www.it-ebooks.info

www.it-ebooks.info

Index

www.it-ebooks.info

www.it-ebooks.info

www.it-ebooks.info

www.it-ebooks.info

www.it-ebooks.info

www.it-ebooks.info

www.it-ebooks.info

www.it-ebooks.info

www.it-ebooks.info

www.it-ebooks.info

www.it-ebooks.info

www.it-ebooks.info

www.it-ebooks.info